# The Space-Efficient Core of Vadalog

**Citation for published version:**
Berger, G, Gottlob, G, Pieris, A & Sallinger, E 2019, The Space-Efficient Core of Vadalog. in *PODS '19 Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems.* ACM, New York, pp. 270-284 , ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD 2019), Amsterdam, Netherlands, 30/06/19. https://doi.org/10.1145/3294052.3319688

**Digital Object Identifier (DOI):**
10.1145/3294052.3319688

**Link:**
Link to publication record in Edinburgh Research Explorer

**Document Version:**
Peer reviewed version

**Published In:**
PODS '19 Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems

# The Space-Efficient Core of Vadalog

Gerald Berger
Institute of Logic and Computation
TU Wien

Georg Gottlob
Department of Computer Science
University of Oxford & TU Wien

Andreas Pieris
School of Informatics
University of Edinburgh

Emanuel Sallinger
Department of Computer Science
University of Oxford & TU Wien

## ABSTRACT

Vadalog is a system for performing complex reasoning tasks such as those required in advanced knowledge graphs. The logical core of the underlying Vadalog language is the warded fragment of tuple-generating dependencies (TGDs). This formalism ensures tractable reasoning in data complexity, while a recent analysis focusing on a practical implementation led to the reasoning algorithm around which the Vadalog system is built. A fundamental question that has emerged in the context of Vadalog is the following: can we limit the recursion allowed by wardedness in order to obtain a formalism that provides a convenient syntax for expressing useful recursive statements, and at the same time achieves space-efficiency? After analyzing several real-life examples of warded sets of TGDs provided by our industrial partners, as well as recent benchmarks, we observed that recursion is often used in a restricted way: the body of a TGD contains at most one atom whose predicate is mutually recursive with a predicate in the head. We show that this type of recursion, known as piece-wise linear in the Datalog literature, is the answer to our main question. We further show that piece-wise linear recursion alone, without the wardedness condition, is not enough as it leads to the undecidability of reasoning. We finally study the relative expressiveness of the query languages based on (piece-wise linear) warded sets of TGDs.

## CCS CONCEPTS

• **Information systems → Query languages**;

ACM ISBN 978-1-4503-6227-6/19/06…$15.00
https://doi.org/10.1145/3294052.3319688

## KEYWORDS

reasoning, query answering, Datalog, tuple-generating dependencies, complexity, expressive power

**ACM Reference Format:**
Gerald Berger, Georg Gottlob, Andreas Pieris, and Emanuel Sallinger. 2019. The Space-Efficient Core of Vadalog. In *38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3294052.3319688

## 1 INTRODUCTION

In recent times, thousands of companies world-wide wish to manage their own knowledge graphs (KGs), and are looking for adequate knowledge graph management systems (KGMS). The term knowledge graph originally only referred to Google's Knowledge Graph, i.e., "a knowledge base used by Google and its services to enhance its search engine's results with information gathered from a variety of sources.[1]" In the meantime, several other large companies have constructed their own knowledge graphs, and many more companies would like to maintain a private corporate knowledge graph incorporating large amounts of data in form of database facts, both from corporate and public sources, as well as rule-based knowledge. Such a corporate knowledge graph is expected to contain relevant business knowledge, for example, knowledge about customers, products, prices, and competitors, rather than general knowledge from Wikipedia and similar sources. It should be managed by a KGMS, that is, a knowledge base management system, which performs complex rule-based reasoning tasks over very large amounts of data and, in addition, provides methods and tools for data analytics and machine learning [6].

### 1.1 The Vadalog System

Vadalog is a system for performing complex reasoning tasks such as those required in advanced knowledge graphs [7]. It is Oxford's contribution to the VADA research project,[2]

[1]https://en.wikipedia.org/wiki/Knowledge_Graph
[2]http://vada.org.uk/

a joint effort of the universities of Oxford, Manchester, and Edinburgh, as well as around 20 industrial partners such as Facebook, BP, and the NHS (UK national health system). One of the most fundamental reasoning tasks performed by Vadalog is *ontological query answering*: given a database $D$, an ontology $\Sigma$ (which is essentially a set of logical assertions that allow us to derive new intensional knowledge from $D$), and a query $q(\bar{x})$ (typically a conjunctive query), the goal is to compute the certain answers to $q$ w.r.t. the knowledge base consisting of $D$ and $\Sigma$, i.e., the tuples of constants $\bar{c}$ such that, for every relational instance $I \supseteq D$ that satisfies $\Sigma$, $I$ satisfies the Boolean query $q(\bar{c})$ obtained after instantiating $\bar{x}$ with $\bar{c}$. Due to Vadalog's ability to perform ontological query answering, it is currently used as the core deductive database component of the overall Vadalog KGMS, as well as at various industrial partners including the finance, security, and media intelligence industries.

The logical core of the underlying Vadalog language is a rule-based formalism known as *warded Datalog*$^\exists$ [17], which is a member of the Datalog$^\pm$ family of knowledge representation languages [11]. Warded Datalog$^\exists$ generalizes Datalog with existential quantification in rule heads, and at the same time applies a restriction on how certain "dangerous" variables can be used; details are given in Section 3. Such a restriction is needed as basic reasoning tasks, e.g., ontological query answering, under arbitrary Datalog$^\exists$ rules become undecidable; see, e.g., [5, 10]. Let us clarify that Datalog$^\exists$ rules are essentially *tuple-generating dependencies* (TGDs) of the form $\forall\bar{x}\forall\bar{y}(\varphi(\bar{x},\bar{y}) \rightarrow \exists\bar{z}\,\psi(\bar{x},\bar{z}))$, where $\varphi$ (the *body*) and $\psi$ (the *head*) are conjunctions of atoms. Therefore, knowledge representation and reasoning should be seen as a modern application of TGDs, which have been introduced decades ago as a unifying framework for database integrity constraints.

The key properties of warded Datalog$^\exists$, which led to its adoption as the logical core on top of which the Vadalog language is built, can be summarized as follows:

(1) *Recursion over KGs.* It is able to express full recursion and joins, needed to express complex reasoning tasks over KGs. Moreover, navigational capabilities, empowered by recursion, are vital for graph-based structures.
(2) *Ontological Reasoning over KGs.* After adding a very mild and easy to handle negation, the language is able to express SPARQL reasoning under the OWL 2 QL entailment regime. Recall that SPARQL is the standard language for querying the Semantic Web,[3] while OWL 2 QL is a prominent profile of the OWL 2 Web Ontology Language, the standard formalism for modeling Semantic Web ontologies.[4]
(3) *Low Complexity.* Reasoning, in particular, ontological query answering, is tractable (in fact, polynomial time) in data complexity, which is a minimal requirement for allowing scalability over large volumes of data.

Warded Datalog$^\exists$ turned out to be powerful enough for expressing all the tasks given by our industrial partners, while a recent analysis of it focusing on a practical implementation led to the reasoning algorithm around which the Vadalog system is built [7].

## 1.2 Research Challenges

With the aim of isolating more refined formalisms, which will lead to yet more efficient reasoning algorithms, the following fundamental question has emerged in the context of Vadalog:

*Can we limit the recursion allowed by wardedness in order to obtain a formalism that provides a convenient syntax for expressing useful statements, importantly, most of the scenarios provided by our partners, and at the same time achieves space-efficiency, in particular,* NLogSpace *data complexity?*

Let us stress that NLogSpace data complexity is the best that we can hope for, since navigational capabilities are vital for graph-based structures, and already graph reachability is NLogSpace-hard. It is known that NLogSpace is contained in the class $NC_2$ of highly parallelizable problems. This means that reasoning in the more refined formalism that we are aiming is principally parallelizable, unlike warded Datalog$^\exists$, which is PTime-complete and intrinsically sequential. Our ultimate goal is to exploit this in the future for the parallel execution of reasoning tasks in both multi-core settings and in the map-reduce model. In fact, we are currently in the process of implementing a multi-core implementation for the refined formalism proposed by the present work.

Extensive benchmark results are available for the Vadalog system, based on a variety of scenarios, both synthetic and industrial scenarios, including: ChaseBench [8], a benchmark that targets data exchange and query answering problems; iBench, a data exchange benchmark developed at the University of Toronto [4]; iWarded, a benchmark specifically targeted at warded sets of TGDs; a DBpedia based benchmark; and a number of other synthetic and industrial scenarios [7]. Let us stress that all the above benchmarks contain only warded sets of TGDs. In fact, a good part of them are not *warded by chance*, i.e., they contain joins among "harmful" variables, which is one of the distinctive features of wardedness [7]. After analyzing the above benchmarks, we observed that recursion is often used in a restricted way. Approximately 70% of the TGD-sets use recursion in the following way: the body of a TGD contains at most one atom whose predicate is mutually recursive with a predicate in the head. More specifically, approximately 55% of the TGD-sets

[3]http://www.w3.org/TR/rdf-sparql-query
[4]https://www.w3.org/TR/owl2-overview/

directly use the above type of recursion, while 15% can be transformed into warded sets of TGDs that use recursion as above. This transformation relies on a standard elimination procedure of unnecessary non-linear recursion. For example,

$$\begin{aligned}\forall x \forall y (E(x,y) &\rightarrow T(x,y)) \\ \forall x \forall y \forall z (T(x,y) \wedge T(y,z) &\rightarrow T(x,z)),\end{aligned}$$

which compute the transitive closure of the binary relation $E$ using non-linear recursion, can be rewritten as the set

$$\begin{aligned}\forall x \forall y (E(x,y) &\rightarrow T(x,y)) \\ \forall x \forall y \forall z (E(x,y) \wedge T(y,z) &\rightarrow T(x,z))\end{aligned}$$

that uses linear recursion. Interestingly, the type of recursion discussed above has been already studied in the context of Datalog, and is known as *piece-wise linear*; see, e.g., [1]. It is a refinement of the well-known *linear* recursion [24, 25], already mentioned in the above example, which allows only one intensional predicate to appear in the body, while all the other predicates are extensional.

Based on this key observation, the following research questions have immediately emerged:

(1) Does warded Datalog$^{\exists}$ with piece-wise linear recursion achieve space-efficiency for query answering?[5]
(2) Is the combination of wardedness and piece-wise linearity justified? In other words, can we achieve the same with piece-wise linear Datalog$^{\exists}$ without the wardedness condition?
(3) What is the expressiveness of the query language based on warded Datalog$^{\exists}$ with piece-wise linear recursion relative to prominent languages such as Datalog?

These are top-priority questions in the context of the Vadalog system since they may provide useful insights towards more efficient reasoning algorithms, in particular, towards parallel execution of reasoning tasks. The ultimate goal of this work is to analyze piece-wise linearity, and provide definite answers to the above questions.

### 1.3 Summary of Contributions

Our main results can be summarized as follows:

(1) Ontological query answering under warded Datalog$^{\exists}$ with piece-wise linear recursion is NLogSpace-complete in data complexity, and PSpace-complete in combined complexity, which provides a definite answer to our first question. This is a rather involved result that heavily relies on a novel notion of resolution-based proof tree, which is of independent interest. In particular, we show that ontological query answering under warded Datalog$^{\exists}$ with piece-wise linear recursion boils down to the problem of checking whether a proof tree that enjoys certain properties exists, which in turn can be done via a space-bounded non-deterministic algorithm. Interestingly, our machinery allows us to re-establish the complexity of ontological query answering under warded Datalog$^{\exists}$ via an algorithm that is significantly simpler than the one employed in [17]. This algorithm is essentially the non-determinisitc algorithm for piece-wise linear warded Datalog$^{\exists}$ with the crucial difference that it employs alternation.
(2) To our surprise, ontological query answering under piece-wise linear Datalog$^{\exists}$, without the wardedness condition, is undecidable. This result, which is shown via a reduction from the unbounded tiling problem, provides a definite answer to our second question: the combination of wardedness and piece-wise linearity is indeed justified.
(3) We finally investigate the relative expressive power of the query language based on warded Datalog$^{\exists}$ with piece-wise linear recursion, which consists of all the queries of the form $Q = (\Sigma, q)$, where $\Sigma$ is a warded set of TGDs with piece-wise linear recursion, and $q$ is a conjunctive query, while the evaluation of $Q$ over a database $D$ is precisely the certain answers to $q$ w.r.t. $D$ and $\Sigma$. By exploiting our novel notion of proof tree, we show that it is equally expressive to piece-wise linear Datalog. The same approach allows us to elucidate the relative expressiveness of the query language based on warded Datalog$^{\exists}$ (with arbitrary recursion), showing that it is equally expressive to Datalog. We also adopt the more refined notion of program expressive power, introduced in [2], which aims at the decoupling of the set of TGDs and the actual conjunctive query, and show that the query language based on warded Datalog$^{\exists}$ (with piece-wise linear recursion) is strictly more expressive than Datalog (with piece-wise linear recursion). This exposes the advantage of value invention that is available in Datalog$^{\exists}$-based languages.

## 2 PRELIMINARIES

**Basics.** We consider the disjoint countably infinite sets $\mathbf{C}$, $\mathbf{N}$, and $\mathbf{V}$ of *constants*, *(labeled) nulls*, and *variables*, respectively. The elements of $(\mathbf{C} \cup \mathbf{N} \cup \mathbf{V})$ are called *terms*. An *atom* is an expression of the form $R(\bar{t})$, where $R$ is an $n$-ary predicate, and $\bar{t}$ is an $n$-tuple of terms. We write $\text{var}(\alpha)$ for the set of variables in an atom $\alpha$; this notation extends to sets of atoms. A *fact* is an atom that contains only constants. A *substitution* from a set of terms $T$ to a set of terms $T'$ is a function $h \colon T \rightarrow T'$. 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

[5]The idea of combining wardedness with piece-wise linearity has been already mentioned in the invited paper [6], while the obtained formalism is called strongly warded.

in $B$ such that $h$ is the identity on $\mathbf{C}$, and $R(t_1, \dots, t_n) \in A$ implies $h(R(t_1, \dots, t_n)) = R(h(t_1), \dots, h(t_n)) \in B$. We write $h(A)$ for the set of atoms $\{h(\alpha) \mid \alpha \in A\}$. For brevity, we may write $[n]$ for the set $\{1, \dots, n\}$, where $n \geq 0$.

**Relational Databases.** A *schema* $\mathbf{S}$ is a finite set of relation symbols (or predicates), each having an associated *arity*. We write $R/n$ to denote that $R$ has arity $n \geq 0$. A *position* $R[i]$ in $\mathbf{S}$, where $R/n \in \mathbf{S}$ and $i \in [n]$, identifies the $i$-th argument of $R$. An *instance* over $\mathbf{S}$ is a (possibly infinite) set of atoms over $\mathbf{S}$ that contain constants and nulls, while a *database* over $\mathbf{S}$ is a finite set of facts over $\mathbf{S}$. The *active domain* of an instance $I$, denoted $\mathsf{dom}(I)$, is the set of all terms occurring in $I$.

**Conjunctive Queries.** A *conjunctive query* (CQ) over $\mathbf{S}$ is a first-order formula of the form

$$q(\bar{x}) \ := \ \exists \bar{y} \left(R_1(\bar{z}_1) \wedge \cdots \wedge R_n(\bar{z}_n)\right),$$

where each $R_i(\bar{z}_i)$, for $i \in [n]$, is an atom without nulls over $\mathbf{S}$, each variable mentioned in the $\bar{z}_i$'s appears either in $\bar{x}$ or $\bar{y}$, and $\bar{x}$ are the *output variables* of $q$. For convenience, we adopt the rule-based syntax of CQs, i.e., a CQ as the one above will be written as the rule

$$Q(\bar{x}) \ \leftarrow \ R_1(\bar{z}_1), \dots, R_n(\bar{z}_n),$$

where $Q$ is a predicate used only in the head of CQs. Let $\mathsf{atoms}(q) = \{R_1(\bar{z}_1), \dots, R_n(\bar{z}_n)\}$. The *evaluation* of $q(\bar{x})$ over an instance $I$, denoted $q(I)$, is the set of all tuples $h(\bar{x})$ of constants, where $h$ is a homomorphism from $\mathsf{atoms}(q)$ to $I$.

**Tuple-Generating Dependencies.** A *tuple-generating dependency* (TGD) $\sigma$ is a first-order sentence

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

where $\bar{x}, \bar{y}, \bar{z}$ are tuples of variables of $\mathbf{V}$, and $\phi, \psi$ are conjunctions of atoms without constants and nulls. For brevity, we write $\sigma$ 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 $\phi$ and $\psi$ as the *body* and *head* of $\sigma$, denoted $\mathsf{body}(\sigma)$ and $\mathsf{head}(\sigma)$, respectively. The *frontier* of the TGD $\sigma$, denoted $\mathsf{front}(\sigma)$, is the set of variables that appear both in the body and the head of $\sigma$. We also write $\mathsf{var}_{\exists}(\sigma)$ for the existentially quantified variables of $\sigma$. The schema of a set $\Sigma$ of TGDs, denoted $\mathsf{sch}(\Sigma)$, is the set of predicates in $\Sigma$. An instance $I$ satisfies a TGD $\sigma$ as the one above, written $I \models \sigma$, if the following holds: whenever there exists a homomorphism $h$ such that $h(\phi(\bar{x}, \bar{y})) \subseteq I$, then there exists $h' \supseteq h_{|\bar{x}}$ such that $h'(\psi(\bar{x}, \bar{z})) \subseteq I$.[6] The instance $I$ satisfies a set $\Sigma$ of TGDs, written $I \models \Sigma$, if $I \models \sigma$ for each $\sigma \in \Sigma$.

**Query Answering under TGDs.** The main reasoning task under TGD-based languages is *conjunctive query answering*. Given a database $D$ and a set $\Sigma$ of TGDs, a *model* of $D$ and $\Sigma$ is an instance $I$ such that $I \supseteq D$ and $I \models \Sigma$. Let $\mathsf{mods}(D, \Sigma)$ be the set of all models of $D$ and $\Sigma$. The *certain answers* to a CQ $q$ w.r.t. $D$ and $\Sigma$ is

$$\mathsf{cert}(q, D, \Sigma) \ := \ \bigcap \{q(I) \mid I \in \mathsf{mods}(D, \Sigma)\}.$$

Our main task is to compute the certain answers to a CQ w.r.t. a database and a set of TGDs from a certain class $\mathsf{C}$ of TGDs; concrete classes of TGDs are discussed below. As is customary when studying the complexity of this problem, we focus on its decision version:

| | |
|---|---|
| PROBLEM : | $\mathsf{CQAns}(\mathsf{C})$ |
| INPUT : | A database $D$, a set $\Sigma \in \mathsf{C}$ of TGDs, a CQ $q(\bar{x})$, and a tuple $\bar{c} \in \mathsf{dom}(D)^{\lvert \bar{x} \rvert}$. |
| QUESTION : | Is it the case that $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$? |

We consider the standard complexity measures: *combined complexity* and *data complexity*, where the latter measures the complexity of the problem assuming that the set of TGDs and the CQ are fixed.

A useful algorithmic tool for tackling the above problem is the well-known *chase procedure*; see, e.g., [10, 15, 19, 23]. We start by defining a single chase step. Let $I$ be an instance and $\sigma = \phi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z}\, \psi(\bar{x}, \bar{z})$ a TGD. We say that $\sigma$ is *applicable* w.r.t. $I$ if there exists a homomorphism $h$ such that $h(\phi(\bar{x}, \bar{y})) \subseteq I$. In this case, *the result of applying $\sigma$ over $I$ with $h$* is the instance $J = I \cup \{h'(\psi(\bar{x}, \bar{z}))\}$, where $h'(z)$ is a fresh null not occurring in $I$, for every $z \in \bar{z}$. Such a single chase step is denoted $I\langle \sigma, h \rangle J$. Consider now an instance $I$, and a set $\Sigma$ of TGDs. A *chase sequence for $I$ under $\Sigma$* is a sequence $(I_i \langle \sigma_i, h_i \rangle I_{i+1})_{i \geq 0}$ of chase steps such that: (1) $I = I_0$; (2) for each $i \geq 0$, $\sigma_i \in \Sigma$; and (3) $\bigcup_{i \geq 0} I_i \models \Sigma$. We call $\bigcup_{i \geq 0} I_i$ the *result* of this chase sequence, which always exists. Although the result of a chase sequence is not necessarily unique (up to isomorphism), each such result is equally useful for query answering purposes, since it can be homomorphically embedded into every other result. Hence, we denote by $\mathsf{chase}(I, \Sigma)$ the result of an arbitrary chase sequence for $I$ under $\Sigma$. The following is a classical result:

Proposition 2.1. *Given a database $D$, a set $\Sigma$ of TGDs, and a CQ $q$,* $\mathsf{cert}(q, D, \Sigma) = q(\mathsf{chase}(D, \Sigma))$.

## 3 THE LOGICAL CORE OF VADALOG

A crucial component of the Vadalog system is its reasoning engine, which in turn is built around the Vadalog language, a general-purpose formalism for knowledge representation and reasoning. The logical core of this language is the well-behaved class of warded sets of TGDs [3, 17].

**An Intuitive Description.** Wardedness applies a syntactic restriction on how certain "dangerous" variables of a set of TGDs are used. These are body variables that can be unified

[6] By abuse of notation, we sometimes treat a tuple of variables as a set of variables, and a conjunction of atoms as a set of atoms.

with a null during the chase, and that are also propagated to the head. For example, given

$$P(x) \to \exists z\, R(x,z) \quad \text{and} \quad R(x,y) \to P(y)$$

the variable $y$ in the body of the second TGD is dangerous. Indeed, once the chase applies the first TGD, an atom of the form $R(\_,\bot)$ is generated, where $\bot$ is a null value, and then the second TGD is triggered with the variable $y$ being unified with $\bot$ that is propagated to the obtained atom $P(\bot)$. It has been observed that the liberal use of dangerous variables leads to a prohibitively high computational complexity of the main reasoning tasks, in particular of CQ answering [10]. The main goal of wardedness is to limit the use of dangerous variables with the aim of taming the way that null values are propagated during the execution of the chase procedure. This is achieved by posing the following conditions:

(1) all the dangerous variables should appear together in a single body atom $\alpha$, called a ward, and
(2) $\alpha$ can share only harmless variables with the rest of the body, i.e., variables that unify only with constants.

We proceed to formalize the above description.

**The Formal Definition.** We first need some auxiliary notions. The set of positions of a schema $\mathbf{S}$, denoted $\mathsf{pos}(\mathbf{S})$, is defined as $\{R[i] \mid R/n \in \mathbf{S},\ \text{with } n \geq 1,\ \text{and } i \in [n]\}$. Given a set $\Sigma$ of TGDs, we write $\mathsf{pos}(\Sigma)$ instead of $\mathsf{pos}(\mathsf{sch}(\Sigma))$. The set of *affected positions* of $\mathsf{sch}(\Sigma)$, denoted $\mathsf{aff}(\Sigma)$, is inductively defined as follows:

- if there exists $\sigma \in \Sigma$ and a variable $x \in \mathsf{var}_{\exists}(\sigma)$ at position $\pi$, then $\pi \in \mathsf{aff}(\Sigma)$, and
- if there exists $\sigma \in \Sigma$ and a variable $x \in \mathsf{front}(\sigma)$ in the body of $\sigma$ only at positions of $\mathsf{aff}(\Sigma)$, and $x$ appears in the head of $\sigma$ at position $\pi$, then $\pi \in \mathsf{aff}(\Sigma)$.

Let $\mathsf{nonaff}(\Sigma) = \mathsf{pos}(\Sigma) \setminus \mathsf{aff}(\Sigma)$. We can now classify the variables in the body of a TGD into harmless, harmful, and dangerous. Fix a TGD $\sigma \in \Sigma$ and a variable $x$ in $\mathsf{body}(\sigma)$:

- $x$ is *harmless* if at least one occurrence of it appears in $\mathsf{body}(\sigma)$ at a position of $\mathsf{nonaff}(\Sigma)$,
- $x$ is *harmful* if it is not harmless, and
- $x$ is *dangerous* if it is harmful and belongs to $\mathsf{front}(\sigma)$.

We are now ready to formally introduce wardedness.

*Definition 3.1* (**Wardedness**). A set $\Sigma$ of TGDs is *warded* if, for each $\sigma \in \Sigma$, there are no dangerous variables in $\mathsf{body}(\sigma)$, or there is an atom $\alpha \in \mathsf{body}(\sigma)$, called a *ward*, such that: (i) all the dangerous variables in $\mathsf{body}(\sigma)$ occur in $\alpha$, and (ii) each variable of $\mathsf{var}(\alpha) \cap \mathsf{var}(\mathsf{body}(\sigma) \setminus \{\alpha\})$ is harmless. Let $\mathsf{WARD}$ be the class of all (finite) warded sets of TGDs. ■

The problem of CQ answering under warded sets of TGDs has been recently investigated in [3, 17]:

Proposition 3.2. $\mathsf{CQAns}(\mathsf{WARD})$ *is* ExpTime-*complete in combined complexity, and* PTime-*complete in data complexity.*

Note that [3, 17] deals only with data complexity. However, the same algorithm provides an ExpTime upper bound in combined complexity. The lower bounds are inherited from Datalog since a set of Datalog rules (seen as TGDs) is warded.

**A Key Application.** One of the distinctive features of wardedness, which is crucial for the purposes of the Vadalog system, is the fact that it can express every SPARQL query under the OWL 2 QL direct semantics entailment regime, which is inherited from the OWL 2 direct semantics entailment regime; for details, see [2, 17, 22]. Recall that SPARQL is the standard language for querying the Semantic Web,[7] while OWL 2 QL is a prominent profile of OWL 2.[8]

## 4 LIMITING RECURSION

We now focus on our main research question: can we limit the recursion allowed by wardedness in order to obtain a formalism that provides a convenient syntax for expressing useful recursive statements, and at the same time achieve space-efficiency? The above question has been extensively studied in the 1980s for Datalog programs, with *linear Datalog* being a key fragment that achieves a good balance between expressivity and complexity; see, e.g., [24, 25]. A Datalog program $\Sigma$ is *linear* if, for each rule in $\Sigma$, its body contains at most one intensional predicate, i.e., a predicate that appears in the head of at least one rule of $\Sigma$. In other words, linear Datalog allows only for linear recursion, which is able to express many real-life recursive queries. However, for our purposes, linear recursion does not provide the convenient syntax that we are aiming at. After analyzing several real-life examples of warded sets of TGDs, provided by our industrial partners, we observed that the employed recursion goes beyond linear recursion. On the other hand, most of the examples coming from our industrial partners use recursion in a restrictive way: each TGD has at most one body atom whose predicate is mutually recursive with a predicate occurring in the head of the TGD. Interestingly, this more liberal version of linear recursion has been already investigated in the context of Datalog, and it is known as *piece-wise linear*; see, e.g., [1]. Does this type of recursion lead to the space-efficient fragment of warded sets of TGDs that we are looking for? The rest of this section is devoted to showing this rather involved result.

Let us start by formally defining the class of piece-wise linear sets of TGDs. To this end, we need to define when two predicates are mutually recursive, which in turn relies on the well-known notion of the predicate graph. The *predicate graph* of a set $\Sigma$ of TGDs, denoted $\mathsf{pg}(\Sigma)$, is a directed graph

[7]http://www.w3.org/TR/rdf-sparql-query
[8]https://www.w3.org/TR/owl2-overview/

$(V, E)$, where $V = \text{sch}(\Sigma)$, and there exists an edge from a predicate $P$ to a predicate $R$, i.e., $(P, R) \in E$, iff there exists a TGD $\sigma \in \Sigma$ such that $P$ occurs in $\text{body}(\sigma)$ and $R$ occurs in $\text{head}(\sigma)$. Two predicates $P, R \in \text{sch}(\Sigma)$ are *mutually recursive* (w.r.t. $\Sigma$) if there exists a cycle in $\text{pg}(\Sigma)$ that contains both $P$ and $R$ (i.e., $R$ is reachable from $P$, and vice versa). We are now ready to define piece-wise linearity for TGDs.

*Definition 4.1* (**Piece-wise Linearity**). A set $\Sigma$ of TGDs is *piece-wise linear* if, for each TGD $\sigma \in \Sigma$, there exists at most one atom in $\text{body}(\sigma)$ whose predicate is mutually recursive with a predicate in $\text{head}(\sigma)$. We write $\mathsf{PWL}$ for the class of (finite) piece-wise linear sets of TGDs. ∎

The main result of this section follows:

Theorem 4.2. $\mathsf{CQAns}(\mathsf{WARD} \cap \mathsf{PWL})$ *is* PSpace-*complete in combined, and* NLogSpace-*complete in data complexity.*

The lower bounds are inherited from linear Datalog. The difficult task is to establish the upper bounds. This relies on a novel notion of proof tree, which is of independent interest. As we shall see, our notion of proof tree leads to space-bounded algorithms that allow us to show the upper bounds in Theorem 4.2, and also re-establish in a transparent way the upper bounds in Proposition 3.2. Moreover, in Section 6, we are going to use proof trees for studying the relative expressive power of (piece-wise linear) warded sets of TGDs.

## 4.1 Query Answering via Proof Trees

It is known that given a CQ $q$ and a set $\Sigma$ of TGDs, we can unfold $q$ using the TGDs of $\Sigma$ into an infinite union of CQs $q_\Sigma$ such that, for every database $D$, $\text{cert}(q, D, \Sigma) = q_\Sigma(D)$; see, e.g., [16, 21]. Let us clarify that in our context, an unfolding, which is essentially a resolution step, is more complex than in the context of Datalog due to the existentially quantified variables in the head of TGDs. The intention underlying our notion of proof tree is to encode in a tree the sequence of CQs, generated during the unfolding of $q$ with $\Sigma$, that leads to a certain CQ $q'$ of $q_\Sigma$ in such a way that each intermediate CQ, as well as $q'$, is carefully decomposed into smaller subqueries that form the nodes of the tree, while the root corresponds to $q$ and the leaves to $q'$. As we shall see, if we focus on well-behaved classes of TGDs such as (piece-wise linear) warded sets of TGDs, we can establish upper bounds on the size of these subqueries, which in turn allow us to devise space-bounded algorithms for query answering. In what follows, we define the notion of proof tree (Definition 4.7), and establish its correspondence with query answering (Theorem 4.8). To this end, we need to introduce the main building blocks of a proof tree: chunk-based resolution (Definition 4.3), a query decomposition technique (Definition 4.5), and the notion of specialization for CQs (Definition 4.6).

**Chunk-based Resolution.** Let $A$ and $B$ be non-empty sets of atoms that mention only constants and variables. The sets $A$ and $B$ *unify* if there is a substitution $\gamma$, which is the identity on $\mathbf{C}$, called *unifier for $A$ and $B$*, such that $\gamma(A) = \gamma(B)$. A *most general unifier* (MGU) for $A$ and $B$ is a unifier for $A$ and $B$, denoted $\gamma_{A,B}$, such that, for each unifier $\gamma$ for $A$ and $B$, $\gamma = \gamma' \circ \gamma_{A,B}$ for some substitution $\gamma'$. Notice that if two sets of atoms unify, then there exists always a MGU, which is unique (modulo variable renaming).

Given a CQ $q(\bar{x})$ and a set of atoms $S \subseteq \text{atoms}(q)$, we say that a variable $y \in \text{var}(S)$ is *shared*, if $y \in \bar{x}$ or $y \in \text{var}(\text{atoms}(q) \setminus S)$. A *chunk unifier* of $q$ with a TGD $\sigma$ (where $q$ and $\sigma$ do not share variables) is a triple $(S_1, S_2, \gamma)$, where $\emptyset \subset S_1 \subseteq \text{atoms}(q)$, $\emptyset \subset S_2 \subseteq \text{head}(\sigma)$, and $\gamma$ is a unifier for $S_1$ and $S_2$ such that, for each $x \in \text{var}(S_2) \cap \text{var}_\exists(\sigma)$,

(1) $\gamma(x) \notin \mathbf{C}$, i.e., $\gamma(x)$ is not constant, and
(2) for every variable $y$ different from $x$, $\gamma(x) = \gamma(y)$ implies $y$ occurs in $S_1$ and is not shared.

The chunk unifier $(S_1, S_2, \gamma)$ is *most general* (MGCU) if $\gamma$ is an MGU for $S_1$ and $S_2$. Notice that the variables of $\text{var}_\exists(\sigma)$ occurring in $S_2$ unify (via $\gamma$) only with non-shared variables of $S_1$. This ensures that $S_1$ is a "chunk" of $q$ that can be resolved as a whole via $\sigma$ using $\gamma$.[9] Without the additional conditions on the substitution $\gamma$, we may get unsound resolution steps. Consider, e.g., the CQ and TGD

$$Q(x) \leftarrow R(x, y), S(y) \quad \text{and} \quad P(x') \rightarrow \exists y'\, R(x', y').$$

Resolving the atom $R(x, y)$ in the query with the given TGD using $\gamma = \{x \mapsto x', y \mapsto y'\}$ would be an unsound step since the shared variable $y$ is lost. This is because $y'$ is unified with the shared variable $y$. On the other hand, $R(x, y), S(y)$ can be resolved with the TGD $\sigma = P(x') \rightarrow \exists y'\, R(x', y'), S(y')$ using $\gamma$; in fact, the chunk unifier is $(\text{atoms}(q), \text{head}(\sigma), \gamma)$.

*Definition 4.3* (**Chunk-based Resolution**). Let $q(\bar{x})$ be a CQ and $\sigma$ a TGD. A *$\sigma$-resolvent* of $q$ is a CQ $q'(\gamma(\bar{x}))$ with $\text{body}(q') = \gamma((\text{atoms}(q) \setminus S_1) \cup \text{body}(\sigma))$ for a MGCU $(S_1, S_2, \gamma)$ of $q$ with $\sigma$. ∎

**Query Decomposition.** As discussed above, the purpose of a proof tree is to encode a finite branch of the unfolding of a CQ $q$ with a set $\Sigma$ of TGDs, which is obtained by applying chunk-based resolution. Such a branch is a sequence $q_0, \ldots, q_n$ of CQs, where $q = q_0$, while, for each $i \in [n]$, $q_i$ is a $\sigma$-resolvent of $q_{i-1}$ for some $\sigma \in \Sigma$. Here is a simple example, which will serve as a running example in the rest of the section, that illustrates the notion of unfolding.

*Example 4.4.* Consider the set $\Sigma$ of TGDs consisting of

$$R(x) \quad \rightarrow \quad \exists y\, T(y, x)$$

[9] A similar notion known as piece unifier has been defined in [21].

$$T(x, y), S(y, z) \quad \rightarrow \quad T(x, z)$$
$$T(x, y), P(y) \quad \rightarrow \quad G()$$

and the CQ that simply asks whether $G()$ is entailed, i.e., the CQ $Q \leftarrow G()$. Since the unfolding of $q$ with $\Sigma$ should give the correct answer for *every* input database, and thus for databases of the form

$$\{R(c^{n-1}), S(c^{n-1}, c^{n-2}), \ldots, S(c^2, c^1), P(c^1)\} \quad \text{for some } n > 1,$$

one of its branches should be $q = q_0, q_1, \ldots, q_n$, where

$$q_1 \ = \ Q \leftarrow T(x, y^1), P(y^1)$$

obtained by resolving $q_0$ using the third TGD,

$$q_i \ = \ Q \leftarrow T(x, y^i), S(y^i, y^{i-1}), \ldots, S(y^2, y^1), P(y^1),$$

for $i \in \{2, \ldots, n-1\}$, obtained by resolving $q_{i-1}$ using the second TGD, and finally

$$q_n \ = \ Q \leftarrow R(y^{n-1}), S(y^{n-1}, y^{n-2}), \ldots, S(y^2, y^1), P(y^1)$$

obtained by resolving $q_{n-1}$ using the first TGD. ■

At this point, one may think that the proof tree that encodes the branch $q_0, \ldots, q_n$ of the unfolding of $q$ with $\Sigma$ is the finite labeled path $v_0, \ldots, v_n$, where each $v_i$ is labeled by $q_i$. However, another crucial goal of such a proof tree, which is not achieved via the naive path encoding, is to split each resolvent $q_i$, for $i > 0$, into smaller subqueries $q_i^1, \ldots, q_i^{n_i}$, which are essentially the children of $q_i$, in such a way that they can be processed independently by resolution. The crux of this encoding is that it provides us with a mechanism for keeping the CQs that must be processed by resolution small. It should be clear from Example 4.4 that by following the naive path encoding, without splitting the resolvents into smaller subqueries, we may get CQs of unbounded size.

The key question here is how a CQ $q$ can be decomposed into subqueries that can be processed independently. The subtlety is that, after splitting $q$, occurrences of the same variable may be separated into different subqueries. Thus, we need a way to ensure that a variable in $q$, which appears in different subqueries after the splitting, is indeed treated as the same variable, i.e., it has the same meaning. We deal with this issue by restricting the set of variables in $q$ of which occurrences can be separated during the splitting step. In particular, we can only separate occurrences of an output variable. This relies on the convention that output variables correspond to fixed constant values of $\mathbf{C}$, and thus their name is "frozen" and never renamed by subsequent resolution steps. Hence, we can separate occurrences of an output variable into different subqueries, i.e., different branches of the proof tree, without losing the connection between them.

Summing up, the idea underlying query decomposition is to split the CQ at hand into smaller queries that keep together all the occurrences of a non-output variable, but with the freedom of separating occurrences of an output variable.

*Definition 4.5* (**Query Decomposition**). Given a CQ $q(\bar{x})$, a *decomposition* of $q$ is a set of CQs $\{q_1(\bar{y}_1), \ldots, q_n(\bar{y}_n)\}$, where $n \geq 1$ and $\bigcup_{i \in [n]} \mathsf{atoms}(q_i) = \mathsf{atoms}(q)$, such that, for each $i \in [n]$: (1) $\bar{y}_i$ is the restriction of $\bar{x}$ on the variables in $q_i$, and (2) for every $\alpha, \beta \in \mathsf{atoms}(q)$, if $\alpha \in \mathsf{atoms}(q_i)$ and $\mathsf{var}(\alpha) \cap \mathsf{var}(\beta) \not\subseteq \bar{x}$, then $\beta \in \mathsf{atoms}(q_i)$. ■

**Query Specialization.** From the above discussion, one expects that a proof tree of a CQ $q$ w.r.t. a set $\Sigma$ of TGDs can be constructed by starting from $q$, which is the root, and applying two steps: resolution and decomposition. Unfortunately, this is not enough for our purposes as we may run into the following problem: some of the subqueries will mistakenly remain large since we have no way to realize that a non-output variable corresponds to a fixed constant value, which in turn allows us to "freeze" its name and separate different occurrences of it during the decomposition step. This is illustrated by Example 4.4. Observe that the size of the CQs $\{q_i\}_{i>0}$ grows arbitrarily, while our query decomposition has no effect on them since they are Boolean queries, i.e., queries without output variables, and thus, we cannot split them into smaller subqueries. The above issue can be solved by having an intermediate step between resolution and decomposition, the so-called specialization step. A specialization of a CQ is obtained by converting some non-output variables of it into output variables, while keeping their name, or taking the name of an existing output variable.

*Definition 4.6* (**Query Specialization**). Let $q(\bar{x})$ be a CQ with $\mathsf{atoms}(q) = \{\alpha_1, \ldots, \alpha_n\}$. A *specialization* of $q$ is a CQ

$$Q(\bar{x}, \bar{y}) \ \leftarrow \rho_{\bar{z}}(\alpha_1, \ldots, \alpha_n)$$

where $\bar{y}, \bar{z}$ are (possibly empty) disjoint tuples of non-output variables of $q$, and $\rho_{\bar{z}}$ is a substitution from $\bar{z}$ to $\bar{x} \cup \bar{y}$. ■

Consider, for example, the CQ $q_1$ from Example 4.4

$$Q \leftarrow T(x, y^1), P(y^1)$$

obtained by resolving $q = q_0$ using the third TGD. The query decomposition cannot split it onto smaller subqueries since the variable $y^1$ is a non-output variable, and thus, all its occurrences should be kept together. We can consider the following specialization of $q_1$

$$Q(y^1) \leftarrow T(x, y^1), P(y^1),$$

which simply converts $y^1$ into an output variable, and now by query decomposition we can split it into the atomic queries

$$Q(y^1) \leftarrow T(x, y^1) \qquad Q(y^1) \leftarrow P(y^1),$$

which represent the original query $q_1$.

**Proof Trees.** We are now ready to introduce our new notion of proof tree. We first explain the high-level idea by exploiting our running example. Consider the set $\Sigma$ of TGDs and the CQ $q$ from Example 4.4. The branch $q_0, \ldots, q_n$ of the

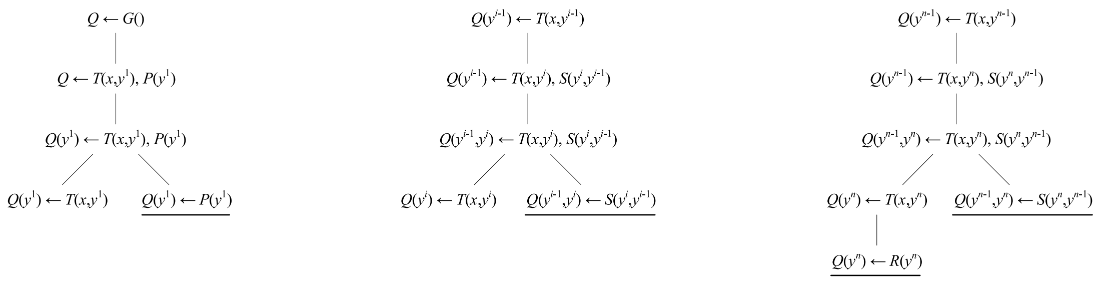
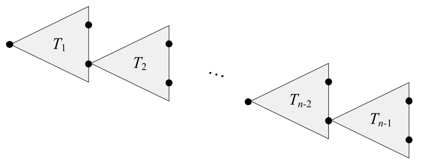

$Q \leftarrow G()$
$Q \leftarrow T(x,y^1), P(y^1)$
$Q(y^1) \leftarrow T(x,y^1), P(y^1)$
$Q(y^1) \leftarrow T(x,y^1)$
$\underline{Q(y^1) \leftarrow P(y^1)}$
$Q(y^{i-1}) \leftarrow T(x,y^{i-1})$
$Q(y^{i-1}) \leftarrow T(x,y^i), S(y^i,y^{i-1})$
$Q(y^{i-1},y^i) \leftarrow T(x,y^i), S(y^i,y^{i-1})$
$Q(y^i) \leftarrow T(x,y^i)$
$\underline{Q(y^{i-1},y^i) \leftarrow S(y^i,y^{i-1})}$
$Q(y^{n-1}) \leftarrow T(x,y^{n-1})$
$Q(y^{n-1}) \leftarrow T(x,y^n), S(y^n,y^{n-1})$
$Q(y^{n-1},y^n) \leftarrow T(x,y^n), S(y^n,y^{n-1})$
$Q(y^n) \leftarrow T(x,y^n)$
$\underline{Q(y^{n-1},y^n) \leftarrow S(y^n,y^{n-1})}$
$\underline{Q(y^n) \leftarrow R(y^n)}$

**Figure 1: Partial trees of the proof tree that encodes the branch $q = q_0, \dots, q_n$ of the unfolding of $q$ with $\Sigma$ from Example 4.4.**

unfolding of $q$ with $\Sigma$ given in Example 4.4 is encoded via a proof tree of the form



where each $T_i$, for $i \in [n-1]$, is a rooted tree with only two leaf nodes. The actual trees are depicted in Figure 1; the left one is $T_1$, the middle one is $T_i$ for $i \in \{2, \dots, n-2\}$, while the right one is $T_{n-1}$. For each $i \in [n-1]$, the child of the root of $T_i$ is obtained via resolution, then we specialize it by converting the variable $y^i$ into an output variable, and then we decompose the specialized CQ into two subqueries. In $T_{n-1}$, we also apply an additional resolution step in order to obtain the leaf node $Q(y^{n-1}) \leftarrow R(y^{n-1})$. The underlined CQs are actually the subqueries that represent the CQ $q_n$ of the unfolding. Indeed, the conjunction of the atoms occurring in the underlined CQs is precisely the CQ $q_n$.

We proceed to give the formal definition. Given a partition $\pi = \{S_1, \dots, S_m\}$ of a set of variables, we write $\mathrm{eq}_\pi$ for the substitution that maps the variables of $S_i$ to the same variable $x_i$, where $x_i$ is a distinguished element of $S_i$. We should not forget the convention that output variables cannot be renamed, and thus, a resolution step should use a MGCU that preserves the output variables. In particular, given a CQ $q$ and a TGD $\sigma$, a $\sigma$-resolvent of $q$ is called *IDO* if the underlying MGCU uses a substitution that is the identity on the output variables of $q$ (hence the name IDO). Finally, given a TGD $\sigma$ and some arbitrary object $o$ (e.g., $o$ can be the node of a tree, or an integer number), we write $\sigma_o$ for the TGD obtained by renaming each variable $x$ in $\sigma$ into $x_o$. This is a simple mechanism for uniformly renaming the variables of a TGD in order to avoid undesirable clatter among variables during a resolution step.

*Definition 4.7* (**Proof Tree**). Let $q(\bar{x})$ be a CQ with $\mathrm{atoms}(q) = \{\alpha_1, \dots, \alpha_n\}$, and $\Sigma$ a set of TGDs. A *proof tree* of $q$ w.r.t. $\Sigma$ is a triple $\mathcal{P} = (T, \lambda, \pi)$, where $T = (V, E)$ is a finite rooted tree, $\lambda$ a labeling function that assigns a CQ to each node of $T$, and $\pi$ a partition of $\bar{x}$, such that, for $v \in V$:

(1) If $v$ is the root node of $T$, then $\lambda(v)$ is the CQ $Q(\mathrm{eq}_\pi(\bar{x})) \leftarrow \mathrm{eq}_\pi(\alpha_1, \dots, \alpha_m)$.
(2) If $v$ has only one child $u$, $\lambda(u)$ is an IDO $\sigma_v$-resolvent of $\lambda(v)$ for some $\sigma \in \Sigma$, or a specialization of $\lambda(v)$.
(3) If $v$ has the children $u_1, \dots, u_k$ for $k > 1$, then $\{\lambda(u_1), \dots, \lambda(u_k)\}$ is a decomposition of $\lambda(v)$.

Assuming that $v_1, \dots, v_m$ are the leaf nodes of $T$, the *CQ induced by* $\mathcal{P}$ is defined as

$$Q(\mathrm{eq}_\pi(\bar{x})) \ \leftarrow \ \alpha_1, \dots, \alpha_\ell,$$

where $\{\alpha_1, \dots, \alpha_\ell\} = \bigcup_{i \in [m]} \mathrm{atoms}(\lambda(v_i))$. ∎

The purpose of the partition $\pi$ is to indicate that some output variables correspond to the same constant value – this is why variables in the same set of $\pi$ are unified via the substitution $\mathrm{eq}_\pi$. This unification step is crucial in order to safely use, in subsequent resolution steps, substitutions that are the identity on the output variables. If we omit this initial unification step, we may lose important resolution steps, and thus being incomplete for query answering purposes. The main result of this section, which exposes the connection between proof trees and CQ answering, follows. By abuse of notation, we write $\mathcal{P}$ for the CQ induced by $\mathcal{P}$.

Theorem 4.8. *Consider a database $D$, a set $\Sigma$ of TGDs, a CQ $q(\bar{x})$, and $\bar{c} \in \mathrm{dom}(D)^{|\bar{x}|}$. The following are equivalent:*

*(1) $\bar{c} \in \mathrm{cert}(q, D, \Sigma)$.*
*(2) There is a proof tree $\mathcal{P}$ of $q$ w.r.t. $\Sigma$ such that $\bar{c} \in \mathcal{P}(D)$.*

The proof of the above result relies on the soundness and completeness of chunk-based resolution. Given a set $\Sigma$ of TGDs and a CQ $q(\bar{x})$, by exhaustively applying chunk-based resolution, we can construct a (possibly infinite) union of CQs $q_\Sigma$ such that, for every database $D$, $\mathrm{cert}(q, D, \Sigma) = q_\Sigma(D)$; implicit in [16, 21]. In other words, given a tuple $\bar{c} \in \mathrm{dom}(D)^{|\bar{x}|}$, $\bar{c} \in \mathrm{cert}(q, D, \Sigma)$ iff there exists a CQ $q'(\bar{x})$ in $q_\Sigma$ such that $\bar{c} \in q'(D)$. It is now not difficult to show that the latter statement is equivalent to the existence of a proof tree $\mathcal{P}$ of $q$ w.r.t. $\Sigma$ such that $\bar{c} \in \mathcal{P}(D)$, and the claim follows.

## 4.2 Well-behaved Proof Trees

Theorem 4.8 states that checking whether a tuple $\bar{c}$ is a certain answer boils down to deciding whether there exists a proof tree $\mathcal{P}$ such that $\bar{c}$ is an answer to the CQ induced by $\mathcal{P}$ over the given database. Of course, the latter is an undecidable problem in general. However, if we focus on (piece-wise linear) warded sets of TGDs, it suffices to check for the existence of a well-behaved proof tree with certain syntactic properties, which in turn allows us to devise a decision procedure. We proceed to make this more precise. For technical clarity, we assume, w.l.o.g., TGDs with only one atom in the head since we can always convert a warded set of TGDs into one with single-atom heads, while certain answers are preserved; for the transformation see, e.g., [12].

**Piece-wise Linear Warded Sets of TGDs.** For piece-wise linear warded sets of TGDs, we can strengthen Theorem 4.8 by focussing on a certain class of proof trees that enjoy two syntactic properties: (i) they have a path-like structure, and (ii) the size of the CQs that label their nodes is bounded by a polynomial. The first property is formalized via linear proof trees. Let $\mathcal{P} = (T, \lambda, \pi)$, where $T = (V, E)$, be a proof tree of a CQ $q$ w.r.t. a set $\Sigma$ of TGDs. We call $\mathcal{P}$ *linear* if, for each node $v \in V$, there exists at most one node $u \in V$ such that $(v, u) \in E$ and $u$ is not a leaf in $T$, i.e., $v$ has at most one child that is not a leaf. For example, the proof tree given above, which consists of the partial trees depicted in Figure 1, is linear. The second property relies on the notion of node-width of a proof tree. The *node-width* of $\mathcal{P}$ is

$$\mathsf{nwd}(\mathcal{P}) \ := \ \max_{v \in V}\{|\lambda(v)|\},$$

i.e., the size of the largest CQ that labels a node of $T$.

Before we strengthen Theorem 4.8, let us define the polynomial that will allow us to bound the node-width of the linear proof trees that we need to consider. This polynomial relies on the notion of predicate level. Consider a set $\Sigma$ of TGDs. For a predicate $P \in \mathsf{sch}(\Sigma)$, we write $\mathsf{rec}(P)$ for the set of predicates of $\mathsf{sch}(\Sigma)$ that are mutually recursive to $P$ according to $\mathsf{pg}(\Sigma) = (V, E)$. Let $\ell_\Sigma \colon \mathsf{sch}(\Sigma) \to \mathbb{N}$ be the unique function that satisfies

$$\ell_\Sigma(P) \ = \ \max\{\ell_\Sigma(R) \mid (R, P) \in E, R \notin \mathsf{rec}(P)\} + 1,$$

with $\ell_\Sigma(P)$ being the *level* (w.r.t. $\Sigma$) of $P$, for each $P \in \mathsf{sch}(\Sigma)$. We define the polynomial

$$f_{\mathsf{WARD} \cap \mathsf{PWL}}(q, \Sigma) := (|q| + 1) \cdot \max_{P \in \mathsf{sch}(\Sigma)}\{\ell_\Sigma(P)\} \cdot \max_{\sigma \in \Sigma}\{|\mathsf{body}(\sigma)|\}.$$

We can now strengthen Theorem 4.8. Let us first clarify that, in the case of piece-wise linear warded sets of TGDs, apart from only one atom in the head, we also assume, w.l.o.g., that the level of a predicate in the body of TGD $\sigma$ is $k$ or $k-1$, where $k$ is the level of the predicate in the head of $\sigma$.

THEOREM 4.9. *Consider a database $D$, a set $\Sigma \in \mathsf{WARD} \cap \mathsf{PWL}$ of TGDs, a CQ $q(\bar{x})$, and $\bar{c} \in \mathsf{dom}(D)^{|\bar{x}|}$. The following are equivalent:*

*(1) $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$.*
*(2) There is a linear proof tree $\mathcal{P}$ of $q$ w.r.t. $\Sigma$ with $\mathsf{nwd}(\mathcal{P}) \leq f_{\mathsf{WARD} \cap \mathsf{PWL}}(q, \Sigma)$ such that $\bar{c} \in \mathcal{P}(D)$.*

**Warded sets of TGDs.** Now, in the case of arbitrary warded sets of TGDs, we cannot focus only on linear proof trees. Nevertheless, we can still bound the node-width of the proof trees that we need to consider by the following polynomial, which, unsurprisingly, does not rely anymore on the notion of predicate level:

$$f_{\mathsf{WARD}}(q, \Sigma) := 2 \cdot \max\left\{|q|, \max_{\sigma \in \Sigma}\{|\mathsf{body}(\sigma)|\}\right\}.$$

Theorem 4.8 can be strengthened as follows:

THEOREM 4.10. *Consider a database $D$, a set $\Sigma \in \mathsf{WARD}$ of TGDs, a CQ $q(\bar{x})$, and $\bar{c} \in \mathsf{dom}(D)^{|\bar{x}|}$. The following are equivalent:*

*(1) $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$.*
*(2) There exists a proof tree $\mathcal{P}$ of $q$ w.r.t. $\Sigma$ with $\mathsf{nwd}(\mathcal{P}) \leq f_{\mathsf{WARD}}(q, \Sigma)$ such that $\bar{c} \in \mathcal{P}(D)$.*

**A Proof Sketch.** Let us now provide some details on how Theorems 4.9 and 4.10 are shown. For both theorems, (2) implies (1) readily follows from Theorem 4.8. We thus focus on the other direction. The main ingredients of the proof can be described as follows:

- We introduce the auxiliary notion of chase tree, which can be seen as a concrete instantiation of a proof tree. It serves as an intermediate structure between proof trees and chase derivations, which allows us to use the chase as our underlying technical tool. Note that the notions of linearity and node-width can be naturally defined for chase trees.
- We then show that, if the given tuple of constants $\bar{c}$ is a certain answer to the given CQ $q$ w.r.t. the given database $D$ and (piece-wise linear) warded set $\Sigma$ of TGDs, then there exists a (linear) chase tree for the image of $q$ to $\mathsf{chase}(D, \Sigma)$ such that its node-width respects the bounds given in the above theorems (Lemma 4.12).
- We finally show that the existence of a (linear) chase tree for the image of $q$ to $\mathsf{chase}(D, \Sigma)$ with node-width at most $m$ implies the existence of a (linear) proof tree $\mathcal{P}$ of $q$ w.r.t. $\Sigma$ with node-width at most $m$ such that $\bar{c} \in \mathcal{P}(D)$ (Lemma 4.13).

Let us make the above description more formal. In order to introduce the notion of chase tree, we first need to recall the notion of chase graph, then introduce the notion of unraveling of the chase graph, and finally introduce the notions of unfolding and decomposition for sets of atoms in the unraveling of the chase graph.

Fix a chase sequence $\delta = (I_i\langle\sigma_i, h_i\rangle I_{i+1})_{i\geq 0}$ for a database $D$ under a set $\Sigma$ of TGDs. The *chase graph* for $D$ and $\Sigma$ (w.r.t. $\delta$) is a directed edge-labeled graph $\mathcal{G}^{D,\Sigma} = (V, E, \lambda)$, where $V = \mathsf{chase}(D, \Sigma)$, and an edge $(\alpha, \beta)$ labeled with $(\sigma_k, h_k)$ belongs to $E$ iff $\alpha \in h_k(\mathsf{body}(\sigma_k))$ and $\beta \in I_{k+1} \setminus I_k$, for some $k \geq 0$. In other words, $\alpha$ has an edge to $\beta$ if $\beta$ is derived using $\alpha$, and if $\beta$ is new in the sense that it has not been derived before. Notice that $\mathcal{G}^{D,\Sigma}$ has no directed cycles. Notice also that $\mathcal{G}^{D,\Sigma}$ depends on $\delta$ – however, we can assume a fixed sequence $\delta$ since, as discussed in Section 2, every chase sequence is equally useful for our purposes.

We now discuss the notion of unraveling of the chase graph; due to space reasons, we keep this discussion informal. Given a set $\Theta \subseteq \mathsf{chase}(D, \Sigma)$, the *unraveling of* $\mathcal{G}^{D,\Sigma}$ *around* $\Theta$ is a directed node- and edge-labeled forest $\mathcal{G}_{\Theta}^{D,\Sigma}$ that has a tree for each $\alpha \in \Theta$ whose branches are backward-paths in $\mathcal{G}$ from $\alpha$ to a database atom. Intuitively, $\mathcal{G}_{\Theta}^{D,\Sigma}$ is a forest-like reorganization of the atoms of $\mathsf{chase}(D, \Sigma)$ that are needed to derive $\Theta$. Due to its forest-like shape, it may contain multiple copies of atoms of $\mathsf{chase}(D, \Sigma)$. The edges between nodes are labeled by pairs $(\sigma, h)$ just like in $\mathcal{G}^{D,\Sigma}$, while the nodes are labeled by atoms and, importantly, the atoms along the paths in $\mathcal{G}^{D,\Sigma}$ may be duplicated and labeled nulls are given new names. We write $U(\mathcal{G}^{D,\Sigma}, \Theta)$ for the set of all atoms that appear as labels in $\mathcal{G}_{\Theta}^{D,\Sigma}$, and $\mathsf{succ}_{\sigma,h}(v)$ for the set of children of a node $v$ of $\mathcal{G}^{D,\Sigma}$ whose incoming edge is labeled with $(\sigma, h)$. It is important to say that there exists a homomorphism $h_\Theta$ that maps $\Theta$ to $U(\mathcal{G}^{D,\Sigma}, \Theta)$.

Let us now introduce the notions of unfolding and decomposition. For sets $\Gamma, \Gamma' \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$, $\Gamma'$ *is an unfolding of* $\Gamma$, if there are $\alpha \in \Gamma$ and $\beta_1, \ldots, \beta_k \in U(\mathcal{G}^{D,\Sigma}, \Theta)$ such that

(1) $\mathsf{succ}_{\sigma,h}(v) = \{\beta_1, \ldots, \beta_k\}$, for some $\sigma \in \Sigma$ and $h$, and some node $v$ of $\mathcal{G}_{\Theta}^{D,\Sigma}$ labeled with $\alpha$,
(2) for every null that occurs in $\alpha$, either it does not appear in $\Gamma \setminus \{\alpha\}$, or it appears in $\{\beta_1, \ldots, \beta_k\}$, and
(3) $\Gamma' = (\Gamma \setminus \{\alpha\}) \cup \{\beta_1, \ldots, \beta_k\}$.

Let $\Gamma \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$ be a non-empty set. A *decomposition* of $\Gamma$ is a set $\{\Gamma_1, \ldots, \Gamma_n\}$, where $n \geq 1$, of non-empty subsets of $\Gamma$ such that (i) $\Gamma = \bigcup_{i\in[n]} \Gamma_i$, and (ii) $i \neq j$ implies that $\Gamma_i$ and $\Gamma_j$ do not share a labeled null. We can now define the key notion of chase tree:

*Definition 4.11* (**Chase Tree**). Consider a database $D$, a set $\Sigma$ of TGDs, and a set $\Theta \subseteq \mathsf{chase}(D, \Sigma)$. A *chase tree* for $\Gamma \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$ (w.r.t. $\mathcal{G}_{\Theta}^{D,\Sigma}$) is a pair $C = (T, \lambda)$, where $T = (V, E)$ is a finite rooted tree, and $\lambda$ a labeling function that assigns a subset of $U(\mathcal{G}^{D,\Sigma}, \Theta)$ to each node of $T$, such that, for each $v \in V$:

(1) If $v$ is the root node of $T$, then $\lambda(v) = \Gamma$.
(2) If $v$ has only one child $u$, $\lambda(u)$ is an unfolding of $\lambda(v)$.
(3) If $v$ has children $u_1, \ldots, u_k$ for $k > 1$, then $\{\lambda(u_1), \ldots, \lambda(u_k)\}$ is a decomposition of $\lambda(v)$.
(4) If $v$ is a leaf node, then $\lambda(v) \subseteq D$.

The *node-width* of $C$ is $\mathsf{nwd}(C) := \max_{v\in V}\{|\lambda(v)|\}$. Moreover, we say that $C$ is *linear* if, for each $v \in V$, there exists at most one $u \in V$ such that $(v, u) \in E$ and $u$ is not a leaf. ■

We can now state our auxiliary technical lemmas. In what follows, fix a database $D$, and a set $\Sigma$ of TGDs.

Lemma 4.12. *Let* $\Theta \subseteq \mathsf{chase}(D, \Sigma)$ *and* $\Gamma \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$*:*

*(1) If* $\Sigma \in \mathsf{WARD} \cap \mathsf{PWL}$*, then there exists a linear chase tree* $C$ *for* $\Gamma$ *such that* $\mathsf{nwd}(C) \leq f_{\mathsf{WARD}\cap\mathsf{PWL}}(\Gamma, \Sigma)$.
*(2) If* $\Sigma \in \mathsf{WARD}$*, then there exists a chase tree* $C$ *for* $\Gamma$ *such that* $\mathsf{nwd}(C) \leq f_{\mathsf{WARD}}(\Gamma, \Sigma)$.

The next technical lemma exposes the connection between chase trees and proof trees:

Lemma 4.13. *Consider a set* $\Theta \subseteq \mathsf{chase}(D, \Sigma)$*, and let* $q'(\bar{x})$ *be a CQ and* $\bar{c}$ *a tuple of constants such that* $h'(\mathsf{atoms}(q')) \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta)$ *and* $h'(\bar{x}) = \bar{c}$*, for some homomorphism* $h'$*. If there is a (linear) chase tree* $C$ *for* $h'(\mathsf{atoms}(q'))$ *with* $\mathsf{nwd}(C) \leq m$*, then there is a (linear) proof tree* $\mathcal{P}$ *for* $q'$ *w.r.t.* $\Sigma$ *such that* $\mathsf{nwd}(\mathcal{P}) \leq m$ *and* $\bar{c} \in \mathcal{P}(D)$.

We can now show Theorem 4.9, while Theorem 4.10 can be shown analogously. Consider a CQ $q(\bar{x})$ and a tuple $\bar{c} \in \mathsf{dom}(D)^{|\bar{x}|}$ such that $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$. We need to show that if $\Sigma \in \mathsf{WARD} \cap \mathsf{PWL}$, then there exists a linear proof tree $\mathcal{P}$ of $q$ w.r.t. $\Sigma$ with $\mathsf{nwd}(\mathcal{P}) \leq f_{\mathsf{WARD}\cap\mathsf{PWL}}(q, \Sigma)$ such that $\bar{c} \in \mathcal{P}(D)$. By hypothesis, there is a homomorphism $h$ such that $h(\mathsf{atoms}(q)) \subseteq \mathsf{chase}(D, \Sigma)$ and $h(\bar{x}) = \bar{c}$. Let $\Theta_q$ be the set of atoms $h(\mathsf{atoms}(q))$. Recall that there is a homomorphism $h_{\Theta_q}$ that maps $\Theta_q$ to $U(\mathcal{G}^{D,\Sigma}, \Theta_q)$. Thus, the homomorphism $h' = h_{\Theta_q} \circ h$ is such that $h'(\mathsf{atoms}(q)) \subseteq U(\mathcal{G}^{D,\Sigma}, \Theta_q)$ and $h'(\bar{x}) = \bar{c}$. By Lemma 4.12, there exists a chase tree $C$ for $h'(\mathsf{atoms}(q))$ with $\mathsf{nwd}(C) \leq f_{\mathsf{WARD}\cap\mathsf{PWL}}(h'(\mathsf{atoms}(q)), \Sigma)$. By Lemma 4.13, there exists a linear proof tree $\mathcal{P}$ of $q$ w.r.t $\Sigma$ with $\mathsf{nwd}(\mathcal{P}) \leq f_{\mathsf{WARD}\cap\mathsf{PWL}}(h'(\mathsf{atoms}(q)), \Sigma) \leq f_{\mathsf{WARD}\cap\mathsf{PWL}}(q, \Sigma)$ such that $\bar{c} \in \mathcal{P}(D)$, and the claim follows.

## 4.3 Complexity Analysis

We now have all the tools for showing that CQ answering under piece-wise linear warded sets of TGDs is in PSpace in combined complexity, and in NLogSpace in data complexity, and also for re-establishing the complexity of warded sets of TGDs (see Proposition 3.2) in a more transparent way than the approach of [3, 17].

**The Case of** $\mathsf{CQAns}(\mathsf{WARD} \cap \mathsf{PWL})$**.** Given a database $D$, a set $\Sigma \in \mathsf{WARD} \cap \mathsf{PWL}$ of TGDs, a CQ $q(\bar{x})$, and a tuple

```
Input: $D, \Sigma \in \mathsf{WARD} \cap \mathsf{PWL}$, $q(\bar{x})$, $\bar{c} \in \mathsf{dom}(D)^{|\bar{x}|}$
Output: Accept if $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$; otherwise, Reject

$p := Q \leftarrow \alpha_1, \ldots, \alpha_n$ with $\mathsf{atoms}(q(\bar{c})) = \{\alpha_1, \ldots, \alpha_n\}$
repeat
    guess $op \in \{\mathsf{r}, \mathsf{d}, \mathsf{s}\}$
    if $op = \mathsf{r}$ then
        guess a TGD $\sigma \in \Sigma$
        if $\mathsf{mgcu}(p, \sigma) = \emptyset$ then
            Reject
        else
            guess $U \in \mathsf{mgcu}(p, \sigma)$
            if $|p[\sigma, U]| > f_{\mathsf{WARD} \cap \mathsf{PWL}}(q, \Sigma)$ then
                Reject
            else
                $p' := p[\sigma, U]$
    if $op = \mathsf{d}$ then
        $p' := p[-D]$
    if $op = \mathsf{s}$ then
        guess $V \subseteq \mathsf{var}(p)$ and $\gamma : V \rightarrow \mathsf{dom}(D)$
        $p' := \gamma(p)$
    $p := p'$
until $\mathsf{atoms}(p) \subseteq D$;
return Accept
```

$\bar{c} \in \mathsf{dom}(D)^{|\bar{x}|}$, by Theorem 4.9, our problem boils down to checking whether there exists a linear proof tree $\mathcal{P}$ of $q$ w.r.t. $\Sigma$ with $\mathsf{nwd}(\mathcal{P}) \leq f_{\mathsf{WARD} \cap \mathsf{PWL}}(q, \Sigma)$ such that $\bar{c} \in \mathcal{P}(D)$. This can be easily checked via a space-bounded algorithm that is trying to build such a proof tree in a level-by-level fashion. Essentially, the algorithm builds the $i$-th level from the $(i-1)$-th level of the proof tree by non-deterministically applying the operations introduced above, i.e., resolution, decomposition and specialization.

The algorithm is depicted in the box above. Here is a semi-formal description of it. The first step is to store in $p$ the Boolean CQ obtained after instantiating the output variables of $q$ with $\bar{c}$. The rest of the algorithm is an iterative procedure that non-deterministically constructs $p'$ (the $i$-th level) from $p$ (the $(i-1)$-th level) until it reaches a level that is a subset of the database $D$. Notice that $p$ and $p'$ always hold one CQ since at each level of a linear proof tree only one node has a child, while all the other nodes are leaves, which essentially means that their atoms appear in the database $D$. At each iteration, the algorithm constructs $p'$ from $p$ by applying resolution (r), decomposition (d), or specialization (s):

**Resolution.** It guesses a TGD $\sigma \in \Sigma$. If the set $\mathsf{mgcu}(p, \sigma)$, i.e., the set of all MGCUs of $p$ with $\sigma$, is empty, then rejects; otherwise, it guesses $U \in \mathsf{mgcu}(p, \sigma)$. If the size of the $\sigma$-resolvent of $p$ obtained via $U$, denoted $p[\sigma, U]$, does not exceed the bound given by Theorem 4.9, then it assigns $p[\sigma, U]$ to $p'$; otherwise, it rejects. Recall that during a resolution step we need to rename variables in order to avoid undesirable clutter. However, we cannot blindly use new variables at each step since this will explode the space used by the algorithm. Instead, we should reuse variables that have been lost due to their unification with an existentially quantified variable. We only need polynomially many variables, while this polynomial depends only on $q$ and $\Sigma$.

**Decomposition.** It deletes from $p$ the atoms that occur in $D$, and it assigns the obtained CQ $p[-D]$ to $p'$. Notice that $p[-D]$ may be empty in case $\mathsf{atoms}(p) \subseteq D$. Essentially, the algorithm decomposes $p$ in such a way that the subquery of $p$ consisting of $\mathsf{atoms}(p) \cap D$ forms a child of $p$ that is a leaf, while the subquery consisting of $\mathsf{atoms}(p) \setminus D$ is the non-leaf child.

**Specialization.** It assigns to $p'$ a specialized version of $p$, where some variables are instantiated by constants of $\mathsf{dom}(D)$. The convention that output variables correspond to constants is implemented by directly instantiating them with actual constants from $\mathsf{dom}(D)$.

After constructing $p'$, the algorithm assigns it to $p$, and this ends one iteration. If $\mathsf{atoms}(p) \subseteq D$, then a linear proof tree $\mathcal{P}$ such that $\bar{c} \in \mathcal{P}(D)$ has been found, and the algorithm accepts; otherwise, it proceeds with the next iteration.

It is easy to see that the algorithm uses polynomial space in general. Moreover, in case the set of TGDs and the CQ are fixed, the algorithm uses logarithmic space, which is the space needed for representing constantly many elements of $\mathsf{dom}(D)$; each element of $\mathsf{dom}(D)$ can be represented using logaritmically many bits. The desired upper bounds claimed in Theorem 4.2 follow.

**The Case of $\mathsf{CQAns}(\mathsf{WARD})$.** The non-deterministic algorithm discussed above cannot be directly used for warded sets of TGDs since it is not enough to search for a linear proof tree as in the case of piece-wise linear warded sets of TGDs. However, by Theorem 4.10, we can search for a proof tree that has bounded node-width. This allows us to devise a space-bounded algorithm, which is similar in spirit as the one presented above, with the crucial difference that it constructs in a level-by-level fashion the branches of the proof tree in parallel universal computations using alternation. Since this alternating algorithm uses polynomial space in general, and logarithmic space when the set of TGDs and the CQ are fixed, we immediately get an EXPTIME upper bound in combined, and a PTIME upper bound in data complexity. This confirms Proposition 3.2 established in [3, 17]. However, our new algorithm is significantly simpler than the one employed in [3, 17], while Theorem 4.10 reveals the main property of warded sets of TGDs that leads to the desirable complexity upper bounds.

## 5 A JUSTIFIED COMBINATION

It is interesting to observe that the class of piece-wise linear warded sets of TGDs generalizes the class of *intensionally linear* sets of TGDs, denoted $\mathsf{IL}$, where each TGD has at most one body atom whose predicate is intensional. Therefore, Theorem 4.2 immediately implies that $\mathsf{CQAns}(\mathsf{IL})$ is PSPACE-complete in combined complexity, and NLOGSPACE-complete in data complexity. Notice that $\mathsf{IL}$ generalizes linear Datalog, which is also PSPACE-complete in combined complexity, and NLOGSPACE-complete in data complexity. Thus, we can extend linear Datalog by allowing existentially quantified variables in rule heads, which essentially leads to $\mathsf{IL}$, without affecting the complexity of query answering.

At this point, one maybe tempted to think that the same holds for piece-wise linear Datalog, i.e., we can extend it with existentially quantified variables in rule heads, which leads to $\mathsf{PWL}$, without affecting the complexity of query answering, that is, PSPACE-complete in combined, and NLOGSPACE-complete in data complexity. However, if this is the case, then wardedness becomes redundant since the formalism that we are looking for is the class of piece-wise linear sets of TGDs, without the wardedness condition. It turned out that this is not the case. To our surprise, the following holds:

THEOREM 5.1. $\mathsf{CQAns}(\mathsf{PWL})$ *is undecidable in data complexity.*

To show the above result we exploit an undecidable tiling problem [9]. A *tiling system* is a tuple $\mathbb{T} = (T, L, R, H, V, a, b)$, where $T$ is a finite set of tiles, $L, R \subseteq T$ are special sets of left and right border tiles, respectively, with $L \cap R = \emptyset$, $H, V \subseteq T^2$ are the horizontal and vertical constraints, and $a, b$ are distinguished tiles of $T$ called the start and the finish tile, respectively. A *tiling* for $\mathbb{T}$ is a function $f : [n] \times [m] \to T$, for some $n, m > 0$, such that $f(1,1) = a$, $f(1,m) = b$, $f(1,i) \in L$ and $f(n,i) \in R$, for every $i \in [m]$, and $f$ respects the horizontal and vertical constraints. In other words, the first and the last rows of a tiling for $\mathbb{T}$ start with $a$ and $b$, respectively, while the leftmost and rightmost columns contain only tiles from $L$ and $R$, respectively. We reduce from the $\mathsf{UnboundedTiling}$ problem, that is, given a tiling system $\mathbb{T}$, decide whether there is a tiling for $\mathbb{T}$. Given a tiling system $\mathbb{T} = (T, L, R, H, V, a, b)$, the goal is to construct in polynomial time a database $D_{\mathbb{T}}$, a set of TGDs $\Sigma \in \mathsf{PWL}$, and a Boolean CQ $q$, such that $\mathbb{T}$ has a tiling iff $() \in \mathsf{cert}(q, D_{\mathbb{T}}, \Sigma)$; $()$ is the empty tuple. Note that $\Sigma$ and $q$ should not depend on $\mathbb{T}$.

**The Database $D_{\mathbb{T}}$.** It simply stores the tiling system $\mathbb{T}$:

$$\begin{aligned} & \{\mathrm{Tile}(t) \mid t \in T\} \cup \{\mathrm{Left}(t) \mid t \in L\} \cup \{\mathrm{Right}(t) \mid t \in R\} \\ \cup\quad & \{H(t,t') \mid (t,t') \in H\} \cup \{V(t,t') \mid (t,t') \in V\} \\ \cup\quad & \{\mathrm{Start}(a), \mathrm{Finish}(b)\}. \end{aligned}$$

**The Set of TGDs $\Sigma$.** It is responsible for generating all the candidate tilings for $\mathbb{T}$, i.e., tilings without the condition $f(1,m) = b$, of arbitrary width and depth. Whether there exists a candidate tiling for $\mathbb{T}$ that satisfies the condition $f(1,m) = b$ will be checked by the CQ $q$. The set $\Sigma$ essentially implements the following idea: construct rows of size $\ell$ from rows of size $\ell - 1$, for $\ell > 1$, that respect the horizontal constraints, and then construct all the candidate tilings by combining compatible rows, i.e., rows that respect the vertical constraints. A row $r$ is encoded as an atom $\mathrm{Row}(p, c, s, e)$, where $p$ is the id of the row from which $r$ has been obtained, i.e., the previous one, $c$ is the id of $r$, i.e., the current one, $s$ is the starting tile of $r$, and $e$ is the ending tile of $r$. We write $\mathrm{Row}(c, c, s, s)$ for rows consisting of a single tile, which do not have a previous row (hence the id of the previous row coincides with the id of the current row), and the starting tile is the same as the ending tile. The following two TGDs construct all the rows that respect the horizontal constraints:

$$\begin{aligned} \mathrm{Tile}(x) &\;\to\; \exists z\, \mathrm{Row}(z, z, x, x), \\ \mathrm{Row}(\_, x, y, z), H(z, w) &\;\to\; \exists u\, \mathrm{Row}(x, u, y, w). \end{aligned}$$

Analogously to Prolog, we write "_" for a "don't-care" variable that occurs only once in the TGD. The next set of TGDs constructs all the pairs of compatible rows, i.e., pairs of rows $(r_1, r_2)$ such that we can place $r_2$ below $r_1$ without violating the vertical constraints. This is done inductively as follows:

$$\begin{aligned} &\mathrm{Row}(x, x, y, y), \mathrm{Row}(x', x', y', y'), V(y, y') \;\to\; \mathrm{Comp}(x, x'), \\ &\mathrm{Row}(x, y, \_, z), \mathrm{Row}(x', y', \_, z'), \\ &\qquad\qquad \mathrm{Comp}(x, x'), V(z, z') \;\to\; \mathrm{Comp}(y, y'). \end{aligned}$$

We finally compute all the candidate tilings, together with their bottom-left tile, using the following two TGDs:

$$\begin{aligned} &\mathrm{Row}(\_, x, y, z), \mathrm{Start}(y), \mathrm{Right}(z) \;\to\; \mathrm{CTiling}(x, y), \\ &\mathrm{CTiling}(x, \_), \mathrm{Row}(\_, y, z, w), \mathrm{Comp}(x, y), \\ &\qquad\qquad \mathrm{Left}(z), \mathrm{Right}(w) \;\to\; \mathrm{CTiling}(y, z). \end{aligned}$$

This concludes the definition of $\Sigma$.

**The Boolean CQ $q$.** Recall that $q$ is responsible for checking whether there exists a candidate tiling such that its bottom-left tile is $b$. This can be easily done via the query

$$Q \;\leftarrow\; \mathrm{CTiling}(x, y), \mathrm{Finish}(y).$$

By construction, $\Sigma \in \mathsf{PWL}$. Moreover, there is a tiling for $\mathbb{T}$ iff $() \in \mathsf{cert}(q, D_{\mathbb{T}}, \Sigma)$, and Theorem 5.1 follows.

## 6 EXPRESSIVE POWER

A class of TGDs naturally gives rise to a declarative database query language. More precisely, we consider queries of the form $(\Sigma, q)$, where $\Sigma$ is a set of TGDs, and $q$ a CQ over $\mathsf{sch}(\Sigma)$. The *extensional (database) schema* of $\Sigma$, denoted $\mathsf{edb}(\Sigma)$, is the set of extensional predicates of $\mathsf{sch}(\Sigma)$, i.e., the predicates that do not occur in the head of a TGD of $\Sigma$. Given a query $Q = (\Sigma, q)$ and a database $D$ over $\mathsf{edb}(\Sigma)$, the *evaluation* of $Q$ over $D$, denoted $Q(D)$, is defined as $\mathsf{cert}(q, D, \Sigma)$. We write $(\mathsf{C}, \mathsf{CQ})$ for the query language consisting of all the queries $(\Sigma, q)$, where $\Sigma \in \mathsf{C}$, and $q$ is a CQ. The evaluation problem for such a query language, dubbed $\mathsf{Eval}(\mathsf{C}, \mathsf{CQ})$, is defined in the usual way. By definition, $\bar{c} \in Q(D)$ iff $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$. Therefore, the complexity of $\mathsf{Eval}(\mathsf{C}, \mathsf{CQ})$ when $\mathsf{C} = \mathsf{WARD} \cap \mathsf{PWL}$ and $\mathsf{C} = \mathsf{WARD}$ is immediately inherited from Theorem 4.2 and Proposition 3.2, respectively:

THEOREM 6.1. *The following statements hold:*

*(1)* $\mathsf{Eval}(\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$ *is* PSPACE*-complete in combined, and* NLOGSPACE*-complete in data complexity.*
*(2)* $\mathsf{Eval}(\mathsf{WARD}, \mathsf{CQ})$ *is* EXPTIME*-complete in combined, and* PTIME*-complete in data complexity.*

The main goal of this section is to understand the relative expressive power of $(\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$ and $(\mathsf{WARD}, \mathsf{CQ})$. To this end, we are going to adopt two different notions of expressive power namely the classical one, which we call combined expressive power since it considers the set of TGDs and the CQ as one composite query, and the program expressive power, which aims at the decoupling of the set of TGDs from the actual CQ. We proceed with the details starting with the combined expressive power.

## 6.1 Combined Expressive Power

Consider a query $Q = (\Sigma, q)$, where $\Sigma$ is a set of TGDs and $q(\bar{x})$ a CQ over $\mathsf{sch}(\Sigma)$. The *expressive power* of $Q$, denoted $\mathsf{ep}(Q)$, is the set of pairs $(D, \bar{c})$, where $D$ is a database over $\mathsf{edb}(\Sigma)$, and $\bar{c} \in \mathsf{dom}(D)^{|\bar{x}|}$, such that $\bar{c} \in Q(D)$. The *combined expressive power* of a query language $(\mathsf{C}, \mathsf{CQ})$, where $\mathsf{C}$ is a class of TGDs, is defined as the set

$$\mathsf{cep}(\mathsf{C}, \mathsf{CQ}) = \{\mathsf{ep}(Q) \mid Q \in (\mathsf{C}, \mathsf{CQ})\}.$$

Given two query languages $\mathrm{Q}_1, \mathrm{Q}_2$, we say that $\mathrm{Q}_2$ is *more expressive (w.r.t. the combined expressive power)* than $\mathrm{Q}_1$, written $\mathrm{Q}_1 \leq_{\mathsf{cep}} \mathrm{Q}_2$, if $\mathsf{cep}(\mathrm{Q}_1) \subseteq \mathsf{cep}(\mathrm{Q}_2)$. We say that $\mathrm{Q}_1$ and $\mathrm{Q}_2$ are *equally expressive (w.r.t. the combined expressive power)*, written $\mathrm{Q}_1 =_{\mathsf{cep}} \mathrm{Q}_2$, if $\mathrm{Q}_1 \leq_{\mathsf{cep}} \mathrm{Q}_2$ and $\mathrm{Q}_2 \leq_{\mathsf{cep}} \mathrm{Q}_1$.

The next easy lemma states that $\mathrm{Q}_1 =_{\mathsf{cep}} \mathrm{Q}_2$ is equivalent to say that every query of $\mathrm{Q}_1$ can be equivalently rewritten as a query of $\mathrm{Q}_2$, and vice versa. Given two query languages $\mathrm{Q}_1$ and $\mathrm{Q}_2$, we write $\mathrm{Q}_1 \leq \mathrm{Q}_2$ if, for every $Q = (\Sigma, q) \in \mathrm{Q}_1$, there exists $Q' = (\Sigma', q') \in \mathrm{Q}_2$ such that, for every $D$ over $\mathsf{edb}(\Sigma)$, $Q(D) = Q'(D)$.

LEMMA 6.2. *Consider two query languages* $\mathrm{Q}_1$ *and* $\mathrm{Q}_2$*. It holds that* $\mathrm{Q}_1 \leq_{\mathsf{cep}} \mathrm{Q}_2$ *iff* $\mathrm{Q}_1 \leq \mathrm{Q}_2$*.*

We are now ready to state the main result of this section, which reveals the expressiveness of $(\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$ and $(\mathsf{WARD}, \mathsf{CQ})$ relative to Datalog. Let us clarify that a Datalog query is essentially a pair $(\Sigma, q)$, where $\Sigma$ is a Datalog program, or a set of *full* TGDs, i.e., TGDs without existentially quantified variables, that have only one head atom, and $q$ a CQ. We write $\mathsf{FULL}_1$ for the above class of TGDs. In other words, piece-wise linear Datalog, denoted $\mathsf{PWL\text{-}DATALOG}$, is the language $(\mathsf{FULL}_1 \cap \mathsf{PWL}, \mathsf{CQ})$, while Datalog, denoted $\mathsf{DATALOG}$, is the language $(\mathsf{FULL}_1, \mathsf{CQ})$, and thus we can refer to their combined expressive power.

THEOREM 6.3. *The following statements hold:*

*(1)* $\mathsf{PWL\text{-}DATALOG} =_{\mathsf{cep}} (\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$.
*(2)* $\mathsf{DATALOG} =_{\mathsf{cep}} (\mathsf{WARD}, \mathsf{CQ})$.

Let us explain how (1) is shown; the proof for (2) is similar. We need to show that: (a) $\mathsf{PWL\text{-}DATALOG} \leq_{\mathsf{cep}} (\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$, and (b) $(\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ}) \leq_{\mathsf{cep}} \mathsf{PWL\text{-}DATALOG}$. By definition, $\mathsf{FULL}_1 \cap \mathsf{PWL} \subseteq \mathsf{WARD} \cap \mathsf{PWL}$. Thus, $(\mathsf{FULL}_1 \cap \mathsf{PWL}, \mathsf{CQ}) \leq (\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$, which, together with Lemma 6.2, implies (a). For showing (b), by Lemma 6.2, it suffices to show that:

LEMMA 6.4. $(\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ}) \leq \mathsf{PWL\text{-}DATALOG}$.

The key idea underlying the above lemma is to convert a linear proof tree $\mathcal{P}$ of a CQ $q(\bar{x})$ w.r.t. a set $\Sigma \in \mathsf{WARD} \cap \mathsf{PWL}$ of TGDs into a piece-wise linear Datalog query $Q = (\Sigma', q'(\bar{x}))$ such that, for every database $D$ over $\mathsf{edb}(\Sigma)$, $\mathcal{P}(D) = Q(D)$. Roughly, each node of $\mathcal{P}$ together with its children, is converted into a full TGD that is added to $\Sigma'$. Assume that the node $v$ has the children $u_1, \ldots, u_k$ in $\mathcal{P}$, where $v$ is labeled by $p_0(\bar{x}_0)$ and, for $i \in [k]$, $u_i$ is labeled by the CQ $p_i(\bar{x}_i)$ with $\bar{x}_0 \subseteq \bar{x}_i$. We then add to $\Sigma'$

$$C_{[p_1]}(\bar{x}_1), \ldots, C_{[p_k]}(\bar{x}_k) \rightarrow C_{[p_0]}(\bar{x}_0),$$

where $C_{[p_i]}$ is a predicate that corresponds to the CQ $p_i$, while $[p_i]$ refers to a *canonical renaming* of $p_i$. The intention underlying such a canonical renaming is the following: if $p_i$ and $p_j$ are the same up to variable renaming, then $[p_i] = [p_j]$. We also add to $\Sigma'$ a full TGD

$$R(x_1, \ldots, x_n) \rightarrow C_{[p_R]}(x_1, \ldots, x_n)$$

for each $n$-ary predicate $R \in \mathsf{edb}(\Sigma)$, where $p_R(x_1, \ldots, x_n)$ is the atomic query consisting of the atom $R(x_1, \ldots, x_n)$. Since in $\mathcal{P}$ we may have several CQs that are the same up to variables renaming, the set $\Sigma'$ is recursive, but due to the linearity of $\mathcal{P}$, the employed recursion is piece-wise linear, i.e., $\Sigma' \in \mathsf{FULL}_1 \cap \mathsf{PWL}$. The CQ $q'(\bar{x})$ is simply the atomic query $C_{[q]}(\bar{x})$. It should not be difficult to see that indeed $\mathcal{P}(D) = Q(D)$, for every database $D$ over $\mathsf{edb}(D)$.

Having the above transformation of a linear proof tree into a piece-wise linear Datalog query in place, we can easily rewrite every query $Q = (\Sigma, q) \in (\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$ into an equivalent query that falls in $\mathsf{PWL\text{-}DATALOG}$. We exhaustively convert each linear proof tree $\mathcal{P}$ of $q$ w.r.t. $\Sigma$

such that $\mathsf{nwd}(\mathcal{P}) \leq f_{\mathsf{WARD}\cap\mathsf{PWL}}(q,\Sigma)$ into a piece-wise linear Datalog query $Q_{\mathcal{P}}$, and then we take the union of all those queries. Since we consider the canonical renaming of the CQs occurring in a proof tree, and since the size of those CQs is bounded by $f_{\mathsf{WARD}\cap\mathsf{PWL}}(q,\Sigma)$, we immediately conclude that we need to explore finitely many CQs. Thus, the above iterative procedure will eventually terminate and construct a finite piece-wise linear Datalog query that is equivalent to $Q$, as needed.

## 6.2 Program Expressive Power

The *expressive power* of a set $\Sigma$ of TGDs, denoted $\mathsf{ep}(\Sigma)$, is the set of triples $(D, q(\bar{x}), \bar{c})$, where $D$ is a database over $\mathsf{edb}(\Sigma)$, $q(\bar{x})$ is a CQ over $\mathsf{sch}(\Sigma)$, and $\bar{c} \in \mathsf{dom}(D)^{|\bar{x}|}$, such that $\bar{c} \in \mathsf{cert}(q, D, \Sigma)$. The *program expressive power* of a query language $(\mathsf{C}, \mathsf{CQ})$, where $\mathsf{C}$ is a class of TGDs, is defined as

$$\mathsf{pep}(\mathsf{C}, \mathsf{CQ}) \; = \; \{\mathsf{ep}(\Sigma) \mid \Sigma \in \mathsf{C}\}.$$

Given two query languages $\mathsf{Q}_1, \mathsf{Q}_2$, we say that $\mathsf{Q}_2$ is *more expressive (w.r.t. program expressive power)* than $\mathsf{Q}_1$, written $\mathsf{Q}_1 \leq_{\mathsf{pep}} \mathsf{Q}_2$, if $\mathsf{pep}(\mathsf{Q}_1) \subseteq \mathsf{pep}(\mathsf{Q}_2)$. Moreover, we say that $\mathsf{Q}_2$ is *strictly more expressive (w.r.t. the program expressive power)* that $\mathsf{Q}_2$, written $\mathsf{Q}_1 <_{\mathsf{pep}} \mathsf{Q}_2$, if $\mathsf{Q}_1 \leq_{\mathsf{pep}} \mathsf{Q}_2$ and $\mathsf{Q}_2 \not\leq_{\mathsf{pep}} \mathsf{Q}_1$.

Let us now establish a useful lemma, analogous to Lemma 6.2, which reveals the essence of the program expressive power. For brevity, given two classes of TGDs $\mathsf{C}_1$ and $\mathsf{C}_2$, we write $\mathsf{C}_1 \preceq \mathsf{C}_2$ if, for every $\Sigma \in \mathsf{C}_1$, there exists $\Sigma' \in \mathsf{C}_2$ such that, for every $D$ over $\mathsf{edb}(\Sigma)$, and CQ $q$ over $\mathsf{sch}(\Sigma)$, $Q(D) = Q'(D)$, where $Q = (\Sigma, q)$ and $Q' = (\Sigma', q)$.

Lemma 6.5. *Consider two query languages* $\mathsf{Q}_1 = (\mathsf{C}_1, \mathsf{CQ})$ *and* $\mathsf{Q}_2 = (\mathsf{C}_2, \mathsf{CQ})$*. Then,* $\mathsf{Q}_1 \leq_{\mathsf{pep}} \mathsf{Q}_2$ *iff* $\mathsf{C}_1 \preceq \mathsf{C}_2$.

We are now ready to study the expressiveness (w.r.t. the program expressive power) of $(\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$ and $(\mathsf{WARD}, \mathsf{CQ})$ relative to Datalog. In particular, we show that:

Theorem 6.6. *The following statements hold:*
*(1)* $\mathsf{PWL\text{-}DATALOG} <_{\mathsf{pep}} (\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$.
*(2)* $\mathsf{DATALOG} <_{\mathsf{pep}} (\mathsf{WARD}, \mathsf{CQ})$.

Let us explain how (1) is shown; the proof for (2) is similar. We need to show that: (a) $\mathsf{PWL\text{-}DATALOG} \leq_{\mathsf{pep}} (\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ})$, and (b) $(\mathsf{WARD} \cap \mathsf{PWL}, \mathsf{CQ}) \not\leq_{\mathsf{pep}} \mathsf{PWL\text{-}DATALOG}$. Since, by definition, $\mathsf{FULL}_1 \cap \mathsf{PWL} \subseteq \mathsf{WARD} \cap \mathsf{PWL}$, we immediately get that $\mathsf{FULL}_1 \cap \mathsf{PWL} \preceq \mathsf{WARD} \cap \mathsf{PWL}$, and thus, by Lemma 6.5, (a) follows. For showing (b), by Lemma 6.5, it suffices to show that:

Lemma 6.7. $\mathsf{WARD} \cap \mathsf{PWL} \not\preceq \mathsf{FULL}_1 \cap \mathsf{PWL}$.

By contradiction, assume the opposite. We define the set of TGDs $\Sigma = \{P(x) \rightarrow \exists y\, R(x, y)\}$, the database $D = \{P(c)\}$, and the CQs $q_1 = Q \leftarrow R(x, y)$ and $q_2 = Q \leftarrow R(x, y), P(y)$. By hypothesis, there exists $\Sigma' \in \mathsf{FULL}_1 \cap \mathsf{PWL}$ such that $Q_1(D) = Q_1'(D)$ and $Q_2(D) = Q_2'(D)$, where $Q_i = (\Sigma, q_i)$ and $Q_i' = (\Sigma', q_i)$, for $i \in \{1, 2\}$. Clearly, $Q_1(D) \neq \emptyset$ and $Q_2(D) = \emptyset$, which implies that $Q_1'(D) \neq \emptyset$ and $Q_2'(D) = \emptyset$. However, it is easy to see that $Q_1'(D) \neq \emptyset$ implies $Q_2'(D) \neq \emptyset$, which is a contradiction, and the claim follows.

## 7 IMPLEMENTATION AND FUTURE WORK

The Vadalog system is currently optimized for piece-wise linear warded sets of TGDs in three ways: (i) the first one is related to the way that existential quantifiers interact with recursion; (ii) the second one is related to the optimizer, which detects and uses piece-wise linearity for the purpose of join ordering; (iii) the third way is related to the architecture of the system. Here are some directions for future research:

(1) As said in Section 1, NLogSpace is contained in the class $\mathrm{NC}_2$ of highly parallelizable problems. This means that reasoning under piece-wise linear warded sets of TGDs is principally parallelizable, unlike warded sets of TGDs. We plan to exploit this for the parallel execution of reasoning tasks in multi-core settings and in the map-reduce model.
(2) Reasoning with piece-wise linear warded sets of TGDs is LogSpace-equivalent to reachability in directed graphs. Reachability in very large graphs has been well-studied and many algorithms and heuristics have been designed that work well in practice; see, e.g., [13, 18, 20]. We are confident that several of these algorithms can be adapted for our purposes.
(3) Reachability in directed graphs is known to be in the *dynamic* parallel complexity class Dyn-FO [14, 26]. This means that by maintaining suitable auxiliary data structures when updating a graph, reachability testing can actually be done in FO, and thus in SQL. We plan to analyze whether reasoning under piece-wise linear warded sets of TGDs, or relevant subclasses thereof, can be shown to be in Dyn-FO or some other dynamic complexity classes.

**Acknowledgements.** We thank the anonymous referees for their useful feedback. This work was supported by the EPSRC Programme Grant EP/M025268/ VADA, the EPSRC grant EP/S003800/1 EQUID, the WWTF grant VRG18-013, the EU Horizon 2020 grant 809965, the FWF grant W1255-N23, and a DOC fellowship of the Austrian Academy of Sciences.

## REFERENCES

[1] Foto N. Afrati, Manolis Gergatsoulis, and Francesca Toni. 2003. Linearisability on datalog programs. *Theor. Comput. Sci.* 308, 1-3 (2003), 199–226.
[2] Marcelo Arenas, Georg Gottlob, and Andreas Pieris. 2014. Expressive languages for querying the semantic web. In *PODS*. 14–26.
[3] Marcelo Arenas, Georg Gottlob, and Andreas Pieris. 2018. Expressive Languages for Querying the Semantic Web. *ACM Trans. Database Syst.* 43, 3 (2018), 13:1–13:45.

[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. 1981. The Implication Problem for Data Dependencies. In *ICALP*. 73–85.

[6] Luigi Bellomarini, Georg Gottlob, Andreas Pieris, and Emanuel Sallinger. 2017. Swift Logic for Big Data and Knowledge Graphs. In *IJCAI*. 2–10.

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

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

[9] Peter Van Emde Boas. 1997. The Convenience of Tilings. In *Complexity, Logic, and Recursion Theory*. 331–363.

[10] 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.

[11] Andrea Calì, Georg Gottlob, Thomas Lukasiewicz, Bruno Marnette, and Andreas Pieris. 2010. Datalog+/-: A Family of Logical Knowledge Representation and Query Languages for New Applications. In *LICS*. 228–242.

[12] Andrea Calì, Georg Gottlob, and Andreas Pieris. 2012. Towards more expressive ontology languages: The query answering problem. *Artif. Intell.* 193 (2012), 87–128.

[13] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.

[14] Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick, and Thomas Zeume. 2015. Reachability is in DynFO. In *ICALP*. 159–170.

[15] 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.

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

[17] Georg Gottlob and Andreas Pieris. 2015. Beyond SPARQL under OWL 2 QL Entailment Regime: Rules to the Rescue. In *IJCAI*. 2999–3007.

[18] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. 2008. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*. 595–608.

[19] David S. Johnson and Anthony C. Klug. 1984. Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies. *J. Comput. Syst. Sci.* 28, 1 (1984), 167–189.

[20] Valerie King. 1999. Fully Dynamic Algorithms for Maintaining All-Pairs Shortest Paths and Transitive Closure in Digraphs. In *FOCS*. 81–91.

[21] Mélanie König, Michel Leclère, Marie-Laure Mugnier, and Michaël Thomazo. 2015. Sound, complete and minimal UCQ-rewriting for existential rules. *Semantic Web* 6, 5 (2015), 451–475.

[22] Roman Kontchakov, Martin Rezk, Mariano Rodriguez-Muro, Guohui Xiao, and Michael Zakharyaschev. 2014. Answering SPARQL Queries over Databases under OWL 2 QL Entailment Regime. In *ISWC*. 552–567.

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

[24] Jeffrey F. Naughton. 1986. Data Independent Recursion in Deductive Databases. In *PODS*. 267–279.

[25] Jeffrey F. Naughton and Yehoshua Sagiv. 1987. A Decidable Class of Bounded Recursions. In *PODS*. 227–236.

[26] Sushant Patnaik and Neil Immerman. 1997. Dyn-FO: A Parallel, Dynamic Complexity Class. *J. Comput. Syst. Sci.* 55, 2 (1997), 199–209.