Semi-Oblivious Chase Termination for Linear Existential Rules: An Experimental Study – EA&B

Marco Calautti
University of Milan
marco.calautti@unimi.it

Mostafa Milani
University of Western Ontario
mostafa.milani@uwo.ca

Andreas Pieris
University of Edinburgh &
University of Cyprus
apieris@inf.ed.ac.uk

ABSTRACT

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 as input a database and a set of constraints, and iteratively completes the database as dictated by the constraints. A key challenge, though, is the fact that it 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, and precise complexity results. 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.

PVLDB Reference Format:
Marco Calautti, Mostafa Milani, and Andreas Pieris. Semi-Oblivious Chase Termination for Linear Existential Rules: An Experimental Study – EA&B. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

The *chase procedure* (or simply chase) is a fundamental algorithmic tool that has been successfully applied to several database problems such as checking logical implication of constraints [5, 16], containment of queries under constraints [3], computing data exchange solutions [12], and ontological query answering [9], to name a few. The chase 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)) 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. Somehow 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 [11]. Indeed, many database problems can be solved by simply exhibiting a universal model.

Roughly speaking, the chase adds new tuples to the database D (possibly with 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 ways for formalizing this simple idea, which lead to different versions of the chase: *oblivious*, *semi-oblivious*, and *restricted*. The key difference between the various versions of the chase is when a TGD is considered applicable. In a nutshell, the (semi-)oblivious versions apply a TGD whenever the body is satisfied, while the restricted version applies a TGD if the body is satisfied but the head is not. For a more detailed discussion about the various versions of the chase procedure and their differences, see, e.g., [8].

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 recently observed that for RAM-based implementations the restricted chase is the indicated approach since the benefit from producing smaller instances justifies the additional effort for checking whether a TGD is already satisfied; see, e.g., [6, 14]. However, as discussed in [6], 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.

Chase Termination and Linear TGDs. There are indeed efficient implementations of the semi-oblivious and restricted chase that allow us to solve important database problems by adopting a materialization-based approach [6, 14, 18, 20]. Nevertheless, for this to be feasible in practice we need a guarantee that the chase terminates, which is not always the case. 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 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 [11] for the restricted chase, and it was observed a year later in [17] that the same proof shows undecidability also for the semi-oblivious chase. The undecidability proof given in [11],

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

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 attracted a considerable attention due to its simplicity, and also the fact that it strikes a good balance between expressiveness and complexity, is that of linear TGDs proposed in [9]. 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 to express database integrity constraints, as well as ontological axioms. In particular, we know that referential integrity constraints (a.k.a. inclusion dependencies) that form a central class of constraints [1], can be easily expressed as simple-linear TGDs. Moreover, the important ontology language DL-Lite$_R$ [10], which is based on Description Logics 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 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, and precise complexity results [7]. On the other hand, for the restricted version of the chase, we only have a decidability result via an algorithm that runs in double-exponential time, under the assumption that the head of the linear TGDs consists of a single atom [15]. 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.

Main Objective. 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 existing 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 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.

From the chase termination literature, we can inherit what we call *acyclicity-based algorithms* for the semi-oblivious chase termination problem in the presence of (simple-)linear TGDs [7], which are the main subjects of our experimental evaluation. These 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, we know that 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, 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 provides conceptually simple chase termination algorithms: construct the dependency graph of Σ, and if it has a "bad" cycle, then conclude that the chase does not terminate; otherwise, it does.

Main Outcome and 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 200K TGDs). Now, concerning the 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 presence 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 the above outcome concerning the acyclicity-based algorithms, we had to overcome a couple of 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 further discussed in Section 4. Hence, we had to revisit and refine the theoretical algorithms from [7] in order to obtain algorithms that are amenable to efficient implementations. The low-level implementation details of the refined algorithms are discussed in Section 5.

► In order to stress test the algorithms in question, 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 as 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.

An extended version of the paper with additional details, as well as the experimental infrastructure and the source code, can be found at https://github.com/mostafamilani/chase-termination.

2 PRELIMINARIES

We consider the disjoint countably infinite sets **C**, **N**, and **V** of *constants*, *(labeled) nulls*, and *variables*, respectively. We refer to constants, nulls and variables as *terms*. For an integer $n > 0$, we write $[n]$ for the set of integers $\{1, \ldots, n\}$.

Relational Databases. A *schema* **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 $\mathrm{ar}(R)$ for the integer n. A *(predicate) position* of **S** is a pair (R, i), where $R/n \in \mathbf{S}$ and $i \in [n]$, that essentially identifies the i-th argument of R. We write $\mathrm{pos}(\mathbf{S})$ for the set of positions of S, that is, the set $\{(R, i) \mid R/n \in \mathsf{S} \text{ and } i \in [n]\}$. An *atom* over S is an expression of the form $R(\bar{t})$, where $R/n \in \mathsf{S}$ 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 $\bar{t} = (t_1, \dots, t_n)$, let $\mathsf{pos}(R(\bar{t}), x) = \{(R, i) \mid t_i = x\}$. We write $\mathsf{var}(R(\bar{t}))$ for the set of variables in $\bar{t}$. The notations $\mathsf{pos}(\cdot, x)$ and $\mathsf{var}(\cdot)$ extend to sets of atoms. An *instance* over S is a (possibly infinite) set of atoms over S with constants and nulls. A *database* over S is a finite set of facts over S. The *active domain* of an instance I, denoted $\mathsf{dom}(I)$, is the set of terms (constants and nulls) occurring in I.

Homomorphisms. A *homomorphism* from a set of atoms A to a set of atoms B is a function 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 $R(t_1, \dots, t_n) \in A$ implies $h(R(t_1, \dots, t_n)) = R(h(t_1), \dots, h(t_n)) \in B$.

Tuple-Generating Dependencies. A *tuple-generating dependency* (TGD) σ is a (constant-free) sentence $\forall \bar{x} \forall \bar{y} \, (\phi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} \, \psi(\bar{x}, \bar{z}))$, 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 $\mathsf{body}(\sigma)$ and $\mathsf{head}(\sigma)$, respectively. The *frontier* of the TGD σ, denoted $\mathsf{fr}(\sigma)$, 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 $\mathsf{sch}(\Sigma)$, 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 an extension of h that is a homomorphism from $\psi(\bar{x}, \bar{z})$ to I; 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 called *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 call a linear TGD *simple* if no variable occurs more than once in its body, and the obtained class is denoted SL.

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 central notion in this context is that of 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 $\mathsf{body}(\sigma)$ to I. The *result* of (σ, h), denoted $\mathsf{result}(\sigma, h)$, is the set $\mu(\mathsf{head}(\sigma))$, where $\mu : \mathsf{var}(\mathsf{head}(\sigma)) \rightarrow \mathbf{C} \cup \mathbf{N}$ is defined as follows: $\mu(x) = h(x)$ if $x \in \mathsf{fr}(\sigma)$, and $\mu(x) = \bot^{x}_{\sigma, h_{|\mathsf{fr}(\sigma)}}$ if $x \notin \mathsf{fr}(\sigma)$, where $\bot^{x}_{\sigma, h_{|\mathsf{fr}(\sigma)}}$ is a null. Let $T(\Sigma, I)$ be the set of triggers for Σ on I. ■

Observe that in the definition of $\mathsf{result}(\sigma, h)$, each existentially quantified variable x of $\mathsf{head}(\sigma)$ 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 $\mathsf{result}(\sigma, h)$. 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 $\mathsf{chase}^0(D, \Sigma) = D$, and for each $i > 0$, let

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

where $S = T(\Sigma, \mathsf{chase}^{i-1}(D, \Sigma))$. We finally define *the result of the chase of D w.r.t. Σ* as the instance $\mathsf{chase}(D, \Sigma) = \bigcup_{i \geq 0} \mathsf{chase}^i(D, \Sigma)$.

Chase Termination. The result of the chase may be infinite even for very simple settings: it is easy to see that for $D = \{R(a, b)\}$ and $\Sigma = \{R(x, y) \rightarrow \exists z \, R(y, z)\}$, $\mathsf{chase}(D, \Sigma)$ is infinite. This leads to the following problem, parameterized by a class C of TGDs:

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

This problem has been recently studied in [7] 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 [7], which in turn lead to simple algorithms for checking the finiteness of the chase. Note that, for clarity, in the rest of the paper we assume TGDs with a non-empty frontier. This assumption can be made without loss of generality since, given a database D and a set Σ of TGDs, we can easily construct a set Σ' of TGDs with a non-empty frontier by slightly modifying Σ such that $\mathsf{chase}(D, \Sigma)$ is finite iff $\mathsf{chase}(D, \Sigma')$ is finite.

Non-Uniform Weak-Acyclicity. Weak-acyclicity was introduced in [12] 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 [7]. 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 a directed multigraph $\mathsf{dg}(\Sigma) = (N, E)$, where $N = \mathsf{pos}(\mathsf{sch}(\Sigma))$ and E contains *only* the following edges. For each TGD $\sigma \in \Sigma$ with $\mathsf{head}(\sigma) = \{\alpha_1, \dots, \alpha_k\}$, for each $x \in \mathsf{fr}(\sigma)$, and for each position $\pi \in \mathsf{pos}(\mathsf{body}(\sigma), x)$:

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

We further need to define when a predicate is reachable from another predicate. Given predicates $R, P \in \mathsf{sch}(\Sigma)$, *$P$ is reachable from R (w.r.t. Σ)* if $R = P$, or there exists a path in $\mathsf{dg}(\Sigma)$ 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 $\mathsf{dg}(\Sigma)$ 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 $\mathsf{dg}(\Sigma)$ with a special edge. We say that Σ is *weakly-acyclic* if there is no cycle in $\mathsf{dg}(\Sigma)$ with a special edge. ■

Characterizing the Finiteness of the Chase. It is not very difficult to show that whenever a set Σ of TGDs (not necessarily linear) is D-weakly-acyclic, then the instance $\mathsf{chase}(D, \Sigma)$ is finite. In other words, the D-weak-acyclicity of Σ is a sufficient condition for the finiteness of $\mathsf{chase}(D, \Sigma)$. 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 $\mathsf{chase}(D, \Sigma)$. This leads to the following characterization established in [7]:

Theorem 3.3. *Consider a database D and a set $\Sigma \in \mathsf{SL}$ of TGDs. It holds that* $\mathsf{chase}(D, \Sigma)$ *is finite iff Σ is D-weakly-acyclic.*

As shown in [7], non-uniform weak-acyclicity is not powerful enough for characterizing the finiteness of the chase instance in the case of linear TGDs. To obtain a characterization analogous to Theorem 3.3, the authors of [7] used the technique of *simplification* to convert linear TGDs into simple-linear TGDs, while preserving the finiteness of the chase instance. We recall this technique. Let $\bar{t} = (t_1, \ldots, t_n)$ be a tuple of (not necessarily distinct) terms. We write $\mathsf{unique}(\bar{t})$ 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 $\mathsf{unique}(\bar{t}) = (x, y, z)$. For each $i \in [n]$, the *identifier of t_i in $\bar{t}$*, denoted $\mathsf{id}_{\bar{t}}(t_i)$, is the integer that identifies the position of $\mathsf{unique}(\bar{t})$ at which t_i appears. We write $\mathsf{id}(\bar{t})$ for the tuple $(\mathsf{id}_{\bar{t}}(t_1), \ldots, \mathsf{id}_{\bar{t}}(t_n))$. For example, if $\bar{t} = (x, y, x, z, y)$, then $\mathsf{id}(\bar{t}) = (1, 2, 1, 3, 2)$. For an atom $\alpha = R(\bar{t})$, the *simplification of α*, denoted $\mathsf{simple}(\alpha)$, is the atom $R_{\mathsf{id}(\bar{t})}(\mathsf{unique}(\bar{t}))$, whereas the *shape of α*, denoted $\mathsf{shape}(\alpha)$, is the predicate $R_{\mathsf{id}(\bar{t})}$. We can naturally refer to the simplification and the shape of a set of atoms. 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 3.4. 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 simple-linear 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. ■

We can now recall the characterization for the finiteness of the chase instance for linear TGDs, established in [7], which is similar to the one for simple-linear TGDs, with the key difference that first we need to simplify both the database and the set of linear TGDs:

Theorem 3.5. *Consider a database D and a set $\Sigma \in \mathsf{L}$ of TGDs. Then,* $\mathsf{chase}(D, \Sigma)$ *is finite iff* $\mathsf{simple}(\Sigma)$ *is* $\mathsf{simple}(D)$*-weakly-acyclic.*

It is clear that Theorems 3.3 and 3.5 provide simple algorithms for checking whether the chase instance is finite. Our goal is to experimentally evaluate those 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 obtained 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. Hence, we need to convert the inherited theoretical algorithms into practical algorithms that are amenable to efficient implementations.

```
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
```

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, i.e., whether $\mathsf{chase}(D, \Sigma)$ 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 say 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 3.4. The goal of the dynamic simplification is to keep only TGDs of $\mathsf{simple}(\Sigma)$ that are needed for checking whether the chase is finite. We present the above algorithms at a high-level, whereas their implementation details are discussed in Section 5.

4.1 Simple-Linear TGDs

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. IsChaseFinite[SL], which is depicted in Algorithm 1, starts by building the dependency graph G of the input set Σ of 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 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 2. 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); 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 thus the algorithm returns false; otherwise, it returns true. 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. It holds that* IsChaseFinite[SL]$(D, \Sigma) =$ true *iff* $\mathsf{chase}(D, \Sigma)$ *is finite.*

4.2 Linear TGDs

Although the algorithm $\mathsf{IsChaseFinite[SL]}$ together with the simplification technique (see Definition 3.4) immediately give 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 sets of linear TGDs coming from the literature 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 $\mathsf{IsChaseFinite[SL]}$ 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 the instance $\mathsf{chase}(\mathsf{simple}(D), \mathsf{simple}(\Sigma))$ is finite iff the instance $\mathsf{chase}(\mathsf{simple}(D), \mathsf{simple}_D(\Sigma))$ is finite, which essentially tells us that 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 mathematically 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 coming 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 1000 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 $\mathbf{S}$. 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 $DB[S] = \{R(1,2), P(1,1,2)\}$. 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, $\Gamma_\Sigma^0(S) = S$ and $\Gamma_\Sigma^i(S) = \Gamma_\Sigma(\Gamma_\Sigma^{i-1}(S))$, for each $i > 0$, and we finally let $\Sigma(S) = \bigcup_{i \geq 0} \Gamma_\Sigma^i(S)$. 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 $\Sigma(S) = \Gamma_\Sigma^{|\mathsf{simple}(\Sigma)|}(S)$. Therefore, since $\mathsf{shape}(\Sigma)$ is finite, $\Sigma(S)$ can be obtained after finitely many steps. We now have all the ingredients to 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 $\Sigma(\mathsf{shape}(D))$. We now 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.*

Since, by definition, $\mathsf{simple}_D(\Sigma) \subseteq \mathsf{simple}(\Sigma)$, it is clear that (1) implies (2) holds trivially. The interesting direction is (2) implies (1), which can be shown via an inductive argument. Another crucial property of dynamic simplification, which will help us to further improve the performance of the termination algorithm, is that the $\mathsf{simple}(D)$-weak-acyclicity of $\mathsf{simple}_D(\Sigma)$ coincides with the weak-acyclicity of $\mathsf{simple}_D(\Sigma)$, which in turn leads to the following result:

[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 $\mathsf{chase}(D, \Sigma)$.

```
Algorithm 2: DynSimplification
  Input: A database $D$ and a set $\Sigma \in \mathsf{L}$ of TGDs
  Output: The $D$-simplification of $\Sigma$
1  $S \longleftarrow \mathsf{FindShapes}(D)$;
2  $\Sigma_s \longleftarrow \emptyset$;
3  $\Delta S \longleftarrow S$;
4  while $\Delta S \neq \emptyset$ do
5      $\Sigma_{aux} \longleftarrow \mathsf{Applicable}(\Delta S, \Sigma)$;
6      $S_{aux} \longleftarrow \{R_{\mathsf{id}(\bar{t})} \in \mathsf{shape}(\Sigma) \mid$ there exists a TGD $\sigma \in \Sigma_{aux}$ such that $R_{\mathsf{id}(\bar{t})}$ occurs in $\mathsf{head}(\sigma)\}$;
7      $\Sigma_s \longleftarrow \Sigma_s \cup \Sigma_{aux}$;
8      $\Delta S \longleftarrow S_{aux} \setminus S$;
9      $S \longleftarrow S \cup \Delta S$;
10 return $\Sigma_s$;
```

Lemma 4.4. *Consider a database D and a set* $\Sigma \in \mathsf{L}$ *of TGDs. The following are equivalent:*

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

An Algorithm for Dynamic Simplification. We now 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 present the algorithm DynSimplification, depicted in Algorithm 2. 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, i.e., they can be applied during the construction of chase(simple(D), simple(Σ)), that are added to Σ_s (lines 5 and 7). This is done via Applicable, which takes as input a set of shapes $\hat{S}$ and a set of linear TGDs $\hat{\Sigma}$, and returns the set

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

In essence, the procedure Applicable computes the set of TGDs of simple($\hat{\Sigma}$) 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 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 and all the applicable TGDs on S are applied during the first iteration. DynSimplification is correct by construction:

Lemma 4.5. *Consider a database D and a set* $\Sigma \in \mathsf{L}$ *of TGDs. It holds that* $\mathsf{DynSimplification}(D, \Sigma) = \mathsf{simple}_D(\Sigma)$.

```
Algorithm 3: IsChaseFinite[L]
  Input: A database $D$ and a set $\Sigma \in \mathsf{L}$ of TGDs
  Output: true if $\mathsf{chase}(D, \Sigma)$ is finite and false otherwise
  $\Sigma_s \longleftarrow \mathsf{DynSimplification}(D, \Sigma)$;
  $G \longleftarrow \mathsf{BuildDepGraph}(\Sigma_s)$;
  if $\mathsf{FindSpecialSCC}(G) \neq \emptyset$ then return false;
  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 3.5, Lemma 4.3, and Lemma 4.4, imply the correctness of the algorithm IsChaseFinite[L], and the next lemma follows:

Lemma 4.6. *Given a database D and a set* $\Sigma \in \mathsf{L}$ *of TGDs, it holds that* $\mathsf{IsChaseFinite[L]}(D, \Sigma) = \mathtt{true}$ *iff* $\mathsf{chase}(D, \Sigma)$ *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, 5.2, and 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 helps when checking the support of positions in special SCCs, as explained 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.

5.2 Find Special SCCs

To implement FindSpecialSCC we adapt the well-known *Tarjan's algorithm* for finding SCCs in directed graphs [19]; we assume the reader is familiar with this algorithm. Actually, FindSpecialSCC is a simple extension of Tarjan's algorithm. Such an extension is needed as we need a mechanism that allows us to check whether a SCC obtained by Tarjan's algorithm is indeed special. This is done by pushing a dummy token in the algorithm's stack (the stack that stores the visited nodes in the current component) whenever the algorithm traverses a special edge. 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 predicates in the database, 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 predicates occurring in the database, 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 predicates faster and without accessing the actual data. For step (2), we start from the given set of positions P and traverse the graph in the reverse order using the reverse links in the adjacency list of the dependency graph. The procedure returns `true` if the graph traversal in the second step reaches a node (i.e., a position) in P_D; otherwise, it returns `false`.

5.4 Dynamic Simplification

DynSmplification is an iterative procedure that uses 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.

The Procedure FindShapes**.** We have two kinds of implementations for this procedure, that is, *in-memory* and *in-database*.

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 $R(\bar{c})$. For relations that cannot be entirely loaded into the main memory, we split them into smaller relations processed separately.

In-database Implementation

The in-database implementation does not load the relations, 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 to 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 [2]. 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. For example, the first query Q' for $R_{(1,1,2)}$ is the query 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 $R \in$ sch(Σ) 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

As already said, our goal is to experimentally evaluate the behaviour of the termination algorithms presented in Section 5. To this end, we are going to conduct extensive experiments with synthetic data and sets of TGDs. 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 databases (Section 6.1) and sets of TGDs (Section 6.2). Let us clarify that in the rest of the paper, 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 and DataFiller. 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 that overcomes the above limitation.

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 $\mathsf{dom}(D)$), 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 $\mathsf{S} = \{R \mid R(\bar{c}) \in D\}$, $|\mathsf{S}| = \mathit{preds}$, the predicates of S have arity between *min* and *max*, $|\mathsf{dom}(D)| = \mathit{dsize}$, and, for each $R \in \mathsf{S}$, $|\{\bar{c} \mid R(\bar{c}) \in D\}| = \mathit{rsize}$. To this end, it first generates a set S consisting of *preds* different predicates, and it randomly selects an arity for each such predicate from the range $[\mathit{min}, \mathit{max}]$. It then generates a database by adding *rsize* tuples to each predicate of 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 *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, [4, 6]) 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 just linear TGDs). In particular, the TGD generator takes as input a set 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 $\mathsf{sch}(\Sigma) \subseteq \mathsf{S}$, $|\mathsf{sch}(\Sigma)| = \mathit{ssize}$, the predicates of $\mathsf{sch}(\Sigma)$ have arity between *min* and *max*, $|\Sigma| = \mathit{tsize}$, and Σ falls in the class *tclass*. To this end, it first chooses a subset S' of S such that $|\mathsf{S}'| = \mathit{ssize}$ 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 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 described below is designed to generate single-head TGDs.

Simple-Linear TGDs

To generate a simple-linear TGD, the generator randomly selects two predicates from S' that will be used to form the body- and the head-atom, respectively. The selection is with repetition to allow the same predicate to appear in both the body and the head. 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 or a universally quantified variable that has been 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 new 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 3.00GHz CPU and 16GB 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 $\mathsf{IsChaseFinite[SL]}$, 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 clarification remark.

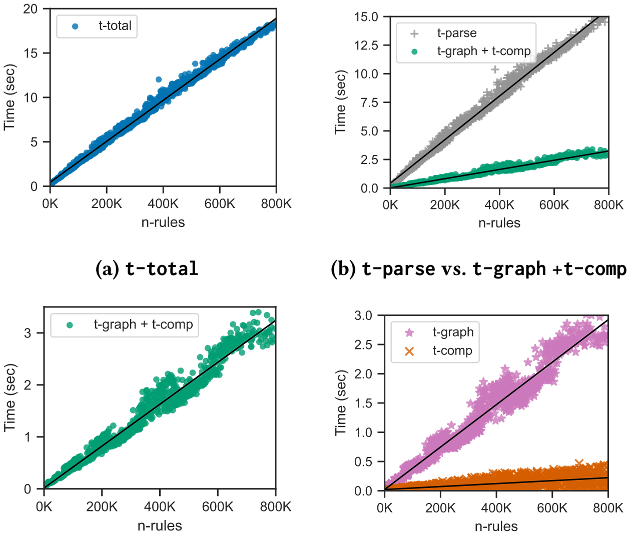
Remark. In our analysis, we neglect the time taken by the procedure Supports, which, as explained in Section 5.3, consists of two steps: (1) find the predicates occurring in the input database, and (2) traverse the dependency graph starting from the positions in the special SCCs in the reverse order to reach the positions of the predicates 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. Therefore, Supports takes insignificant time compared to the rest of the algorithm, which we can simply ignore without affecting our analysis for IsChaseFinite[SL]. Hence, 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 also 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. In fact, D_Σ has an atom $R(c_1, \dots, c_n)$, where $c_1, \dots, c_n$ are distinct constants, for each predicate $R \in \mathsf{sch}(\Sigma)$.

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 $200 \leq \mathsf{sch}(\Sigma) \leq 400$, each predicate of $\mathsf{sch}(\Sigma)$ 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 as follows. We have first constructed the underlying schema $\mathbf{S}$ by generating 1000 predicates, while their arities were randomly selected from [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 $\mathbf{S}$ and the tuple of values for the tuning parameters (ssize, 1, 5, tsize, SL), where ssize and tsize were randomly chosen from $[x, y]$ 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. 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, i.e., each point in the plots corresponds to one of the 900 sets of TGDs. Figure 1a shows the total runtime (`t-total`) for sets of TGDs with various sizes (`n-rules`). Figure 1b 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 1c zooms in `t-graph` + `t-comp`, which are shown separately in Figure 1d.



(a) `t-total`

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

(c) `t-graph` + `t-comp`

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

Figure 1: Runtime of IsChaseFinite[SL].

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 1d) 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 quite efficient that runs in linear time in the size of the underlying 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 the 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. We proceed to 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 database, present our experimental results for the two components of IsChaseFinite[L], and discuss the take-home messages.

8.1 Generating Databases and Linear TGDs

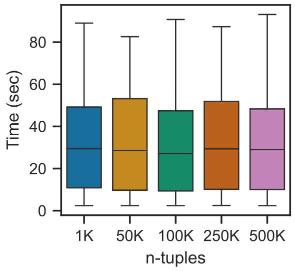
The goal is to generate a family of pairs of the form (D, Σ), where D is a database and Σ a set of linear TGDs, that will serve as the input to IsChaseFinite[L] during the experimental evaluation. To this end, we first constructed a very large database, which we dub $D^\star$, by using our data generator. In particular, we called the data generator with input (1000, 1, 5, 500K, 500K), and obtained $D^\star$ that mentions 1000 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. This is turn ensures that the virtual databases defined via the views have a variety of shapes, which is crucial for our purposes. Having the database $D^\star$ 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 5 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*, L), where *ssize* and *tsize* were randomly chosen from $[x, y]$ 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^1_Σ, D^{50}_Σ, D^{100}_Σ, D^{250}_Σ, and D^{500}_Σ, respectively, leading to five pairs. Summing up, we generated the family $\{(D^s_\Sigma, \Sigma) \mid \Sigma \in \Sigma^\star \text{ and } s \in \{1, 50, 100, 250, 500\}\}$ consisting of 225 pairs.

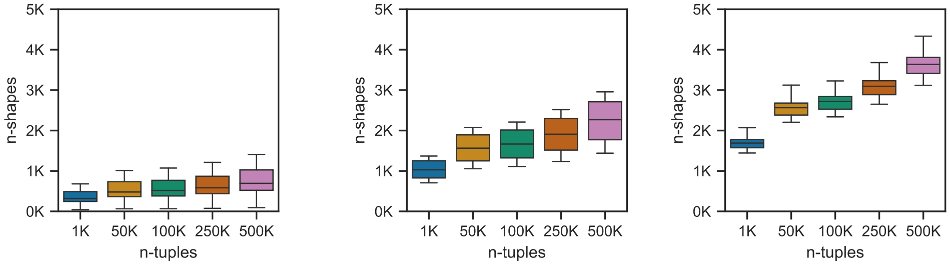
8.2 Experimental Evaluation

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.

Separate the Two Components. The figure below depicts the average time over all generated pairs, consisting of a database D^s_Σ 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^s_Σ and finding the special SCCs:

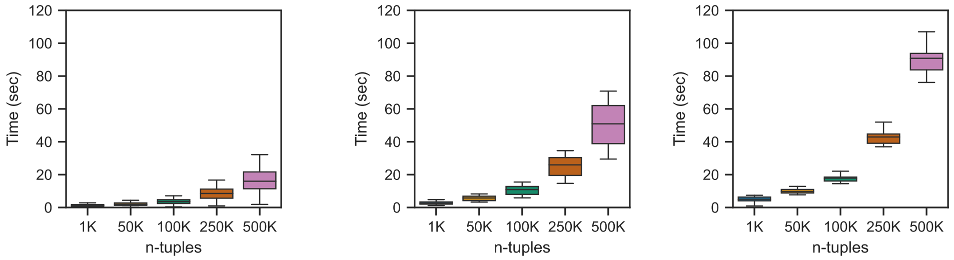


Interestingly, it confirms that the database size does not impact the time to build the dependency graph and find the special SCCs; thus, it does not impact the end-to-end runtime of the db-independent component, as claimed above. This can be explained by the fact that the number of shapes in a database increases very slowly as we increase the size of the database; this is illustrated in Figure 2. In particular, the bar plots in Figure 2 show the average number of shapes over all databases D^s_Σ of a certain size s, where each plot corresponds to a certain predicate profile $[x, y]$, i.e., Σ falls in the predicate profile $[x, y]$. It is clear that the number of shapes increases



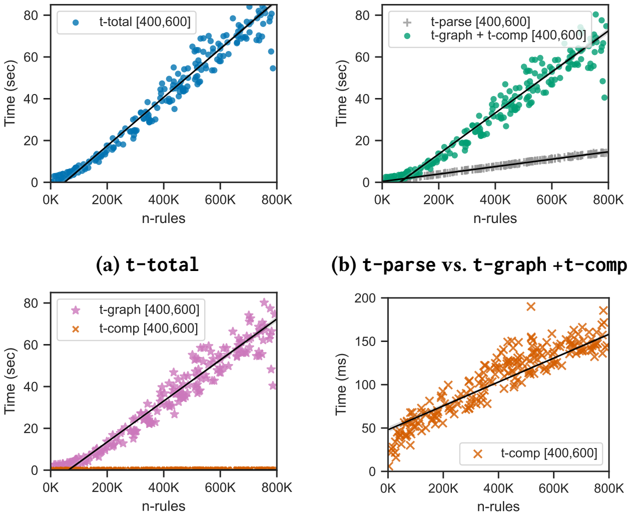
(a) [5,200] **(b) [200,400]** **(c) [400,600]**

Figure 2: Number of Shapes.



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

Figure 3: Runtime of FindShapes **(in-database implementation).**



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

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

Figure 4: Runtime of the db-independent component.

as we increase the size of the database, which was expected. The interesting outcome, however, is the fact that this increase is very slow, which should be attributed to the fact that many tuples are likely to induce the same shape. Moreover, a new shape gives rise to only a few simplified TGDs, and it does not significantly affect the time for building and processing the dependency graph.

Let us finally observe that, by comparing the three bar plots, it is clear that the number of predicates, reflected in the predicate profile, impacts the number of shapes, which is rather expected as with more predicates there would be more shapes. This means that the number of predicates of the underlying schema is a parameter that affects the number of shapes, which explains why in our analysis above we had to separately consider the three predicate profiles.

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 Figure 3 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-database implementation, where each plot corresponds to a certain predicate profile. Due to space constraints, we do not report the analogous plots for the in-memory implementation. Note, however, 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 3 that the time to find the shapes increases while the database size increases, which is not surprising since, as discussed above, the number of shapes increases while the database size increases. Observe, however, that the time to find the shapes grows much faster than the actual number shapes, 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, also 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 4 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. Due to space constraints, we omit the analogous plots for the smaller predicate profiles [5,200] and [200,400], which depict similar trends. Figure 4a shows the total runtime (`t-total`) for sets of TGDs with various sizes (`n-rules`). Figure 4b 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 4c shows separately `t-graph` and `t-comp`. Figure 4d 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`). It should not be forgotten, however, that for linear TGDs we need to focus on a single predicate profile in order to get these linear trends; otherwise, if we consider all the predicate profiles at once, there is no trend that can be observed. Note also that the absolute running time increases compared to the case of simple-linear TGDs.

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. On the one hand, we have the db-dependent component that is only affected by the size of the database. On the other hand, we have the db-independent component 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 quite reasonable, which should be seen as 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 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

With the aim of validating the main outcome of the stress test analysis performed in Sections 7 and 8, we have also run experiments using databases and sets of TGDs that are available in the literature. Due to space constraints, we cannot present this analysis in full details, and we refer the reader to the extended version of the paper. We proceed to briefly discuss the essence of this validation analysis.

Adopted Scenarios. We considered three families of databases and sets of TGDs from the literature: (i) the family Deep from [6] that collects sets of simple-linear TGDs that are at the same time weakly-acyclic, which has been developed to test data exchange scenarios, (ii) LUBM, which is a popular benchmark consisting of an ontology modelled using the central Description Logic (DL) EL, called Univ-Bench, and a data generator, called UBA, for generating synthetic data over the vocabulary of Univ-Bench [13], and (iii) iBench, which is a framework for generating TGDs with tuning parameters that can control a wide range of properties [4].

Discussion. Concerning IsChaseFinite[SL] for simple-linear TGDs, we observed 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 can be done very efficiently. We have also seen that the validation analysis confirms the main outcome of the analysis for the algorithm IsChaseFinite[L] performed in Section 8. In particular, we observed that indeed the costly task is finding the database shapes, whereas the time taken by the db-independent component is negligible. Moreover, we have seen 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 some members of LUBM, it takes less than a second. Another interesting takeaway from the validation analysis 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 an offline computation of the database shapes might be preferred.

10 CONCLUSIONS AND FUTURE WORK

Our work provides the first systematic attempt to experimentally evaluate termination algorithms for the semi-oblivious chase. 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 ones. 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 without affecting the db-independent component. An interesting direction, which we are planning to explore, is to materialize and incrementally keep updated the shapes in a database.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

[2] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast Algorithms for Mining Association Rules in Large Databases. In *VLDB*. 487–499.

[3] 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.

[4] Patricia C. Arocena, Boris Glavic, Radu Ciucanu, and Renée J. Miller. 2015. The iBench Integration Metadata Generator. *PVLDB* 9, 3 (2015), 108–119.

[5] Catriel Beeri and Moshe Y. Vardi. 1984. A Proof Procedure for Data Dependencies. *J. ACM* 31, 4 (1984), 718–741.

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

[7] Marco Calautti, Georg Gottlob, and Andreas Pieris. 2022. Non-Uniformly Terminating Chase: Size and Complexity. In *PODS*. 369–378.

[8] Marco Calautti and Andreas Pieris. 2021. Semi-Oblivious Chase Termination: The Sticky Case. *Theory Comput. Syst.* 65, 1 (2021), 84–121.

[9] 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.

[10] 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.

[11] Alin Deutsch, Alan Nash, and Jeff B. Remmel. 2008. The Chase Revisisted. In *PODS*. 149–158.

[12] 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.

[13] 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.

[14] Markus Krötzsch, Maximilian Marx, and Sebastian Rudolph. 2019. The Power of the Terminating Chase (Invited Talk). In *ICDT*. 3:1–3:17.

[15] 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.

[16] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. 1979. Testing Implications of Data Dependencies. *ACM Trans. Database Syst.* 4, 4 (1979), 455–469.

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

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

[19] Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

[20] 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.