Semi-Oblivious Chase Termination for Linear Existential Rules and Beyond: An Experimental Analysis

AMIRHOSSEIN ALIZAD, University of Western Ontario, London, Canada
MARCO CALAUTTI, University of Milan, Milano, Italy
MOSTAFA MILANI, University of Western Ontario, London, Canada
ANDREAS PIERIS, University of Cyprus, Nicosia, Cyprus and University of Edinburgh, Edinburgh, United Kingdom of Great Britain and Northern Ireland

The chase procedure is a fundamental algorithmic tool in databases that allows us to reason with constraints, such as existential rules, with a plethora of applications. It takes a database and a set of constraints as input and iteratively completes the database as dictated by the constraints. A key challenge, though, is the fact that the chase may not terminate, which leads to the problem of checking whether it terminates given a database and a set of constraints. In this work, we focus on the semi-oblivious version of the chase, which is well-suited for practical implementations, and linear existential rules, a central class of constraints with several applications. In this setting, there is a mature body of theoretical work that provides syntactic characterizations of when the chase terminates, algorithms for checking chase termination, precise complexity results, and worst-case optimal bounds on the size of the result of the chase (whenever it is finite). Our main objective is to experimentally evaluate the existing chase termination algorithms with the aim of understanding which input parameters affect their performance, clarifying whether they can be used in practice, and revealing their performance limitations. Concerning guarded existential rules, a natural generalization of linear existential rules, one can reuse the machinery for linear existential rules by first applying the so-called linearization technique, that is, the technique of converting guarded existential rules into linear existential rules without affecting the termination of the chase. A secondary objective of this work is to understand how realistic is the use of the linearization technique in the context of the semi-oblivious chase termination problem.

CCS Concepts: • **Information systems → Data management systems;**

Additional Key Words and Phrases: Semi-oblivious chase, linear existential rules, termination

ACM Reference Format:
Amirhossein Alizad, Marco Calautti, Mostafa Milani, and Andreas Pieris. 2026. Semi-Oblivious Chase Termination for Linear Existential Rules and Beyond: An Experimental Analysis. *ACM Trans. Datab. Syst.* 51, 3, Article 16 (March 2026), 48 pages. https://doi.org/10.1145/3785662

This work was funded by the European Union - Next Generation EU under the MUR PRIN-PNRR grant P2022KHTX7 "DISTORT" and the NSERC Discovery Grant 2021-04120.
Authors' Contact Information: Amirhossein Alizad, University of Western Ontario, London, Ontario, Canada; e-mail: aaliza8@uwo.ca; Marco Calautti, University of Milan, Milano, Italy; e-mail: marco.calautti@unimi.it; Mostafa Milani, University of Western Ontario, London, Ontario, Canada; e-mail: mostafa.milani@uwo.ca; Andreas Pieris (corresponding author), University of Cyprus, Nicosia, Cyprus and University of Edinburgh, Edinburgh, United Kingdom of Great Britain and Northern Ireland; e-mail: pieris.andreas@ucy.ac.cy.

ACM 0362-5915/2026/03-ART16
https://doi.org/10.1145/3785662

1 Introduction

The *chase procedure* (or chase) is a fundamental algorithmic tool that has been successfully applied to several database problems such as checking logical implication of constraints [Beeri and Vardi 1984; Maier et al. 1979], containment of queries under constraints [Aho et al. 1979], computing data exchange solutions [Fagin et al. 2005], and ontological query answering [Calì et al. 2012], to name a few. It takes as input a database D and a set Σ of constraints, which, for this work, are *existential rules* (a.k.a. ***tuple-generating dependencies*** (**TGDs**)), that is, first-order sentences of the form

$$\forall \bar{x} \forall \bar{y} \, (\phi(\bar{x}, \bar{y}) \ \rightarrow \ \exists \bar{z} \, \psi(\bar{x}, \bar{z})) \, ,$$

where ϕ (the body) and ψ (the head) are conjunctions of relational atoms, and it produces an instance D_Σ that is a *universal model* of D and Σ, i.e., a model that can be homomorphically embedded into every other model of D and Σ. Intuitively, D_Σ acts as a representative of all the models of D and Σ. This is the reason for the ubiquity of the chase in databases, as discussed in Deutsch et al. [2008]. Indeed, many database problems can be solved by simply exhibiting a universal model.

1.1 The Chase in a Nutshell

Roughly, the chase adds new tuples to the database D (possibly involving null values that act as witnesses for the existentially quantified variables), as dictated by the TGDs of Σ, and it keeps doing this until all the TGDs of Σ are satisfied. There are, in principle, three different ways for formalizing this simple idea, which lead to different versions of the chase procedure.

Oblivious Chase. The first one, which leads to the *oblivious chase*, is as follows: for each pair $(\bar{t}, \bar{u})$ of tuples of terms from the instance I constructed so far, apply a TGD $\sigma = \forall \bar{x} \forall \bar{y} \, (\phi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} \, \psi(\bar{x}, \bar{z}))$ if $\phi(\bar{t}, \bar{u}) \subseteq I$, and σ has not been applied in a previous chase step due to the same pair $(\bar{t}, \bar{u})$, and add to I the set of atoms $\psi(\bar{t}, \bar{v})$, where $\bar{v}$ is a tuple of new terms not occurring in I.

Semi-Oblivious Chase. The second one, which is a refinement of the oblivious chase, and it gives rise to the *semi-oblivious chase*, is as follows: for each pair $(\bar{t}, \bar{u})$ of tuples of terms from the instance I constructed so far, apply a TGD $\sigma = \forall \bar{x} \forall \bar{y} \, (\phi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} \, \psi(\bar{x}, \bar{z}))$ if $\phi(\bar{t}, \bar{u}) \subseteq I$, and σ has not been applied in a previous chase step due to a pair of tuples $(\bar{t}, \bar{u}')$, where $\bar{u}$ and $\bar{u}'$ might be different, and add to I the set of atoms $\psi(\bar{t}, \bar{v})$, where $\bar{v}$ is a tuple of new terms not in I. In other words, a TGD σ of the above form is applied only once due to a certain witness $\bar{t}$ for the variables $\bar{x}$.

Restricted Chase. The third one, which leads to the *restricted* (a.k.a. *standard*) *chase*, is as follows: for each pair $(\bar{t}, \bar{u})$ of tuples of terms from the instance I constructed so far, apply a TGD $\sigma = \forall \bar{x} \forall \bar{y} \, (\phi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} \, \psi(\bar{x}, \bar{z}))$ if $\phi(\bar{t}, \bar{u}) \subseteq I$, and there is no tuple $\bar{u}'$ of terms from I such that $\psi(\bar{t}, \bar{u}') \subseteq I$, i.e., the TGD is not already satisfied, and add to I the set of atoms $\psi(\bar{t}, \bar{v})$, where $\bar{v}$ is a tuple of new terms not in I.

Summing up, the key difference between the oblivious, the semi-oblivious, and the restricted versions of the chase procedure is that the first two essentially apply a TGD whenever the body is satisfied, whereas the latter applies a TGD if the body is satisfied but the head is not.

1.2 Restricted vs. (Semi-)Oblivious Chase

It is not difficult to verify that the restricted chase, in general, builds smaller instances than the (semi-)oblivious one. In fact, it is easy to devise an example where, according to the restricted chase, none of the TGDs should be applied, while the (semi-)oblivious chase builds an infinite instance. Here is such an example taken from Calautti and Pieris [2021]:

Example 1.1. Consider the database $D = \{R(a, a)\}$ and the TGD

$$\forall x \forall y (R(x, y) \ \rightarrow \ \exists z \, R(z, x)).$$

The restricted chase will detect that the database already satisfies the TGD and do nothing, while the (semi-)oblivious chase will build the instance

$$\{R(a,a), R(\bot_1,a), R(\bot_2,\bot_1), R(\bot_3,\bot_2), \ldots\},$$

where $\bot_1, \bot_2, \bot_3, \ldots$ are (labeled) nulls. ∎

It is also easy to devise examples where the semi-oblivious chase does not apply any TGD, whereas the oblivious chase builds an infinite instance. It is generally agreed that the oblivious version of the chase, although a very useful theoretical tool, has no practical applications due to the fact that it infers a lot of redundant information, which in turn leads to very large instances that are very often infinite. Concerning the other variants of the chase, the restricted one has a clear advantage over the semi-oblivious one, as it generally builds smaller instances. But, of course, this advantage does not come for free: at each step, the restricted chase has to check that there is no way to satisfy the head of the TGD at hand, and this can be very costly in practice. It has been observed that for RAM-based implementations, the restricted chase is the indicated approach since the benefit from producing smaller instances justifies the additional effort to check whether a TGD is already satisfied; see, e.g., Benedikt et al. [2017]; Krötzsch et al. [2019]. However, as discussed in Benedikt et al. [2017], an RDBMS-based implementation of the restricted chase is quite challenging, whereas an efficient implementation of the semi-oblivious chase is feasible. Hence, both the semi-oblivious and restricted versions of the chase are relevant tools for practical implementations.

1.3 Chase Termination for Linear TGDs and Beyond

There are indeed efficient implementations of the semi-oblivious and restricted chase that allow us to solve central database problems by adopting a materialization-based approach [Benedikt et al. 2017; Krötzsch et al. 2019; Nenov et al. 2015; Urbani et al. 2018]. Nevertheless, for this to be feasible in practice, we need a guarantee that the chase terminates. There are settings where the termination of the chase is guaranteed by design. For example, in data exchange, there is an *a priori* assumption that the chase terminates as the adopted languages (e.g., weakly-acyclic TGDs) are deliberately designed to ensure the termination of the chase [Fagin et al. 2005]. There are settings, however, where such an *a priori* assumption cannot be realistically made. Such a setting, which is of special interest for our work, is that of ontological reasoning. Indeed, even lightweight TGD-based ontology languages allow us to express non-acyclic statements with value invention, a combination that may lead to an infinite chase; see, e.g., Grau et al. [2013]. This fact motivated a long line of research on the chase termination problem, that is, given a database D and a set Σ of TGDs, to devise sound and complete algorithms that check whether the semi-oblivious or restricted chase of D with Σ terminates. It is known that, in general, this is an undecidable problem. This has been established in Deutsch et al. [2008] for the restricted chase, and it was observed a year later in Marnette [2009] that the same proof shows undecidability also for the semi-oblivious chase. The undecidability proof given in Deutsch et al. [2008], however, constructs a sophisticated set of TGDs that goes beyond existing well-behaved classes of TGDs that enjoy certain syntactic properties. This observation leads to the obvious question: Is the chase termination problem algorithmically solvable whenever we focus on well-behaved classes of TGDs?

A well-behaved class of TGDs, which forms a central ontology language that attracted a considerable attention due to its simplicity and the fact that it strikes a good balance between expressiveness and complexity, is that of linear TGDs proposed in Calì et al. [2012]. A TGD is *linear* if it has only one atom in its body, whereas the head can be an arbitrary conjunction of atoms. Such a TGD is called *simple-linear* if each variable in its body occurs only once, whereas variables in its head can repeat without any restriction. Although at first glance, (simple-)linear TGDs may

look very inexpressive, it turns out that they are powerful enough for modelling ontologies. In particular, the practically important ontology language DL-Lite$_R$ [Calvanese et al. 2007], which is based on **Description Logics** (**DLs**) and forms the logical underpinning of OWL 2 QL, one of the popular profiles of the W3C committee's **Web Ontology Language** (**OWL**) standard for ontology languages, can be easily embedded into the class of simple-linear TGDs.

The chase termination problem in the presence of (simple-)linear TGDs has been extensively studied over the last few years. Concerning the semi-oblivious version of the chase, there is a mature body of theoretical work that provides syntactic characterizations of when the chase terminates based on suitable acyclicity notions, algorithms for checking chase termination, precise complexity results, and worst-case optimal bounds on the size of the chase instance (whenever it is finite) [Calautti et al. 2022]. On the other hand, for the restricted version of the chase, we only have a decidability result via an algorithm that runs in elementary time under the assumption that the head of the linear TGDs consists of a single atom. Actually, this has been independently shown in Leclère et al. [2019] using an approach based on derivation trees, and in Gogacz et al. [2020] by providing a reduction to the satisfiability problem of Monadic Second Order Logic over infinite trees [Gogacz et al. 2020]. Needless to say that the syntactic characterizations of the termination of the semi-oblivious chase, as well as the existing algorithms for checking the termination of the semi-oblivious chase, are far from being applicable to the restricted chase, and there is no way to be merely adapted in order to be applicable to the restricted chase. This striking difference on the progress that has been achieved should be attributed to the fact that the chase termination problem is significantly more challenging in the case of the restricted chase.

A natural generalization of linear TGDs, which has been the subject of several research projects during the last decade or so, is the class of guarded TGDs proposed in Calì et al. [2013]. A TGD is *guarded* if it has an atom in its body that contains all the universally quantified variables. It is known that the central DL EL [Baader et al. 2005], which forms the logical underpinning of OWL 2 EL, another popular profile of OWL, can be embedded (up to a certain normal form) into the class of guarded TGDs. The chase termination problem in the presence of guarded TGDs has been extensively studied in the literature [Calautti et al. 2015, 2022; Gogacz et al. 2023]. Concerning the semi-oblivious version of the chase, it has been recently shown in Calautti et al. [2022] that one can effectively use the machinery for linear TGDs by first converting the given set of guarded TGDs into a set of linear TGDs via the so-called *linearization technique* without affecting the termination of the semi-oblivious chase. On the other hand, for the restricted version of the chase, we only have a decidability result via a sophisticated algorithm that relies on Monadic Second Order Logic and runs in elementary time, under the assumption that the head of the guarded TGDs consists of a single atom [Gogacz et al. 2023]. As for linear TGDs, there is a noticeable difference in the progress that has been achieved for the two versions of the chase due to the remarkably more challenging nature of the problem in the case of the restricted chase.

1.4 Our Objectives

Having a complete theoretical understanding of the semi-oblivious chase termination problem in the presence of (simple-)linear TGDs, the next step is to experimentally evaluate the proposed algorithms with the aim of understanding which input parameters affect their performance, clarifying whether they can be applied in a practical context, and revealing their performance limitations. This is precisely the main objective of this work. Note that, we do not consider the restricted chase, as this will be very premature due to the lack of a good theoretical understanding of the problem in question; the latter is the subject of an ongoing research activity.

From the chase termination literature, we can inherit two types of algorithms for the semi-oblivious chase termination problem in the presence of (simple-)linear TGDs, that is, *materialization-based* and *acyclicity-based* [Calautti et al. 2022], which can be described as follows:

Materialization-based. The materialization-based algorithms exploit the existence of worst-case optimal bounds on the size of the chase instance (whenever it is finite). In particular, given a database D and a set Σ of (simple-)linear TGDs, we have an integer $k_{D,\Sigma}$ such that the chase of D with Σ terminates iff the size of the chase instance (i.e., the number of its atoms) is at most $k_{D,\Sigma}$. In particular, $k_{D,\Sigma} = ||D|| \cdot (n+1) \cdot ||\Sigma||^{2 \cdot m \cdot (n+1)}$, where $||D||$ denotes the size of D, $||\Sigma||$ denotes the size of Σ, m is the maximum arity over all predicates occurring in Σ, and $n = \ell \cdot m$ (resp., $n = \ell \cdot m^{m+1}$) if Σ is a set of simple-linear TGDs (resp., a set of linear TGDs), where ℓ is the number of predicates occurring in Σ. This leads to a conceptually simple chase termination algorithm: simply run the semi-oblivious chase of D with Σ and keep a counter for the number of generated atoms, and if the count exceeds $k_{D,\Sigma}$, then conclude that the chase does not terminate; otherwise, it does.

Acyclicity-based. On the other hand, the acyclicity-based algorithms exploit the syntactic characterizations of when the chase terminates via suitable acyclicity notions. In particular, given a database D and a set Σ of (simple-)linear TGDs, the chase of D with Σ terminates iff the dependency graph of Σ (a standard way of representing a set of TGDs as a graph, which is defined in Section 3) does not contain a "bad" cycle that is supported by the database D, where a "bad" cycle witnesses the fact that during the chase of D with Σ we eventually fall in a cyclic chase derivation that leads to non-termination. This again leads to a conceptually simple chase termination algorithm: construct the dependency graph of Σ, and if it has a "bad" cycle supported by D, then the chase does not terminate; otherwise, it does.

We performed an exploratory analysis revealing that the algorithms based on materialization are too expensive in practice; our implementation of the materialization-based algorithms exploits VLog, a state-of-the-art chase engine [Urbani et al. 2018]. We considered 483 pairs (D, Σ), where D is a database and Σ a set of simple-linear TGDs, obtained from the ontology repository of the Data and Knowledge Group of the University of Oxford by keeping from the available ontologies, which are actually modelled using DLs, only the ontological axioms that can be expressed as simple-linear TGDs. For each such pair, we checked whether the semi-oblivious chase of D with Σ terminates by using both the materialization- and the acyclicity-based algorithms. We observed that for 28% of the scenarios, the materialization-based algorithm runs out of memory, whereas the acyclicity-based algorithm provides an answer in 0.5 seconds in the worst-case. This is because the worst-case upper bound on the size of the result of the chase from [Calautti et al. 2022] is very large, and thus, the materialization-based algorithm is forced, in general, to construct extremely large instances before being able to safely recognize that the chase does not terminate. For the remaining 72% of the scenarios, the acyclicity-based algorithm consistently outperforms the materialization-based algorithm. Overall, for 50% of the considered scenarios, either the materialization-based algorithm fails, or the acyclicity-based algorithm is at least 10 times faster. Therefore, towards our main objective, we focused our attention on the acyclicity-based algorithms.

As a side comment, let us say that the above exploratory analysis allows us to draw some conclusions concerning the overhead produced by the acyclicity-based algorithm when we want to actually compute the chase whenever the chase termination check succeeds. This is because the time needed to actually compute the chase corresponds to the runtime of the materialization-based termination algorithm when the input pair (D, Σ) ensures the termination of the chase. By analyzing the results of our exploratory analysis focusing on the scenarios (D, Σ) that guarantee

the termination of the chase, we observed that whenever the chase terminates very quickly (within 0.25 seconds), the acyclicity-based algorithm is also very fast, with a comparable runtime of at most 0.2 seconds. On the other hand, as the time for computing the chase increases, the runtime of the acyclicity-based algorithm remains consistently low. More precisely, we have observed that, when the computation of the chase exceeds 0.25 seconds, the chase termination check via the acyclicity-based algorithm contributes only about 3% of the total time on average for checking for chase termination and then computing the chase whenever termination is guaranteed.

A secondary objective of this work is to understand how realistic is the use of the linearization technique in the context of the semi-oblivious chase termination problem. In other words, the goal is to evaluate the algorithm that first linearizes the given set of guarded TGDs, and then checks whether the dependency graph of the linearized set of TGDs contains a "bad" cycle.

1.5 Main Outcome and Technical Challenges

Our experimental analysis revealed that for simple-linear TGDs, the primary parameter impacting the runtime of the acyclicity-based algorithm is the size of the input set of TGDs, whereas the size of the input database does not play any crucial role. Interestingly, the algorithm is very fast (in the order of seconds) even for large sets of simple-linear TGDs (with up to 100K TGDs).

Now, concerning the technically more interesting case of linear TGDs, our analysis showed that the acyclicity-based algorithm consists of two components that are of different nature. In particular, there is a database-dependent component whose performance is solely impacted by the size of the database, and a database-independent component whose runtime is primarily affected by the size of the set of TGDs. Interestingly, the overall runtime of the algorithm is quite reasonable, which is a strong evidence that fast checking for the termination of the semi-oblivious chase in the case of linear TGDs is not an unrealistic goal. Note that most of the total end-to-end runtime of the algorithm is taken by the database-dependent component, which indicates that our future efforts should be focused on improving that component.

Towards our secondary objective concerning guarded TGDs and the linearization technique, we considered a special fragment of guarded TGDs that corresponds to the widely used DL EL; recall that EL can be embedded (up to a certain normal form) into the class of guarded TGDs. Our analysis revealed that, even with this lightweight fragment of guarded TGDs, we cannot hope to go far in practice by relying on the linearization technique, unless we focus on very small sets of TGDs with a few hundred TGDs.

Technical Challenges. Towards the above outcome concerning the acyclicity-based algorithms, we had to overcome several technical challenges that led to results of independent interest:

- It would not be possible to obtain the above insightful conclusions by naively implementing the algorithms in question, as this would lead to poor performance; this is discussed in Section 4 for the case of (simple-)linear TGDs and in Section 10 for the case of guarded TGDs. Hence, we had to revisit and adapt the theoretical algorithms from Calautti et al. [2022] in order to obtain algorithms that are amenable to efficient implementations. The low-level implementation details of the adapted algorithms for (simple-)linear TGDs can be found in Section 5 and for guarded TGDs in Section 10.
- In order to stress test the algorithms in question for (simple-)linear TGDs, we had to synthetically generate databases and sets of TGDs. However, as discussed in Section 6, existing data and TGD generators are not suitable for our purposes since they do not allow us to tune certain parameters that are crucial for evaluating our chase termination algorithms. To this end, we developed our own data and TGD generators, and used them to carefully generate the databases and TGDs that have been employed in our experimental analysis.

All the relevant details can be found at https://github.com/chasetermination/Chase-Termination.

2 Preliminaries

We consider the disjoint countably infinite sets $\mathbf{C}$, $\mathbf{N}$, and $\mathbf{V}$ of *constants*, *(labeled) nulls*, and *variables*; we refer to constants, nulls and variables as *terms*. For $n > 0$, let $[n]$ be the set $\{1, \ldots, n\}$.

Relational Databases. A *schema* $\mathbf{S}$ is a finite set of relation symbols (or predicates) with associated arity. We write R/n to denote that R has arity $n > 0$; we may also write $\text{ar}(R)$ for the integer n. A *(predicate) position* of $\mathbf{S}$ is a pair (R, i), where $R/n \in \mathbf{S}$ and $i \in [n]$, that essentially identifies the ith argument of R. We write $\text{pos}(\mathbf{S})$ for the set of positions of $\mathbf{S}$, that is, the set $\{(R, i) \mid R/n \in \mathbf{S} \text{ and } i \in [n]\}$. An *atom* over $\mathbf{S}$ is an expression of the form $R(\bar{t})$, where $R/n \in \mathbf{S}$ and $\bar{t}$ is an n-tuple of terms. A *fact* is an atom whose arguments consist only of constants. For a variable x in $\bar{t} = (t_1, \ldots, t_n)$, let $\text{pos}(R(\bar{t}), x) = \{(R, i) \mid t_i = x\}$. We write $\text{var}(R(\bar{t}))$ for the set of variables in $\bar{t}$. The notations $\text{pos}(\cdot, x)$ and $\text{var}(\cdot)$ extend to sets of atoms. An *instance* over $\mathbf{S}$ is a (possibly infinite) set of atoms over $\mathbf{S}$ with constants and nulls. A *database* over $\mathbf{S}$ is a finite set of facts over $\mathbf{S}$. The *active domain* of an instance I, denoted $\text{dom}(I)$, is the set of terms (constants and nulls) occurring in I. For a singleton instance $\{\alpha\}$, we simply write $\text{dom}(\alpha)$ instead of $\text{dom}(\{\alpha\})$.

Substitutions and Homomorphisms. A *substitution* from a set of terms T to a set of terms T' is a function $h : T \to T'$. Henceforth, we treat a substitution h as the set of mappings $\{t \mapsto h(t) \mid t \in T\}$. The restriction of h to a subset S of T, denoted $h_{|S}$, is the substitution $\{t \mapsto h(t) \mid t \in S\}$. A *homomorphism* from a set of atoms A to a set of atoms B is a substitution h from the set of terms in A to the set of terms in B such that h is the identity on $\mathbf{C}$, and $R(t_1, \ldots, t_n) \in A$ implies $h(R(t_1, \ldots, t_n)) = R(h(t_1), \ldots, h(t_n)) \in B$.

TGDs. A TGD σ is a (constant-free) first-order sentence of the form

$$\forall \bar{x} \forall \bar{y} \, (\phi(\bar{x}, \bar{y}) \;\rightarrow\; \exists \bar{z} \, \psi(\bar{x}, \bar{z})) \,,$$

where $\bar{x}$, $\bar{y}$ and $\bar{z}$ are tuples of variables of $\mathbf{V}$, and $\phi(\bar{x}, \bar{y})$ and $\psi(\bar{x}, \bar{z})$ are non-empty conjunctions of atoms that mention only variables from $\bar{x} \cup \bar{y}$ and $\bar{x} \cup \bar{z}$, respectively. Note that, by abuse of notation, we may treat a tuple of variables as a set of variables. We write σ as $\phi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} \, \psi(\bar{x}, \bar{z})$, and use comma instead of $\wedge$ for joining atoms. We refer to $\phi(\bar{x}, \bar{y})$ and $\psi(\bar{x}, \bar{z})$ as the *body* and *head* of σ, denoted $\text{body}(\sigma)$ and $\text{head}(\sigma)$, respectively. The *frontier* of the TGD σ, denoted $\text{fr}(\sigma)$, is the set of variables $\bar{x}$, i.e., the variables that appear both in the body and the head of σ. The *schema* of a set Σ of TGDs, denoted $\text{sch}(\Sigma)$, is the set of predicates occurring in Σ.

An instance I satisfies a TGD σ as the one above, written $I \models \sigma$, if whenever there exists a homomorphism h from $\phi(\bar{x}, \bar{y})$ to I, then there is $h' \supseteq h_{|\bar{x}}$ that is a homomorphism from $\psi(\bar{x}, \bar{z})$ to I; by abuse of notation, we may treat a conjunction of atoms as a set of atoms. The instance I satisfies a set Σ of TGDs, written $I \models \Sigma$, if $I \models \sigma$ for each $\sigma \in \Sigma$.

Linearity. A TGD is *linear* if it has only one body-atom, and the corresponding class that collects all the finite sets of linear TGDs is denoted L. We further call a linear TGD *simple* if no variable occurs more than once in its body, and the corresponding class is denoted SL. It is clear that $\mathsf{SL} \subsetneq \mathsf{L}$.

3 The Semi-oblivious Chase Procedure

The semi-oblivious chase (or simply chase) takes as input a database D and a set Σ of TGDs, and constructs an instance that contains D and satisfies Σ. A key notion in this context is that of a trigger.

Definition 3.1. Given a set Σ of TGDs and an instance I, a *trigger* for Σ on I is a pair (σ, h), where $\sigma \in \Sigma$ and h is a homomorphism from $\text{body}(\sigma)$ to I. The *result* of (σ, h), denoted $\text{result}(\sigma, h)$, is the

set $\mu(\text{head}(\sigma))$, where $\mu : \text{var}(\text{head}(\sigma)) \to \mathbf{C} \cup \mathbf{N}$ is defined as follows:

$$\mu(x) = \begin{cases} h(x) & \text{if } x \in \text{fr}(\sigma) \\ \perp^{x}_{\sigma,h_{|\text{fr}(\sigma)}} & \text{otherwise,} \end{cases}$$

where $\perp^{x}_{\sigma,h_{|\text{fr}(\sigma)}} \in \mathbf{N}$. Let $T(\Sigma, I)$ be the set of triggers for Σ on I. ∎

Observe that in the definition of $\text{result}(\sigma, h)$, each existentially quantified variable x of $\text{head}(\sigma)$ is mapped by μ to a null value of $\mathbf{N}$ whose name is uniquely determined by the trigger (σ, h) and the variable x itself. This means that, given a trigger (σ, h), we can unambiguously construct the set of atoms $\text{result}(\sigma, h)$. The central idea of the chase is, starting from a database D, to exhaustively apply triggers for the given set Σ of TGDs on the instance constructed so far. More precisely, given a database D and a set Σ of TGDs, let

$$\text{chase}^0(D, \Sigma) = D,$$

and for each $i > 0$, let

$$\text{chase}^i(D, \Sigma) = \text{chase}^{i-1}(D, \Sigma) \cup \bigcup_{(\sigma,h)\in S} \text{result}(\sigma, h),$$

where $S = T(\Sigma, \text{chase}^{i-1}(D, \Sigma))$. We finally define *the result of the chase of D w.r.t. Σ* as the instance

$$\text{chase}(D, \Sigma) = \bigcup_{i\geq 0} \text{chase}^i(D, \Sigma).$$

3.1 Chase Termination

The result of the chase may be infinite even for very simple settings: it is easy to see that for

$$D = \{R(a, b)\} \quad \text{and} \quad \Sigma = \{R(x, y) \to \exists z\, R(y, z)\},$$

$\text{chase}(D, \Sigma)$ is infinite. This leads to the following problem, parameterized by a class C of TGDs such as SL (the class of simple-linear TGDs) and L (the class of linear TGDs):

INPUT :	A database D and a set Σ of TGDs from C.
QUESTION :	Is the instance $\text{chase}(D, \Sigma)$ finite?

This problem has been recently studied in Calautti et al. [2022] for the classes of simple-linear and linear TGDs. Interestingly, for both classes, the finiteness of the result of the chase has been syntactically characterized by exploiting the notion of non-uniform weak-acyclicity. We proceed to recall this acyclicity notion, and then present the characterizations established in Calautti et al. [2022], which in turn lead to simple algorithms for checking the finiteness of the result of the chase.

For the sake of clarity, in the rest of the article, we assume TGDs with a non-empty frontier, i.e., we assume that there is at least one variable in a TGD σ that occurs both in $\text{body}(\sigma)$ and $\text{head}(\sigma)$. This assumption can be made without loss of generality since, given a database D and a set Σ of TGDs, we can easily construct in linear time a database D' and a set Σ' of TGDs with a non-empty frontier by slightly modifying D and Σ such that $\text{chase}(D, \Sigma)$ is finite iff $\text{chase}(D', \Sigma')$ is finite. This can be done by simply extending each predicate of the schema with one position and add a new "dummy" constant (respectively, variable) at this new position in each atom of D (respectively, Σ).

3.2 Non-uniform Weak-acyclicity

Weak-acyclicity was introduced in Fagin et al. [2005] as the main formalism for data exchange purposes, which guarantees the finiteness of the result of the chase for *every* input database. Non-uniform weak-acyclicity is the database-dependent variant of weak-acyclicity introduced in Calautti et al. [2022]. We proceed to give the formal definitions. We first need to recall the notion of the *dependency graph* of a set Σ of TGDs, defined as the directed multigraph $\mathrm{dg}(\Sigma) = (N, E)$, where $N = \mathrm{pos}(\mathrm{sch}(\Sigma))$ and E contains *only* the following edges. For each TGD $\sigma \in \Sigma$ with $\mathrm{head}(\sigma) = \{\alpha_1, \ldots, \alpha_k\}$, for each $x \in \mathrm{fr}(\sigma)$, and for each position $\pi \in \mathrm{pos}(\mathrm{body}(\sigma), x)$:

- For each $i \in [k]$ and for each $\pi' \in \mathrm{pos}(\alpha_i, x)$, there exists a *normal* edge $(\pi, \pi') \in E$.
- For each existentially quantified variable z in σ, $i \in [k]$, and $\pi' \in \mathrm{pos}(\alpha_i, z)$, there is a *special* edge $(\pi, \pi') \in E$.

We further need to define when a predicate is reachable from another predicate. Given predicates $R, P \in \mathrm{sch}(\Sigma)$, *$P$ is reachable from R (w.r.t. Σ)* if $R = P$, or there exists a path in $\mathrm{dg}(\Sigma)$ from a position of the form (R, i) to a position of the form (P, j). Given a database D, we say that a (not necessarily simple and possibly cyclic) path C in $\mathrm{dg}(\Sigma)$ is *D-supported* if there exists an atom $R(\bar{t}) \in D$ and a node of the form (P, i) in C such that P is reachable from R. We are now ready to recall (non-uniform) weak-acyclicity.

Definition 3.2. Consider a database D and a set Σ of TGDs. We say that Σ is *weakly-acyclic w.r.t. D*, or *D-weakly-acyclic*, if there is no D-supported cycle in $\mathrm{dg}(\Sigma)$ with a special edge. We say that Σ is *weakly-acyclic* if there is no cycle in $\mathrm{dg}(\Sigma)$ with a special edge. ∎

3.3 Characterizing the Finiteness of the Chase

It is not difficult to show that whenever a set Σ of TGDs (not necessarily linear) is D-weakly-acyclic, then the instance $\mathrm{chase}(D, \Sigma)$ is finite. In other words, the D-weak-acyclicity of Σ is a sufficient condition for the finiteness of $\mathrm{chase}(D, \Sigma)$. What is more interesting is that, assuming that Σ is a set of simple-linear TGDs, the D-weak-acyclicity of Σ is also a necessary condition for the finiteness of $\mathrm{chase}(D, \Sigma)$. This leads to the following characterization established in Calautti et al. [2022]:

Theorem 3.3. *Consider a database D and a set $\Sigma \in \mathsf{SL}$ of TGDs. The following are equivalent:*
(1) $\mathrm{chase}(D, \Sigma)$ *is finite.*
(2) Σ *is D-weakly-acyclic.*

For linear TGDs, non-uniform weak-acyclicity is not enough for characterizing the finiteness of the chase instance. Here is an example from Calautti et al. [2022] that illustrates this fact:

Example 3.4. Consider the database $D = \{R(a, b)\}$ and the (non-simple) linear TGD

$$\sigma = R(x, x) \rightarrow \exists z\, R(z, x).$$

It is easy to see that there is no trigger for $\{\sigma\}$ on D. This means that $\mathrm{chase}(D, \{\sigma\}) = D$ is finite, whereas $\{\sigma\}$ is *not* D-weakly-acyclic. ∎

To obtain a characterization analogous to Theorem 3.3, the authors of Calautti et al. [2022] used the technique of *simplification* to convert linear TGDs into simple-linear TGDs, while preserving the finiteness of the chase instance. We proceed to recall this technique. Let $\bar{t} = (t_1, \ldots, t_n)$ be a tuple of (not necessarily distinct) terms. We write $\mathrm{unique}(\bar{t})$ for the tuple obtained from $\bar{t}$ by keeping only the first occurrence of each term in $\bar{t}$. For example, if $\bar{t} = (x, y, x, z, y)$, then $\mathrm{unique}(\bar{t}) = (x, y, z)$. For each $i \in [n]$, the *identifier of t_i in $\bar{t}$*, denoted $\mathrm{id}_{\bar{t}}(t_i)$, is the integer that identifies the position of $\mathrm{unique}(\bar{t})$ at which t_i appears. We write $\mathrm{id}(\bar{t})$ for the tuple $(\mathrm{id}_{\bar{t}}(t_1), \ldots, \mathrm{id}_{\bar{t}}(t_n))$. For example, if $\bar{t} = (x, y, x, z, y)$, then $\mathrm{id}(\bar{t}) = (1, 2, 1, 3, 2)$. For an atom $\alpha = R(\bar{t})$, the *shape of α*, denoted

shape(α), is the predicate $R_{\mathsf{id}(\bar{t})}$, whereas the *simplification of* α, denoted simple(α), is the atom $R_{\mathsf{id}(\bar{t})}(\mathsf{unique}(\bar{t}))$. For example, assuming that α is $R(x, y, x, z)$, shape(α) is the predicate $R_{(1,2,1,3)}$, which essentially describes how the terms occurring in α are related, that is, there are three distinct terms, while the first and the third are the same. The simplification of α is the atom $R_{(1,2,1,3)}(x, y, z)$, which fully describes the atom α without repeating variables. Indeed, given $R_{(1,2,1,3)}(x, y, z)$, we can unambiguously construct the atom of which $R_{(1,2,1,3)}(x, y, z)$ is the simplification of, that is, the atom with predicate R having at its first and third position the first variable (i.e., x), at its second position the second variable (i.e., y), and at its fourth position the third variable (i.e., z). We can naturally refer to the simplification and the shape of a set of atoms A, which are clearly sets of simplified atoms and shapes, denoted simple(A) and shape(A), respectively. For a tuple of variables $\bar{x} = (x_1, \ldots, x_n)$, a *specialization of* $\bar{x}$ is a function f from $\bar{x}$ to $\bar{x}$ such that $f(x_1) = x_1$, and $f(x_i) \in \{f(x_1), \ldots, f(x_{i-1}), x_i\}$, for each $i \in \{2, \ldots, n\}$. We write $f(\bar{x})$ for $(f(x_1), \ldots, f(x_n))$. We are now ready to recall how a set of linear TGDs is converted into a set of simple-linear TGDs.

Definition 3.5. Consider a linear TGD σ of the form

$$R(\bar{x}) \rightarrow \exists \bar{z}\, \psi(\bar{y}, \bar{z}),$$

where $\bar{y} \subseteq \bar{x}$, and a specialization f of $\bar{x}$. The *simplification of* σ *induced by* f is the TGD

$$\mathsf{simple}(R(f(\bar{x}))) \rightarrow \exists \bar{z}\, \mathsf{simple}(\psi(f(\bar{y}), \bar{z})).$$

We write $\mathsf{simple}(\sigma)$ for the set of all simplifications of σ induced by some specialization of $\bar{x}$. For a set $\Sigma \in \mathsf{L}$ of TGDs, the *simplification of* Σ is defined as the set

$$\mathsf{simple}(\Sigma) \;=\; \bigcup_{\sigma \in \Sigma} \mathsf{simple}(\sigma)$$

consisting only of simple-linear TGDs. ∎

Here is a simple example that illustrates the notions of specialization and simplification.

Example 3.6. Consider the set Σ consisting of the linear TGDs

$$\sigma_1 \;:\; R(x, y, x, z) \;\rightarrow\; \exists w Q(x, w) \qquad \sigma_2 \;:\; Q(x, y) \;\rightarrow\; R(x, x, y, y).$$

For a database D, during the construction of chase(D, Σ), the TGD σ_1 can be triggered due to an atom α of the form $R(t_1, t_2, t_3, t_4)$ as long as $t_1 = t_3$. Hence, even if $t_1 = t_2 = t_3$ or $t_1 = t_2 = t_3 = t_4$, the atom α also triggers σ_1. The goal of simplifying a linear TGD, for example σ_1, is to convert it into a set of linear TGDs that account for all possible ways of triggering σ_1, but without repeating variables in their bodies, i.e., they are simple-linear. Each specialization f of the body-variables of σ_1 encodes one way of triggering σ_1. For example, the function f such that $f(x) = f(y) = x$ and $f(z) = z$ induces the atom $R(x, x, x, z)$, which represents a "more specific" way of triggering σ_1 w.r.t. the original body-atom $R(x, y, x, z)$, since it not only requires the first and third position of the atom to unify to the same term, but the second position as well. Similarly, if f is such that $f(x) = f(y) = f(z) = x$, then it encodes the atom $R(x, x, x, x)$ that is even more specific. Hence, the simplification $\mathsf{simple}(\sigma_1)$ of σ_1 consists of the following simple-linear TGDs:

$$\begin{aligned}
R_{(1,2,1,3)}(x, y, z) &\rightarrow \exists w Q_{(1,2)}(x, w) \\
R_{(1,1,1,2)}(x, z) &\rightarrow \exists w Q_{(1,2)}(x, w) \\
R_{(1,2,1,1)}(x, y) &\rightarrow \exists w Q_{(1,2)}(x, w) \\
R_{(1,2,1,2)}(x, y) &\rightarrow \exists w Q_{(1,2)}(x, w) \\
R_{(1,1,1,1)}(x) &\rightarrow \exists w Q_{(1,2)}(x, w).
\end{aligned}$$

The above simple-linear TGDs take into account all possible ways σ_1 can be triggered during the construction of the chase; the head atoms are simplified accordingly. Similarly, $\mathsf{simple}(\sigma_2)$, the simplification of σ_2, consists of the following simple-linear TGDs:

$$Q_{(1,2)}(x,y) \rightarrow R_{(1,1,2,2)}(x,y) \qquad Q_{(1,1)}(x) \rightarrow R_{(1,1,1,1)}(x).$$

The chase of D w.r.t. Σ can be faithfully simulated via the chase of $\mathsf{simple}(D)$ w.r.t. $\mathsf{simple}(\Sigma)$. ∎

We can now recall the characterization for the finiteness of the chase instance for linear TGDs, established in Calautti et al. [2022], which is similar to the one for simple-linear TGDs, with the crucial difference that first we need to simplify both the database and the set of linear TGDs:

Theorem 3.7. *Consider a database D and a set $\Sigma \in \mathsf{L}$ of TGDs. The following are equivalent:*

(1) $\mathsf{chase}(D,\Sigma)$ *is finite.*
(2) $\mathsf{simple}(\Sigma)$ *is* $\mathsf{simple}(D)$*-weakly-acyclic.*

It is clear that Theorems 3.3 and 3.7 provide simple algorithms for checking whether the chase instance is finite. In particular, given a database D and a set Σ of simple-linear TGDs, we simply need to check whether Σ is D-weakly-acyclic, in which case the algorithm returns `true`; otherwise, it returns `false`. The same holds when Σ is a set of linear TGDs, with the difference that the algorithm first needs to simplify D and Σ, and then perform the acyclicity check. Our goal is to experimentally evaluate the above algorithms with the aim of understanding which input parameters affect their performance, clarifying whether they can be applied in a practical context, and revealing their performance limitations. Of course, a naive implementation of the above algorithms, especially for linear TGDs where the expensive simplification must be applied, will lead to poor performance, and thus, will not be very useful towards our goal. Indeed, simplification is expensive as, in general, it leads to a very large number of simple-linear TGDs. Consider, for example, the TGD σ of the form $P(x,x,y_1,\ldots,y_n) \rightarrow T(x)$, for $n > 0$. One can verify that the number of TGDs in $\mathsf{simple}(\sigma)$ coincides with the number of ways one can partition a set of n elements. This is known as the nth Bell number, denoted B_n, where $B_0 = 1$ and $B_{i+1} = \sum_{k=0}^{i} \binom{i}{k} B_k$. Thus, $B_n = |\mathsf{simple}(\sigma)| \geq 2^{n-1}$, i.e., is at least exponential in n. Hence, we first need to reconsider and adapt the above theoretical algorithms in order to obtain practical algorithms that are amenable to efficient implementations.

4 Practical Termination Algorithms

We first present the algorithm IsChaseFinite[SL] that accepts as input a database D and a set Σ of simple-linear TGDs, and checks whether Σ is D-weakly-acyclic, which is equivalent to saying that the instance $\mathsf{chase}(D,\Sigma)$ is finite. Note that a naive search for a "bad" cycle in a dependency graph will be too costly since we may have to go through exponentially many cycles. Thus, IsChaseFinite[SL] relies on a refined machinery that searches for *strongly connected components* with a special edge. We then proceed to give an analogous algorithm, dubbed IsChaseFinite[L], for linear TGDs, which essentially simplifies the given database D and set Σ of linear TGDs, and then checks whether $\mathsf{simple}(\Sigma)$ is $\mathsf{simple}(D)$-weakly-acyclic, which is equivalent to saying that $\mathsf{chase}(D,\Sigma)$ is finite. Note, however, that IsChaseFinite[L] relies on a refined notion of simplification that *dynamically simplifies* Σ by leveraging the given database D, instead of doing it statically as in Definition 3.5 without taking any database into account. For instance, consider the set Σ of TGDs from Example 3.6 and the database $D = \{R(a,b,a,c)\}$. We have that $\mathsf{simple}(D) = \{R_{(1,2,1,3)}(a,b,c)\}$. It is clear that most of the TGDs of $\mathsf{simple}(\Sigma)$ are not needed to construct $\mathsf{chase}(\mathsf{simple}(D),\mathsf{simple}(\Sigma))$ as the only TGDs being triggered, starting from $\mathsf{simple}(D)$, are

$$R_{(1,2,1,3)}(x,y,z) \rightarrow \exists w Q_{(1,2)}(x,w) \qquad Q_{(1,2)}(x,y) \rightarrow R_{(1,1,2,2)}(x,y).$$

ALGORITHM 1: IsChaseFinite[SL]

Input: A database D and a set $\Sigma \in \mathsf{SL}$ of TGDs
Output: `true` if $\mathsf{chase}(D, \Sigma)$ is finite and `false` otherwise

1 $G \leftarrow \mathsf{BuildDepGraph}(\Sigma)$;
2 $S \leftarrow \mathsf{FindSpecialSCC}(G)$;
3 $P \leftarrow \bigcup_{C \in S} \{v_C\}$;
4 **if** $\mathsf{Supports}(D, P, G)$ **then return** `false`;
5 **return** `true`

The dynamic simplification aims to keep only TGDs of $\mathsf{simple}(\Sigma)$ that are really needed for checking whether the chase instance is finite. Note that in this section, we present the above algorithms at a high-level without delving into implementation details; the latter will be the subject of Section 5.

4.1 Simple-linear TGDs

Before presenting IsChaseFinite[SL], we need to introduce a couple of auxiliary notions. A ***strongly connected component*** (**SCC**) in a directed graph G is a maximal subgraph of G in which there is a (directed) path between every pair of nodes. A *special SCC* in a dependency graph is an SCC with at least one special edge. We are now ready to discuss IsChaseFinite[SL] depicted in Algorithm 1. It starts by building the dependency graph G of the input set Σ of simple-linear TGDs (line 1). It then collects the special SCCs of G in a set S (line 2), which can clearly form "bad" cycles that violate the condition underlying non-uniform weak-acyclicity. Of course, for the latter to happen, some nodes (i.e., predicate positions) in a special SCC must be supported by the given database D as defined in Section 3. To check this, the algorithm first collects exactly one node v_C from each special SCC C of G in a set P (line 3); note that it is not important how v_C is selected. It then checks if D supports any of the nodes of P (line 4). If this is the case, then there is a D-supported cycle in G with a special edge and the algorithm returns `false`; otherwise, it returns `true`. The implementation details of BuildDepGraph, FindSpecialSCC, and Supports are discussed in Sections 5.1–5.3, respectively. The correctness of IsChaseFinite[SL] follows by Theorem 3.3:

Lemma 4.1. *Consider a database D and a set $\Sigma \in \mathsf{SL}$ of TGDs. The following are equivalent:*

(1) $\mathsf{IsChaseFinite[SL]}(D, \Sigma) = \mathtt{true}$.
(2) $\mathsf{chase}(D, \Sigma)$ *is finite.*

4.2 Linear TGDs

Although IsChaseFinite[SL] together with the simplification technique (see Definition 3.5) immediately gives rise to a simple algorithm for checking the finiteness of the chase instance for linear TGDs, a naive implementation of the simplification technique leads to poor performance. Indeed, we performed exploratory experiments on real-world sets of linear TGDs and observed that a naive implementation is not scalable as the algorithm quickly runs out of memory when dealing with large sets of TGDs. This is because by statically simplifying a set of linear TGDs Σ, without taking into account the underlying database, leads to an exponentially large set of simple-linear TGDs; in particular, the size of the set $\mathsf{simple}(\Sigma)$ is exponential in the maximum arity of the predicates in $\mathsf{sch}(\Sigma)$. Thus, the algorithm IsChaseFinite[SL] becomes impractical due to the very large size of the dependency graph of $\mathsf{simple}(\Sigma)$, which exceeds the capacity of the main memory.

Dynamic Simplification. We refine the notion of simplification by taking into account the underlying database, which leads to the technique of dynamic simplification. In particular, given a

database D and a set Σ of linear TGDs, the goal is to define a set $\mathsf{simple}_D(\Sigma)$, which is a subset of $\mathsf{simple}(\Sigma)$, that enjoys two crucial properties:

(1) It holds that $\mathsf{chase}(\mathsf{simple}(D), \mathsf{simple}(\Sigma))$ is finite iff $\mathsf{chase}(\mathsf{simple}(D), \mathsf{simple}_D(\Sigma))$ is finite, i.e., the technique of dynamic simplification preserves the finiteness of the chase.
(2) The set $\mathsf{simple}_D(\Sigma)$ is, in general, orders of magnitude smaller than the set $\mathsf{simple}(\Sigma)$ obtained by statically simplifying Σ.

Item (1) is established by Lemma 4.3 below. Item (2) cannot be formally proved as there are cases where both static and dynamic simplification build the same set of linear TGDs. However, we have experimentally verified that for existing databases and sets of TGDs from the literature (in fact, those used in Section 9), the size of the dynamically simplified sets of TGDs is, on average, 5 times smaller than the size of the corresponding statically simplified sets of TGDs. The absolute difference varies with the dynamically simplified sets being up to 1,000 times smaller in the best case.

The key idea of dynamic simplification is to exploit the shapes of the atoms occurring in the given database to guide the simplification. More precisely, given a database D and a set Σ of linear TGDs, we first collect the shapes that can be derived from $\mathsf{shape}(D)$ using the TGDs of Σ; we denote this set as $\Sigma(\mathsf{shape}(D))$. Then, $\mathsf{simple}_D(\Sigma)$ keeps from the set $\mathsf{simple}(\Sigma)$ only those simple-linear TGDs such that the predicate of their body-atom belongs to $\Sigma(\mathsf{shape}(D))$, as these are the only TGDs that can be applied during the construction of the instance $\mathsf{chase}(\mathsf{simple}(D), \mathsf{simple}(\Sigma))$. All the other TGDs of $\mathsf{simple}(\Sigma)$ are superfluous whenever the input database is D in the sense that they will never be applied during the construction of $\mathsf{chase}(\mathsf{simple}(D), \mathsf{simple}(\Sigma))$. We proceed to formalize this idea. To this end, we need to introduce some auxiliary notions.

For a schema $\mathbf{S}$, let $\mathsf{shape}(\mathbf{S})$ be the set of all shapes mentioning a predicate of $\mathbf{S}$, that is, the set

$$\mathsf{shape}(\mathbf{S}) \;=\; \left\{R_{\mathsf{id}(\bar{t})} \mid R \in \mathbf{S} \text{ and } \bar{t} \in \{1, \ldots, \mathsf{ar}(R)\}^{\mathsf{ar}(R)}\right\}.$$

For a set of shapes $S \subseteq \mathsf{shape}(\mathbf{S})$, the *database induced by* S, denoted $DB[S]$, is the database $\{R(\mathsf{id}(\bar{t})) \mid R_{\mathsf{id}(\bar{t})} \in S\}$. For example, assuming that $S = \{R_{(1,2)}, P_{(1,1,2)}\}$, then the database $DB[S]$ is $\{R(1,2), P(1,\ 1,\ 2)\}$. Consider now a linear TGD $\sigma = R(x_1, \ldots, x_n) \rightarrow \exists \bar{z}\, \psi(\bar{y}, \bar{z})$ and let h be a homomorphism from $\{R(x_1, \ldots, x_n)\}$ to $\{R(i_1, \ldots, i_n)\} \subseteq DB[\mathsf{shape}(\{R\})]$. The *$h$-specialization* of the tuple $(x_1, \ldots, x_n)$ is the (unique) specialization f of $(x_1, \ldots, x_n)$ such that $f(x_i) = f(x_j)$ iff $h(x_i) = h(x_j)$, for every $i, j \in [n]$. For example, assuming that h is a homomorphism from $\{R(x, y, x, z)\}$ to $\{R(1, 1, 1, 2)\}$, the h-specialization of (x, y, x, z) is the function f such that $f(x) = x$, $f(y) = x$, and $f(z) = z$. We can now proceed with the formalization of dynamic simplification.

Consider a set Σ of linear TGDs and a set of shapes $S \subseteq \mathsf{shape}(\Sigma)$; for brevity, we write $\mathsf{shape}(\Sigma)$ for $\mathsf{shape}(\mathsf{sch}(\Sigma))$. A shape $R_{\mathsf{id}(\bar{t})} \in \mathsf{shape}(\Sigma)$ is an *immediate consequence* of S and Σ if:

(1) $R_{\mathsf{id}(\bar{t})} \in S$, or
(2) there is a TGD $R(\bar{x}) \rightarrow \exists \bar{z}\, \psi(\bar{y}, \bar{z})$ in Σ and a homomorphism h from $\{R(\bar{x})\}$ to $DB[S]$ such that $R_{\mathsf{id}(\bar{t})}$ occurs in the head of the simplification of σ induced by the h-specialization of $\bar{x}$.

In simple words, item (2) tells us that there exists a TGD in $\mathsf{simple}(\Sigma)$ of the form

$$R'_{\mathsf{id}(\bar{t}')}(\bar{x}) \;\rightarrow\; \exists \bar{z}\, \ldots, R_{\mathsf{id}(\bar{t})}(\bar{y}), \ldots$$

with $R'_{\mathsf{id}(\bar{t}')} \in S$. The *immediate consequence operator* of Σ is the function $\Gamma_\Sigma : 2^{\mathsf{shape}(\Sigma)} \rightarrow 2^{\mathsf{shape}(\Sigma)}$ (as usual, 2^X denotes the powerset of a set X) such that

$$\Gamma_\Sigma(S) \;=\; \left\{R_{\mathsf{id}(\bar{t})} \mid R_{\mathsf{id}(\bar{t})} \text{ is an immediate consequence of } S \text{ and } \Sigma\right\}.$$

By iterative applications of the above operator, we can compute the shapes that can be derived from S using the TGDs of Σ. Formally,

$$\Gamma^0_\Sigma(S) \;=\; S \qquad \text{and} \qquad \Gamma^i_\Sigma(S) \;=\; \Gamma_\Sigma(\Gamma^{i-1}_\Sigma(S)) \;\text{ for }\; i > 0$$

and we finally let

$$\Sigma(S) \;=\; \bigcup_{i \geq 0} \Gamma^i_\Sigma(S).$$

At first glance, the construction of $\Sigma(S)$ requires infinitely many iterations. However, since $\Sigma(S) \subseteq \mathsf{shape}(\Sigma)$, in the worst-case $\Sigma(S)$ is obtained after $|\mathsf{shape}(\Sigma)|$ iterations. It is actually easy to verify that $\Sigma(S) = \Gamma^{|\mathsf{shape}(\Sigma)|}_\Sigma(S)$. Therefore, since $\mathsf{shape}(\Sigma)$ is finite, we conclude that $\Sigma(S)$ can be obtained after finitely many steps. We can now formally define dynamic simplification.

Definition 4.2. Consider a database D and a set Σ of linear TGDs.[1] The *dynamic simplification of* Σ *relative to* D (or *D-simplification of* Σ), denoted $\mathsf{simple}_D(\Sigma)$, is defined as the set

$$\Big\{\mathsf{simple}(R(f(\bar{x}))) \rightarrow \exists \bar{z}\, \mathsf{simple}(\psi(f(\bar{y}), \bar{z})) \mid R(\bar{x}) \rightarrow \exists \bar{z}\, \psi(\bar{y}, \bar{z}) \in \Sigma \text{ and } f \text{ is the } h\text{-specialization of } \bar{x} \text{ for some homomorphism } h \text{ from } \{R(\bar{x})\} \text{ to } DB(\Sigma(\mathsf{shape}(D)))\Big\}$$

consisting only of simple-linear TGDs. ∎

It is not difficult to verify that the D-simplification of Σ essentially collects all the TGDs of $\mathsf{simple}(\Sigma)$ such that the predicate of their body-atom belongs to $\Sigma(\mathsf{shape}(D))$. Consider, for instance, the set Σ of TGDs from Example 3.6 and the database $D = \{R(a, b, a, c)\}$. It is easy to see that

$$\Sigma(\mathsf{shape}(D)) \;=\; \{R_{(1,2,1,3)}, R_{(1,1,2,2)}, Q_{(1,2)}\}.$$

Therefore, from the set $\mathsf{simple}(\Sigma)$, $\mathsf{simple}_D(\Sigma)$ keeps only the TGDs

$$R_{(1,2,1,3)}(x, y) \rightarrow \exists w Q_{(1,2)}(x, w) \qquad Q_{(1,2)}(x, y) \rightarrow R_{(1,1,2,2)}(x, y).$$

We proceed to show that indeed dynamic simplification preserves the finiteness of the chase.

Lemma 4.3. *Consider a database D and a set $\Sigma \in \mathsf{L}$ of TGDs. The following are equivalent:*

(1) $\mathsf{chase}(\mathsf{simple}(D), \mathsf{simple}(\Sigma))$ *is finite.*
(2) $\mathsf{chase}(\mathsf{simple}(D), \mathsf{simple}_D(\Sigma))$ *is finite.*

Proof. Since, by definition, $\mathsf{simple}_D(\Sigma) \subseteq \mathsf{simple}(\Sigma)$, it is clear that (1) implies (2). The interesting direction is (2) implies (1). We proceed to establish the contrapositive of the implication in question, that is, if the instance $\mathsf{chase}(\mathsf{simple}(D), \mathsf{simple}(\Sigma))$ is infinite, then the instance $\mathsf{chase}(\mathsf{simple}(D), \mathsf{simple}_D(\Sigma))$ is also infinite. To this end, we need an auxiliary technical lemma:

Lemma 4.4. *Consider a path $(R_1, i_1), \ldots, (R_n, i_n)$ in the dependency graph of* $\mathsf{simple}(\Sigma)$ *such that there is an atom of the form $R_1(\bar{c})$ in* $\mathsf{simple}(D)$*. It holds that $\{R_1, \ldots, R_n\} \subseteq \Sigma(\mathsf{shape}(D))$.*

Proof. We proceed by induction on the length n of the path.

Base Case. Clearly, $\mathsf{shape}(D) \subseteq \Sigma(\mathsf{shape}(D))$. Since $\mathsf{shape}(D)$ coincides with the set of predicates used by the atoms of $\mathsf{simple}(D)$ and, by hypothesis, R_1 is used by an atom of $\mathsf{simple}(D)$, we get that $R_1 \in \Sigma(\mathsf{shape}(D))$, as needed.

[1] We assume, without loss of generality, that the atoms of D mention only predicates of $\mathsf{sch}(\Sigma)$, and thus, $\mathsf{shape}(D) \subseteq \mathsf{shape}(\Sigma)$. Indeed, the atoms of D with a predicate not in $\mathsf{sch}(\Sigma)$ do not affect in any way the size of the instance $\mathsf{chase}(D, \Sigma)$.

Inductive Step. Consider a path $(R_1, i_1), \ldots, (R_n, i_n)$ in the dependency graph of simple(Σ) with R_1 being a predicate used by an atom of simple(D). By induction hypothesis, $\{R_1, \ldots, R_{n-1}\} \subseteq \Sigma(\text{shape}(D))$. It remains to show that $R_n \in \Sigma(\text{shape}(D))$. Assume that σ is the TGD witnessing the edge $((R_{n-1}, i_{n-1}), (R_n, i_n))$ in the dependency graph of simple(Σ). There is $i \geq 0$ such that $R_{n-1} \in \Gamma^i_\Sigma(\text{shape}(D))$. Since R_{n-1} is the predicate of the body-atom of σ, $R_n \in \Gamma^{i+1}_\Sigma(\text{shape}(D))$, and therefore, $R_n \in \Sigma(\text{shape}(D))$, as needed. □

We can now show the desired implication. Since, by hypothesis, chase(simple(D), simple(Σ)) is infinite, Theorem 3.3 allows us to conclude that there exists a simple(D)-supported cycle with a special edge in the dependency graph of simple(Σ). Let $(R_1, i_1), \ldots, (R_n, i_n)$, with $(R_1, i_1) = (R_n, i_n)$, be such a cycle. Since this cycle is simple(D)-supported, there exists an atom $P(\bar{c}) \in D$ and a node (R_j, i_j) in the cycle such that R_j is reachable from P. Since the TGDs of Σ have a non-empty frontier, in the dependency graph of simple(Σ) there exists a path $(P_1, k_1), \ldots, (P_m, k_m)$ with $P_1 = P$ and $(P_m, k_m) = (R_j, i_j)$. Summing up, in the dependency graph of simple(Σ), there exists a path

$$(P_1, k_1), \ldots, (P_{m-1}, k_{m-1}), (R_j, i_j), \ldots, (R_{n-1}, i_{n-1}), (R_1, i_1), \ldots, (R_{j-1}, i_{j-1}), (R_j, i_j).$$

Assume that this path is witnessed by the TGDs $\sigma_1, \ldots, \sigma_{n+m-2}$. By Lemma 4.4, $\{P_1, \ldots, P_{m-1}, R_1, \ldots, R_{n-1}\} \subseteq \Sigma(\text{shape}(D))$. By the definition of dynamic simplification (see Definition 4.2), we get that the TGDs $\sigma_1, \ldots, \sigma_{n+m-2}$ belong to $\text{simple}_D(\Sigma)$. Therefore, the above simple(D)-supported cycle with a special edge also occurs in the dependency graph of $\text{simple}_D(\Sigma)$. Hence, by Theorem 3.3, chase(simple(D), $\text{simple}_D(\Sigma)$) is infinite, and the claim follows. □

Another crucial property of dynamic simplification, which will help us to further improve the performance of the termination algorithm for linear TGDs, is that the simple(D)-weak-acyclicity of $\text{simple}_D(\Sigma)$ coincides with the weak-acyclicity of $\text{simple}_D(\Sigma)$. This holds since, by construction, all the predicates occurring in the TGDs of $\text{simple}_D(\Sigma)$ are reachable from a predicate occurring in simple(D), and thus, each cycle with a special edge in the dependency graph of $\text{simple}_D(\Sigma)$ is trivially simple(D)-supported. The above property immediately implies Lemma 4.5 below since checking for the finiteness of chase(simple(D), $\text{simple}_D(\Sigma)$), which is equivalent to the simple(D)-weak-acyclicity of $\text{simple}_D(\Sigma)$ by Theorem 3.7, boils down to checking if $\text{simple}_D(\Sigma)$ is weakly-acyclic, without having to explicitly check for simple(D)-supportedness, and thus, avoiding the expensive task of simplifying D.

Lemma 4.5. *Consider a database D and a set $\Sigma \in \mathsf{L}$ of TGDs. The following are equivalent:*

(1) chase(simple(D), $\text{simple}_D(\Sigma)$) *is finite.*

(2) $\text{simple}_D(\Sigma)$ *is weakly-acyclic.*

An Algorithm for Dynamic Simplification. Dynamic simplification indeed allows us to filter out from the set obtained by statically simplifying a set of linear TGDs superfluous simple-linear TGDs by exploiting the given database. It remains, however, to provide a concrete algorithm that performs the dynamic simplification of a set of linear TGDs that is amenable to an efficient implementation. To this end, we proceed to present the algorithm DynSimplification, depicted in Algorithm 2, that takes as input a database D and a set Σ of linear TGDs, and constructs the D-simplification of Σ. The algorithm starts by finding the shapes of the atoms occurring in D, namely, it computes the set shape(D) (line 1). It then initializes the set of simplified TGDs Σ_s (line 2) and the set of new shapes ΔS (line 3). Then, the algorithm iteratively generates simplified TGDs and collects the new shapes that are added to ΔS, and continues this until a fixpoint is reached, i.e., $\Delta S = \emptyset$ (line 4). In particular, at each iteration, the algorithm computes simplified TGDs that are not superfluous,

ALGORITHM 2: DynSimplification

Input: A database D and a set $\Sigma \in \mathsf{L}$ of TGDs
Output: The D-simplification of Σ

1 $S \leftarrow \mathsf{FindShapes}(D)$;
2 $\Sigma_s \leftarrow \emptyset$;
3 $\Delta S \leftarrow S$;
4 **while** $\Delta S \neq \emptyset$ **do**
5 $\quad \Sigma_{aux} \leftarrow \mathsf{Applicable}(\Delta S, \Sigma)$;
6 $\quad S_{aux} \leftarrow \{R_{\mathrm{id}(\bar{t})} \in \mathsf{shape}(\Sigma) \mid$ there is $\sigma \in \Sigma_{aux}$ such that $R_{\mathrm{id}(\bar{t})}$ occurs in $\mathsf{head}(\sigma)\}$;
7 $\quad \Sigma_s \leftarrow \Sigma_s \cup \Sigma_{aux}$;
8 $\quad \Delta S \leftarrow S_{aux} \setminus S$;
9 $\quad S \leftarrow S \cup \Delta S$;
10 **return** Σ_s;

i.e., they can be applied during the construction of $\mathsf{chase}(\mathsf{simple}(D), \mathsf{simple}(\Sigma))$, that are added to Σ_s (lines 5 and 7). This is done via the procedure Applicable, which takes as input a set of shapes $\hat{S}$, together with the set Σ of linear TGDs, and returns the set

$$\{\mathsf{simple}(R(f(\bar{x}))) \rightarrow \exists \bar{z}\, \mathsf{simple}(\psi(f(\bar{y}), \bar{z})) \mid R(\bar{x}) \rightarrow \exists \bar{z}\, \psi(\bar{y}, \bar{z}) \in \Sigma \text{ and } f \text{ is the } h\text{-specialization of } \bar{x} \text{ for some homomorphism } h \text{ from } \{R(\bar{x})\} \text{ to } DB[\hat{S}]\}.$$

In essence, the procedure Applicable computes the set of TGDs of $\mathsf{simple}(\Sigma)$ such that the predicate of their body belongs to $\hat{S}$. The algorithm also collects the newly generated shapes, that is, the predicates occurring in the head of the TGDs of $\mathsf{Applicable}(\Delta S, \Sigma)$, that are added to ΔS (lines 6 and 8). Note that at each iteration, the algorithm applies the TGDs on ΔS, not on S, with the exception of the first iteration where $S = \Delta S$. This works because there are no new applicable TGDs on S after the first iteration since the TGDs are linear (one body-atom) and all the applicable TGDs on S are applied during the first iteration. The implementation details of FindShapes and Applicable are discussed in Sections 5.1 and 5.2, respectively. A simple example that illustrates how the algorithm DynSimplification works follows:

Example 4.6. Consider again the set Σ of TGDs from Example 3.6 and the database $D = \{R(a, b, a, c)\}$. We proceed to describe how the dynamic simplification procedure operates on D and Σ. The sets S and ΔS are initialized to the set of shapes of D, that is, $S = \Delta S = \{R_{(1,2,1,3)}\}$, and $\Sigma_s = \emptyset$. At the first iteration of the algorithm, we have that σ_1 is "applicable" to ΔS, i.e., the body-atom of σ_1 is "coherent" with the (only) shape present in ΔS, i.e., $R_{(1,2,1,3)}$. No other TGDs of Σ are applicable, and thus, we conclude that the TGD $R_{(1,2,1,3)}(x, y, z) \rightarrow \exists w Q_{(1,2)}(x, w)$, which is constructed in line 5 and added to Σ_s in line 7, will be actually be triggered the chase. We then have $S = \{R_{(1,2,1,3)}, Q_{(1,2)}\}$, while the new shapes are $\Delta S = \{Q_{(1,2)}\}$. At the second iteration, we have that σ_2 is applicable to ΔS. Thus, the TGD $Q_{(1,2)}(x, y) \rightarrow R_{(1,1,2,2)}(x, y)$ is added to Σ_s and we have $S = \{R_{(1,2,1,3)}, Q_{(1,2)}, R_{(1,1,2,2)}\}$ and $\Delta S = \{R_{(1,1,2,2)}\}$. Finally, at the third iteration, no TGDs of Σ are applicable since there is no way to specialize the body-atom of σ_1 in a way that it becomes coherent with the shape $R_{(1,1,2,2)}$ in ΔS. The algorithm then terminates and Σ_s collects a subset of $\mathsf{simple}(\Sigma)$ that is sufficient to construct $\mathsf{chase}(\mathsf{simple}(D), \mathsf{simple}(\Sigma))$. ∎

It is easy to see that DynSimplification is correct by construction:

Lemma 4.7. *For a database D and a set $\Sigma \in \mathsf{L}$ of TGDs,* $\mathsf{DynSimplification}(D, \Sigma) = \mathsf{simple}_D(\Sigma)$.

ALGORITHM 3: IsChaseFinite[L]

Input: A database D and a set $\Sigma \in \mathsf{L}$ of TGDs
Output: `true` if chase(D, Σ) is finite and `false` otherwise

```
1 Σ_s ← DynSimplification(D, Σ);
2 G ← BuildDepGraph(Σ_s);
3 if FindSpecialSCC(G) ≠ ∅ then return false;
4 return true
```

Termination Algorithm. Having in place DynSimplification, it is now straightforward to devise the algorithm IsChaseFinite[L], depicted in Algorithm 3, that checks for the finiteness of the chase in the case of linear TGDs. The correctness of DynSimplification, Theorem 3.7, Lemma 4.3, and Lemma 4.5, imply the correctness of the algorithm IsChaseFinite[L], and the next lemma follows:

Lemma 4.8. *Consider a database D and a set $\Sigma \in \mathsf{L}$ of TGDs. The following are equivalent:*

(1) IsChaseFinite[L]$(D, \Sigma) =$ `true`.
(2) chase(D, Σ) *is finite.*

5 Implementation Details

We proceed to discuss the implementation details of the algorithms for checking the finiteness of the chase instance presented in Section 4. In particular, we discuss the implementation choices that help to improve the performance of the algorithms, but are missing from the descriptions given in Section 4. In Sections 5.1–5.3, we discuss the procedures BuildDepGraph, FindSpecialSCC, and Supports, respectively, used in Algorithm 1. The details of DynSimplification used in Algorithm 3 are discussed in Section 5.4.

5.1 Build Dependency Graphs

The procedure BuildDepGraph takes as input a set Σ of TGDs and returns the dependency graph of Σ in the form of an adjacency list. Recall that an adjacency list for a directed graph is a list of lists. Each list corresponds to a node v of the graph, and the members in such a list represent the outgoing edges of v. Towards an implementation of such an adjacency list, we store a dependency graph as a list of *node objects*. Each node object represents a node in the graph, and has a list of *edge objects* representing the edges of the graph. For each edge object, we additionally store a binary value that specifies whether the corresponding edge is special or not. Although a singly linked list suffices for storing a dependency graph, we implement a dependency graph's adjacency list as a doubly linked list for performance purposes. By implementing a doubly linked list, each node object, in addition to a list of edge objects, has a list of reverse edge objects representing the edges in the opposite direction. These reverse edge objects enable traversing the graph in the opposite direction of the edges, which significantly helps when checking the support of positions in special SCCs, as we explain in Section 5.3. Now, given a set Σ of simple-linear TGDs, BuildDepGraph iterates over all TGDs and constructs the dependency graph of Σ by creating new elements for newly visited positions and linking them as dictated by the TGDs. To speed up the process of building dependency graphs, the procedure also uses an index structure that maps predicate positions to their corresponding elements in the adjacency list. This index allows for fast access to the elements (nodes) for adding new links (edges) while parsing new TGDs. Using the index structure, the procedure creates dependency graphs in linear time w.r.t. the size of the input set of TGDs.

5.2 Find Special SCCs

To implement FindSpecialSCC we adapt *Tarjan's algorithm* for finding SCCs in directed graphs [Tarjan 1972], which we briefly recall below. Other algorithms for finding SCCs can be found in the literature (see, e.g., [Dijkstra 1982; Sharir 1981]), some of which are simpler than Tarjan's algorithm, such as Kosaraju's algorithm [Sharir 1981]. We build on Tarjan's algorithm due to its efficiency.

Tarjan's Algorithm. Given a directed graph, Tarjan's algorithm constructs a *spanning forest*, that is, a set of trees that contains all the nodes of the graph, while each tree corresponds to an SCC. To find the trees, the algorithm runs a depth-first search starting from an arbitrary node in the graph and creates the trees by traversing the nodes in each tree and then moving to the next tree. During the search for nodes in a tree, the algorithm traverses two categories of edges: (i) edges that lead to new nodes, and (ii) edges that lead to already visited nodes. The edges of the first category are called *tree edges* as they add new nodes to the tree. The edges of the second category are classified into three subcategories: (ii.a) the edges that move from an ancestor in the tree to one of its descendants, which are ignored by the algorithm, (ii.b) the edges that move from a descendant to its ancestor, called *fronds*, and (ii.c) the edges that move from one subtree to another subtree in the same tree, called *cross-links*. Tarjan's algorithm assigns numbers to the nodes in the order they are visited during the search (i.e., a node with a smaller number is visited first) and pushes them in a stack. This stack, which we call *SCC stack*, differs from the stack used to implement the depth-first search. A visited node is removed from the search stack during backtracking when all its child nodes are visited in the depth-first search. However, such a node remains in the SCC stack as long as there is a path from it to a node below it in the stack. This allows the SCC stack to keep track of the nodes in the current tree. The algorithm uses the assigned numbers to understand whether a node is a root node after visiting all its children. This is done by checking if the traversed edges, while visiting the descendant nodes, are fronds and cross-links. After finding a root, the algorithm creates an SCC by popping nodes from the top of the SCC stack and adding them to the SCC until it reaches the root of the current tree. The algorithm continues until all the nodes in the graph are visited and returns the obtained trees (i.e., the SCCs of the graph).

The Procedure FindSpecialSCC. It is a simple extension of Tarjan's algorithm for finding the special SCCs in a dependency graph. Such an extension is needed as we need a mechanism that allows us to check whether an SCC obtained by Tarjan's algorithm is special. This is done by pushing a dummy token in the stack (the stack that stores the visited nodes in the current component) whenever the algorithm traverses a special edge. Note that the algorithm pushes this dummy token even if the special edge is ignored, as we explained in Tarjan's algorithm. After finding the root of the current SCC and popping the nodes to create the SCC, the algorithm labels the SCC as special if there is a dummy token between the popped nodes. Only the special SCCs are stored and returned.

5.3 Check for Positions Support

The procedure Supports consists of two steps: (1) query the database to find the positions of the extensional predicates, and (2) traverse the dependency graph starting from the positions in the special SCCs in the reverse order to reach the positions computed in the first step. Step (1) has been implemented via a single SQL query that returns the list of non-empty relations, which we then use to create the set of positions of the extensional predicates, denoted P_D. The SQL query has been implemented using the catalog of the DBMS that stores the database, which allows the query to find the set of relations names faster and without accessing the actual data. For step (2), we start from the given set of positions P, i.e., the set of positions in the special SCCs, and traverse

the graph in the reverse order using the reverse links in the adjacency list of the dependency graph, as explained in Section 5.1. The procedure returns `true` if the graph traversal in the second step reaches a node (i.e., a position) in P_D from an extensional predicate; otherwise, it returns `false`.

5.4 Dynamic Simplification

The procedure DynSmplification takes as input a database D and a set Σ of linear TGDs, and returns the set $\text{simple}_D(\Sigma)$ of simple-linear TGDs. It is an iterative procedure that uses two sub-procedures: FindShapes that computes the set of shapes S of the atoms of D, and Applicable that computes the simplified TGDs using the shapes of S. We proceed to discuss their implementation details.

The Procedure FindShapes**.** We have both an *in-memory* and an *in-database* implementation, which we discuss below. In Sections 8 and 9, we conduct experiments comparing the performance of the two implementations of FindShapes, and we discuss when one outperforms the other.

In-memory Implementation

For the in-memory implementation, we run an SQL query for each relation R of D to load all the tuples of R into the main memory. We then construct the set of shapes of the atoms in each relation R by iterating over its tuples $\bar{c}$, and generating the shape of each atom $R(\bar{c})$. The relations that cannot be loaded into main memory are partitioned into smaller subrelations of a fixed size, which has been determined through preliminary experiments.

In-database Implementation

The in-database implementation does not load the relations into the main memory, but instead runs SQL queries to find the shapes of the atoms in each relation. In particular, we translate each possible shape to a Boolean query that evaluates to `true` if the shape exists in the database. The query for a shape of a predicate R is of the following general form:

```
SELECT CASE WHEN EXISTS (SELECT * FROM R WHERE Equality_Conditions
                         AND Diseqality_Conditions) THEN 1 ELSE 0 END
```

where the equality and disequality conditions are consistency checks according to the given shape. For example, the shape $R_{[1,1,2]}$ translates into the following SQL query Q:

```
SELECT CASE WHEN EXISTS
        (SELECT * FROM R WHERE a1=a2 AND a2!=a3) THEN 1 ELSE 0 END
```

where we assume that the predicate R comes with the attributes `a1`, `a2`, and `a3`. Of course, running an SQL query per shape results in many queries for predicates with high arity. However, depending on the database D, many of these queries may be unnecessary since do not find any shapes. To avoid running some of those queries, we use the *Apriori* algorithm's idea to find association rules [Agrawal and Srikant 1994]. We start by running queries for checking the existence of more general shapes (e.g., $R_{(1,1,2)}$) before we check more specific shapes (e.g., $R_{(1,1,1)}$). For each shape, we run a pair of queries. The first query is a relaxed version of the general query explained above without the disequality conditions; e.g., the first query Q' for $R_{(1,1,2)}$ is Q above without the underlined condition. We only continue to run Q if Q' evaluates to `true`. Additionally, we do not run the query for more specific shapes, e.g., $R_{(1,1,1)}$, if Q' evaluates to `false`. This allows us to avoid the execution of many queries for specific shapes by running a single query for more general shapes.

The Procedure Applicable**.** Recall that Applicable takes as input a set S of shapes and a set Σ of linear TGDs, and returns the set of TGDs of simple(Σ) such that the predicate of their body

belongs to S. As explained in Section 4, this is done by iterating over all TGDs $\sigma \in \Sigma$ of the form $R(\bar{x}) \rightarrow \exists \bar{z}\, \psi(\bar{y}, \bar{z})$ and homomorphisms h from $\{R(\bar{x})\}$ to $DB[S]$, and collecting the simplification of σ induced by the h-specialization of $\bar{x}$. Applying the above iterative process with a large set of shapes and a large set of TGDs can be very costly. Thus, to improve the performance of this process, the implementation uses an index structure that enables fast access to the TGDs. The index structure maps each predicate $R \in \text{sch}(\Sigma)$ to the set of TGDs of Σ that their body-atom uses R. This allows the procedure to iterate over the current shapes in each iteration and quickly access the relevant TGDs in the index. However, checking whether a relevant TGD is applicable remains costly. This task requires checking the shapes of the body-atoms in all the simplified TGDs obtained from the TGD with the current shape. To facilitate this, we generate and store the shape of the body-atom of each TGD. Additionally, we store an array of strings representing all possible identifiers of tuples up to the maximum arity of the schema, that allows us to quickly find the shapes of the body-atoms in simplified TGDs.

6 Experimental Infrastructure

Our goal is to experimentally evaluate the behaviour of the termination algorithms presented in Section 5 with the aim of clarifying whether they can be applied in a practical context, and if not, reveal their limitations. To this end, we are going to conduct extensive experiments with synthetic data and sets of TGDs. Note that, we are also going to conduct experiments with real-world sets of TGDs in Section 9. Therefore, we need a way to generate databases and sets of TGDs that are suitable for such an experimental evaluation. We proceed to discuss our tools for generating data (Section 6.1) and sets of TGDs (Section 6.2). Let us clarify that in the rest of the article, whenever we say that we randomly select an element from a certain space of elements, we actually mean that we select such an element uniformly at random.

6.1 Data Generator

There are freely available data generators such as TPC-H[2] and DataFiller.[3] However, none of the existing tools is suitable for our purposes. To effectively evaluate the dynamic simplification procedure presented above, which is a key component of the termination algorithm for linear TGDs, we need to make sure that the generated database contains a variety of shapes. This is precisely the limitation of the existing data generators, as they do not allow us to control the shape of the generated atoms. Hence, we had to implement our own data generator.

Our data generator has tuning parameters that allow us to determine key properties of the generated database D: the number of predicates in D, the minimum and maximum arity of those predicate, the size of the database domain (i.e., the number of values in $\text{dom}(D)$), and the number of tuples in each relation of D. In particular, the generator takes as input a tuple of integer values for the tuning parameters ($preds$, min, max, $dsize$, $rsize$), and constructs a database D such that, with $\mathbf{S} = \{R \mid R(\bar{c}) \in D\}$, $|\mathbf{S}| = preds$, the predicates of $\mathbf{S}$ have arity between min and max, $|\text{dom}(D)| = dsize$, and, for each $R \in \mathbf{S}$, $|\{\bar{c} \mid R(\bar{c}) \in D\}| = rsize$. To this end, it first generates a set $\mathbf{S}$ consisting of $preds$ different predicates, and it randomly selects an arity for each such predicate from the range $[min, max]$. It then generates a database by adding $rsize$ tuples to each predicate of $\mathbf{S}$ that are formed using $dsize$ different constant values. Now, to ensure that the obtained database contains different shapes, each tuple is generated by randomly selecting a shape and filling the positions by randomly picking values from the database domain of size

[2] https://www.tpc.org/tpch/
[3] https://github.com/memsql/datafiller

dsize without repetition, that is, a shape determines how many times the same value is repeated in a tuple.

6.2 TGD Generator

As for the data generator, existing TGD generators (see, for example, [Arocena et al. 2015; Benedikt et al. 2017]) are not suitable for our purposes since they do not allow us to control the shape of the atoms occurring in the bodies of the generated TGDs, which is crucial for generating sets of TGDs that are suitable for our experiments. Thus, we had to implement our own TGD generator that supports this key feature.

Our TGD generator has tuning parameters that allows us to determine key properties of the generated set Σ of TGDs: the size of the schema (that is, the size of $\mathsf{sch}(\Sigma)$), the minimum and maximum arity of the predicates of $\mathsf{sch}(\Sigma)$, the number of TGDs in Σ, and the underlying class of Σ (that is, whether Σ is a set of simple-linear or linear TGDs). In particular, the TGD generator takes as input a set $\mathbf{S}$ of predicates and a tuple of values for the tuning parameters ($ssize$, min, max, $tsize$, $tclass$), and constructs a set Σ of TGDs such that $\mathsf{sch}(\Sigma) \subseteq \mathbf{S}$, $|\mathsf{sch}(\Sigma)| = ssize$, the predicates of $\mathsf{sch}(\Sigma)$ have arity between min and max, $|\Sigma| = tsize$, and Σ falls in the class $tclass$. To this end, it first chooses a subset $\mathbf{S}'$ of $\mathbf{S}$ such that $|\mathbf{S}'| = ssize$ and its predicates have arity between min and max, and then generates the desired set of simple-linear or linear TGDs using all the predicates of $\mathbf{S}'$; the actual generation is described below. Let us stress that in our experiments, we consider only single-head TGDs, that is, TGDs with only one head-atom, despite the fact that our termination algorithms work with multi-head TGDs, that is, TGDs that have several atoms in their heads. This is because the number of atoms occurring in the head of a TGD is typically negligible compared to the number of TGDs, and having several atoms in the heads of TGDs does not affect the number of shapes occurring in the database. As we shall see in Sections 7 and 8, the number of TGDs and the number of database shapes are the main parameters impacting the runtime of the algorithms, and thus, our experimental evaluation provides conclusive results even if we consider single-head TGDs. Hence, for clarity, our TGD generator is designed to generate single-head TGDs.

Simple-linear TGDs. To generate a simple-linear TGD, the generator randomly selects two predicates from $\mathbf{S}'$ that will be used for forming the body- and the head-atom, respectively. The random selection is with repetition to allow the same predicate to appear in both the body and the head of the TGD. Since we target a simple-linear TGD, we use different variables to fill the positions in the body-atom. Now, for the head-atom, we fill each position with either an existentially quantified variable, or a universally quantified variable that has been already used in the body-atom. In particular, for each position π of the head-atom, the generator classifies π as an existential position with probability 10%. In this case, π is filled with a fresh variable that does not appear in the body-atom; otherwise, it is filled with a randomly selected variable from the body-atom.

Linear TGDs. Generating linear TGDs is done in the same way as for simple-linear TGDs, with the crucial difference that, after randomly selecting the predicates of the body- and the head-atom, the generator randomly chooses a shape for the body-atom and then applies similar steps as for simple-linear TGDs to fill the positions of the body- and the head-atom. Note that the selection of the body-variables is guided by the chosen shape, which in turn allows for the repetition of variables in the body-atom of the generated linear TGD.

We are now ready to proceed with our experimental evaluation. Note that for the experiments, we used a server with an Intel Core i5 3.00 GHz CPU and 16 GB RAM, all the databases in our

experiments are stored in a PostgreSQL 11.5 instance, and the termination algorithms in question have been implemented in Java SE 11.

7 Evaluation for Simple-linear TGDS

We start with the experimental evaluation of IsChaseFinite[SL], which is depicted in Algorithm 1. Towards a refined analysis, we are going to break down its end-to-end runtime, which we denote by `t-total`, into the following three time parameters:

- `t-parse`: time to parse the TGDs from an input file,
- `t-graph`: time to build the dependency graph G of the input set of TGDs, and
- `t-comp`: time to find the special SCCs in the graph G.

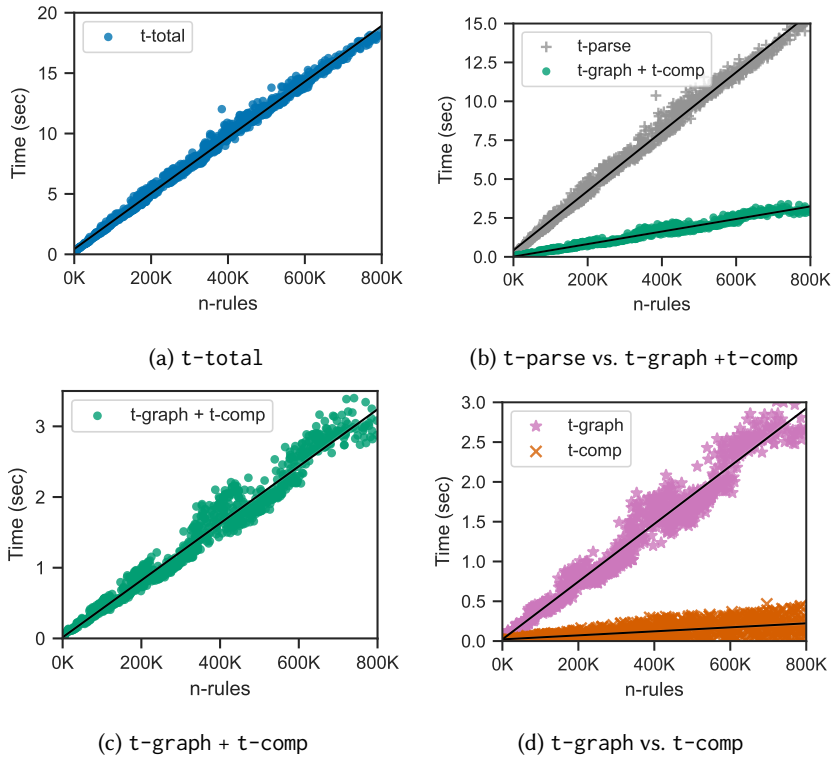
In the rest of the section, we explain how we generate the sets of simple-linear TGDs that are used in our experimental evaluation, and then present our experimental results and discuss the take-home messages. But let us first give a couple of clarification remarks.

Remark 1. In our analysis, we neglect the time taken by the procedure Supports, which, as explained in Section 5.3, consists of two steps: (1) find the predicates occurring in the input database, and (2) traverse the dependency graph starting from the positions in the special SCCs in the reverse order to reach the positions of the relations computed in the first step. Step (1) is performed by running a fast query on the catalog of the DBMS storing the input database, and can be safely ignored as it does not impact the rest of the algorithm. Concerning step (2), the time to traverse the dependency graph is negligible compared to the time needed to find the special SCCs, which is already in the order of milliseconds. Summing up, the procedure Supports takes insignificant time compared to the rest of the algorithm, which we can simply ignore without affecting our analysis for IsChaseFinite[SL]. Therefore, in our experiments, we assume that all the predicates used by the set of simple-linear TGDs occur in the database, and thus, all the positions in the special SCCs are trivially supported. This, in turn, simplifies our experiments as they can be conducted using a very simple database that can be induced by the set Σ of simple-linear TGDs, denoted D_Σ, without using our data generator discussed above. In fact, D_Σ has an atom $R(c_1, \ldots, c_n)$, where $c_1, \ldots, c_n$ are distinct constants, for each predicate $R \in \text{sch}(\Sigma)$.

Remark 2. The parsing time (`t-parse`) clearly depends on the parser's performance, and is not so crucial for evaluating the performance of IsChaseFinite[SL]. However, we include it in our analysis in order to have an accurate figure for the end-to-end runtime of the algorithm, and also compare it with the other two time parameters.

7.1 Generating Simple-linear TGDs

We now discuss how the sets of simple-linear TGDs used in our experiments are generated. To systematically generate a representative family of sets of TGDs, without favouring any of the two key parameters, namely, the size of the underlying schema and the number of TGDs, we consider three *predicate profiles* consisting of sets of TGDs that mention [5,200], [200,400], and [400,600] predicates of arity between 1 and 5, and we further consider three *TGD profiles* consisting of sets of TGDs with [1,333K], [333K,666K], and [666K,1M] TGDs. Note that our choice to fix the arity of the predicates between 1 and 5 is consistent with what we observe in real-life scenarios, where the arity is typically small. The combination of those predicate and TGD profiles gives rise to nine *combined profiles* consisting of sets of TGDs with similar syntactic properties. For example, the combined profile obtained from the predicate profile [200,400] and the TGD profile [333K,666K] consists of sets of TGDs Σ such that $200 \leq \text{sch}(\Sigma) \leq 400$, each predicate of $\text{sch}(\Sigma)$ has arity between 1 and 5, and $333\text{K} \leq |\Sigma| \leq 666\text{K}$. For our experiments, we generated 100 sets of TGDs for each of the nine combined profiles, totalling 900 sets of simple-linear TGDs. This was done as follows. We



(a) `t-total` (b) `t-parse` vs. `t-graph` +`t-comp`

(c) `t-graph` + `t-comp` (d) `t-graph` vs. `t-comp`

Fig. 1. Runtime of IsChaseFinite[SL].

first constructed the underlying schema **S** by generating 1,000 predicates, while their arities were randomly selected from the range [1,5]. Then, for the combined profile induced by the predicate profile $[x, y]$ and the TGD profile $[z, w]$, we have generated 100 sets of simple-linear TGDs by repeatedly executing our TGD generator with input the schema **S** and the tuple of values for the tuning parameters $(ssize, 1, 5, tsize, \text{SL})$, where *ssize* and *tsize* were randomly chosen from the range of values $[x, y]$ and $[z, w]$, respectively.

7.2 Experimental Evaluation

The algorithm IsChaseFinite[SL] was run for each one of the 900 sets of TGDs of the combined profiles discussed above. Recall that the database is induced by the set of TGDs, i.e., for the set of TGDs Σ, the database is D_Σ. The scatter plots in Figure 1 show the runtime of IsChaseFinite[SL](D_Σ, Σ), for each set Σ of TGDs from the combined profiles. In particular, each point in the plots corresponds to one of the 900 sets of TGDs. Figure 1(a) shows the total runtime (`t-total`) for sets of TGDs with various sizes (`n-rules`). Figure 1(b) breaks down `t-total` into the time to parse the TGDs (`t-parse`) and the time to build their dependency graph and find the special SCCs (`t-graph` + `t-comp`). Figure 1(c) zooms in `t-graph` + `t-comp`, which are shown separately in Figure 1(d).

It is evident from the above scatter plots that the time parameters `t-parse` and `t-graph` increase linearly as long as we increase `n-rules`, whereas `t-comp` increases very slowly. Let us remark that we have not observed such a linear relationship (in fact, we have not observed any correlation) between the time parameters `t-parse` and `t-graph`, and the number of predicates of the underlying schema. The linear relationship between `t-parse` and `n-rules` is because parsing each TGD takes

constant time since the arity of the predicates falls in the limited range [1,5], and each TGD has one atom in its body and one atom in its head. Note that allowing multi-heads will not change this since, as discussed in Section 6, the number of head-atoms is negligible compared to the number of TGDs. The linear relationship with `t-graph` (as shown in Figure 1(d)) is because the algorithm iterates over the TGDs and spends constant time to process each TGD and update the graph by adding new nodes and edges. Again, since the arity of the predicates falls in [1,5], and each TGD has one atom in its body and one atom in its head, the number of nodes and edges added in the dependency graph due to a certain TGD is in a small fixed range, and thus, the time to update the graph w.r.t. each TGD is constant. The fact that `t-comp` increases very slowly is because finding the special SCCs solely depends on the dependency graph, which is, in general, much smaller than the set of TGDs, while Tarjan's algorithm is efficient that runs in linear time in the size of the graph.

7.3 Take-home Messages

The main takeaway from the experimental results for simple-linear TGDs is that the primary parameter impacting the runtime of IsChaseFinite[SL] is the number of TGDs (`n-rules`), and we have also observed that the algorithm is very fast even for extremely large sets of TGDs. In fact, most of the end-to-end runtime is spent on parsing (`t-parse`) and building the dependency graph (`t-graph`), whereas the time to find special SCCs (`t-comp`) is insignificant compared to `t-parse` and `t-graph`. To be more precise, `t-parse` is much larger than `t-graph`, and it actually takes most of the total end-to-end runtime of the algorithm. This illustrates the effectiveness of IsChaseFinite[SL] as the actual check for the finiteness of the chase instance for large sets of TGDs is much faster than even reading and parsing the TGDs from the input file.

8 Evaluation for Linear TGDs

We now proceed with the evaluation of IsChaseFinite[L], depicted in Algorithm 3. Differently from IsChaseFinite[SL], where the input database did not play any crucial role, we now have a component that heavily relies on the database, that is, the procedure that computes the database shapes, which is part of dynamic simplification. In other words, we have the *database-dependent component* of IsChaseFinite[L], that is, find the database shapes, and the *database-independent component*, that is, simplify the given set of linear TGDs by using the database shapes, build the dependency graph of the simplified set of TGDs, and find the special SCCs in this graph. We claim that these two components, which from now on we call db-dependent and db-independent, respectively, should be evaluated separately as their runtime is impacted by different parameters. Concerning the db-dependent component, it is obvious that it is only affected by the database, whereas the set of TGDs plays no role. On the other hand, although it is clear that the db-independent component is affected by the set of TGDs, it is not straightforward to see that it is not affected by the input database since it operates on a dynamically simplified set of TGDs. Interestingly, we experimentally confirm below that this is indeed the case. Consequently, towards a refined analysis of the algorithm IsChaseFinite[L], we are going to consider the following four time parameters:

- `t-shapes`: time to find the database shapes,
- `t-parse`: time to parse the TGDs from an input file,
- `t-graph`: time to build the dependency graph G of the simplified version of the input set of TGDs (including the time for the simplification using the database shapes), and
- `t-comp`: time to find the special SCCs in the graph G.

Clearly, `t-shapes` refers to the runtime of the db-dependent component, whereas `t-parse` + `t-graph` + `t-comp`, which we denote by `t-total`, refers to the end-to-end runtime of the db-independent component. In the rest of the section, we explain how we generate the databases

and the sets of linear TGDs that are used in our experimental evaluation, confirm that the db-independent component is not affected by the input database, and then present our experimental results for the two components of IsChaseFinite[L] and discuss the take-home messages. Note that, as stated in Remark 2 in Section 7, although the parsing time is not crucial for evaluating the performance of the db-independent component, we consider it in order to have an accurate figure for the end-to-end runtime, and also compare it with `t-graph` and `t-comp`.

8.1 Generating Databases and Linear TGDs

The goal is to generate a family of pairs of the form (D, Σ), where D is a database and Σ a set of linear TGDs, that will serve as the input to IsChaseFinite[L] during the experimental evaluation. To this end, we first constructed a very large database, which we dub $D^\star$, by using our data generator. In particular, we called the data generator with input (1000, 1, 5, 500K, 500K), and obtained $D^\star$ that mentions 1,000 predicates of arity between 1 and 5, and each such predicate has 500K tuples, resulting in a very large database with 500M tuples in total. We further devised views over $D^\star$ that allow us to define on-demand virtual databases with 1K, 50K, 100K, 250K, and 500K tuples per predicate, resulting in databases with 1M, 50M, 100M, 250M, and 500M tuples in total, respectively. Those views are actually implemented as a simple SQL query that simply keeps the first 1K, 50K, 100K, 250K, and 500K tuples per predicate, respectively. Note that the tuples in $D^\star$ are lexicographically sorted, which means that the different shapes in a relation of $D^\star$ are evenly distributed. Having $D^\star$ and the database views in place the desired family of pairs was generated as described below.

For each one of the nine combined profiles used in the generation of simple-linear TGDs in Section 7, we generated five sets of linear TGDs, totalling 45 sets. In particular, for the combined profile induced by the predicate profile $[x, y]$ and the TGD profile $[z, w]$, we have generated 5 sets of linear TGDs by repeatedly executing our TGD generator with input the schema $\{R \mid R(\bar{c}) \in D^\star\}$, i.e., the 1000 predicates occurring in $D^\star$, and the tuple of values for the tuning parameters $(ssize, 1, 5, tsize, \mathsf{L})$, where *ssize* and *tsize* were randomly chosen from the range of values $[x, y]$ and $[z, w]$, respectively. Let $\Sigma^\star$ be the family that collects the 45 generated sets of linear TGDs. Then, for each set $\Sigma \in \Sigma^\star$ of linear TGDs, by exploiting the database views discussed above, we obtained five virtual databases of varying size (1K, 50K, 100K, 250K, and 500K tuples per predicate), denoted D_Σ^1, D_Σ^{50}, D_Σ^{100}, D_Σ^{250}, and D_Σ^{500}, respectively, leading to five pairs. Summing up, we generated the family

$$\left\{\left(D_\Sigma^s, \Sigma\right) \mid \Sigma \in \Sigma^\star \text{ and } s \in \{1, 50, 100, 250, 500\}\right\}$$

consisting of 225 pairs that will serve as the input to IsChaseFinite[L].

8.2 Experimental Evaluation

Before delving into the evaluation of the two components of the algorithm IsChaseFinite[L], let us first confirm that indeed the db-independent component is not affected by the input database, which in turn justifies our decision to separately evaluate the two components as their runtime is impacted by different parameters.

Separate the Two Components. Figure 2 depicts the average time over all generated pairs, consisting of a database D_Σ^s of a certain size $s \in \{1, 50, 100, 250, 500\}$ and a set Σ of linear TGDs, for building the dependency graph of the dynamically simplified version of Σ using the shapes of D_Σ^s and finding the special SCCs. Interestingly, it confirms that the database size does not impact the time to build the dependency graph and find the special SCCs; thus, it does not impact the end-to-end runtime of the db-independent component, as claimed above. This is because the number of shapes in a database is bounded by the underlying schema. Indeed, the maximum number of shapes of a certain predicate in a database only depends on the arity of the predicate. Moreover,

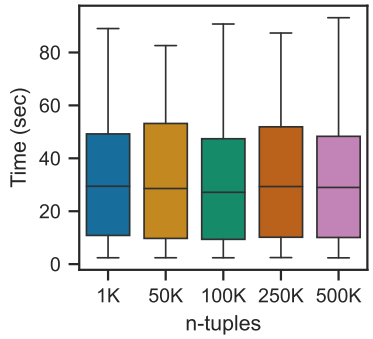
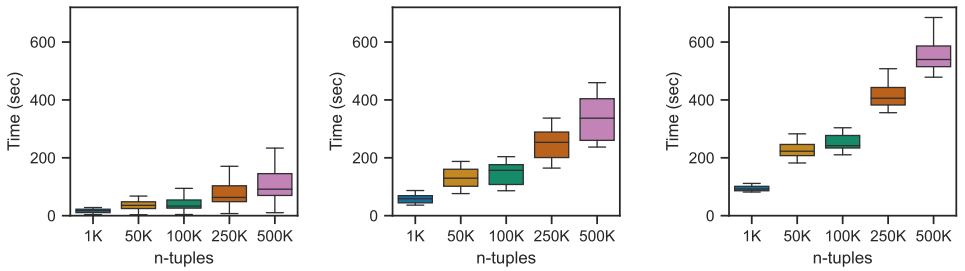
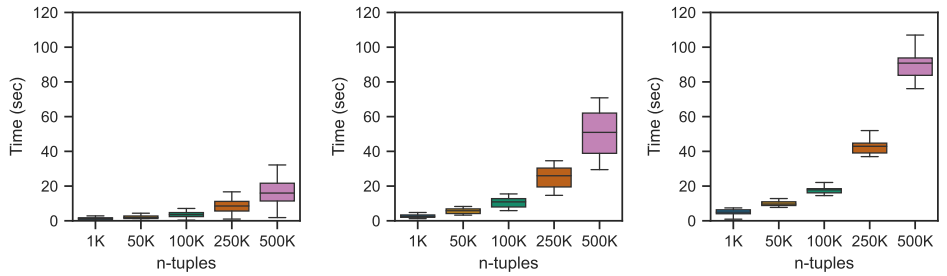


Fig. 2. Average time vs. number of tuples per predicate.



(a) [5,200] (b) [200,400] (c) [400,600]

Fig. 3. Runtime of FindShapes (in-memory implementation).

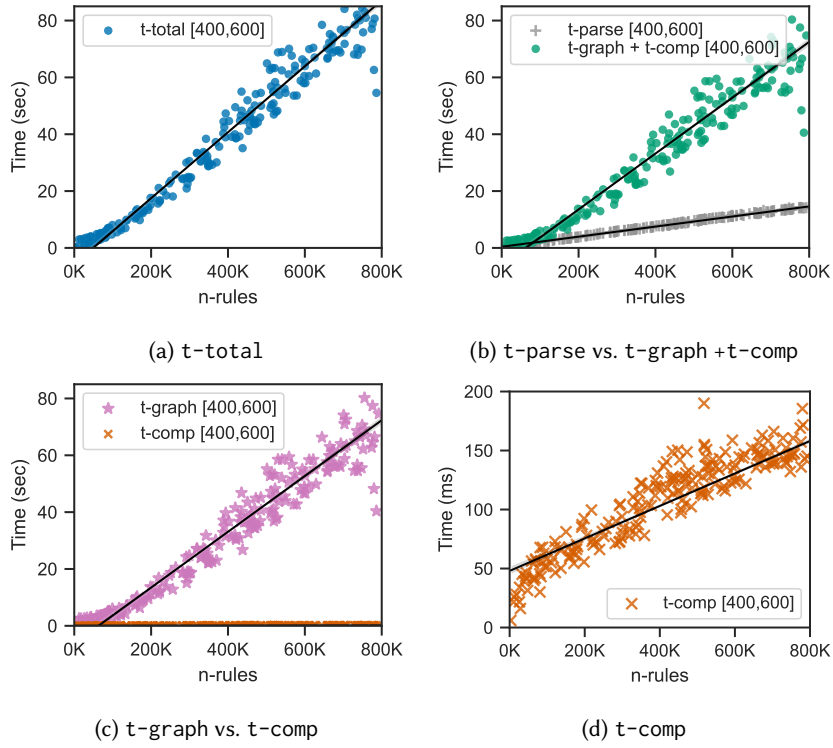


(a) [5,200] (b) [200,400] (c) [400,600]

Fig. 4. Runtime of FindShapes (in-database implementation).

each shape gives rise to only a few simplified TGDs, and it does not significantly affect the time for building and processing the dependency graph.

Evaluation of the DB-dependent Component. We run the procedure FindShapes for each one of the databases D^s_Σ, where $\Sigma \in \Sigma^\star$ and $s \in \{1, 50, 100, 250, 500\}$; 225 executions in total. Recall that we have two kinds of implementations for the procedure FindShapes, namely, in-memory and in-database (see Section 5.4). The bar plots in Figures 3 and 4 show the average runtime over all databases D^s_Σ of a certain size s for finding the shapes in the case of the in-memory and in-database



(a) `t-total` (b) `t-parse` vs. `t-graph` +`t-comp`

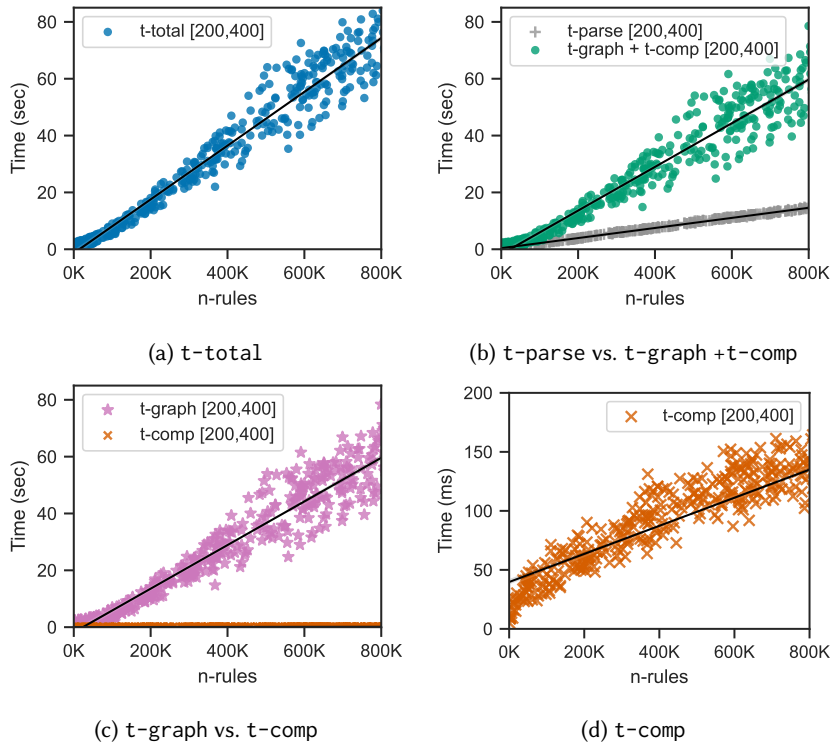
(c) `t-graph` vs. `t-comp` (d) `t-comp`

Fig. 5. Runtime of the db-independent component for the predicate profile [400,600].

implementation, respectively, where each plot corresponds to a certain predicate profile. Note that for both implementations, we observed a similar trend, with the in-database implementation outperforming the in-memory one.

It is evident from the bar plots in Figure 4 that the time to find the shapes increases while the database size increases, which is not surprising. Observe, however, that the time to find the shapes grows quite fast, which should be attributed to the fact that for finding the shapes, we actually need to scan the whole database. Let us finally observe that, by comparing the three bar plots, it is apparent that the number of predicates, reflected in the underlying predicate profile, impacts the time to find the shapes, which explains why we had to analyze each predicate profile separately.

Evaluation of the DB-independent Component. The scatter plots in Figure 5 show the runtime of the db-independent component of the algorithm IsChaseFinite[L] when executed with input (D^s_Σ, Σ), for each set $\Sigma \in \Sigma^\star$ falling in the predicate profile [400,600] and $s \in \{1, 50, 100, 250, 500\}$. In particular, each point in the plots corresponds to a pair (D^s_Σ, Σ). Let us stress that, unlike the analogous Figure 1 for simple-linear TGDs, we focus on a particular predicate profile since otherwise we do not obtain any trend of the runtime w.r.t. the number of TGDs. In other words, the apparent linear trend observed in those plots only holds for sets of TGDs from the same predicate profile. This is because the number of predicates of the underlying schema impacts the number of shapes, which in turn affects the process of dynamic simplification and the size of the dependency graph, and thus, the time parameters `t-graph` and `t-comp` are impacted. This explains why we had to analyze each predicate profile separately; analogous plots for the predicate profiles [5,200] and [200,400] are presented and discussed below.



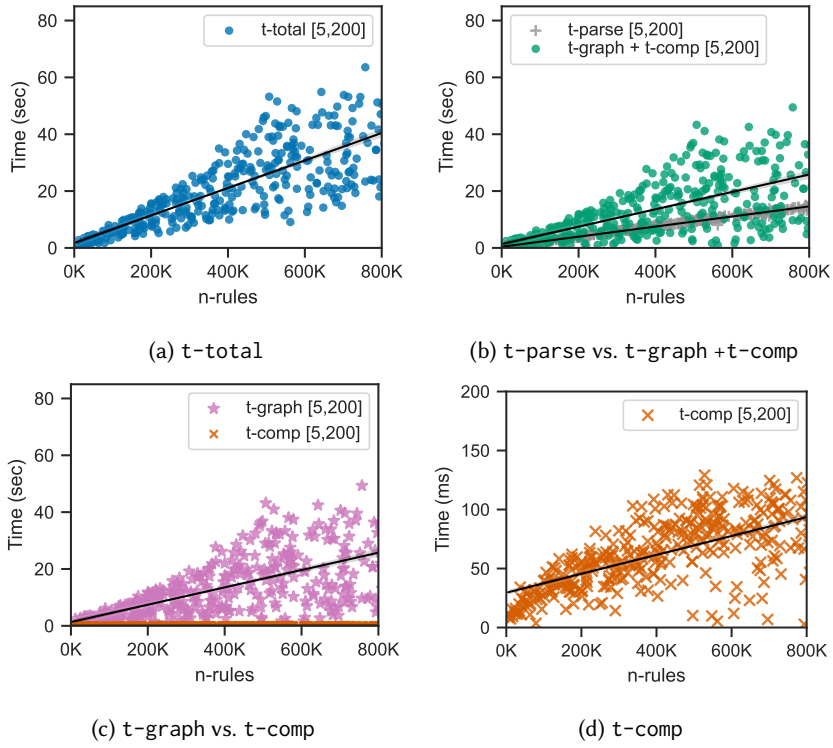
(a) t-total (b) t-parse vs. t-graph +t-comp

(c) t-graph vs. t-comp (d) t-comp

Fig. 6. Runtime of the db-independent component for the predicate profile [200,400].

Figure 5(a) shows the total runtime (t-total) for sets of TGDs with various sizes (n-rules). Figure 5(b) breaks down t-total into the time to parse the TGDs (t-parse) and the time to build their dependency graph and find the special SCCs (t-graph + t-comp), whereas Figure 5(c) shows separately t-graph and t-comp. Figure 5(d) zooms in t-comp. It is evident from the above scatter plots that the time parameters t-parse and t-graph increase linearly as long as we increase n-rules, whereas t-comp increases very slowly. This is essentially what we have observed for simple-linear TGDs in Figure 1, with the key difference that the time needed to parse the TGDs (t-parse) is now much less compared to the time for building the dependency graph and finding the special SCCs (t-graph + t-comp).

The plot for the predicate profile [200,400] is shown in Figure 6, whereas for the profile [5,200] in Figure 7. We can observe, similarly to the predicate profile [400,600], that the time parameters t-parse and t-graph increase linearly as long as we increase n-rules, whereas t-comp increases very slowly. However, it is clear from the plots that, as long as we move to smaller predicate profiles and larger sets of TGDs, the above linear trends become less apparent. This phenomenon can be explained as follows. As we decrease the number of predicates that appear in the schema, it is likely that a large set of TGDs gives rise to a limited number of edges in the underlying dependency. This is because many TGDs simply lead to the same edges, which are, of course, considered once in the graph. This is confirmed by the plot in Figure 8, which depicts the average number of edges in the dependency graphs of the sets of linear TGDs used in our experimental evaluation. For the smaller predicate profiles, increasing the number of TGDs leads to a smaller increase in the graph size compared with the larger predicate profiles. In particular, for the profile [5,200], the



(a) `t-total` (b) `t-parse` vs. `t-graph` +`t-comp`

(c) `t-graph` vs. `t-comp` (d) `t-comp`

Fig. 7. Runtime of the db-independent component for the predicate profile [5,200].

number of edges in the dependency graphs remains almost the same as we increase the number of TGDs.

Let us conclude this analysis by briefly discussing a relevant implementation-independent statistic, that is, how the number of TGDs grows after simplification. This is illustrated by the plot in Figure 9. An interesting observation is that, although dynamic simplification can, in principle, cause an exponential blow-up, we do not observe such a behavior in our data-driven setting.

8.3 Take-home Messages

The main takeaway is that the algorithm IsChaseFinite[L] consists of two components that are of different nature in the sense that their runtime is impacted by different parameters of the input. On the one hand, we have the db-dependent component, which is responsible for finding the database shapes, that is only affected by the size of the database. On the other hand, we have the db-independent component, that is, simplify the given set of linear TGDs by using the database shapes, build the dependency graph of the simplified set of TGDs, and find the special SCCs in this graph, whose runtime is primarily affected by the number of TGDs (`n-rules`). Having said that, we have also observed that the number of predicates also affects the runtime of the db-independent component since it impacts the number of shapes, which in turn affects the process of dynamic simplification and the size of the dependency graph. We conclude by observing that the total runtime of IsChaseFinite[L] is reasonable, which is a strong evidence that fast checking for the finiteness of the chase instance in the case of linear TGDs is not an unrealistic goal. Note

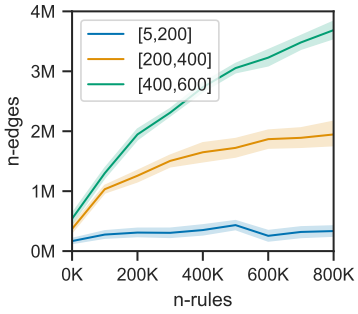


Fig. 8. The average number of edges vs the number of linear TGDs.

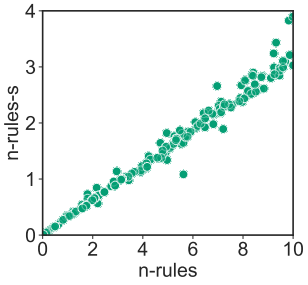


Fig. 9. The number of simplified TGDs (`n-rules-s`) vs. the number of TGDs before simplification (`n-rules`); the x and y axes are scaled by 10^5 and 10^6, respectively.

that most of the total end-to-end runtime of the algorithm is spent on finding database shapes, which indicates that our future efforts should be concentrated on improving the db-dependent component.

9 Validation of Results

In Sections 7 and 8, we have presented an experimental evaluation of IsChaseFinite[SL] and IsChaseFinite[L] using synthetically generated databases and sets of TGDs. In this final section on (simple-)linear TGDs, we use synthetic databases and sets of TGDs that are available in the literature with the aim of validating the main outcome of the stress test analysis performed above.

9.1 Adopted Scenarios

We considered three families of databases and sets of TGDs that are discussed below. They are also summarized in Table 1, where we report statistics about (i) the underlying schema, i.e., number of predicates (`n-pred`) and arity of predicates (`arity`), (ii) the size of the database, i.e., number of atoms (`n-atoms`) and number of shapes (`n-shapes`), and (iii) the number of TGDs (`n-rules`).

Deep. This family collects sets of simple-linear TGDs that are also weakly-acyclic [Benedikt et al. 2017]. It has been developed to test scenarios with a large number of chase applications and large source instances, as well as a significant number of source-to-target TGDs and target TGDs in a data exchange setting.

LUBM. This is a popular benchmark consisting of an ontology modelled using the central DL EL, called Univ-Bench, and a data generator, called UBA, for generating synthetic data over the vocabulary of Univ-Bench [Guo et al. 2005]. We use four members of the LUBM family, namely, `LUBM-1`, `LUBM-10`, `LUBM-100`, and `LUBM-1K`. Let us clarify that the DL axioms occurring in Univ-Bench can be easily converted into TGDs with one atom in the head that do not repeat variables in an atom. However, not all the axioms lead to linear TGDs, and thus, we kept only those that can be converted into linear TGDs (which are also simple-linear).

iBench. This is a framework for generating dependencies such as TGDs with tuning parameters that can control a wide range of properties [Arocena et al. 2015]. For our experiments, we use the following sets of simple-linear TGDs generated using iBench: (i) STB-128, derived from an earlier STBenchmark [Alexe et al. 2008], and is the smaller scenario of the family, and (ii) ONT-256, a scenario that has several times larger source instances. For both of those sets of TGDs, we used the

Table 1. The Families Deep, LUMB, and iBench

Family	Name	n-pred	arity	n-atoms	n-shapes	n-rules
Deep	Deep-100	1,299	4	1,000	1,000	4,241
	Deep-200					4,541
	Deep-300					4,841
LUBM	LUBM-1	104	[1,2]	99,547	30	137
	LUBM-10			1,272,575		
	LUBM-100			13,405,381		
	LUBM-1K			133,573,854		
iBench	STB-128	287	[1,10]	1,109,037	129	231
	ONT-256	662	[1,11]	2,146,490	245	785

databases from Benedikt et al. [2017] that were generated using the data generator in Barbosa et al. [2002] and consists of 1,000 tuples per source relation.

Let us remark that in what follows we discuss our experimental evaluation of the algorithm IsChaseFinite[L] for linear TGDs. Concerning the algorithm IsChaseFinite[SL] for simple-linear TGDs, there is not much to discuss other than the fact that it runs in a few milliseconds for all the scenarios discussed above. This is a confirmation that for simple-linear TGDs, checking for the finiteness of the chase instance can be done very efficiently.

9.2 Experimental Evaluation

We run the algorithm IsChaseFinite[L] with all the scenarios discussed above, and the experimental results are summarized in Table 2. Note that t-total refers to the end-to-end runtime for checking the finiteness of the chase, i.e, t-total = t-parse + t-graph + t-comp + t-shapes. Let us clarify that this measure differs from t-total defined in Section 7.2, where t-shapes was excluded to remain consistent with the stress tests of IsChaseFinite[SL], which ignore the database-dependent component; the latter component was analyzed separately. Here, we include t-shapes to report the end-to-end runtime and demonstrate the applicability of our algorithms on these benchmarks. We highlight in a box the best end-to-end runtime obtained by considering either the in-memory or the in-database implementation for finding the database shapes.

The parsing time (t-parse) is insignificant in all scenarios, as the number of rules (n-rules), which is the main parameter that impacts the parsing time, is at most 4,000. The time to build the dependency graph (t-graph) and the time to find the special SCCs (t-comp) are also negligible. This is due to the limited number of shapes (n-shapes) in these scenarios, which means that the dynamic simplification that uses those shapes does not construct a large set of simplified TGDs, and thus, the induced dependency graph is rather small. It is evident from Table 2 that the most costly task is finding the database shapes, which is consistent with what we observed in Section 7.

We report the time to find the shapes (t-shapes) for both the in-memory and the in-database implementations of the procedure FindShape. The in-memory implementation is faster for the Deep family since the underlying database has many singleton relations, i.e., relations with only one tuple, and thus, loading the tuples and finding the shapes can be done efficiently. On the other hand, the in-database implementation takes more time because it runs one query per relation, which results in many queries. For the LUBM and iBench families, the in-database implementation is faster as it finds the shapes by running a few queries due to the small number of predicates. Note that, in general, the runtime of the in-database implementation increases with the number of tuples in the relations, which is evident from the LUBM scenarios. As a result, finding the shapes

Table 2. Runtime of IsChaseFinite[L] in Milliseconds

				In-db		In-memory	
Name	t-parse	t-graph	t-comp	t-shapes	t-total	t-shapes	t-total
Deep-100	214	90	10	6,641	6,957	447	**763**
Deep-200	265	116	9		7,033	447	**839**
Deep-300	234	100	11		6,986	500	**846**
LUBM-1	84	10	1	221	**318**	2,724	2,820
LUBM-10	46	10	1	830	**889**	10,943	11,002
LUBM-100	45	11	1	6,396	**6,454**	70,131	70,189
LUBM-1K	43	231	80	65,578	**65,932**	854,015	854,369
STB-128	78	18	7	4,991	**5,096**	7,379	7,484
ONT-256	179	35	8	11,726	**11,949**	15,761	15,984

for LUBM-1K takes a significant time. However, it is still much lower than the time for finding the database shapes using the in-memory implementation that requires loading many tuples.

9.3 Discussion

It is fair to conclude that the experimental evaluation performed in this section confirms the main outcome of the analysis for the algorithm IsChaseFinite[L] performed in Section 8. In particular, we observe that indeed the costly task is finding the database shapes, whereas the time taken by the db-independent component is negligible. Moreover, we see that checking for the finiteness of the chase instance can be done rather efficiently in practice. In particular, for sets consisting of thousands of TGDs, such as the Deep scenarios, and millions of facts, such as LUBM-10, it takes less than a second. For schemas with a large number of predicates (e.g., STB-128 and ONT-256) and databases with a large number of atoms (e.g., LUBM-100), it takes less than 10 seconds. Finally, for very large databases with hundreds of millions of atoms, such as LUBM-1K, it takes around a minute, which is a reasonable time taking into account the actual size of the input.

Another interesting takeaway from the experimental evaluation of this section is that there is no clear way to go regarding the implementation of FindShape among the in-memory and the in-database options. In particular, the in-memory implementation is preferred when there are a few tuples per relation in the database, whereas the in-database implementation performs better when the underlying schema has a few predicates of small arity. For schemas with many predicates, each of which has many tuples in the input database, both implementations require significant time, and the offline computation of the database shapes might be preferred.

9.4 Feasibility in the Real World

We have also considered real-world scenarios with the aim of analysing the feasibility of our algorithms in a more realistic context. We considered 483 pairs (D, Σ), where D is a database and Σ a set of simple-linear TGDs, obtained from the ontology repository of the Data and Knowledge Group of the University of Oxford by keeping from the available ontologies only the ontological axioms that can be expressed as simple-linear TGDs. Let us stress that these are real-world ontologies modelled using DLs, and include, among others, a large subset of the Gardiner ontology corpus, several Phenoscape ontologies, and a number of ontologies from two versions of the **Open Biomedical Ontology** (**OBO**) corpus [Grau et al. 2013]. Both IsChaseFinite[SL] and IsChaseFinite[L] performed swiftly with the maximum time taken to be 1.77 and 0.44 seconds, respectively. On average, IsChaseFinite[SL] and IsChaseFinite[L] took approximately 0.85 and 0.22 seconds, respectively,

across different scenarios. A rather unexpected finding is that IsChaseFinite[L], which employs dynamic simplification, outperformed IsChaseFinite[SL]. This should be attributed to two reasons: (i) the underlying schema consists of only unary and binary predicates, which in turn allows for a fast computation of shapes, and (ii) dynamic simplification uses a small number of simple-linear TGDs for building the dependency graph, which is a consequence of the fact that the databases mention a small number of predicates from the underlying schema.

10 Chase Termination for Guarded TGDs via Linearization

Guarded TGDs is a well-known and well-studied generalization of linear TGDs, introduced in Calì et al. [2013], inspired by the guarded fragment of first-order logic. A TGD σ is called *guarded* if it has an atom in its body, called the *guard* and denoted $\text{guard}(\sigma)$, that contains all the universally quantified variables of σ. By convention, in case there are more than one atoms in $\text{body}(\sigma)$ that contain all the universally quantified variables of σ, then $\text{guard}(\sigma)$ is defined as the left-most such atom. It is clear that a linear TGD is trivially guarded. The chase termination problem in the presence of guarded TGDs has been extensively studied in the literature [Calautti et al. 2015, 2022; Gogacz et al. 2023]. Interestingly, it has been recently shown in Calautti et al. [2022] that one can effectively use the machinery for linear TGDs by first converting the given set of guarded TGDs into a set of linear TGDs via the so-called *linearization technique* without affecting the termination of the semi-oblivious chase. Our goal in this final section is to understand how realistic is the use of the linearization technique in the context of the semi-oblivious chase termination problem. To this end, we are going to consider a special fragment of guarded TGDs, which, as we explain below, corresponds to the widely used DL EL [Baader et al. 2005]. As we shall see, even with this lightweight fragment of guarded TGDs, we cannot go far in practice by relying on the linearization technique. Having said that, we will also present a limited number of realistic scenarios where the lineariztion technique is good enough in practice.

In the rest of the section, we will formally introduce the fragment of guarded TGDs that corresponds to the DL EL, discuss how the finiteness of the chase can be characterized via the linearization technique and obtain a theoretical algorithm, discuss how the theoretical algorithm can lead to a practical algorithm that is amenable to an efficient implementation, and finally discuss the implementation details and the experimental evaluation of the presented practical algorithm.

10.1 A Lightweight Fragment of Guarded TGDs

DLs form a family of Knowledge Representation languages that has emerged from symbolic AI, underlies the OWL 2 recommendation for ontology languages on the web, and whose members can be seen as decidable fragments of (two-variable guarded) first-order logic [Baader et al. 2017]. A well-known DL that has become very popular due to its efficient reasoning properties is EL [Baader et al. 2005]. An *EL-concept* is formed according to the rule

$$A, B \ := \ \top \mid C \mid A \sqcap B \mid \exists R.B,$$

where $\top$ represents the universal concept, C denotes a concept name (i.e., a unary predicate), and R denotes a role name (i.e., a binary predicate). An *EL-ontology* is a finite set of EL-concept inclusions of the form $A \sqsubseteq B$ with A, B being EL-concepts. Given an instance I, each EL-concept A is associated with an extension A^I in the obvious way:

$$\begin{aligned} \top^I &= \text{dom}(I) \\ C^I &= \{a \in \text{dom}(I) \mid C(a) \in I\} \\ (A \sqcap B)^I &= A^I \cap B^I \\ (\exists R.A)^I &= \{a \in \text{dom}(I) \mid \text{there is } b \in \text{dom}(I) \text{ such that } R(a,b) \in I \text{ and } b \in A^I\}. \end{aligned}$$

We say that I satisfies an EL-concept inclusion $A \sqsubseteq B$, written $I \models A \sqsubseteq B$, if $A^I \subseteq B^I$. We further say that I satisfies an EL-ontology O, written $I \models O$, if I satisfies each EL-concept inclusion of O.

It is well-known that, for standard reasoning tasks, an EL-ontology can always be normalized in polynomial time into an EL-ontology that contains only EL-concepts in one of the following forms:

$$A \sqsubseteq B \qquad A \sqcap B \sqsubseteq C \qquad \exists R.A \sqsubseteq B \qquad A \sqsubseteq \exists R.B,$$

where A, B, C are concept names and R is a role name. Thus, we can always assume that an EL-ontology is in normal form. Interestingly, an EL-ontology O in normal form can be equivalently expressed as a finite set of guarded TGDs of a special form. This is done by simply applying the transformation $\tau(\cdot)$ on each EL-concept inclusion of O defined as follows:

$$\begin{aligned}
\tau(A \sqsubseteq B) &= A(x) \rightarrow B(x) \\
\tau(A \sqcap B \sqsubseteq C) &= A(x), B(x) \rightarrow C(x) \\
\tau(\exists R.A \sqsubseteq B) &= R(x,y), A(y) \rightarrow B(x) \\
\tau(A \sqsubseteq \exists R.B) &= A(x) \rightarrow \exists y R(x,y), B(y).
\end{aligned}$$

Note that the variables x, y in the above definition are arbitrarily chosen from the set $\mathbf{V}$. By abuse of notation, we write $\tau(O)$ for the obtained set of guarded TGDs. It is straightforward to see that, given an EL-ontology O in normal form, for every instance I, it holds that $I \models O$ iff $I \models \tau(O)$, i.e., O and $\tau(O)$ are logically equivalent, which means that they are satisfied by the same instances.

From the above brief introduction on EL and how EL-ontologies in normal form can be equivalently expressed as guarded TGDs, it should now be clear how the lightweight fragment of guarded TGDs that corresponds to EL, and is of special interest for the purposes of this section, can be defined. In particular, we defined the class TEL of TGDs (the TGD representation of EL-ontologies in normal form) that collects (up to variable renaming) all the finite sets Σ of guarded TGDs such that there is an EL-ontology O in normal form with $\Sigma = \tau(O)$.

10.2 Characterizing the Finiteness of the Chase via Linearization

We proceed to present a characterization of the finiteness of the chase instance analogous to Theorem 3.3 for the class TEL of TGDs. This characterization relies on the so-called linearization technique, originally introduced in Gottlob et al. [2014] for query answering purposes, and then used in Calautti et al. [2022] in the context of chase termination. Note that in Calautti et al. [2022], the linearization technique is used in its full generality for characterizing the finiteness of the chase instance for arbitrary guarded TGDs. For our purposes, we present a simplified version of the linearization technique that deals only with the lightweight fragment TEL of guarded TGDs. The essence of the linearization technique is to convert a database D and a set $\Sigma \in \mathsf{TEL}$ of TGDs into a database D' and a set Σ' of simple-linear TGDs, respectively, such that $\mathsf{chase}(D, \Sigma)$ is finite iff $\mathsf{chase}(D', \Sigma')$ is finite. With the latter equivalence in place, we can then use the characterization for simple-linear TGDs provided by Theorem 3.3 in order to obtain a characterization for TEL.

Before delving into the low-level details of linearization, let us first give a high-level description. Given a database D, a set $\Sigma \in \mathsf{TEL}$ of TGDs, and an atom $\alpha \in \mathsf{chase}(D, \Sigma)$, the *type of α (w.r.t. D and Σ)* is the set of atoms in $\mathsf{chase}(D, \Sigma)$ that use a unary predicate and mention a term from α [Calì et al. 2013]. The key property of the type states that the set of atoms in $\mathsf{chase}(D, \Sigma)$ that can be derived from α (used as a guard) is determined by the type of α. Now, for linearizing the database D (relative to Σ), we encode each atom $\alpha \in D$ and its type as an atom of the form $[\tau](\cdot)$, where τ is a symbolic representation of α and its type. For example, if $\alpha = R(a, b)$ and $\{S(a), S(b), P(b)\}$ is its type, we encode $R(a, b)$ and its type as $[R(1, 2), \{S(1), S(2), P(2)\}](a, b)$.

For the linearization of Σ, the intention is, for a TGD $\sigma \in \Sigma$, to encode the shape of the type τ of the guard of σ in a predicate $[\tau]$, and then replace σ with a linear TGD that uses in its body an atom $[\tau](\cdot)$. We need an effective way, though, to compute the type of an atom α by completing its known part, which is inherited from the type of the guard atom that generates α, with atoms that mention the new null values invented in α. This exploits the main property of the type.

To formally present the linearization technique, we first need to recall some auxiliary notions. Fix a database D and a set $\Sigma \in \mathsf{TEL}$ of TGDs. For an atom $\alpha \in \mathsf{chase}(D, \Sigma)$, the *type of α w.r.t. D and Σ* is defined as the set of atoms

$$\mathsf{type}_{D,\Sigma}(\alpha) \;=\; \{R(t) \mid R(t) \in \mathsf{chase}(D, \Sigma) \text{ and } t \in \mathsf{dom}(\alpha)\},$$

i.e., it collects all the atoms of $\mathsf{chase}(D, \Sigma)$ with a unary predicate that mention a term that appears in α. Similarly, the *completion of D w.r.t.* Σ is the set of all atoms of $\mathsf{chase}(D, \Sigma)$ with a unary predicate that mention a constant that occurs in D, that is, the set

$$\mathsf{complete}(D, \Sigma) \;=\; \{R(c) \in \mathsf{chase}(D, \Sigma) \mid c \in \mathsf{dom}(D)\}.$$

A *Σ-type* τ is a pair (α, T), where α is either $R(1)$ or $P(1,1)$ or $P(1,2)$ with $R/1, P/2 \in \mathsf{sch}(\Sigma)$, and $T \subseteq \{S(t) \mid S/1 \in \mathsf{sch}(\Sigma) \text{ and } t \in \mathsf{dom}(\alpha)\} \setminus \{\alpha\}$. We write $\mathsf{guard}(\tau)$ for the atom α, $\mathsf{atoms}(\tau)$ for the set $T \cup \{\alpha\}$, and $\mathsf{ar}(\tau)$ for the integer $\mathsf{ar}(\mathsf{guard}(\tau))$, i.e., the *arity of τ*. Intuitively, τ encodes the shape of a guard and its type. Assuming α is of the form $R(1)$, given a unary tuple (t), the *instantiation of τ with* (t), denoted $\tau((t))$, is the set of atoms obtained from $\mathsf{atoms}(\tau)$ after replacing 1 with t. Analogously, assuming that α is of the form $P(1,1)$ (respectively, $P(1,2)$), given a tuple $\bar{t}$ of the form (t,t) (respectively, (t,u) with $t \neq u$), the *instantiation of τ with $\bar{t}$*, denoted $\tau(\bar{t})$, is the set of atoms obtained from $\mathsf{atoms}(\tau)$ after replacing 1 with t (respectively, 1 with t and 2 with u). Finally, for $\alpha = R(t)$ (respectively, $\alpha = P(t,t)$, $\alpha = P(t,u)$ with $t \neq u$), we write Σ-types(α) for the set of all Σ-types τ such that $\mathsf{guard}(\tau)$ is of the form $R(1)$ (respectively, $P(1,1)$, $P(1,2)$). We are now ready to present the linearization technique.

The *linearization of D relative to* Σ is defined as

$$\mathsf{linear}_{\Sigma}(D) \;=\; \left\{[\tau](\bar{t}) \mid R(\bar{t}) \in D,\ \tau \in \Sigma\text{-types}(R(\bar{t})),\ \tau(\bar{t}) = \mathsf{type}_{D,\Sigma}(R(\bar{t}))\right\},$$

where $[\tau]$ is a new predicate not occurring in $\mathsf{sch}(\Sigma)$ that serves as a symbolic representation of the atom $R(\bar{t})$ and its type w.r.t. D and Σ.

We now explain how the TGDs of Σ are linearized. Consider a TGD $\sigma \in \Sigma$. For a Σ-type τ such that there exists a homomorphism h from $\mathsf{body}(\sigma)$ to $\mathsf{atoms}(\tau)$ and $h(\mathsf{guard}(\sigma)) = \mathsf{guard}(\tau)$, the *linearization of σ induced by τ and h*, denoted $\sigma[\tau, h]$, is defined as follows:

- If σ is of the form $A(x) \to B(x)$, then $\sigma[\tau, h]$ is the TGD

$$[\tau](x) \;\to\; [\tau'](x),$$

  where $\tau' = (B(1), T \setminus \{B(1)\})$ with

$$T \;=\; \{\beta \in \mathsf{complete}(\{B(1)\} \cup \mathsf{atoms}(\tau), \Sigma) \mid \mathsf{dom}(\beta) = \{1\}\}.$$

- If σ is of the form $A(x), B(x) \to C(x)$, then $\sigma[\tau, h]$ is the TGD

$$[\tau](x) \;\to\; [\tau'](x),$$

  where $\tau' = (C(1), T \setminus \{C(1)\})$ with

$$T \;=\; \{\beta \in \mathsf{complete}(\{C(1)\} \cup \mathsf{atoms}(\tau), \Sigma) \mid \mathsf{dom}(\beta) = \{1\}\}.$$

- If σ is of the form $R(x,y), A(y) \to B(x)$, then $\sigma[\tau, h]$ is the TGD

$$[\tau](x,y) \;\to\; [\tau'](x),$$

where $\tau' = (B(1), T \setminus \{B(1)\})$ with

$$T \;=\; \{\beta \in \mathsf{complete}(\{B(1)\} \cup \mathsf{atoms}(\tau), \Sigma) \mid \mathsf{dom}(\beta) = \{1\}\}.$$

- If σ is of the form $A(x) \rightarrow \exists y R(x, y), B(y)$, then $\sigma[\tau, h]$ is the TGD

$$[\tau](x) \;\rightarrow\; \exists y [\tau_1](x, y), [\tau_2](y),$$

where $\tau_1 = (R(1, 2), T_1 \setminus \{R(1, 2)\})$ with

$$T_1 \;=\; \{\beta \in \mathsf{complete}(\{R(1, 2), B(2)\} \cup \mathsf{atoms}(\tau), \Sigma) \mid \mathsf{dom}(\beta) \subseteq \{1, 2\}\}$$

and $\tau_2 = (B(2), T_2 \setminus \{B(2)\})$ with

$$T_2 \;=\; \{\beta \in \mathsf{complete}(\{B(2)\} \cup \mathsf{atoms}(\tau), \Sigma) \mid \mathsf{dom}(\beta) = \{2\}\}.$$

We write $\mathsf{linear}_\Sigma(\sigma)$ for the set of all linearizations of σ induced by some Σ-type τ and homomorphism h from $\mathsf{body}(\sigma)$ to $\mathsf{atoms}(\tau)$ such that $h(\mathsf{guard}(\sigma)) = \mathsf{guard}(\tau)$. Finally, the *linearization of* Σ is defined as the (finite) set of TGDs

$$\mathsf{linear}(\Sigma) \;=\; \bigcup_{\sigma \in \Sigma} \mathsf{linear}_\Sigma(\sigma).$$

We proceed to give an example that illustrates the notion of linearization introduced above.

Example 10.1. Consider the set $\Sigma \in \mathsf{TEL}$ consisting of the TGDs

$$\sigma_1 \;:\; R(x, y), A(y) \;\rightarrow\; B(x) \qquad \sigma_2 \;:\; B(x) \;\rightarrow\; A(x).$$

For a database D, during the construction of $\mathsf{chase}(D, \Sigma)$, the TGD σ_1 can be triggered whenever an atom α of the form $R(t, u)$ (possibly with $t = u$) exists in $\mathsf{chase}(D, \Sigma)$, whose type also contains the atom $A(u)$. The goal of linearizing σ_1 is to convert it into a set of linear TGDs that account for all possible ways of triggering σ_1, i.e., by considering all possible shapes that the atom α, together with its type, could possibly have in $\mathsf{chase}(D, \Sigma)$, for every database D. Hence, each linearization $\sigma_1[\tau, h]$ of σ_1 induced by some Σ-type τ and some homomorphism h from $\mathsf{body}(\sigma_1)$ to $\mathsf{atoms}(\tau)$ with $h(\mathsf{guard}(\sigma_1)) = \mathsf{guard}(\tau)$, encodes one of such possible ways of triggering σ_1. For example, the Σ-type $\tau = (R(1, 2), \{A(2), B(2)\})$ and the homomorphism h such that $h(x) = 1$ and $h(y) = 2$ capture the case when σ_1 is triggered using an atom α of the form $R(t, u)$, together with the atom $A(u)$, assuming the type of α also contains $B(u)$. Thus, the TGD $\sigma_1[\tau, h]$ is of the form

$$[(R(1, 2), \{A(2), B(2)\}](x, y) \quad\rightarrow\quad [(B(1), S)](x).$$

The type S of $B(1)$, in the head of $\sigma_1[\tau, h]$, is computed coherently with the assumption made in τ regarding the type of the guard of σ_1. In this case, S contains all the atoms, other than $B(1)$, with a unary predicate and only the constant 1 as their argument that can be derived from $\{B(1)\} \cup \mathsf{atoms}(\tau) = \{R(1, 2), A(2), B(1), B(2)\}$ using the TGDs of Σ. In particular,

$$S \;=\; \{\beta \in \mathsf{complete}(\{R(1, 2), A(2), B(1), B(2)\}, \Sigma) \mid \mathsf{dom}(\beta) = \{1\}\} \setminus \{B(1)\} = \{A(1)\}.$$

By considering all possible Σ-types and homomorphisms, the linearization $\mathsf{linear}_\Sigma(\sigma_1)$ of σ_1 consists of the following simple-linear TGDs:

$$[(R(1, 2), \{A(2)\} \cup S)](x, y) \quad\rightarrow\quad [(B(1), \{A(1)\})](x),$$

for each $S \subseteq \{A(1), B(1), B(2)\}$. Similarly, $\mathsf{linear}_\Sigma(\sigma_2)$ consists of the following simple-linear TGDs:

$$\begin{aligned} [(B(1), \emptyset)](x) \quad&\rightarrow\quad [(A(1), \{B(1)\})](x) \\ [(B(1), \{A(1)\})](x) \quad&\rightarrow\quad [(A(1), \{B(1)\})](x). \end{aligned}$$

The chase of a database D w.r.t. Σ can be simulated via the chase of $\mathsf{linear}_\Sigma(D)$ w.r.t. $\mathsf{linear}(\Sigma)$. ∎

It is implicit in Calautti et al. [2022] that $\mathsf{chase}(D, \Sigma)$ is finite iff $\mathsf{chase}(\mathsf{linear}_\Sigma(D), \mathsf{linear}(\Sigma))$ is finite. Since $\mathsf{linear}(\Sigma)$ consists of simple-linear TGDs, from Theorem 3.3, we immediately get the following characterization for the finiteness of the chase instance for the class of TGDs TEL:

Theorem 10.2. *Consider a database D and a set $\Sigma \in \mathsf{TEL}$ of TGDs. The following are equivalent:*

(1) $\mathsf{chase}(D, \Sigma)$ *is finite.*
(2) $\mathsf{linear}(\Sigma)$ *is* $\mathsf{linear}_\Sigma(D)$*-weakly-acyclic.*

It is clear that Theorem 10.2 provides a simple algorithm for checking whether the chase instance is finite. In particular, given a database D and a set Σ from TEL, we simply need to check whether $\mathsf{linear}(\Sigma)$ is $\mathsf{linear}_\Sigma(D)$-weakly-acyclic, in which case the algorithm returns `true`; otherwise, it returns `false`. Our goal is to experimentally evaluate the above algorithm with the aim of understanding how realistic is the use of the linearization technique in the context of the semi-oblivious chase termination problem and revealing its limitations in practice. Of course, a naive implementation of the above algorithm will lead to poor performance, and thus, will not be very useful towards our goal. Indeed, linearization is expensive as it can lead to a very large number of simple-linear TGDs. Consider, for example, a set $\Sigma \in \mathsf{TEL}$ mentioning $n > 0$ unary predicates, and consider a TGD $\sigma \in \Sigma$ of the form $A(x) \rightarrow B(x)$. One can verify that the number of Σ-types τ for which there exists a homomorphism h from $\mathsf{body}(\sigma)$ to $\mathsf{atoms}(\tau)$ is 2^{n-1}, and thus $|\mathsf{linear}(\Sigma)| \geq 2^{n-1}$.

Hence, we first need to reconsider the above theoretical algorithm in order to obtain an algorithm that is amenable to an efficient implementation. This is the subject of the next subsection.

10.3 Towards a Practical Termination Algorithm

We present the algorithm IsChaseFinite[TEL] that accepts as input a database D and a set Σ of TGDs from TEL, and checks whether $\mathsf{linear}(\Sigma)$ is $\mathsf{linear}_\Sigma(D)$-weakly-acyclic, which is equivalent to saying that the instance $\mathsf{chase}(D, \Sigma)$ is finite. Let us stress that IsChaseFinite[TEL] relies on a refined notion of linearization that *dynamically linearizes* Σ by leveraging the given database D, instead of doing it statically as explained above without taking any database into account.

Similarly to dynamic simplification employed in the algorithm IsChaseFinite[L], the goal of dynamic linearization is to keep only TGDs of $\mathsf{linear}(\Sigma)$ that are really needed for checking whether the chase instance is finite. For instance, consider the set $\Sigma \in \mathsf{TEL}$ of TGDs from Example 10.1 and the database $D = \{R(a, b), A(b)\}$. We have that $\mathsf{linear}_\Sigma(D) = \{[\tau_1](a, b), [\tau_2](b)\}$ with $\tau_1 = (R(1, 2), \{A(1), A(2), B(1)\})$ and $\tau_2 = (A(1), \{B(1)\})$. It is clear that most of the TGDs of $\mathsf{linear}(\Sigma)$ are not needed to construct $\mathsf{chase}(\mathsf{linear}_\Sigma(D), \mathsf{linear}(\Sigma))$ as the only TGDs being triggered, starting from the database $\mathsf{linear}_\Sigma(D)$, are the following with $\tau_3 = (B(1), \{A(1)\})$

$$[\tau_1](x, y) \rightarrow [\tau_3](x) \qquad [\tau_3](x) \rightarrow [\tau_2](x).$$

In this subsection, we present the algorithm IsChaseFinite[TEL] at a high-level without delving into implementation details; the latter will be the subject of the next subsection.

Dynamic Linearization. We refine the technique of linearization by taking into account the underlying database, which leads to dynamic linearization. In particular, given a database D and a set Σ of TGDs from TEL, the goal is to define a set $\mathsf{linear}_D(\Sigma)$, which is a subset of $\mathsf{linear}(\Sigma)$, that enjoys two crucial properties:

(1) It holds that $\mathsf{chase}(\mathsf{linear}_\Sigma(D), \mathsf{linear}(\Sigma))$ is finite iff $\mathsf{chase}(\mathsf{linear}_\Sigma(D), \mathsf{linear}_D(\Sigma))$ is finite, which tells us that dynamic linearization preserves the finiteness of the chase.
(2) It is, in general, much faster to construct the set $\mathsf{linear}_D(\Sigma)$ instead of the set $\mathsf{linear}(\Sigma)$ obtained by statically linearizing Σ.

Item (1) is established by Lemma 10.4 below. Concerning item (2), we have experimentally verified that for existing databases and sets of TGDs from the literature (in fact, those used in our experimental evaluation presented below), dynamic linearization is indeed much faster than static linearization. In fact, even for small sets of TGDs, it is practically unfeasible to statically linearize Σ due to the very large number of Σ-types that exceeds the capacity of the main memory.

Analogously to dynamic simplification, the key idea of dynamic linearization is to exploit the types occurring in the linearization of the database to guide the linearization of the TGDs. More precisely, given a database D and a set Σ of TGDs from TEL, we first collect the Σ-types that can be derived from the Σ-types occurring in $\mathsf{linear}_\Sigma(D)$ using the TGDs of Σ; we denote this set as $\Sigma(\mathsf{types}_\Sigma(D))$, where $\mathsf{types}_\Sigma(D) = \{\tau \mid \text{there is an atom of the form } [\tau](\bar{t}) \text{ in } \mathsf{linear}_\Sigma(D)\}$. Then, $\mathsf{linear}_D(\Sigma)$ keeps from the set $\mathsf{linear}(\Sigma)$ only those simple-linear TGDs such that the predicate of their body-atom belongs to $\Sigma(\mathsf{type}_\Sigma(D))$, as these are the only TGDs that can be applied during the construction of the instance $\mathsf{chase}(\mathsf{linear}_\Sigma(D), \mathsf{linear}(\Sigma))$. All the other TGDs of $\mathsf{linear}(\Sigma)$ are superfluous whenever the input database is D in the sense that they will never be applied during the construction of $\mathsf{chase}(\mathsf{linear}_\Sigma(D), \mathsf{linear}(\Sigma))$. We proceed to formalize this idea. To this end, we need to introduce some auxiliary notions.

For a set Σ of TGDs from TEL, we write $\mathsf{types}(\Sigma)$ for the set that collects all the Σ-types. Consider a set $T \subseteq \mathsf{types}(\Sigma)$. A Σ-type τ is an *immediate consequence* of T and Σ if:

(1) $\tau \in T$, or
(2) there is a TGD $\sigma \in \Sigma$ and a Σ-type $\tau' \in T$ such that the following holds: there is a homomorphism h from $\mathsf{body}(\sigma)$ to $\mathsf{atoms}(\tau')$ and $h(\mathsf{guard}(\sigma)) = \mathsf{guard}(\tau')$ such that $[\tau]$ occurs in the head of the linearization of σ induced by τ and h.

In simple words, item (2) states that there exists a TGD in $\mathsf{linear}(\Sigma)$ that has one of the forms

$$[\tau'](x) \to [\tau](x) \qquad [\tau'](x,y) \to [\tau](x) \qquad [\tau'](x) \to \exists y [\tau_1](x,y), [\tau_2](y),$$

with $\tau_1 = \tau$ or $\tau_2 = \tau$, where $\tau' \in T$. The *immediate consequence operator* of Σ is the function $\Gamma_\Sigma : 2^{\mathsf{types}(\Sigma)} \to 2^{\mathsf{types}(\Sigma)}$ (recall that 2^X denotes the powerset of a set X) such that

$$\Gamma_\Sigma(T) = \{\tau \mid \tau \text{ is an immediate consequence of } T \text{ and } \Sigma\}.^4$$

By iterative applications of the above operator, we can compute the Σ-types that can be derived from T using the TGDs of Σ. Formally,

$$\Gamma^0_\Sigma(T) = T \qquad \text{and} \qquad \Gamma^i_\Sigma(T) = \Gamma_\Sigma(\Gamma^{i-1}_\Sigma(T)) \text{ for } i > 0$$

and we finally let

$$\Sigma(T) = \bigcup_{i \geq 0} \Gamma^i_\Sigma(T).$$

At first glance, the construction of $\Sigma(T)$ requires infinitely many iterations. However, since $\Sigma(T) \subseteq \mathsf{types}(\Sigma)$, in the worst-case $\Sigma(T)$ is obtained after $|\mathsf{types}(\Sigma)|$ iterations. It is actually easy to verify that $\Sigma(T) = \Gamma^{|\mathsf{types}(\Sigma)|}_\Sigma(T)$. Therefore, since $\mathsf{types}(\Sigma)$ is finite, we conclude that $\Sigma(T)$ can be obtained after finitely many steps. We can now formally define dynamic linearization.

Definition 10.3. Consider a database D and a set Σ of TGDs from TEL. The *dynamic linearization of* Σ *relative to* D (or *D-linearization of* Σ), denoted $\mathsf{linear}_D(\Sigma)$, is defined as the set

[4]Note that the notion of immediate consequence operator of Σ introduced in Section 4 and used in the definition of dynamic simplification is different than the notion of immediate consequence operator introduced here. We adopt the same name for those two different notions as they are completely independent and used in different contexts.

$$\big\{\sigma[\tau, h] \mid \sigma \in \Sigma,\ \tau \in \Sigma(\mathsf{types}_\Sigma(D)) \text{ and } h \text{ is a homomorphism from } \mathsf{body}(\sigma) \text{ to } \mathsf{atoms}(\tau) \text{ with } h(\mathsf{guard}(\sigma)) = \mathsf{guard}(\tau)\big\}$$

consisting only of simple-linear TGDs. ∎

It is not difficult to verify that the D-linearization of Σ essentially collects all the TGDs of $\mathsf{linear}(\Sigma)$ such that the predicate of their body-atom belongs to $\Sigma(\mathsf{types}_\Sigma(D))$. Consider, for instance, the set Σ of TGDs from Example 10.1 and the database $D = \{R(a, b), A(b)\}$. It is easy to see that

$$\Sigma(\mathsf{types}_\Sigma(D)) \;=\; \{\tau_1, \tau_2, \tau_3\},$$

where

$$\begin{aligned} \tau_1 &= (R(1,2), \{A(1), A(2), B(1)\}) \\ \tau_2 &= (A(1), \{B(1)\}) \\ \tau_3 &= (B(1), \{A(1)\}). \end{aligned}$$

Therefore, from the set $\mathsf{linear}(\Sigma)$, $\mathsf{linear}_D(\Sigma)$ keeps only the TGDs

$$[\tau_1](x, y) \to [\tau_3](x) \qquad [\tau_3](x) \to [\tau_2](x).$$

We proceed to show that indeed dynamic linearization preserves the finiteness of the chase. The proof is similar to that of Lemma 4.3, which is the analogous result for dynamic simplification.

Lemma 10.4. *Consider a database D and a set $\Sigma \in \mathsf{TEL}$ of TGDs. The following are equivalent:*

(1) $\mathsf{chase}(\mathsf{linear}_\Sigma(D), \mathsf{linear}(\Sigma))$ *is finite.*

(2) $\mathsf{chase}(\mathsf{linear}_\Sigma(D), \mathsf{linear}_D(\Sigma))$ *is finite.*

Proof. Since, by definition, $\mathsf{linear}_D(\Sigma) \subseteq \mathsf{linear}(\Sigma)$, it is clear that (1) implies (2). The interesting direction is (2) implies (1). We proceed to establish the contrapositive of the implication in question, that is, if the instance $\mathsf{chase}(\mathsf{linear}_\Sigma(D), \mathsf{linear}(\Sigma))$ is infinite, then the instance $\mathsf{chase}(\mathsf{linear}_\Sigma(D), \mathsf{linear}_D(\Sigma))$ is also infinite. To this end, we need an auxiliary technical lemma:

Lemma 10.5. *Consider a path $([\tau_1], i_1), \ldots, ([\tau_n], i_n)$ in the dependency graph of* $\mathsf{linear}(\Sigma)$ *such that there is an atom of the form $[\tau_1](\bar{c})$ in* $\mathsf{linear}_\Sigma(D)$*. It holds that $\{[\tau_1], \ldots, [\tau_n]\} \subseteq \Sigma(\mathsf{types}_\Sigma(D))$.*

Proof. We proceed by induction on the length n of the path.

Base Case. Clearly, $\mathsf{types}_\Sigma(D) \subseteq \Sigma(\mathsf{types}_\Sigma(D))$. Since $[\tau_1](\bar{c}) \in \mathsf{linear}_\Sigma(D)$, we get that $\tau_1 \in \mathsf{types}_\Sigma(D)$. Hence, $[\tau_1] \in \Sigma(\mathsf{types}_\Sigma(D))$, as needed.

Inductive Step. Consider a path $([\tau_1], i_1), \ldots, ([\tau_n], i_n)$ in the dependency graph of $\mathsf{linear}(\Sigma)$ with $[\tau_1]$ being a predicate in $\mathsf{linear}_\Sigma(D)$. By induction hypothesis, $\{([\tau_1], i_1), \ldots, ([\tau_{n-1}], i_{n-1})\} \subseteq \Sigma(\mathsf{types}_\Sigma(D))$. It remains to show that $[\tau_n] \in \Sigma(\mathsf{types}_\Sigma(D))$. Assume that σ is the TGD witnessing the edge $(([\tau_{n-1}], i_{n-1}), ([\tau_n], i_n))$ in the dependency graph of $\mathsf{linear}(\Sigma)$. There is $i \geq 0$ such that $[\tau_{n-1}] \in \Gamma^i_\Sigma(\mathsf{types}_\Sigma(D))$. Since $[\tau_{n-1}]$ is the predicate of the body-atom of σ, $[\tau_n] \in \Gamma^{i+1}_\Sigma(\mathsf{types}_\Sigma(D))$, and therefore, $[\tau_n] \in \Sigma(\mathsf{types}_\Sigma(D))$, as needed. □

We can now show the desired implication. Since, by hypothesis, $\mathsf{chase}(\mathsf{linear}_\Sigma(D), \mathsf{linear}(\Sigma))$ is infinite, Theorem 3.3 allows us to conclude that there exists a $\mathsf{linear}_\Sigma(D)$-supported cycle with a special edge in the dependency graph of $\mathsf{linear}(\Sigma)$. Let $([\tau_1], i_1), \ldots, ([\tau_n], i_n)$, with $([\tau_1], i_1) = ([\tau_n], i_n)$, be such a cycle. Since this cycle is $\mathsf{linear}_\Sigma(D)$-supported, there is an atom $[\tau](\bar{c}) \in \mathsf{linear}_\Sigma(D)$ and a node $([\tau_j], i_j)$ in the cycle such that $[\tau_j]$ is reachable from $[\tau]$. Since the TGDs of Σ have a non-empty frontier, in the dependency graph of $\mathsf{linear}(\Sigma)$ there exists a path $([\tau'_1], k_1), \ldots, ([\tau'_m], k_m)$

ALGORITHM 4: DynLinearization

Input: A database D and a set $\Sigma \in \mathsf{TEL}$ of TGDs
Output: The D-linearization of Σ

1 $T \leftarrow \mathsf{FindTypes}(D, \Sigma)$;
2 $\Sigma_\ell \leftarrow \emptyset$;
3 $\Delta T \leftarrow T$;
4 **while** $\Delta T \neq \emptyset$ **do**
5 $\quad \Sigma_{aux} \leftarrow \mathsf{Linearize}(\Delta T, \Sigma)$;
6 $\quad T_{aux} \leftarrow \{\tau \in \mathsf{types}(\Sigma) \mid$ there is $\sigma \in \Sigma_{aux}$ such that $[\tau]$ occurs in $\mathsf{head}(\sigma)\}$;
7 $\quad \Sigma_\ell \leftarrow \Sigma_\ell \cup \Sigma_{aux}$;
8 $\quad \Delta T \leftarrow T_{aux} \setminus T$;
9 $\quad T \leftarrow T \cup \Delta T$;
10 **return** Σ_ℓ

with $[\tau'_1] = [\tau]$ and $([\tau'_m], k_m) = ([\tau_j], i_j)$. Summing up, in the dependency graph of $\mathsf{linear}(\Sigma)$, there exists a path of the form

$$([\tau'_1], k_1), \ldots, ([\tau'_{m-1}], k_{m-1}), ([\tau_j], i_j), \ldots, ([\tau_{n-1}], i_{n-1}), ([\tau_1], i_1), \ldots, ([\tau_{j-1}], i_{j-1}), ([\tau_j], i_j).$$

Assume that this path is witnessed by the TGDs $\sigma_1, \ldots, \sigma_{n+m-2}$. By Lemma 10.5,

$$\{[\tau'_1], \ldots, [\tau'_{m-1}], [\tau_1], \ldots, [\tau_{n-1}]\} \subseteq \Sigma(\mathsf{types}_\Sigma(D)).$$

By the definition of dynamic linearization (see Definition 10.3), $\sigma_1, \ldots, \sigma_{n+m-2}$ belong to $\mathsf{linear}_D(\Sigma)$. Therefore, the above $\mathsf{linear}_\Sigma(D)$-supported cycle with a special edge also occurs in the dependency graph of $\mathsf{linear}_D(\Sigma)$. Hence, by Theorem 3.3, $\mathsf{chase}(\mathsf{linear}_\Sigma(D), \mathsf{linear}_D(\Sigma))$ is infinite. □

Similarly to dynamic simplification, another useful property of dynamic linearization, which will help us to further improve the performance of the termination algorithm for TGDs from TEL, is that the $\mathsf{linear}_\Sigma(D)$-weak-acyclicity of $\mathsf{linear}_D(\Sigma)$ coincides with the weak-acyclicity of $\mathsf{linear}_D(\Sigma)$. This holds since, by construction, all the predicates occurring in the TGDs of $\mathsf{linear}_D(\Sigma)$ are reachable from a predicate occurring in $\mathsf{linear}_\Sigma(D)$, and thus, each cycle with a special edge in the dependency graph of $\mathsf{linear}_D(\Sigma)$ is trivially $\mathsf{linear}_\Sigma(D)$-supported. The above property immediately implies Lemma 10.6 below since checking for the finiteness of $\mathsf{chase}(\mathsf{linear}_\Sigma(D), \mathsf{linear}_D(\Sigma))$, which is equivalent to the $\mathsf{linear}_\Sigma(D)$-weak-acyclicity of $\mathsf{linear}_D(\Sigma)$ by Theorem 10.2, boils down to checking if $\mathsf{linear}_D(\Sigma)$ is weakly-acyclic, without having to explicitly check for $\mathsf{linear}_\Sigma(D)$-supportedness.

Lemma 10.6. *Consider a database D and a set $\Sigma \in \mathsf{TEL}$ of TGDs. The following are equivalent:*
(1) $\mathsf{chase}(\mathsf{linear}_\Sigma(D), \mathsf{linear}_D(\Sigma))$ *is finite.*
(2) $\mathsf{linear}_D(\Sigma)$ *is weakly-acyclic.*

An Algorithm for Dynamic Linearization. We provide a concrete algorithm that performs the dynamic linearization of a set of TGD from TEL that is amenable to an efficient implementation. To this end, we proceed to present the algorithm DynLinearization, depicted in Algorithm 4, that takes as input a database D and a set $\Sigma \in \mathsf{TEL}$ of TGDs, and constructs the D-linearization of Σ. The algorithm starts by finding the Σ-types that occur in the linearization of D relative to Σ, namely, it computes the set $\mathsf{types}_\Sigma(D)$ (line 1). It then initializes the set of linearized TGDs Σ_ℓ (line 2) and the set of new Σ-types ΔT (line 3). Then, the algorithm iteratively generates linearized TGDs and collects the new Σ-types that are added to ΔT, and continues this until a fixpoint is reached, i.e.,

ALGORITHM 5: IsChaseFinite[TEL]

Input: A database D and a set $\Sigma \in \mathsf{TEL}$ of TGDs
Output: `true` if $\mathsf{chase}(D, \Sigma)$ is finite and `false` otherwise

1 $\Sigma_\ell \leftarrow \mathsf{DynLinearization}(D, \Sigma)$;
2 $G \leftarrow \mathsf{BuildDepGraph}(\Sigma_\ell)$;
3 **if** $\mathsf{FindSpecialSCC}(G) \neq \emptyset$ **then return** `false`;
4 **return** `true`

$\Delta T = \emptyset$ (line 4). In particular, at each iteration, the algorithm computes linearized TGDs that are not superfluous, i.e., they can be applied during the construction of $\mathsf{chase}(\mathsf{linear}_\Sigma(D), \mathsf{linear}(\Sigma))$, that are added to Σ_ℓ (lines 5 and 7). This is done via the procedure Linearize, which takes as input a set of Σ-types $\hat{T}$, together with the set of TGDs Σ, and returns the set

$$\big\{\sigma[\tau, h] \mid \sigma \in \Sigma,\ \tau \in \hat{T} \text{ and } h \text{ is a homomorphism from } \mathsf{body}(\sigma) \text{ to } \mathsf{atoms}(\tau) \text{ with } h(\mathsf{guard}(\sigma)) = \mathsf{guard}(\tau)\big\}.$$

Essentially, the procedure Linearize computes the set of TGDs of $\mathsf{linear}(\Sigma)$ such that the predicate of their body-atom belongs to $\hat{T}$. The algorithm also collects the newly generated Σ-types, that is, the Σ-types τ such that the predicate $[\tau]$ occurs in the head of a TGD of $\mathsf{Linearize}(\Delta T, \Sigma)$, that are added to ΔT (lines 6 and 8). Note that at each iteration, the algorithm considers only the Σ-types of ΔT, not of T, during the linearization, with the exception of the first iteration where $T = \Delta T$. The implementation details of FindTypes and Linearize are discussed in the next subsection.

Example 10.7. Consider the set Σ of TGDs from Example 10.1 and the database $D = \{R(a, b), A(b)\}$. We describe how dynamic linearization operates on D and Σ. The sets T and ΔT are initialized to the set of Σ-types of D, i.e., $T = \Delta T = \{\tau_1, \tau_2\}$ with $\tau_1 = (R(1, 2), \{A(1), A(2), B(1)\})$ and $\tau_2 = (A(1), \{B(1)\})$, and $\Sigma_\ell = \emptyset$. At the first iteration of the algorithm, we have that the body of σ_1 is "coherent" with the Σ-type τ_1 present in ΔT. No other TGDs of Σ have a body which is coherent with a Σ-type in ΔT, and thus, we conclude that the TGD $[\tau_1](x, y) \rightarrow [\tau_3](x)$ with $\tau_3 = (B(1), \{A(1)\})$, which is constructed in line 5 and added to Σ_ℓ in line 7, will be actually triggered by the chase. We then have $T = \{\tau_1, \tau_2, \tau_3\}$, while the new Σ-types are $\Delta T = \{\tau_3\}$. At the second iteration, we have that the body of σ_2 is coherent with the (only) Σ-type τ_3 in ΔT, and thus, the TGD $[\tau_3](x) \rightarrow [\tau_2](x)$ is added to Σ_ℓ. Since no new Σ-types have been produced, the algorithm terminates with Σ_ℓ collecting a subset of $\mathsf{linear}(\Sigma)$ that is sufficient to construct $\mathsf{chase}(\mathsf{linear}_\Sigma(D), \mathsf{linear}(\Sigma))$. ∎

It is easy to see that DynLinearization is correct by construction:

Lemma 10.8. *For a database D and a set $\Sigma \in \mathsf{TEL}$ of TGDs,* $\mathsf{DynLinearization}(D, \Sigma) = \mathsf{linear}_D(\Sigma)$.

Termination Algorithm. Having in place DynLinearization, it is now easy to devise the algorithm IsChaseFinite[TEL], depicted in Algorithm 5, that checks for the finiteness of the chase instance in the case of sets of TGDs from TEL. The correctness of DynLinearization, Theorem 10.2, Lemma 10.4, and Lemma 10.6, imply the correctness of the algorithm IsChaseFinite[TEL]:

Lemma 10.9. *Consider a database D and a set $\Sigma \in \mathsf{TEL}$ of TGDs. The following are equivalent:*

(1) $\mathsf{IsChaseFinite[TEL]}(D, \Sigma) = \mathtt{true}$.
(2) $\mathsf{chase}(D, \Sigma)$ *is finite.*

10.4 Implementation Details

We proceed to discuss the implementation details of the algorithm IsChaseFinite[TEL] presented above. We actually discuss the non-trivial implementation choices that help to improve the performance of the algorithm, but are missing from the descriptions given in the previous subsection. The details for the procedures BuildDepGraph and FindSpecialSCC have been already discussed in Sections 5.1 and 5.2, respectively. Thus, our main task here is to discuss the details of DynLinearization. Recall that DynLinearization takes as input a database D and a set Σ of TGDs from TEL, and returns the set $\mathsf{linear}_D(\Sigma)$, i.e., the D-linearization of Σ. It is an iterative procedure that uses two sub-procedures: FindTypes that computes the set T of Σ-types that form the predicates occurring in $\mathsf{linear}_\Sigma(D)$, i.e., the linearization of D relative to Σ, and Linearize that linearizes TGDs using the Σ-types of T. We proceed to discuss how those two sub-procedures have been implemented.

Before delving into the implementation details, let us first stress that both sub-procedures need to perform *instance checking*, which is a standard reasoning task defined as follows: given a database D, a set Σ of TGDs from TEL, and a fact $C(t)$, where C is a unary predicate of $\mathsf{sch}(\Sigma)$, decide whether $C(t) \in \mathsf{chase}(D, \Sigma)$. In other words, using DL terminology, we are asking whether the value t can be derived as an instance of the basic concept C from D and Σ. FindTypes relies on instance checking in order to compute the type of an atom w.r.t. D and Σ, whereas Linearize needs instance checking to compute the completion of a database w.r.t. Σ. For performing instance checking for sets of TGDs from TEL, we rely on a well-known reasoner for EL-ontologies, called ELK, which is one of the most competitive reasoning systems for EL-ontologies available today [Kazakov et al. 2014]. Note that ELK is suitable for our purposes since, by the definition of the class TEL, a set of TGDs from TEL corresponds to an EL-ontology in normal form, i.e., whenever we need to perform instance checking in the presence of a set Σ of TGDs from TEL, we simply need to convert Σ into an EL-ontology and then use ELK. We are now ready to discuss the two sub-procedures in question.

The Procedure FindTypes**.** Recall that this procedure takes as input a database D and a set $\Sigma \in \mathsf{TEL}$ of TGDs, and computes the set of Σ-types $\mathsf{types}_\Sigma(D)$. To this end, it needs to compute the database $\mathsf{linear}_D(\Sigma)$, and then simply collect the Σ-types τ such that $[\tau]$ is a predicate occurring in $\mathsf{linear}_D(\Sigma)$. To compute $\mathsf{linear}_D(\Sigma)$, the procedure iterates over every atom $\alpha = R(\bar{t}) \in D$ and computes its type w.r.t. D and Σ, i.e., $\mathsf{type}_{D,\Sigma}(\alpha)$. The latter is done by iterating over every atom of the form $C(t)$, where C is a unary predicate from $\mathsf{sch}(\Sigma)$ and t a term from $\bar{t}$, and using ELK to check whether t can be derived as an instance of C from D and Σ, i.e., whether $C(t) \in \mathsf{chase}(D, \Sigma)$, in which case $C(t)$ belongs to $\mathsf{type}_{D,\Sigma}(\alpha)$. The procedure then extracts from $\mathsf{type}_{D,\Sigma}(\alpha)$ the Σ-type τ such that $\tau(\bar{t}) = \mathsf{type}_{D,\Sigma}(\alpha)$, and constructs the atom $[\tau](\bar{t})$ that is added to the database D_ℓ, which was initialized at the beginning to empty. At the end of the computation, $D_\ell = \mathsf{linear}_\Sigma(D)$.

To optimize the computation of $\mathsf{type}_{D,\Sigma}(\alpha)$, which is a costly task as it relies on the ELK reasoner, we exploit the fact that, for every two atoms $\beta, \gamma \in \mathsf{chase}(D, \Sigma)$ with $\mathsf{dom}(\beta) = \mathsf{dom}(\gamma)$, it holds that $\mathsf{type}_{D,\Sigma}(\beta) = \mathsf{type}_{D,\Sigma}(\gamma)$; this immediately follows by the definition of the type of an atom w.r.t. D and Σ. Therefore, we can optimize the procedure by computing $\mathsf{type}_{D,\Sigma}(\alpha)$ once and reusing it for any other atom β such that $\mathsf{dom}(\alpha) = \mathsf{dom}(\beta)$. To this end, we build an index structure with a record for each tuple $\bar{t}$ that appears in an atom of D, where the key is the tuple $\bar{t}$ and the value is the set of atoms of $\mathsf{chase}(D, \Sigma)$ with a unary predicate that mentions a term from $\bar{t}$. This significantly improves the performance of the procedure, and it is applied by default in all of our experiments.

The Procedure Linearize**.** Recall that Linearize takes as input a set of Σ-types T, together with the set Σ of TGDs from TEL, and returns the set of TGDs of $\mathsf{linear}(\Sigma)$ such that the predicate of their body belongs to T. As discussed above, this is done by iterating over all TGDs $\sigma \in \Sigma$, Σ-types $\tau \in T$,

and homomorphisms h from $\text{body}(\sigma)$ to $\text{atoms}(\tau)$ with $h(\text{guard}(\sigma)) = \text{guard}(\tau)$, and collecting the linearization of σ induced by τ and h. Applying the above iterative process with a large set of Σ-types can be very costly. To improve the performance of this process, the implementation uses an index structure that enables fast access to the Σ-types. The index structure maps each predicate $R \in \text{sch}(\Sigma)$ to the set of Σ-types τ of T such that the predicate of $\text{guard}(\tau)$ is R. This allows the procedure to iterate over the TGDs $\sigma \in \Sigma$ and quickly access the relavant Σ-types τ in the index, which in turn allows to easily find the relevant homomorphisms h from $\text{body}(\sigma)$ to $\text{atoms}(\tau)$.

It should not be forgotten that for computing the linearization of σ induced by τ and h we need to compute the set of atoms $\text{complete}(S \cup \text{atoms}(\tau), \Sigma)$, where S is of the form $\{B(1)\}$ or $\{R(1,2), B(2)\}$, where $B/1, R/2 \in \text{sch}(\Sigma)$. The form of S depends on the form of the TGD σ; see the definition of linearization above. This is done by iterating over every atom of the form $C(t)$, where C is a unary predicate from $\text{sch}(\Sigma)$ and $t \in \{1, 2\}$, and using ELK to check whether t can be derived as an instance of C from the database $S \cup \text{atoms}(\tau)$ and Σ, i.e., whether $C(t) \in \text{chase}(S \cup \text{atoms}(\tau), \Sigma)$, in which case $C(t)$ belongs to $\text{complete}(S \cup \text{atoms}(\tau), \Sigma)$. Note that at each call of ELK only the database changes, whereas the set of TGDs Σ remains the same. Therefore, the translation of Σ into the syntax that ELK expects is computed only once.

With the implementation details of IsChaseFinite[TEL] clarified, let us emphasize that our implementation intertwines dynamic linearization, dependency graph construction, and SCC detection to detect the non-terminating cases early and stop the algorithm sooner. The dependency graph is built incrementally as TGDs are linearized, while special SCCs are checked consistently. This approach is similar to our implementation of IsChaseFinite[L] presented in Algorithm 4.

10.5 Experimental Evaluation

To evaluate the performance of IsChaseFinite[TEL], we use real-world ontologies provided by the ontology repository of the Knowledge Representation and Reasoning group of the University of Oxford.[5] The repository contains 787 ontologies, of which 70 are EL-ontologies in normal form, which can be translated into sets of TGDs from TEL. However, most of those ontologies do not come with a suitable ABox (the DL terminology for the notion of database) for effectively testing our algorithm. Therefore, in our experiments, for each set Σ of TGDs obtained from one of the 70 EL-ontologies in question, we use the simple database $D_\Sigma = \{A(c) \mid A/1 \in \text{sch}(\Sigma)\} \cup \{R(c,c) \mid R/2 \in \text{sch}(\Sigma)\}$, where c is an arbitrarily chosen constant of $\mathbf{C}$. Note that, D_Σ is known in the literature as the *critical database* of $\text{sch}(\Sigma)$ and it acts as the worst-case scenario for chase termination purposes, i.e., if there is a database D such that $\text{chase}(D, \Sigma)$ is infinite, then already $\text{chase}(D_\Sigma, \Sigma)$ is infinite [Marnette 2009]. Out of the 70 ontologies, we report results for 59 ontologies for which we could successfully run the algorithm IsChaseFinite[TEL]; the others failed due to memory exhaustion. We observe that, even with the critical database, which is structurally very simple, in most of the cases the obtained results are far from being promising, which is a strong indication that using the linearization technique in the context of the semi-oblivious chase termination problem is rather unrealistic. This is the reason why we did not run experiments with more realistic databases as the critical database was enough to arrive at our negative conclusion concerning the use of the linearization technique for attacking the semi-oblivious chase termination problem in practice.

The results of our experiments are presented in the appendix, where we report for each ontology the time spent on linearization (`t-linearize`) and the total runtime of the algorithm (`t-total`). In Tables 3 and 4, we summarize the running times by grouping the ontologies into profiles based on the number of TGDs and the number of predicates in order to highlight the trend of the running times with respect to those two key parameters.

[5] https://www.cs.ox.ac.uk/isg/ontologies/

Profile	t-linearize	t-total
0–2.5K	42.62 sec	43.96 sec
2.5K–5K	18.22 min	18.31 min
5K–7.5K	43.47 min	43.65 min
7.5K–10K	5.17 hr	5.18 hr
10K–12.5K	6.49 hr	6.51 hr

Table 3. Runtime vs. Number of TGDs

Profile	t-linearize	t-total
0–1.5K	49.45 sec	50.83 sec
1.5K–3K	20.86 min	20.96 min
3K–4.5K	58.21 min	58.45 min
4.5K–6K	4.69 hr	4.7 hr
6K–7.5K	15.17 hr	15.19 hr

Table 4. Runtime vs. Number of Predicates

Take-home Messages. Our main observation is that, in general, the total runtime is dominated by the time taken to linearize the set of TGDs, making it the most computationally expensive component of the algorithm. Additionally, we observe that the total runtime significantly increases as the number of TGDs (resp., number of predicates) grows, which in turn shows that the algorithm faces serious challenges in scaling to thousands of TGDs (resp., predicates). Summing up, our final conclusion is that the use of the linearization technique in the context of the semi-oblivious chase termination problem is rather unrealistic, unless we focus on very small sets of TGDs from TEL with a few hundreds TGDs and predicates. Therefore, devising a practical algorithm for the semi-oblivious chase termination problem in the case of sets of TGDs from TEL (or, generally, guarded TGDs) remains a major open problem. Towards this direction, one has to either completely avoid the computationally expensive task of linearizing the TGDs or devise more efficient reasoning engines that will significantly speed up the process of linearization.

11 Conclusions and Future Work

Our work provides the first systematic attempt to experimentally evaluate algorithms devised for the semi-oblivious chase termination problem in the presence of (simple-)linear TGDs. Our analysis revealed that for simple-linear TGDs, we can efficiently check whether the chase terminates even for very large databases and sets of TGDs. Concerning linear TGDs, the overall runtime of the algorithm is quite reasonable, but there is still room for improvement. Interestingly, our analysis showed that the algorithm for linear TGDs consists of two separate components, the db-dependent and the db-independent components. This modular nature of the algorithm allows us to study and improve the two components separately. In particular, we have observed that the heavy component is the db-dependent one, and thus, we can focus our future efforts to improve the performance of that component. Although our analysis relied on an in-database and an in-memory implementation of the procedure for finding the shapes, we could adopt other techniques depending on the underlying application without affecting the db-independent component. An interesting direction is to materialize and incrementally keep updated the shapes in a database, which will improve the performance of the db-dependent component.

Finally, we performed an experimental analysis with sets of TGDs from a lightweight fragment of guarded TGDs, called TEL, which corresponds to the DL EL. The objective was to understand how realistic is the use of the linearization technique, that is, the technique of converting guarded TGDs into simple-linear TGDs, in the context of the semi-oblivious chase termination problem. We observed that, even with sets of TGDs from TEL, we cannot go far in practice by relying on the linearization technique, unless we focus on very small sets of TGDs with a few hundreds of TGDs and predicates. This is due to the fact that the linearization technique relies on the expensive task of reasoning with guarded TGDs, which in our analysis is performed using ELK, one of the most competitive reasoning systems for EL-ontologies available today [Kazakov et al. 2014]. To the best of our knowledge, the only alternative approach would be to exploit known Datalog rewritings for guarded TGDs, which in turn will allow us to use existing Datalog engines, in order to perform the necessary reasoning tasks. Having said that, it is unlikely that reasoning with guarded TGDs using

Datalog rewritings, which have been designed for arbitrary guarded TGDs, would perform better for our purposes than the reasoner ELK.

References

Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast algorithms for mining association rules in large databases. In *VLDB*. 487–499.

Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. 1979. Efficient optimization of a class of relational expressions. *ACM Trans. Database Syst.* 4, 4 (1979), 435–454.

Bogdan Alexe, Wang Chiew Tan, and Yannis Velegrakis. 2008. STBenchmark: Towards a benchmark for mapping systems. *PVLDB* 1, 1 (2008), 230–244.

Patricia C. Arocena, Boris Glavic, Radu Ciucanu, and Renée J. Miller. 2015. The iBench integration metadata generator. *PVLDB* 9, 3 (2015), 108–119.

Franz Baader, Sebastian Brandt, and Carsten Lutz. 2005. Pushing the EL envelope. In *IJCAI*. 364–369.

Franz Baader, Ian Horrocks, Carsten Lutz, and Ulrike Sattler. 2017. *An Introduction to Description Logic*. Cambridge University Press.

Denilson Barbosa, Alberto Mendelzon, John Keenleyside, and Kelly Lyons. 2002. ToXgene: a template-based data generator for XML. In *SIGMOD*. 616–616.

Catriel Beeri and Moshe Y. Vardi. 1984. A proof procedure for data dependencies. *J. ACM* 31, 4 (1984), 718–741.

Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. 2017. Benchmarking the chase. In *PODS*. 37–52.

Marco Calautti, Georg Gottlob, and Andreas Pieris. 2015. Chase termination for guarded existential rules. In *PODS*. 91–103.

Marco Calautti, Georg Gottlob, and Andreas Pieris. 2022. Non-uniformly terminating chase: Size and complexity. In *PODS*. 369–378.

Marco Calautti and Andreas Pieris. 2021. Semi-oblivious chase termination: The sticky case. *Theory Comput. Syst.* 65, 1 (2021), 84–121.

Andrea Calì, Georg Gottlob, and Michael Kifer. 2013. Taming the infinite chase: Query answering under expressive relational constraints. *J. Artif. Intell. Res.* 48 (2013), 115–174.

Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. 2012. A general datalog-based framework for tractable query answering over ontologies. *J. Web Sem.* 14 (2012), 57–83.

Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. 2007. Tractable reasoning and efficient query answering in description logics: The dl-lite family. *J. Autom. Reasoning* 39, 3 (2007), 385–429.

Alin Deutsch, Alan Nash, and Jeff B. Remmel. 2008. The chase revisisted. In *PODS*. 149–158.

Edsger W Dijkstra. 1982. Finding the maximum strong components in a directed graph. In *Selected Writings on Computing: A Personal Perspective*. 22–30.

Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2005. Data exchange: Semantics and query answering. *Theor. Comput. Sci.* 336, 1 (2005), 89–124.

Tomasz Gogacz, Jerzy Marcinkowski, and Andreas Pieris. 2020. All-instances restricted chase termination for linear TGDs. *Künstliche Intell.* 34, 4 (2020), 465–473.

Tomasz Gogacz, Jerzy Marcinkowski, and Andreas Pieris. 2023. Uniform restricted chase termination. *SIAM J. Comput.* 52, 3 (2023), 641–683.

Georg Gottlob, Marco Manna, and Andreas Pieris. 2014. Polynomial combined rewritings for existential rules. In *KR*.

Bernardo Cuenca Grau, Ian Horrocks, Markus Krötzsch, Clemens Kupke, Despoina Magka, Boris Motik, and Zhe Wang. 2013. Acyclicity notions for existential rules and their application to query answering in ontologies. *J. Artif. Intell. Res.* 47 (2013), 741–808.

Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *J. Web Semant.* 3, 2-3 (2005), 158–182.

Yevgeny Kazakov, Markus Krötzsch, and Frantisek Simancik. 2014. The incredible ELK - from polynomial procedures to efficient reasoning with $\mathcal{EL}$ ontologies. *J. Autom. Reason.* 53, 1 (2014), 1–61.

Markus Krötzsch, Maximilian Marx, and Sebastian Rudolph. 2019. The power of the terminating chase (invited talk). In *ICDT*. 3:1–3:17.

Michel Leclère, Marie-Laure, Michaël Thomazo, and Federico Ulliana. 2019. A single approach to decide chase termination on linear existential rules. In *ICDT*. 18:1–18:19.

David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. 1979. Testing implications of data dependencies. *ACM Trans. Database Syst.* 4, 4 (1979), 455–469.

Bruno Marnette. 2009. Generalized schema-mappings: from termination to tractability. In *PODS*. 13–22.

Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. 2015. RDFox: A highly-scalable RDF store. In *ISWC*. 3–20.

Micha Sharir. 1981. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications* 7, 1 (1981), 67–72.

Robert Endre Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

Jacopo Urbani, Markus Krötzsch, Ceriel J. H. Jacobs, Irina Dragoste, and David Carral. 2018. Efficient model construction for horn logic with vlog—system description. In *IJCAR*. 680–688.

A Experimental Evaluation of Section 10

The results of our experiments are presented in Tables 5 and 6, where we report for each ontology the time spent on linearization (`t-linearize`) and the total runtime of the algorithm (`t-total`). For each ontology, we further report the total number of TGDs obtained after the translation into a set of TGDs from TEL (`n-rules`) and the total number of predicates in the set of TGDs (`n-pred`), as well as the total number of TGDs produced after linearization (`n-rules-l`) and the total number of predicates occurring in the linearized set of TGDs (`n-pred-l`); the plot in Figure 10 illustrates that the number of TGDs after linearization grows linearly.

For each of the 41 ontologies presented in Table 5, we were able to run IsChaseFinite[TEL] five times and present the average running times. On the other hand, for the ontologies presented in Table 6, due to the high runtimes (from 16 minutes to 4 hours), we chose to run the algorithm once.

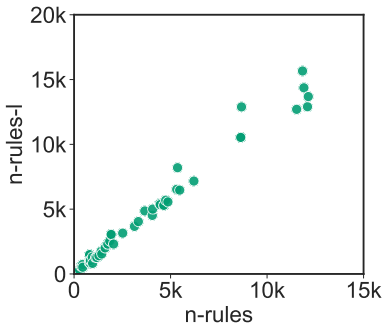


Fig. 10. The number of linearized TGDs (`n-rules-l`) vs. the number of TGDs before linearization (`n-rules`).

Table 5. Runtime of IsChaseFinite[TEL] and Further Statistics Before and After Linearization

Ontology	n-rules	n-pred	n-rules-l	n-pred-l	t-linearize	t-total
Ont-577	46	19	104	63	0.58 sec	1.2 sec
Ont-542	72	39	90	48	0.72 sec	1.32 sec
Ont-543	72	39	90	48	0.7 sec	1.29 sec
Ont-474	104	69	167	123	1 sec	1.68 sec
Ont-505	169	163	166	164	1.28 sec	1.95 sec
Ont-414	200	113	248	137	1.81 sec	2.46 sec
Ont-648	240	237	254	252	2.37 sec	3.24 sec
Ont-580	288	282	294	293	3.44 sec	4.46 sec
Ont-514	381	312	595	501	5.97 sec	6.88 sec
Ont-513	381	312	595	501	5.85 sec	6.72 sec
Ont-358	445	262	738	583	6.21 sec	7.12 sec
Ont-683	452	207	518	238	5.33 sec	6.06 sec
Ont-389	800	376	1,000	446	19.03 sec	19.92 sec
Ont-681	802	341	878	378	11 sec	11.84 sec
Ont-606	804	559	1,489	1,097	24.99 sec	26.41 sec
Ont-605	804	559	1,489	1,097	25.34 sec	26.77 sec
Ont-562	815	644	888	751	25.77 sec	27.15 sec
Ont-538	816	377	1,020	445	18.05 sec	18.93 sec
Ont-454	830	410	836	415	14.34 sec	15.48 sec
Ont-539	832	383	1,038	451	18.5 sec	19.4 sec
Ont-388	848	404	1,042	464	21.53 sec	22.44 sec
Ont-639	947	452	1,296	815	28.31 sec	29.53 sec
Ont-527	962	481	818	637	28.93 sec	30.11 sec
Ont-563	1,116	835	1,276	1,029	50.59 sec	52.44 sec
Ont-570	1,178	974	1,271	1,077	74.41 sec	76.63 sec
Ont-687	1,288	531	1,414	590	30.94 sec	32.02 sec
Ont-417	1,402	715	1,754	894	58.57 sec	59.9 sec
Ont-571	1,435	1,212	1,540	1,331	2.13 min	2.18 min
Ont-418	1,610	818	2,012	1,020	1.34 min	1.37 min
Ont-458	1,747	1,169	2,304	1,605	2.44 min	2.49 min
Ont-457	1,747	1,169	2,304	1,605	2.47 min	2.51 min
Ont-555	1,850	1,102	2,547	1,509	2.5 min	2.54 min
Ont-423	1,921	871	3,048	1,845	2.01 min	2.05 min
Ont-578	1,921	871	3,048	1,845	2.02 min	2.06 min
Ont-679	2,048	944	2,310	1,064	1.86 min	1.89 min
Ont-534	2,520	1,266	3,150	1,582	4.81 min	4.85 min
Ont-685	3,130	1,498	3,672	1,763	7.37 min	7.43 min
Ont-401	3,330	1,553	4,041	1,786	7.58 min	7.64 min
Ont-360	3,664	1,636	4,862	3,163	13.63 min	13.72 min
Ont-574	4,054	1,727	4,520	1,948	10.23 min	10.31 min
Ont-399	4,074	1,804	5,006	2,037	12.66 min	12.74 min

Table 6. Runtime of IsChaseFinite[TEL] and Further Statistics Before and After Linearization

Ontology	n-rules	n-pred	n-rules-l	n-pred-l	t-linearize	t-total
Ont-372	4,432	1,961	5,422	2,202	16.14 min	16.22 min
Ont-377	4,466	2,074	5,348	2,375	16.66 min	16.76 min
Ont-495	4,648	2,316	5,297	3,805	43.33 min	43.49 min
Ont-494	4,648	2,316	5,297	3,805	43.12 min	43.28 min
Ont-396	4,748	2,157	5,692	2,432	21.27 min	21.37 min
Ont-392	4,862	2,380	5,560	2,693	21.78 min	21.9 min
Ont-373	5,318	2,353	6,528	2,647	27.83 min	27.95 min
Ont-402	5,362	3,145	8,198	5,341	70.3 min	70.58 min
Ont-376	5,474	2,529	6,468	2,846	29.61 min	29.74 min
Ont-393	6,208	3,048	7,168	3,490	46.13 min	46.32 min
Ont-522	8,635	4,362	10,537	7,539	4.86 hr	4.87 hr
Ont-523	8,635	4,362	10,537	7,539	5.02 hr	5.03 hr
Ont-403	8,676	5,354	12,889	8,607	5.63 hr	5.64 hr
Ont-677	11,533	4,811	12,703	5,339	3.2 hr	3.2 hr
Ont-669	11,837	6,733	15,662	10,727	15.17 hr	15.19 hr
Ont-397	11,908	5,334	14,363	6,056	5.21 hr	5.22 hr
Ont-438	12,096	5,313	12,904	5,717	4.65 hr	4.66 hr
Ont-500	12,136	5,249	13,686	5,820	4.25 hr	4.26 hr

Received 27 November 2024; revised 6 September 2025; accepted 8 December 2025