The Vadalog System
Datalog-based Reasoning for Knowledge Graphs

Luigi Bellomarini
University of Oxford and
Banca d'Italia and
Università Roma Tre

Emanuel Sallinger
University of Oxford

Georg Gottlob
University of Oxford and
TU Wien

ABSTRACT

Over the past years, there has been a resurgence of Datalog-based systems in the database community as well as in industry. In this context, it has been recognized that to handle the complex knowledge-based scenarios encountered today, such as reasoning over large knowledge graphs, Datalog has to be extended with features such as existential quantification. Yet, Datalog-based reasoning in the presence of existential quantification is in general undecidable. Many efforts have been made to define decidable fragments. Warded Datalog+/- is a very promising one, as it captures PTIME complexity while allowing ontological reasoning. Yet so far, no implementation of Warded Datalog+/- was available. In this paper we present the Vadalog system, a Datalog-based system for performing complex logic reasoning tasks, such as those required in advanced knowledge graphs. The Vadalog system is Oxford's contribution to the VADA research programme, a joint effort of the universities of Oxford, Manchester and Edinburgh and around 20 industrial partners. As the main contribution of this paper, we illustrate the first implementation of Warded Datalog+/-, a high-performance Datalog+/- system utilizing an aggressive termination control strategy. We also provide a comprehensive experimental evaluation.

PVLDB Reference Format:
L. Bellomarini, E. Sallinger, and G. Gottlob. The Vadalog System: Datalog-based Reasoning for Knowledge Graphs. *PVLDB*, 11(2): xxxx-yyyy, 2017.
DOI: https://doi.org/TBD

1. INTRODUCTION

The importance of capitalizing and exploiting corporate knowledge has been clear to decision makers since the late 1970s, when this idea was gradually made concrete in the context of *expert systems*, software frameworks able to harness such knowledge and provide answers to structured business questions. Through *deductive database systems* – database systems that have advanced reasoning capabilities – and in particular the language *Datalog*, the area became of high interest to the database community in the 1980s and 1990s. Yet, even though the importance of harnessing knowledge certainly has grown steadily since then, culminating in today's desire of companies to build and exploit *knowledge graphs*, the interest in deductive databases has faltered.

Many factors have prevented an effective large-scale development and diffusion of deductive database systems: immature hardware technology, unable to cope with the rising challenges and opportunities coming from the Big Data applications; the rigidity of existing database management systems, unable to go beyond the standard requirements of query answering; the lack of knowledge languages expressive enough to address real-world cases.

The situation has rapidly changed in recent times, thanks to accelerations in many fields. The strong need for scalability firstly represented by Internet giants and promptly understood by large and medium-sized companies led to the soar of a flurry of specialized middleware modules having scalability as a central target. The resurgence of *Datalog* in academia and industry [3, 8, 15, 16, 17, 18] turned out to be a key factor. Companies like LogicBlox have proven that a fruitful exchange between academic research and high-performance industrial applications can be achieved based on Datalog [3], and companies like LinkedIn have shown that the interest in Datalog permeates industry [32]. Meanwhile, the recent interest in machine learning brought renewed visibility to AI, raising interest and triggering investments in thousands of companies world-wide, which suddenly wish to collect, encapsulate and exploit their corporate knowledge in the form of a knowledge graph.

The term *knowledge graph* (KG) has no standard definition. It can be seen as referring only to Google's Knowledge Graph, to triple-based models, or to multi-attributed graphs, which represent n-ary relations [28, 36]. As shown by Krötzsch [27], in order to support rule-based reasoning on such data structures, it is sometimes necessary to use tuples of arity higher than three at least for intermediate results. In this paper, we adopt a general notion of KGs by allowing relations of arbitrary arity, to support all of these models and modes of reasoning.

Example 1. *An example of a simple knowledge graph reasoning setting is given in [28]:*

$$\text{Spouse}(x, y, start, loc, end) \rightarrow \text{Spouse}(y, x, start, loc, end)$$

This rule expresses that when a person x is married to a person y at a particular location, starting date and end date, then the same holds for y and x. That is, the graph of persons and their marriage relations is symmetric.

As stated in [28], most modern ontology languages are not able to express this example. Beyond this simple example, there are numerous requirements for a system that allows ontological reasoning over KGs. Navigating graphs is impossible without powerful recursion; ontological reasoning is impossible without existential

Preprint. To appear in PVLDB, 2018.

quantification in rule heads [23]. An analysis of various requirements was given in [9]. In this paper, we isolate three concrete requirements for reasoning over knowledge graphs:

1. **Recursion over KGs**. Should be at least able to express full recursion and joins, i.e., should at least encompass Datalog. Full recursion in combination with arbitrary joins allows to express complex reasoning tasks over KGs. Navigational capabilities, empowered by recursion, are vital for graph-based structures.
2. **Ontological Reasoning over KGs**. Should at least be able to express SPARQL reasoning under the OWL 2 QL entailment regime and set semantics. OWL 2 QL is one of the most adopted profiles of the Web Ontology Language, standardized by W3C.
3. **Low Complexity**. Reasoning should be tractable in data complexity. Tractable data complexity is a minimal requirement for allowing scalability over large volumes of data.

Beyond these specific requirements, the competition between powerful recursion, powerful existential quantification and low complexity has spawned fruitful research throughout the community as reasoning with recursive Datalog is undecidable in the presence of existential quantification. This has been done under a number of different names, but which we shall here call *Datalog*$^{\pm}$, the "+" referring to the additional features (including existential quantification), the "-" to restrictions that have to be made to obtain decidability. Many languages within the Datalog$^{\pm}$ family of languages have been proposed and intensively investigated [5, 7, 15, 16, 17, 18]. Details on the languages are given in Section 2. Depending on the syntactic restrictions, they achieve a different balance between expressiveness and computational complexity.

Figure 1 gives an overview of the main Datalog$^{\pm}$ languages. In fact, most of these candidates, including Linear Datalog$^{\pm}$, Guarded Datalog$^{\pm}$, Sticky Datalog$^{\pm}$ and Weakly Sticky Datalog$^{\pm}$ do not fulfil (1). Datalog itself does not fulfil (2). Warded and Weakly Frontier Guarded Datalog$^{\pm}$ satisfy (1) and (2), thus are expressive enough. However, the expressiveness of Weakly Frontier Guarded Datalog$^{\pm}$ comes at the price of it being EXPTIME-complete [7]. Thus it does not fulfil (3).

Thus, in total, the only known language that satisfies (1), (2) and (3) is Warded Datalog$^{\pm}$. Yet, while Warded Datalog$^{\pm}$ has very good theoretical properties, the algorithms presented in [23] are alternating-time Turing machine algorithms, far away from a practical implementation.

In this paper we present **the Vadalog system**, Oxford's contribution to the VADA research project [37], a joint effort of the universities of Oxford, Manchester and Edinburgh and around 20 industrial partners such as Facebook, BP, and the NHS (UK national health system). The Vadalog system is built around the Vadalog language, with Warded Datalog$^{\pm}$ as its logical core. It is currently used as the core deductive database system of the overall Vadalog Knowledge Graph Management System described in [9] as well as at various industrial partners, including the finance, security, and media intelligence industries. The **main contributions** of this paper are:

- A **novel analysis** of Warded Datalog$^{\pm}$, focused on practical implementation. In particular, we identify a number of *guide structures* that closely exploit the underpinnings of Warded Datalog$^{\pm}$ to guarantee termination while keeping the size of such guide structures limited to ensure a small memory footprint. These three guide structures, the *linear forest*, *warded forest*, and *lifted linear forest* are related, but play complementary roles for exploiting the periodicity of execution. The culmination of this analysis is the **first practical algorithm for Warded Datalog$^{\pm}$**.

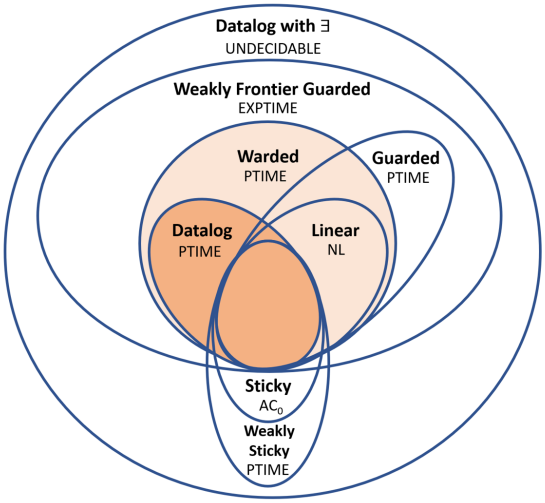

Figure 1: Syntactic containment of Datalog$^{\pm}$ languages. Annotations (non-bold) denote data complexity. All names that do not explicitly mention Datalog refer to the respective Datalog$^{\pm}$ languages. E.g., "Sticky" refers to "Sticky Datalog$^{\pm}$".

- A **system and architecture** that implements this algorithm in a relational database-inspired operator pipeline architecture. The pipeline's operators rely on *termination strategy wrappers* which transparently prevent the generation of facts that may lead to non-termination while ensuring the correctness of the result.
- A full-scale **experimental evaluation** of the Vadalog system on a variety of real-world and synthetic scenarios that thoroughly validate the effectiveness of our techniques on Warded Datalog$^{\pm}$ in absolute terms and comparatively with the top existing systems, which are outperformed by our reasoner.

Overview. In Section 2 we describe the Vadalog language. In Section 3 we illustrate how we use the properties of Warded Datalog$^{\pm}$ to handle recursion and termination. In Section 4 we present the architecture of the Vadalog system. Section 5 is dedicated to a full-scale evaluation of the system. In Section 6 we discuss the related work and compare with other systems. The conclusions are in Section 7. For space reasons, more details, algorithms, examples and proofs can be found in an on-line technical report [10].

2. REASONING WITH VADALOG

Our system adopts Vadalog as a reference language. Its core is based on Datalog$^{\pm}$ and in particular on Warded Datalog$^{\pm}$. Before discussing the details of Warded Datalog$^{\pm}$ and the specific extensions to it that we have in Vadalog, we first recall some foundations underlying Datalog$^{\pm}$.

Let **C**, **N**, and **V** be disjoint countably infinite sets of *constants*, *(labeled) nulls* and (regular) *variables*, respectively. A *(relational) schema* **S** is a finite set of relation symbols (or predicates) with associated arity. A *term* is a either a constant or variable. An *atom* over **S** is an expression of the form $R(\bar{v})$, where $R \in \mathbf{S}$ is of arity $n > 0$ and $\bar{v}$ is an n-tuple of terms. A *database instance* (or simply *database*) over **S** associates to each relation symbol in **S** a relation of the respective arity over the domain of constants and nulls. The members of relations are called *tuples*. By some abuse of notations, we sometimes use the terms atom, tuple and fact interchangeably.

Datalog$^{\pm}$ languages consists of *existential rules*, or *tuple-generating dependencies*, which generalize Datalog rules with existential

quantification in rule heads. Such a rule is a first-order sentence of the form $\forall\bar{x}\forall\bar{y}(\varphi(\bar{x},\bar{y}) \rightarrow \exists\bar{z}\,\psi(\bar{x},\bar{z}))$, where φ (the *body*) and ψ (the *head*) are conjunctions of atoms with constants and variables. For brevity, we write this existential rule as $\varphi(\bar{x},\bar{y}) \rightarrow \exists\bar{z}\,\psi(\bar{x},\bar{z})$ and replace $\wedge$ with comma to denote conjunction of atoms. If there is at most one atom in φ, we call a rule *linear*, otherwise *non-linear*.

The intuitive meaning of such a rule is as follows: if there is a fact $\varphi(\bar{t},\bar{t}')$ that occurs in an instance I, then there exists a tuple $\bar{t}''$ of constants and nulls such that the facts $\psi(\bar{t},\bar{t}'')$ are also in I. Formally, the semantics of a set of existential rules Σ over a database D, denoted $\Sigma(D)$, is defined via the well-known *chase procedure*. Roughly, the chase adds new facts to D (possibly involving null values used to satisfy the existentially quantified variables) until the final result $\Sigma(D)$ satisfies all the existential rules of Σ. Notice that, in general, $\Sigma(D)$ is infinite. Example 2 is a simple such example:

Example 2. $\text{Company}(x) \rightarrow \exists p\ \underline{\text{KeyPerson}}(\hat{p}, x)$
$\text{Control}(x,y), \text{KeyPerson}(\hat{p},x) \rightarrow \underline{\text{KeyPerson}}(\hat{p}, y)$.

The first rule expresses that for every company x there exists a key person p. Then, if company x controls company y and p is a key person for x, then p will be a key person for y as well.

Now, given a pair $Q = (\Sigma, \text{Ans})$, where Σ is a set of existential rules and Ans an n-ary predicate, the evaluation of a query Q over a database D, denoted $Q(D)$, is defined as the set of tuples $Q(D) = \{\bar{t} \in dom(D)^n \mid \text{Ans}(\bar{t}) \in \Sigma(D)\}$. Observe that in Example 2, the predicate KeyPerson has been underlined to mean it is in Ans.

Based on this setting, we are interested in the *universal tuple inference* reasoning task (which we will simply call *reasoning task* for the sake of simplicity): given a database D and a pair $Q = (\Sigma, \text{Ans})$, find an instance J, such that a tuple $\bar{t} \in J$ if and only if $\bar{t} \in Q(D)$ and for every other instance J' such that $\bar{t} \in J'$ if and only if $\bar{t} \in Q(D)$, there is a homomorphism h from J to J'. In other terms, we are interested in finding the most general answer to the reasoning task, which amounts to saying that such answer is homomorphic to any other answer (henceforth we call it *universal*).

Warded Datalog$^\pm$. The core of the VADALOG language is Warded Datalog$^\pm$, a recent member of the Datalog$^\pm$ family. The full VADALOG language is then obtained as an extension to the Warded Datalog$^\pm$ core with features of practical utility.

The idea behind the notion of wardedness is taming the propagation of nulls during the construction of the answer in the chase procedure. Towards a formal definition of the notion of wardedness, let us first present some preliminary notions. We consider a set of rules Σ. For a predicate p used in Σ, we define *position* $p[i]$ as the i-th term of p. Let $\texttt{affected}(\Sigma)$ be a set inductively as follows: 1. it contains all the positions π such that π is an existentially quantified variable for some rule of Σ; 2. it contains all the positions π, such that there is some rule ρ of Σ where a variable v appears only in positions $\pi' \in \texttt{affected}(\Sigma)$ in the body of ρ, and v appears in position π in the head of ρ. Vice versa, we define $\texttt{nonaffected}(\Sigma)$, the non-affected positions of Σ. When convenient, we denote variables in affected positions with the "hat" symbol ($\hat{x}$). In a given rule ρ, the variables can be classified according to their positions. We define a variable v as: *harmless* in ρ, if at least one occurrence in the body of ρ is in a non-affected position; *harmful* in ρ, if in the body of ρ, v always appears in affected positions; *dangerous*, if v is harmful in ρ and also appears in the head of ρ. In Example 2, variable p is harmful and dangerous (as the first rule can inject nulls into the second position of KeyPerson); x and y are harmless.

With these definitions in place, we define a set of rules as warded if the following conditions hold: *1. all the dangerous variables in a rule appear only within a single atom (the ward); 2. the ward shares with other atoms only harmless variables.*

The set of rules in Example 2 is warded, since although variable p is dangerous, in the body it only appears in KeyPerson, which is the ward. We assume the presence of a unary *active constant domain* relation Dom [24], such that if any variable x occurring in $\text{Dom}(\bar{x})$, can bind only to EDB values [10].

Other Datalog$^\pm$ Languages. We now give a brief description of languages mentioned, but not used in this paper. In Linear Datalog$^\pm$, rule bodies are required to consist of only one atom. Guarded Datalog$^\pm$ requires all universally quantified variables to occur within a single body atom. Sticky Datalog$^\pm$ and Weakly Sticky Datalog$^\pm$ pose syntactic restrictions based on a variable marking procedure. Weakly Frontier Guarded Datalog$^\pm$ tames the propagation of nulls by syntactic restrictions. In that sense, it is similar (but less strict) than Warded Datalog$^\pm$: It can be obtained from Warded Datalog$^\pm$ by dropping the requirement that the ward shares with other atoms only harmless variables. More details can be found in [23].

3. TERMINATION AND RECURSION CONTROL

In this section, we present the core of the Vadalog system – a practical algorithm with termination guarantees, yet allowing high performance and a limited memory footprint. This algorithm exploits the deep structural properties of Warded Datalog$^\pm$, so our main focus in this section is to describe the key ideas behind the algorithm. In Section 4, describing the technical architecture, we will show how the algorithm is actually exploited in a processing pipeline.

Let us now consider a set of rules, based on a significant portion of the real-life company control scenario (only advanced features have been omitted), already touched on in Example 2, which we will use throughout the next sections as a running example. In particular, here we consider the "significantly controlled companies", that is, the companies for which there exist significant shareholders who hold more than 20% of the stocks.

Example 3. *Assume we have the following database instance and set of rules, which are textually described afterwards.*

$$D = \{\text{Company(HSBC)}, \text{Company(HSB)}, \text{Company(IBA)}, \text{Controls(HSBC, HSB)}, \text{Controls(HSB, IBA)}\}$$

$$1 : \text{Company}(x) \rightarrow \exists p \exists s\ \text{Owns}(\hat{p}, \hat{s}, x)$$
$$2 : \text{Owns}(\hat{p}, \hat{s}, x) \rightarrow \text{Stock}(x, \hat{s})$$
$$3 : \text{Owns}(\hat{p}, \hat{s}, x) \rightarrow \text{PSC}(x, \hat{p})$$
$$4 : \text{PSC}(x, \hat{p}), \text{Controls}(x, y) \rightarrow \exists s\ \text{Owns}(\hat{p}, \hat{s}, y)$$
$$5 : \text{PSC}(x, \hat{p}), \text{PSC}(y, \hat{p}) \rightarrow \underline{\text{StrongLink}}(x, y)$$
$$6 : \text{StrongLink}(x, y) \rightarrow \exists p \exists s\ \text{Owns}(\hat{p}, \hat{s}, x)$$
$$7 : \text{StrongLink}(x, y) \rightarrow \exists p \exists s\ \text{Owns}(\hat{p}, \hat{s}, y)$$
$$8 : \text{Stock}(x, \hat{s}) \rightarrow \text{Company}(x).$$

For a significantly controlled company x, there exists a person p who owns significant shares s of it (rule 1). The predicate Owns denotes "significant ownership". When p owns s% of x, then s is part of the share split of the company stock (rule 2). If p owns a significant share of x, then p is a "person of significant control" for x (rule 3). Controls(x, y) *holds when a company x exerts some form of significant control over company y; if* Controls(x, y) *holds, for every person p having significant control over x, then there exist some share s ($s > 20\%$) , such that p owns s% of y (rule 4). If two companies share a person with significant control, they have a strong link (rule 5). If there is a strong link between x and y, there*

is some person p who owns significant share s of x (rule 6); the same applies to y (rule 7). Finally, the existence of a stock for x, implies that x is a company (rule 8). We are interested in knowing all the strong links between companies (underlined).

3.1 Warded Forest

The key for guaranteeing termination of a chase-based procedure is defining points at which the chase procedure may be terminated prematurely, while at the same time upholding correctness of the reasoning task, i.e., guaranteeing that all tuples that are part of the output have already been produced. We shall call a principle by which such "cut-off" of the chase tree can be achieved a *termination strategy*. Generally speaking, the earlier this cut-off can be achieved, the better our termination strategy. In particular, for it to be effective, it has to always cut-off after a finite part of the, in general, unboundedly large chase graph. Yet at the same time, the overhead of the strategy itself has to be taken into consideration.

More formally speaking, we define the *chase graph* for a database D and a set of rules Σ, as the directed graph *chase-graph*(D, Σ) having as nodes the facts obtained from chase(D, Σ) and having an edge from a node **a** to **b** if **b** is obtained from **a** and possibly from other facts by the application of one chase step, i.e., of one rule of Σ.

As the main vehicle towards understanding the chase graph for Warded Datalog$^{\pm}$, we introduce our first major tool, namely the *warded forest* of a chase graph: the subgraph that consists of all nodes of the chase graphs, all edges of the chase graph that correspond to the application of linear rules, and one edge for each non-linear rule – namely the one from the fact bound to the ward. Thus, by this definition, the connected components in a warded forest are explicitly determined (i.e., separated from each other) by joins involving constants or, in other words, in each connected component we only have edges representing the application of linear rules or rules with warded dangerous variables. As an effect, every single fact will then inherit all its labelled nulls from exactly one fact, either its direct parent in the application of a linear rule or the ward in the application of a warded rule.

Figure 2 shows a portion of the (infinite) warded forest of the chase graph for our running example (Example 3). In particular, solid edges derive from the application of linear rules, and dash-dotted edges derive from non-linear rules where one fact involved in the join binds to a ward. The gray rectangles denote the subtrees in the warded forest: they are generated by non-linear rules where none of the facts involved in the join bind to a ward, i.e., there is no propagation of dangerous variables to the head.

We say that two facts are *isomorphic* if they refer to the same predicate name, have the same constants in the same positions and there exists a bijection of labelled nulls into labelled nulls. Then, we define a *subtree*(**a**) as the subtree of the warded forest that is rooted in **a**. We say that two subtrees are isomorphic if the usual notion of graph isomorphism holds with the given definition of isomorphism between facts.

THEOREM 1. *Let* **a** *and* **b** *be two facts in a warded forest. If they are isomorphic, then* subtree(**a**) *is isomorphic to* subtree(**b**).

In the warded forest in Figure 2, we see examples of isomorphic facts, which give rise to isomorphic subtrees (each shown only once): those rooted in Company(HSBC), in Owns$(p, s,$ HSBC), and so on. Theorem 1 gives a fundamental result in terms of description of the "topology" of the warded chase forest, since we see that each subtree is uniquely identified, up to isomorphism of the nulls, by its root. It is certainly an important hint for the exploration of the chase graph, as it points out a form of structural periodicity that we will exploit to efficiently guarantee termination.

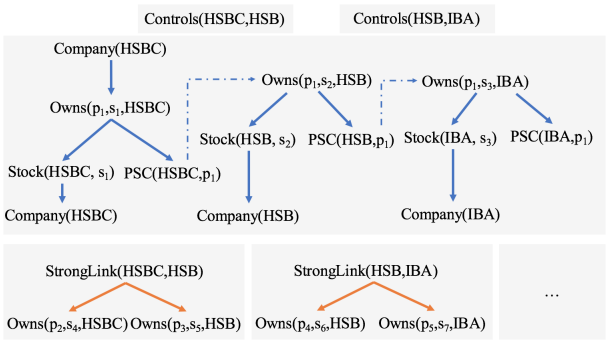

Figure 2: A portion of the warded forest for the chase graph of rules in Example 3. Solid edges denote linear rules and the dash-dotted edges denote joins in warded rules.

3.2 Harmless Warded Datalog$^{\pm}$

We now make a step forward towards the definition of a concrete termination strategy in the application of the chase. Theorem 1 gives a good direction, since it ensures that for isomorphic facts, subtrees rooted at these facts will be isomorphic as well. Therefore, ideally, when the chase produces a fact that is isomorphic to one that has been generated at a previous step, we could avoid exploring its successors, since the derived subtree would certainly be isomorphic to an already generated one and therefore irrelevant for our reasoning task. Unfortunately, this property of subtrees in warded forests does not extend to generic chase graphs (and graph isomorphism), as shown by the following example.

EXAMPLE 4. *From database D of Example 3, we can generate a fact* Owns$(p_1, s_1,$ HSBC$)$ *with rule 1, and* Owns$(p_1, s_2,$ HSB$)$ *by subsequently applying rule 3 (which produces* PSC(HSBC, p_1)*) and rule 4. By applying rule 3 on* Owns$(p_1, s_2,$ HSB$)$*, we also obtain* PSC(HSB, p_1)*. From* Company(HSB)*, via rule 1 and rule 3, we obtain* PSC(HSB, p_2)*. Then, we see that although* PSC(HSB, p_1) *and* PSC(HSB, p_2) *are isomorphic, the respective subtrees are not:* PSC(HSB, p_1) *has* StrongLink(HSBC, HSB) *as a child node, which derives from the join with* PSC(HSBC, p_1) *in rule 5; conversely,* PSC(HSB, p_2) *does not join with any* PSC *facts that regard* HSBC.

Example 4 shows that pruning a chase graph on the basis of isomorphism on labelled nulls is not correct. However, if we consider a syntactical restriction of Warded Datalog$^{\pm}$, namely *Harmless Warded Datalog*$^{\pm}$, where joins on harmful variables are forbidden, then Theorem 1 extends to generic chase graphs.

THEOREM 2. *Let* **a** *and* **b** *be two facts in the chase graph of a set of harmless warded rules. If* **a** *and* **b** *are isomorphic, then* subgraph(**a**) *is isomorphic to* subgraph(**b**).

Theorem 2 immediately suggests a (typically not very efficient) terminating algorithm for Harmless Warded Datalog$^{\pm}$, namely by modifying the chase in two aspects: (1) *detection*, that is, memorizing all facts generated during the chase up to isomorphism and (2) *enforcement*, which consists in cutting off the chase when a fact isomorphic to a memorized one would be generated.

Correctness of the algorithm follows from Theorem 2 and the fact that our reasoning task is not affected by the multiplicity of isomorphic copies of facts. Yet, this algorithm is of course impractical, as memorizing the chase up to isomorphism and performing the full isomorphism check is prohibitive in space and time as we will show experimentally in Section 5.6.

A limitation of Theorem 2, but only an apparent one, is that is restricted to Harmless Warded Datalog$^{\pm}$. Yet, it fully extends to Warded Datalog$^{\pm}$ as a set of warded rules can be always rewritten as an equivalent (i.e., giving the same result for any instance of our reasoning task up to null homomorphism) set of harmless warded rules. For this purpose we devise a *Harmful Joins Elimination Algorithm*, which we show in detail in the technical report [10]. In the following example we show the result of removing harmful joins from rule 5 of Example 3.

Example 5. $\text{Dom}(p), \text{PSC}(x, p) \rightarrow \text{PSC}'(x, p)$
$\text{PSC}'(x, p), \text{PSC}'(y, p) \rightarrow \text{StrongLink}(x, y)$
$\text{Company}(x), \text{Controls}(x, y) \rightarrow \text{StrongLink}(x, y), \text{StrongLink}(y, x)$
$\text{Company}(x), \text{Controls}(x, z), \text{StrongLink}(z, y) \rightarrow$
$\text{StrongLink}(x, y), \text{StrongLink}(y, x).$

Essentially, the first two rules encode the grounded version of the original rule (i.e., one only considering constants). They are obtained by creating a ground copy of the predicate holding the harmful variable. The last two rules encode transitive closure of the StrongLink relation. They are obtained by means of a rewriting-based technique, which replaces atoms that contain harmful variables. Thanks to the definition of Warded Datalog$^{\pm}$, this procedure always terminates.

3.3 Lifted Linear Forest

In Section 3.1 we have seen how Theorem 1 guarantees isomorphism of subtrees in warded forests based on the isomorphism of their roots. In Section 3.2 we showed how this result can be extended to chase graphs (Theorem 2) if the rules do not contain harmful joins (Harmless Warded Datalog$^{\pm}$).

Ideally this would be enough for the implementation of a termination strategy, since the chase could skip the exploration of subgraphs isomorphic to other nodes that have already been explored (*vertical pruning*). This would however have practicability limitations in that the algorithm would rely on the specific ground values in the facts to deduce the isomorphism of the derived subgraphs with the result of very low chances to detect isomorphic subtrees.

In this section we introduce the notion of *lifted linear forest*, a structure that allows to apply the results on isomorphisms independently of the specific ground values by grouping subtrees with the same "topology" into equivalence classes.

We say that two facts are *pattern-isomorphic* if they have the same predicate name, there exists a bijection between the constant values and there exists a bijection between the labelled nulls. For example, $\text{P}(1, 2, x, y)$ is pattern-isomorphic to $\text{P}(3, 4, z, y)$, but not to $\text{P}(5, 5, z, y)$. By extension, we define two graphs as pattern-isomorphic if the nodes are pairwise isomorphic and they have coinciding edges. By means of pattern-isomorphism we can group the facts into equivalence classes and we denote with $\pi(\mathbf{a})$ the representative of the equivalence class to which $\mathbf{a}$ belongs. In other terms, π is a function such that it maps pattern-isomorphic facts into the same element, which we call *pattern*; for example, it could map $\text{P}(1, 2, x, y)$ and $\text{P}(3, 4, z, y)$ into $\text{P}(c_1, c_2, \nu_1, \nu_2)$ and $\text{P}(5, 5, z, y)$ into $\text{P}(c_1, c_1, \nu_1, \nu_2)$, though any other representation for patterns would be acceptable. A notion of *graph pattern-isomorphism* can be thus easily derived by extension.

An immediate use would be grouping subgraphs of the chase graphs into equivalence classes, so that Theorem 2 can be applied more broadly, deriving subgraph pattern-isomorphism on the basis of the pattern-isomorphism of the respective root nodes. This would open to the implementation of a form of *horizontal pruning*, i.e., vertical pruning modulo pattern-isomorphism. Unfortunately this is not possible: given two pattern-isomorphic nodes $\mathbf{a}$ and $\mathbf{b}$, the two derived subgraphs in the chase graph can be different indeed, as one can have a fact $\mathbf{a}'' \in \textit{subgraph}(\mathbf{a})$, pattern-isomorphic to $\mathbf{b}'' \in \textit{subgraph}(\mathbf{b})$ such that there is a third fact $\mathbf{k}$ sharing a constant c with $\mathbf{a}''$ but not with $\mathbf{b}''$. A non-linear rule could then join $\mathbf{a}''$ with $\mathbf{k}$, producing $\mathbf{a}'$. Since $\mathbf{k}$ does not join with $\mathbf{b}''$, in $\textit{subgraph}(\mathbf{b})$ there would be no facts pattern-isomorphic to $\mathbf{a}'$. This means that in harmless warded forests, subgraph isomorphism is independent of the specific values of the labelled nulls but is dependent on the specific values of the constants.

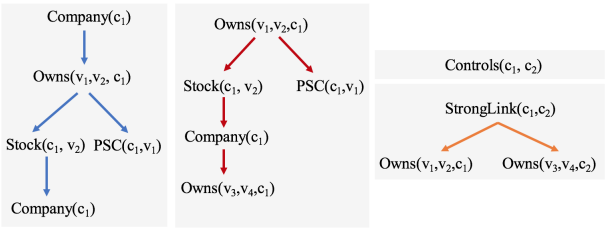

Figure 3: The lifted linear forest for the chase graph of Example 3.

Towards overcoming this limitation, we introduce the notion of *linear forest*. A linear forest for a chase graph is the subgraph that contains all the facts from the graph and only the edges that correspond to the application of linear rules (one atom in the body).

We have an example of linear forest in Figure 2, where it is denoted by the subgraph where only the solid edges are considered. In a linear forest, pattern-isomorphism on roots extends to pattern-isomorphism of subtrees, as shown by the following property.

Theorem 3. *Let* $\mathbf{a}$ *and* $\mathbf{b}$ *be two facts in a linear forest. If they are pattern-isomorphic, then* subtree($\mathbf{a}$) *is pattern-isomorphic to* subtree($\mathbf{b}$).

Inspired by the application of Theorem 3 to the notion of linear forest, we define the *lifted linear forest* for a chase graph as the set of equivalence classes modulo subtree pattern-isomorphism of the linear forest. For each node $\mathbf{n}$ of a linear forest, the corresponding node $\pi(\mathbf{n})$ belongs to the lifted linear forest; the subtrees of the linear forest whose roots are pattern-isomorphic are thus collapsed into the same connected component of the lifted linear forest. The term "lifted" suggests that such data structure seizes the structural symmetries in the linear forests and factors them out. An example of lifted linear forest is presented in Figure 3, where we show how the chase graph for Example 3 is simplified.

In linear forests, a termination strategy could thus feature both vertical and horizontal pruning. This however comes at a price: as trees in linear forests tend to be less deep than in warded forests because of the exclusion of the edges for joins in warded rules, node isomorphisms are less beneficial, as they allow to prevent fewer isomorphism checks. The next section shows how we combine warded and lifted linear structures to achieve the best trade off.

3.4 The Algorithm

With all ingredients ready, we now present the main algorithm of the Vadalog system. Algorithm 1 is a termination strategy in the sense that it guides chase(D, Σ) and returns whether each chase step must be activated or not. Thus it controls the generation of new facts and, hence, the termination of the (in general infinite) chase.

The core principle of it is incrementally building a warded forest as in Theorem 2 and a lifted linear forest, and, at the same time, exploiting the already available portions of these forests in multiple ways. Specifically, we will use the warded forest for an optimized form of local detection of isomorphic facts and the lifted linear for

a highly reusable and compact summary of the detected patterns for their enforcement. This termination strategy can be part of a naïve chase algorithm, but more practically – for achieving good performance – in the Vadalog system is applied in the streaming-based pipeline described in Section 4.

The algorithm uses a number of data structures based on one of the three main concepts introduced so far. In particular, the properties associated to facts (namely root and provenance) are mostly based on the concept of *linear forest*. The ground structure, which is the target of isomorphisms checks, is based on the *warded forest*. We saw a full example of this in Figure 2. Finally, a summary structure that determines when skipping isomorphisms checks is possible, is based on the *lifted linear forest*. We saw an example of this in Figure 3. We describe each of them in detail now.

The **fact structure** is a structured representation of a fact **a**, with four fields. 1. *generating_rule:* The kind of rule of Σ (linear / warded / non-linear) which generated **a**. Here a "warded" rule is a rule where there is actually a join over a harmless variable in a ward and a dangerous variable is propagated to the head. 2. *l_root:* The root of the tree (i.e., of the connected component) containing **a** in the linear forest. 3. *w_root:* The root of the tree (i.e., of the connected component) containing **a** in the warded forest. 4. *provenance:* The provenance of **a** in the linear forest, defined as the list $[\rho_1, \ldots, \rho_n]$ of the rules that have been orderly applied in the chase from **a**.l_root to obtain **a**. In particular we have $\mathbf{a} = \rho_n(\ldots(\rho_1(\mathbf{a}.l\_root)))$, where $\rho(\mathbf{a})$ denotes the application of a rule ρ to a fact **a** in the chase. On provenance we also define the *inclusion relation* $\subseteq$ as the set of pairs $\langle p_i, p_j \rangle$ such that p_i is an ordered (and possibly coinciding) left-subsequence of p_j.

The **ground structure** G stores the nodes of the warded forest incrementally built during the chase, grouping them by the root of the tree in the warded forest to which each node belongs. G is a dictionary of sets of facts. More precisely, each element G[**a**.w_root] represents the set of facts of the tree rooted in **a**.w_root in the warded forest. Based on G, we define the provenance $\lambda = [\rho_1, \ldots, \rho_n]$ for **a** as *non-redundant* if every $\mathbf{a}^{\mathbf{i}} = \rho_i(\ldots(\rho_1(\mathbf{a}.l\_root)))$ for $i = 1, \ldots, n$ is not isomorphic to any $\mathbf{a}^j \in G[\mathbf{a}.w\_root]$. We define λ as a *stop-provenance* if it is a $\subseteq$-maximal non-redundant provenance. In more plain words, λ is non-redundant if no intermediate fact in the chase from **a**.l_root to **a** is isomorphic to any other fact in the entire tree of **a** of the warded forest; it is a stop-provenance if every provenance $\lambda' \supset \lambda$ does not satisfy non-redundancy. Stop-provenances then represent a maximal root-leaf path in a tree of the lifted linear forest, i.e., any extension of it is bound to contain a node that is pattern-isomorphic to another node of the same tree.

The **summary structure** S stores the lifted linear forest, which is incrementally built during the chase. In particular, it memorizes the complete paths from the roots, which are the patterns of the roots of the underlying linear forest, to the leaves in the form of stop-provenances. S is a dictionary of sets of stop-provenances, indexed by common root pattern. At a given point in time, each element S[π(**a**.l_root)] of S thus represents the partial tree of the lifted linear forest, with maximal root-leaf paths.

Let us now analyze the dynamic aspects of the algorithm. We assume that: 1. the rules at hand are harmless warded; 2. existential quantifications appear only in linear rules. For any set of warded Datalog rules both conditions can be always achieved: the first by applying the Harmful Joins Elimination Algorithm (Section 3.2), the second with an elementary logic transformation.

Linear and warded rules produce facts **a**, for which, in the base case, isomorphism checks must be performed. The ground structure allows to restrict this costly check to the local connected component of the warded forest (line 8-10), featuring a form of *local detection*. If an isomorphic fact is found (line 8), the algorithm stores the stop-provenance of **a** in such a way that: 1. whenever the same sequence of rules is applied from **a**.l_root, the result can be discarded (enforcement with vertical pruning) without performing any isomorphism check (line 3); 2. whenever a subsequence of rules is applied from **a**.l_root, no superfluous isomorphism checks are performed (line 5). Moreover, since we want to maximize the reuse of the learnt stop-provenance so as to apply it to as many facts as possible and independently of the specific ground values (enforcement with horizontal pruning), the algorithm maps **a**.l_root into the respective root of the lifted linear forest and stores the provenance in into S w.r.t. to the pattern $\pi(\mathbf{a}.l\_root)$ (line 9). Other, (non-warded) non-linear rules are the roots of new trees (connected components) of the warded forest (line 14). Since isomorphism checks are only local to the trees in the warded forest, the current fact provenance can be forgotten for memory reasons. New trees are generated unless their root is already present in G, which by Theorem 2 would imply that the isomorphism of the entire subtree. As we assume that non-linear rules do not have existential quantification, the condition can be efficiently checked as set containment of ground facts.

Algorithm 1 Termination strategy for the chase step.

```
1:  function CHECK_TERMINATION(a)
2:      if a.generating_rule == {LINEAR or WARDED} then
3:          if ∃λ ∈ S[π(a.l_root)] s.t. λ ⊆ a.provenance then
4:              return false                    ▷ beyond a stop provenance
5:          else if ∃λ ∈ S[π(a.l_root)] s.t. a.provenance ⊂ λ then
6:              return true                     ▷ within a stop provenance
7:          else                                ▷ continue exploration
8:              if ∃g in G[a.w_root] s.t. a isomorphic to g then
9:                  S[π(a.l_root)]=a.provenance
10:                 return false                ▷ isomorphism found
11:             else
12:                 G[a.w_root].append(a)
13:                 return true                 ▷ isomorphism not found
14:     else if a ∉ G then                      ▷ other non-linear generating rules
15:         G[a.w_root].append(a)               ▷ and reset provenance
16:         return true
17:     else                                    ▷ the new tree is redundant
18:         return false
```

Our algorithm guarantees chase termination as a consequence of Theorem 1. Actually, the set of distinct symbols that can be present as terms of facts in *subtree*(**a**) of the warded forest is bounded up to isomorphism of the labelled nulls. Algorithm 1 provides an effective strategy, by applying a form of *lazy pattern recognition* that exploits regularities in the warded forest as soon as they are detected. The roots of the warded trees represent the "handles" for detecting such regularities. Besides, by Theorem 2, we know that relying on such isomorphisms in the chase graph is correct in the sense that fact isomorphism in the guarantees the isomorphism of the respective subgraphs. The roots of the lifted linear trees represent the handles for storing the learnt regularities in a generalized way and enforcing them afterwards. The generalization of isomorphism of trees to patterns-isomorphism based on roots patterns is correct as a consequence of Theorem 3.

The most time-demanding operation in Algorithm 1 is isomorphism check, which in a naïve chase step implementation would require checking each fact against the entire database. We significantly limit the number of isomorphism checks and the cost of the single check. The lifted linear forest allows to minimize the num-

ber of checks, basically avoiding superfluous controls for pattern-isomorphic facts generated in sub- or super-paths of known ones.

In terms of memory footprint, S is extremely compact as the lifted linear forest does not contain the ground values, but only their equivalence classes modulo constant- and variable- isomorphism.

4. ARCHITECTURE

In Section 3.4, we presented a practically useful algorithm for supporting reasoning with Warded Datalog$^{\pm}$. Specifically, Algorithm 1 guarantees the termination of chase-based procedures by controlling the recursion, and is already optimized in the sense it avoids the typical overheads that chase-based algorithms usually need to spend on homomorphism checks.

Yet, the adoption of a traditional chase-based procedure in the presence of large amounts of data has a number of disadvantages, in particular requiring the entirety of original and generated data to be available as possible inputs for chase steps, untenable for large-scale settings. In this section, we show how the limitations of the chase are overcome by presenting the specialized architecture that we devised for the Vadalog system. We show that our termination algorithm plays a fundamental role in it.

Pipeline architecture. One of the core design choices in the implementation of the Vadalog system is the use of the *pipe and filters* architectural style, where the set of logic rules and the queries are together compiled into an active *pipeline* that reads the data from the input sources, performs the needed transformations, and produces the desired output as a result. This is actually done in four steps. 1. A *logic optimizer* applies several logic transformations to the original set of rules, consisting in elementary (e.g., multiple head elimination, elimination of redundancies, etc.) and complex (e.g., elimination of the harmful joins) rewritings of the rules. 2. A *logic compiler* reads the object representation of the rules received from the *rule parser* and transforms it into a *reasoning access plan*. The reasoning access plan is a logic pipeline, where each rule of the program corresponds to a filter (i.e., a node) in the pipeline and there is a pipe (i.e., an edge) from one filter a to a filter b, if rule a has in the body an atom that unifies with the head of a. 3. The *execution optimizer* transforms the reasoning access plans, performing basic optimizations, such as rearranging the join order, pushing selections and projections as close as possible to input nodes and so on. 4. A *query compiler* turns the reasoning access plan into a *reasoning query plan*, an active pipeline (shown in Figure 4), capable to perform the needed transformations (e.g., projections, selections, joins, application of functions, creation of new values, aggregations, etc.) and producing the output data.

Execution model. In VADALOG, some rules are marked as "input" (by specific annotations) as they represent external data sources and therefore are mapped to source filters in the query plan; vice versa, terminal nodes are marked as "output" and represent sink filters in the pipeline. The reasoning process is then realized as a data stream along the pipeline, implemented with a *pull (query-driven) approach*. We implement a generalization of the *volcano iterator model* [25] used in DBMSs, extending it to the entire pipeline. Each filter pulls the required input from the respective sources, which in turn, pull from their sources down to the initial data sources. The initial data sources, use *record managers*, specific components that act as adapters towards external sources, turning input streaming data into facts. Therefore the entire process is triggered and driven by the sink nodes, which issue `open()`, `next()`, `close()` messages to their predecessors, which receive and propagate the messages back down to the source nodes. The `next()` primitive returns a Boolean indication of the presence of facts for a specific

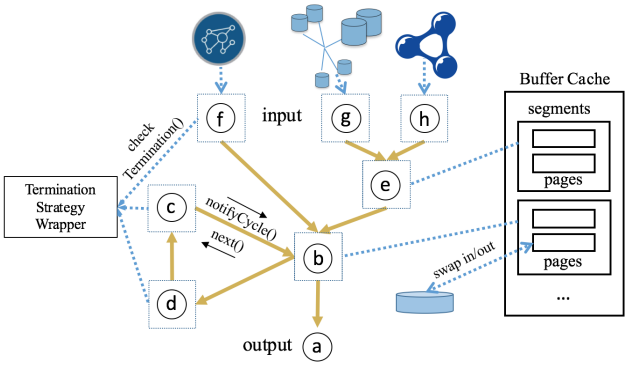

Figure 4: The pipeline architecture of the Vadalog system.

requestor. The facts are then accessed with specific typed `get` primitives. The behavior of the `next()` primitive is driven by the direct availability of facts in the invoked filter as well as further logics that control termination according to Algorithm 1, as we will see.

Since for each filter, multiple parent filters may be available, our execution model adopts a *round-robin strategy*, that is, it tries to pull data from all the available parent filters, in a predefined order. We empirically verified that such strategy guarantees a good balance of the workload among the filters and a uniform propagation of the facts from the sources to the sinks. From another perspective, the round-robin strategy sustains a breadth-first application of the rules, where a rule is re-applied, only when all the others have been applied in turn (successfully or not), and in the same order. In the example in Figure 4, the output filter a sends a `next()` message to b, which is propagated to c, f and e.

There are two factors that make the stream-based processing definitely non-trivial: not surprisingly, the presence of typically "blocking" operations (in particular, the join and the aggregations) and, more important to our case, the possible cycles in the pipeline, which can be induced by the recursion in the rules and can lead to non-termination of the streaming process. As detailed in the technical report [10], we support non-blocking aggregation as a specific feature of the VADALOG language, which avoids the presence of blocking nodes in this case.

Cycle management. Non-terminating sequences directly derive from the presence of recursion (and hence cycles): in absence of further checks in the sink filters, a recursive pipeline may generate infinite facts. To cope with this, we feature *termination strategy wrappers*, components that implement Algorithm 1 and work as follows. When a filter receives a `next()` call, it agnostically preloads a fact $A(c)$, in the sense that it issues all the necessary `next()` primitives to its predecessors, handles the runtime cycles, so that $A(c)$, if available, is stored in the filter data structure to be possibly consumed. The filter issues a `checkTermination(A(c))` message to its local termination wrapper, which applies Algorithm 1. The termination strategy wrappers also manage the *fact*, *ground* and *summary* structures of Section 3.4. Then, if the termination check is negative, $A(c)$ is discarded as it would lead to non-termination.

5. EXPERIMENTAL EVALUATION

In this section we evaluate the performance of the Vadalog system on two synthetic (Sections 5.1 and 5.2) and two real-world scenarios (Sections 5.3 and 5.4), all involving non-trivial warded Datalog$^{\pm}$ rules (with many existentials, harmful joins, null propagation, etc.). We thoroughly validate our theoretical results and the architecture on such cases and show that our reasoner exhibits very good scalability and outperforms existing systems.

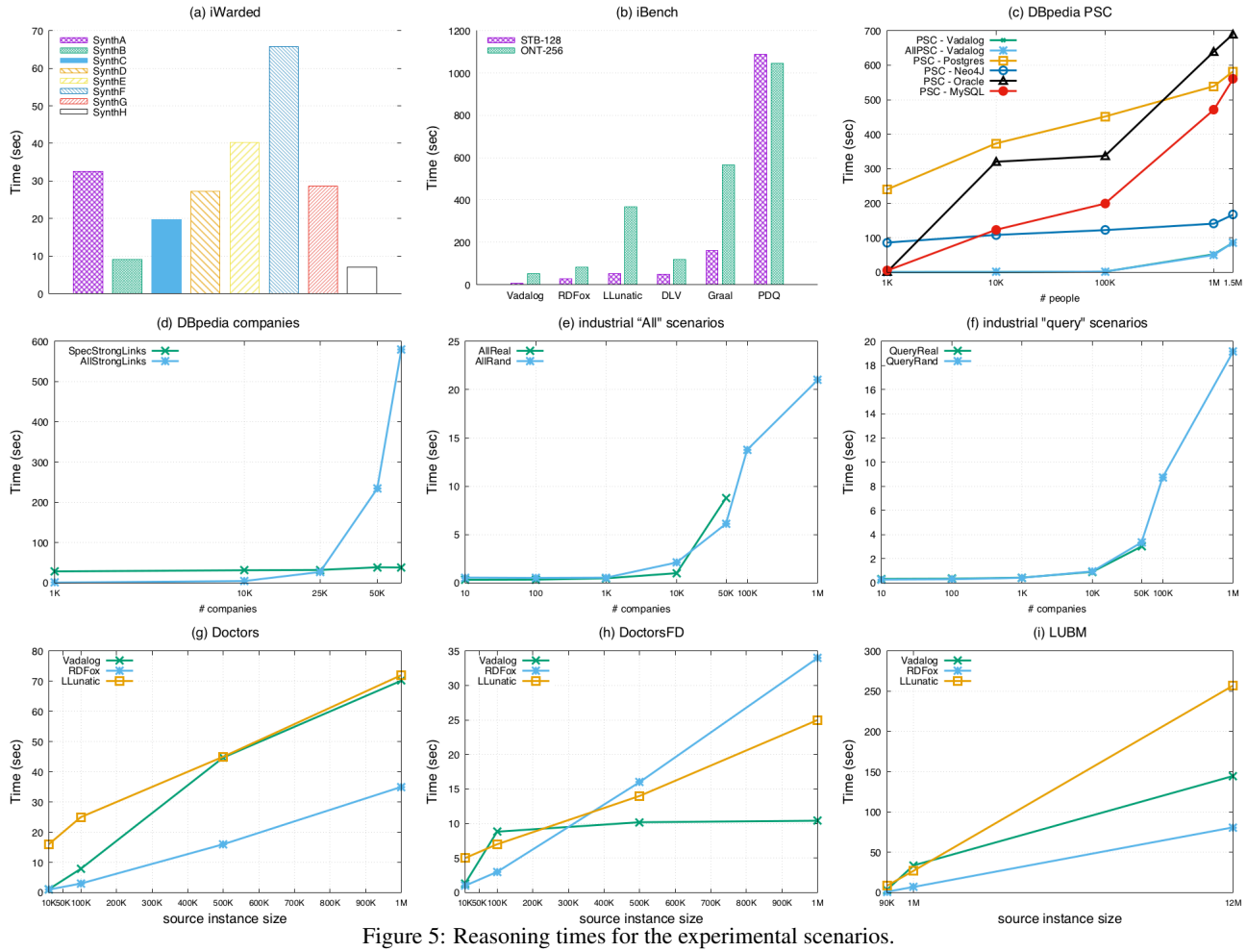

Figure 5: Reasoning times for the experimental scenarios.

As a supplementary experimental contribution (Section 5.5), we then show that the Vadalog system is also a best-in-class general purpose chase/query answering/data integration system, by comparing its results with top-performing systems on sets of rules where the typical characteristics of wardedness cannot be exploited. We highlight the advantage of our approach over pure isomorphism checks with specific experiments (Section 5.6) and evaluate scalability along many different dimensions (Section 5.7).

Test setup. The reasoner was used "as a library" and invoked from specific Java test classes for end-to-end, i.e., storage to storage executions of the reasoning. For the storage, we adopted simple CSV archives to precisely highlight the performance of the reasoner, independently of any optimized back-end physical structures, such as indexes, and so on. Clearly, in some cases our tasks could benefit from back-end rewriting (e.g., performing joins or aggregations in the DBMS), yet here we intentionally execute every operation purely inside of our system. We also used local installations of PostgreSQL 9.6, Oracle 11g, MySQL 5 and Neo4J 3.2.

Hardware configuration. We ran the tests on a Linux server with 8 Xeon v3 cores running at 2.4 GHz and with 16GB of RAM.

5.1 iWarded: Synthetic Warded Scenarios

In this section we investigate the impact of specific properties of the warded rules on the performance of reasoning tasks in the Vadalog system. To this end, we developed iWarded, a flexible generator of warded rules that allows to control the internals related to Warded Datalog$^{\pm}$ such as the number of linear and non-linear rules, the presence of harmful or harmless joins, recursion, etc.

Description of the scenarios. We built eight scenarios, all with 100 rules and the same set of (multi-)queries that activates all the rules. *SynthA* has a prevalence of linear rules and 20% of the total rules have existential quantification; 30% of linear and non-linear rules are recursive, and the joins are equally distributed between harmless-harmful and harmless-harmless, for which we have a prevalence of joins involving wards. *SynthB* is specular to *SynthA*, but has a prevalence of non-linear rules; all the other proportions are respected accordingly. *SynthC* (as well as all the next settings) has a 30%-70% balance between linear and non-linear rules, which we posed to highlight the impact of the (various kinds) of joins on the performance. In this setting, harmful-harmful join are present. *SynthD* stresses the presence of such joins even more. *SynthE* studies the impact of a strong presence of recursion of the non-linear rules. *SynthF* deals with a high recursion incidence on linear ones. *SynthG* has basically the same characteristics as *SynthC* and *SynthD*, but with a prevalence of harmless rules that do not involve wards, therefore its characteristics resemble those of a pure Datalog program. *SynthH* emphasizes the joins on wards. The full table with the detailed composition of the rules is in the report [10].

Results. The results are reported in Figure 5(a). The Vadalog system shows the best performance in scenarios *SynthB* and *SynthH*, for which the execution times are under 10 seconds. The prevalence of join rules, with particularly high values for harmless

joins with wards allows to exploit the wardedness at best. Warded joins produce particularly deep warded forests, making the isomorphism check for each generated fact particularly effective: the deeper the warded forests are, the more likely it is that an isomorphic fact is met within the same structure during the isomorphism check in the ground structure. Therefore the summary structure is updated more frequently, maximizing the pattern learning and letting the algorithm converge sooner. Then, scenario *SynthC* acts as a baseline, with an average number of rules and balanced join types. Although scenario *SynthD* has less than half the harmless joins with ward of *SynthC*, it requires 7 seconds more due to a higher number of harmful joins. Scenario *SynthG*, has 60 harmless joins without wards, i.e., it resembles the result of the harmful join elimination applied to *SynthD* and therefore shows almost the same times, though slightly affected by the presence of more existentials that give rise to more patterns. *SynthG* is indeed very important, since it pinpoints the behavior of the Vadalog system in the presence of generic Datalog programs, where the properties of warded rules cannot be exploited. The performance is indeed good, as is also confirmed in Section 5.5 by the comparison with chase-based systems, though typical optimizations of Datalog (foreseen as a future optimization) will bring improvements. Finally, scenarios *SynthE* and *SynthF* show that the impact of recursion on performance is significant, with times of about 40 and 65 seconds, respectively.

The memory footprint is confirmed to be limited. All the scenarios require less than 400MB of memory. Interestingly, in the top-performing scenarios, all the required memory is immediately allocated. This confirms that deep warded forests cause a rapid growth of ground and summary structures.

5.2 iBench

iBench [4] is a popular tool to generate *data integration scenarios* of which scenarios *STB-128* and *ONT-256* were featured in ChaseBench [11]. These scenarios are particularly relevant for our purposes, as they are highly recursive and consist of non-trivially warded rules, i.e., there are many existentials, labelled nulls are often propagated to the heads and also involved in many joins.

Description of the scenarios. *STB-128*: It is a set of about 250 warded rules, 25% of which contain existentials; there are also 15 cases of harmful joins and 30 cases of propagation of labeled nulls with warded rules. The expected target instance contains 800k facts, with 20% of labelled nulls. On this scenario we ran 16 different queries and averaged the response times. Queries are rather complex: involve on average 5 joins, harmful in 8 cases.

ONT-256: It is a set of 789 warded rules, 35% of which contain existentials; there are 295 cases of harmful joins and more than 300 propagations of labelled nulls. Rules are even more complex than *STB-128*, and contain multiple joins as well as pervasive recursion. The expected target instance contains ~2 million facts, with an incidence of 50% of labelled nulls. On this scenario we ran 11 different queries and averaged the response times. Queries involve an average of 5 joins, which in 5 cases are harmful.

Results. In Figure 5(b) we depict the times obtained for the Vadalog system as well as those reported in ChaseBench for the other mentioned systems in a comparable hardware/software environment. The Vadalog system outperforms all systems in both the scenarios: with 6.59 seconds for *STB-128* and 51.579 seconds for *ONT-256*, it is on average 3 times faster than the best performing chase-based system, RDFox and 7 times faster than LLunatic. In particular, the three best performing systems, RDFox, LLunatic and DLV reported 28, 52 and 48 seconds for *STB-128* and 83, 367, and 118 seconds for *ONT-256*.

5.3 DBpedia

We built four reasoning tasks based on real datasets about companies and persons extracted from DBpedia [20].

Description of the scenarios. For the DBpedia entity `Company`, we considered the company name, the `dbo:parentCompany` property, holding all the controlled companies, and `dbo:keyPerson`, holding all the persons relevant to the company. We mapped parent companies into facts Control(x, y), if company x controls y, and the key persons into facts KeyPerson(x, p), if a company x has a key person p. DBpedia publishes ~67K companies and ~1.5M persons.

PSC: The PSC (*persons with significant control*) are the set of the persons that directly or indirectly have some control on a company. The goal of this reasoning setting is finding all the PSC for all the companies of DBpedia.

Example 6. KeyPerson(x, p), Person(p) → PSC(x, p)
Control(y, x), PSC(y, p) → PSC(x, p).

More precisely, we say that a person p is a PSC for a company x if either p is a key person for x, or x is controlled by a company y and p is a PSC for y. We ran the scenario, reported in Example 6, for all the available companies (67K) and for subsets of 1K, 10K, 100K, 1M and 1.5M of the available persons. We also executed the same scenario with indexed tables in PostgreSQL, MySQL, Oracle and graphs in Neo4J, translating the set of rules of Example 6 into recursive SQL and Cypher, respectively.

AllPSC: This scenario is a variation of the previous one, where we actually want to group in a single set all the PSC for a company.

Towards the next scenarios, we say that if companies x and y share more than a number N of PSC, then there is a *strong link* between them. Every company x has at least one PSC, which can be either a KeyPerson for the company or another, unknown person. Based on the set of rules in Example 7 we built two scenarios.

Example 7. KeyPerson(x, p) → PSC(x, p)
Company(x) → $\exists p$ PSC(x, p)
Control(y, x), PSC(y, p) → PSC(x, p)
PSC(x, p), PSC(y, p), $x > y$, $w \geq N$,
w = mcount(p) → StrongLink(x, y, w)

SpecStrongLinks: We want to obtain all the strong links of a specific company, `Premier_Foods`, with $N = 1$ (which means that if a company shares one PSC with `Premier_Foods`, it is a strong link). We considered 1K, 10K, 25K, 50K and 67K companies.

AllStrongLinks: We want all the strong links for all the possible pairs of companies. In order to contain the explosion of the possible strong links, we set $N = 3$.

Results. The results for scenarios *PSC* and *AllPSC*, graphed in Figure 5(c), are particularly meaningful since they expose the behavior of the Vadalog system in the transitive closure (or reachability) setting. The Vadalog system shows very good performance, with linear growth of times and absolute values under 100 seconds. Since the facts for PSC are recursively generated, the good performance confirms the effectiveness of dynamic indexing in combination with our implementation of the nested loop join. In the figure, the lines for *PSC* and *AllPSC* almost coincide, proving that monotonic aggregations are effective and do not cause overhead. Interestingly, our reasoner resulted two times faster than Neo4J, a best-in-class specialized graph processing system, even for such a typical graph-oriented task. In the figure, we also show that we outperform relational systems, with PostgreSQL, MySQL and Oracle reporting a 6-times worse performance.

The results for the scenarios *SpecStrongLinks* and *AllStrongLinks* are depicted in Figure 5(d). In spite of their apparent simplicity,

the settings are extremely challenging: the transitive closure of PSC along Control produces a very high number of PSC for each company. Moreover, even when a company does not have any defined PSC (because there are no direct or inherited key persons), the existential quantification introduces an artificial one, which is also propagated with the Control relationship to the company subsidiaries. Therefore, PSC tends to have high cardinality, e.g., 38K facts for 67K companies, which results in a theoretical number of ∼700 millions of possible comparisons to be performed in rule 4. Since our reasoner showed very good performance in the computation of a transitive closure, the steep growth curve for *AllStrongLinks* is explained by the huge size of the output (more than 10 million links), which in turn derives from very long control chains and moderate density of the underlying graph. Scenario *SpecStrongLinks* shows almost constant time, always under 40 seconds, which validates our indexing and query planning mechanisms.

5.4 Industrial Validation

As an industrial validation, we evaluated the system against a variation of Example 3 in real-world cases, solving the control problem for the European financial companies, and in synthetic settings, with generated graphs for large-scale evaluation.

Description of the scenarios. We considered five real ownership graphs, built as subsets of the European graph of financial companies, with 10, 100, 1K, 10K and 50K companies and up to $42K$ edges. We chose two real-world scenarios: *AllReal*, where we ask for the control relationship between all companies and report the reasoning time; *QueryReal*, where we make 10 separate queries for specific pairs of companies, reporting the average query time.

For the synthetic settings, we built control graphs as *scale-free networks*, i.e., graphs whose degree distribution asymptotically follows a particular degree law [13, 34]. Actually, real company control and investment networks in general adhere to a scale-free topology for reasons of portfolio differentiation [21]. From the available real-world cases, we learnt the values for the parameters of our scale-free networks with specialized machine-learning techniques [2]. We then considered the following scenarios. *AllRand*, where we ask for all the control relationships; *QueryRand*, where we ask for 10 separate pairs of companies and average the query times. We tested on 7 artificial graphs from 10 to 1M companies.

Results. The results are reported in Figure 5(e-f). We observe that the growth is much slower than in *PSC* where we also explored company graphs. Indeed, the characteristics of the graph of financial companies seem to make it simpler to explore and the reasoning terminates in less than 10 seconds for *AllReal* and less than 4 in *QueryReal* for up to 50K companies. We observe that our synthetic graphs are a very good approximation of the real ones, as the behavior of *AllRand* and *QueryRand* almost overlaps with the respective real cases up to 50K companies. Results for very large graphs (100K and 1M companies) are extremely promising, as the computation terminates in about 20 seconds for both *AllRand* and *QueryRand*. We also point out that unlike *SpecStrongLinks*, here restricting the reasoning to specific pairs of companies does not make an extreme difference, which we motivate with the different graph topology: shorter chains and many hub companies.

5.5 Comparison with Chase-based Tools

In Section 5.2 we have evaluated the Vadalog system on a set of non-trivially warded rules and used such results to validate the performance against a range of chased-based systems. In this section, we complement the contribution with a validation of our system for the cases of sets of rules just "warded by chance", i.e., the ones where there is a prevalence of harmless joins, without any propagation of labelled nulls. In other terms, in this section we conduct a comparative evaluation (still relying on ChaseBench) of settings resembling *SynthG* in Section 5.1, typical of pure Datalog and data exchange/data integration scenarios, modeled, e.g., with s-t tgds.

Description of the scenarios. *Doctors, DoctorsFD*: it is a data integration task from the schema mapping literature [29], even non-recursive yet rather important as a plausible real-world case. We used source instances of 10K, 100K, 500K, 1M facts and ran 9 queries of which we report the average times.

LUBM: The Lehigh University Benchmark (LUBM) [26] is a widely used synthetic benchmark describing the university domain. It is provided with a rule generator parametric in the number of universities. We used source instances of 90K, 1M, 12M, and 120M facts. We ran 14 queries and averaged the answering time.

Results. In Figure 5(g-i) we report the results, comparing our reasoner with the two top-performing systems on the scenarios at hand according to ChaseBench, namely RDFox and LLunatic. The Vadalog system showed a very promising behavior also for these cases where our termination and recursion control algorithm cannot be exploited because of the very high fragmentation of the warded forest induced by the absence of warded joins. In particular, on *DoctorsFD*, the Vadalog system outperforms both the systems, being 3.5 times faster than RDFox and 2.5 faster than LLunatic. On *Doctors* and *LUBM*, our reasoner is up to 2 times faster than Llunatic and one time slower than RDFox. This is motivated by the fact that, unlike RDFox, we do not incorporate yet specific Datalog optimization techniques, such as magic sets [1], which will certainly boost performance in such generic cases.

5.6 Benchmarking the Lifted Linear Forest

With the experimental evaluation we describe in this section, we verify that the adoption of lifted linear forests provides visible advantage over pure isomorphism check on the entire set of generated facts (the "trivial technique" mentioned in Section 3.2).

Description of the scenarios. We implemented a termination strategy based on exhaustive storage of the generated facts and pure isomorphism check, and plugged it in into the Vadalog system. The code has been carefully optimized with hash-based indexing of the facts to have constant-time retrieval of previously generated homomorphic facts. We tested the Vadalog system comparing the performance of our approach (the full technique) with the trivial technique, using the *AllPSC* scenario described in Section 5.3.

Results. As shown in Figure 6, for less than 100k persons, the trivial technique exhibits the same performance as the full one, even being 1 or 2 seconds faster in some cases. For more than 1M persons, the elapsed time for isomorphism check departs from the previous trend: the execution requires 64 seconds vs 52 seconds, observed with the linear forest. For 1.5M persons, the performance divergence grows even more and we observe 290 vs 86 seconds. Finally, to have a clearer picture of the trend, we specifically extended the *AllPSC* scenario and also ran it with 2M persons (from synthetic data) and obtained 549 vs 125 seconds. The experiment shows that the trivial technique does not scale even when a very efficient data structure is adopted. As a matter of fact, isomorphism check requires storing all the generated facts; for small input instances, a very efficient indexing technique pays off. After a "sweet spot" at 100k, where the two algorithms basically have the same performance, for larger input instances isomorphism check times grow by orders of magnitude as the indexing structure eventually becomes inefficient (e.g., in the case of a hash structure, for example, for the soar in the number of hash conflicts) or even overflows the available heap space, requiring costly off-heap and to-disk swappings. As a further confirmation of these results, we ran the

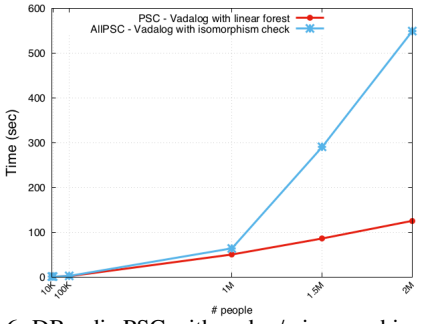

Figure 6: DBpedia PSC with and w/o isomorphism check.

same test again with all our cases, which confirmed the trend: for example for the *AllRand* scenario presented in Section 5.4, elapsed time grows to 62 seconds for 1M persons.

5.7 Scaling Rules and Tuples

In order to fully characterize the scalability of the Vadalog system, we evaluated reasoning performance along further dimensions: size of the source database, number of rules, number of atoms in the rules, arity of the predicates.

Description of the scenarios. All our scenarios are variations of *SynthB*, introduced in Section 5.1.

DbSize: We ran *SynthB* with an initial database of 10k, 50k, 100k and 500k source instances. We generated input facts adopting a uniform distribution of values to induce an average join rate.

Rule#: We considered variants of *SynthB* with 100, 200, 500, 1k rules respectively. We started from the basic version with 100 rules, with 50 input predicates and 10 output predicates. Then, we built larger scenarios by composing blocks of rules. Each block is a copy of the basic *SynthB* (with 100 rules), with rules appropriately renamed and linked to the respective input and output predicates. In this way, each set of rules has the same "reasoning complexity" as the basic case because independent blocks do not interact with each other, yet isolate the effect of scaling up the number of rules.

Atom#: We considered variants of *SynthB* having join rules with an average of 2, 4, 8 and 16 atoms. We started from the basic version with binary joins (bodies with 2 atoms) and scaled up, adding atoms to the bodies. Atoms have been added in such a way that the number of output facts is preserved and are proportionally distributed among the 27 recursive join rules and the 63 non-recursive rules not to overly alter the computational complexity of the original rule set and actually test the effect of "larger" rules. The 10 linear rules have been left unmodified.

Arity: We constructed variants of *SynthB* with atoms having an average arity of 3, 6, 12, 24. In order to test the effect of "larger" tuples in isolation, we increased the arity by adding variables that do not produce new interactions between atoms, so the proportion of harmless and harmful joins is also maintained.

Results. *DbSize* (Figure 7(a)) confirms the polynomial behavior of the warded fragment and good absolute elapsed times. The trend is clearly polynomial (concretely, slightly sublinear): for 10k the obtained elapsed time is 4 seconds; it grows to 9 seconds for 50k, 13 seconds for 100k and 51 seconds for 500k. Let us come to *Rule#* (Figure 7(b)). For 100 rules, we have the usual elapsed time for *SynthB* (9 seconds), stable for 200 rules. We obtained 52 seconds for 500 rules and 101 seconds for 1k rules. The results confirm that the system is capable of handling an increasing number of rules and the performance is not affected by this. In fact, there is a linear growth of the elapsed times when multiple independent reasoning tasks are added as an effect of more rules. In

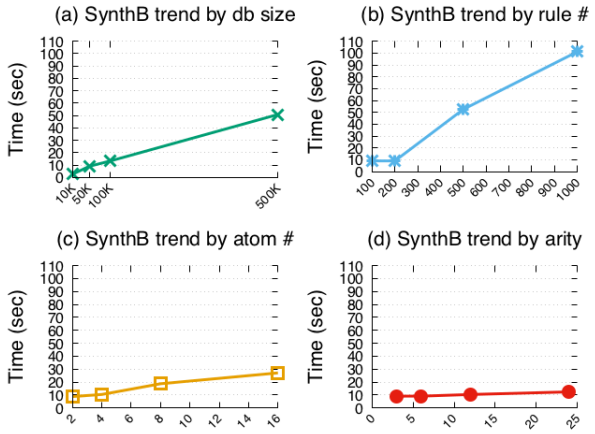

Figure 7: Reasoning time scaling db size, rule size and arity.

other terms, the effect of scaling up the number of rules is negligible and only reflects in a slight increase in the time required for harmful-harmful simplification, construction of the execution plan and so on. For *Atom#* (Figure 7(c)), starting from the base case with 2 body atoms (9 seconds), we obtained 10 seconds for 4 atoms, 18 seconds for 8 atoms and 27 seconds for 16 atoms. The execution optimizer implements multi-joins as a cascade of single-joins, and thus adding more atoms corresponds to lengthening the processing pipeline. As the size of the input instance does not vary, indexes allow to limit the impact of multiple joins, with a growth of execution time clearly polynomial (concretely, slightly sublinear). Finally, *Arity* (Figure 7(d)) shows that the size of facts (source and intermediate results) is almost irrelevant with respect to performance; starting from the base case with the usual elapsed time of 9 seconds, we obtain 9 seconds for arity 6, 10 seconds for arity 12 and 12 seconds for arity 24. The 1-2 seconds difference is due to the natural increase in the data size and the internal hash-based caching structures are not affected by a reasonable growth in tuple size.

6. RELATED WORK AND COMPARISON

A wide range of systems can be seen as related work to the Vadalog system. Let us first focus on the reasoners that share much (but not all) of the expressive power of our system, namely those that support (recursive) Datalog with existential quantification. Such systems have typically been applied in the areas of data exchange, integration, cleaning and query answering. ChaseBench [11] provides both an excellent collection of benchmarks (most of which were presented in the preceding section) as well as a curated selection of systems in the area: Graal [6], LLunatic [22], RDFox [31], and PDQ [12], as well as DEMo [33], Pegasus [30], E [35], and ChaseFUN [14].

PDQ is a system based on query rewriting. More particularly, it reformulates a given query as query containment check under tgds and uses an underlying implementation of the restricted chase (i.e., chase with full homomorphism check) on top of PostgreSQL. The combination of restricted chase and rewriting into target systems language was shown not to be effective performance-wise, as systems fail to efficiently interleave homomorphism checks and updates [11]. PDQ is outperformed by the Vadalog system by three orders of magnitude in the tested scenarios. Actually, PDQ's approach is effective in cases when the query reformulation drastically reduces the size of the input instance; however, this rarely happens in well-designed reasoning tasks (e.g., iBench) where the

rules are compact and non-redundant. Furthermore, as we show in the industrial validation in Section 5.4, RDBMSs perform badly in the presence of recursion and the experimental results we present for PostgreSQL are consistent with the ones of PDQ. **LLunatic** is a system motivated by data cleaning settings, implementing several variants of the chase, including restricted chase with marked nulls. It is implemented on top of PostgreSQL, thus shares many of the characteristics in this regard as PDQ.

Graal is a toolkit for computing certain query answering under constraints. It adopts a saturation algorithm, which is basically a variant of the non-naïve restricted chase, where a fact is generated only if it passes the homomorphism check and the applicable rules are selected on the basis of a dependency graph, which avoids an exhaustive breadth-first search of applicable rules. The approach is based on forward chaining. It also embodies very smart query rewriting algorithms, which carefully take into consideration and implement specific logic fragments. Although Graal recognizes specific logic fragments (e.g., linear rules), such awareness is used only for query optimization purposes (e.g., rewriting), whereas the Vadalog system exploits the fragment structure to avoid homomorphism check, and, as a consequence, the need to store a fresh memory copy of all the generated facts. Conversely, Graal performs homomorphism check, implementing it against several different backends, with both the performance drawbacks of full homomorphism check (shown in Section 3.2) and those of the restricted chase-based implementations [11].

DLV is a mature Datalog system, which supports many different features. It covers disjunctive Datalog under the answer sets semantics (with two forms of negation). In the version considered by [11], existentials have been simulated by Skolemization. DLV implements most advanced Datalog optimization techniques such as magic sets, back-jumping, smart pruning and parallel processing. DLV performs better than PDQ and Graal as it adopts unrestricted chase and a pure in-memory processing. The main limitation of the in-memory version of the system (the only one supporting tgds), is a too large memory footprint due to grounding. This becomes prohibitive for large input sizes and is the principle difference between DLV and the Vadalog system (which adopts a form of *lifted* inference, i.e., does not require grounding).

RDFox is a high-performance in-memory Datalog engine, implementing both restricted and unrestricted chase and fully supporting existentials. The Vadalog system is three times faster than RDFox on iBench. A principle difference between RDFox and Vadalog is that RDFox considers all rule instances, while Vadalog considers – utilizing the lifted linear forest – a reduced number of rule instances. This is particularly relevant if there are a high number of input tuples that are structurally similar (i.e., follow the same *pattern* as described in Section 3).

Many more systems exist than those discussed in detail. In addition to the ones discussed above, [11] considers DEMo, Pegasus, E and ChaseFUN. Of these, ChaseFUN does not support recursive rules, and DEMo performs poorly on all the experiments [11]. E is a constraint solver requiring processing one tuple at a time, and Pegasus "fared badly on all tests" due to it being based on query reformulation [11]. However note that none of these systems was designed or optimized for the considered scenarios. A further interesting system is LogicBlox [3]. Unfortunately, it was not available for benchmarks. Looking further, there are some similarities to the OnTop system [19], the OBDA approach in general, as well as graph databases. Moreover, our basic architecture shares similarities with those of a huge variety of state-of-the-art data processing systems. Yet, as can be expected (but is also shown by our experiments – in particular those in Section 5.3), systems that have no strong optimization for recursion or existential quantification are at a significant disadvantage performance-wise.

Some general themes emerge from the specific systems discussed above as well as those sharing similar properties: (a) **Restricted vs unrestricted chase**. Back-end based implementations of restricted chase are problematic. Let us, for example, consider a small set of rules inspired by *ONT-256* scenario, introduced in Section 5.2.

Example 8. $D = \{\text{Whistle}(1,1,2,3), \text{Young}(1)\}$
$1 : \text{Whistle}(a,a,b,c) \rightarrow \text{Whistle}(b,b,a,c)$
$2 : \text{Whistle}(a,a,b,c) \rightarrow \exists h\, \text{Cow}(a,b,\hat{h})$
$3 : \text{Cow}(a,b,\hat{h}), \text{Young}(a) \rightarrow \text{Cow}(b,a,\hat{h}).$

In systems based on RDBMS backend, homomorphism check is implemented in SQL, so for rule 2, we would have:

```
SELECT * FROM Whistle W WHERE NOT EXISTS
  (SELECT * FROM Cow C WHERE C.a = W.a
      and C.b = W.b) and W.a = W.a1
```

If the query returns an empty result, the check is considered successful. Such an SQL query would have to be executed before each chase step, and running it just once for the entire predicate Whistle would be insufficient. To highlight this aspect, let us suppose rule 1 is first applied to D, generating $\text{Whistle}(2,2,1,3)$. Now, an update in Cow by rule 2, generating $\text{Cow}(1,2,h_1)$, may trigger rule 3, hence generating $\text{Cow}(2,1,h_1)$; this invalidates the result of the previous check for rule 2, which would generate $\text{Cow}(2,1,h_2)$ from $\text{Whistle}(1,1,2,3)$. Although many optimizations are possible, systems implementing backend-based chase (e.g., Graal, PDQ and to some extent LLunatic) pay such query overhead, unlike systems that do not translate homomorphism checks into target system queries. In particular, in this case the Vadalog system would recognize isomorphic copies of $\text{Cow}(1,2,h)$ in the same component of the warded forest and thus inhibit rule 3 from firing. This would be done by recognizing the initial pattern $\text{Whistle}(c_1,c_1,c_2,c_3)$ in the lifted-linear forest and cutting at the stop provenance. (b) **In-memory**. The restriction to in-memory processing is successful, yet it causes a too large memory footprint, in particular in combination with grounding, and fails for Big Data. In contrast, the Vadalog system exploits the specificities of Warded Datalog$^{\pm}$ to implement a restricted chase that optimizes the search for isomorphic facts by restricting to the same component of the warded forest and adopting good guide structures, that also limit memory footprint. For instance, a potentially very large set of components in the lifted-linear forest, pattern-isomorphic to the one in Example 8, can be represented by just a single pattern component with five facts.

7. CONCLUSION

In this paper, we introduced the Vadalog system, the first implementation of Warded Datalog$^{\pm}$. At the core of it, the algorithm exploits the key theoretical underpinnings of the underlying language through novel recursion control techniques. The architecture and system exhibited competitive performance in both real-world and synthetic benchmarks.

As future work, we are working on adding further standard database optimization techniques to the system, such as more sophisticated cost-based query plan optimization. We also are constantly expanding our range of supported data sources so as to encompass more and more data stores, big data platforms and queryable APIs.

Acknowledgements. The work on this paper was supported by EPSRC programme grant EP/M025268/1 VADA. The Vadalog system as presented here is IP of the University of Oxford.

8. REFERENCES

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

[2] D. Agarwal. Discussion of "learning scale free networks by reweighted l1 regularization". In *AISTATS*, volume 15, pages 49–50, 2011.

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

[4] P. C. Arocena, B. Glavic, R. Ciucanu, and R. J. Miller. The ibench integration metadata generator. *PVLDB*, 9(3):108–119, 2015.

[5] J. Baget, M. Leclère, and M. Mugnier. Walking the decidability line for rules with existential variables. In *KR*. AAAI Press, 2010.

[6] J. Baget, M. Leclère, M. Mugnier, S. Rocher, and C. Sipieter. Graal: A toolkit for query answering with existential rules. In *RuleML*, pages 328–344, 2015.

[7] J. Baget, M. Mugnier, S. Rudolph, and M. Thomazo. Walking the complexity lines for generalized guarded existential rules. In *IJCAI*, pages 712–717. IJCAI/AAAI, 2011.

[8] P. Barceló and R. Pichler, editors. *Datalog in Academia and Industry - Second International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings*, volume 7494 of *Lecture Notes in Computer Science*. Springer, 2012.

[9] L. Bellomarini, G. Gottlob, A. Pieris, and E. Sallinger. Swift logic for big data and knowledge graphs. In *IJCAI*, pages 2–10, 2017.

[10] L. Bellomarini, E. Sallinger, and G. Gottlob. The Vadalog System (technical report). `https://drive.google.com/drive/u/0/folders/0B2xKYNJdpJzQaEh5VzlhTURHU3M`.

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

[12] M. Benedikt, J. Leblay, and E. Tsamoura. Querying with access patterns and integrity constraints. *PVLDB*, 8(6):690–701, 2015.

[13] B. Bollobás, C. Borgs, J. Chayes, and O. Riordan. Directed scale-free graphs. In *SODA*, pages 132–139, 2003.

[14] A. Bonifati, I. Ileana, and M. Linardi. Functional dependencies unleashed for scalable data exchange. In *SSDBM*, pages 2:1–2:12. ACM, 2016.

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

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

[17] A. Calì, G. Gottlob, T. Lukasiewicz, B. Marnette, and A. Pieris. Datalog+/-: A family of logical knowledge representation and query languages for new applications. In *LICS*, pages 228–242, 2010.

[18] A. Calì, G. Gottlob, and A. Pieris. Towards more expressive ontology languages: The query answering problem. *Artificial Intelligence*, 193:87–128, 2012.

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

[20] DBpedia. Web site. `http://wiki.dbpedia.org/services-resources/downloads/dbpedia-tables`.

[21] D. Garlaschelli, S. Battiston, M. Castri, V. Servedio, and G. Caldarelli. The scale-free topology of market investments. `http://goo.gl/h7tVVK`, 2004.

[22] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. That's all folks! LLUNATIC goes open source. *PVLDB*, 7(13):1565–1568, 2014.

[23] G. Gottlob and A. Pieris. Beyond SPARQL under OWL 2 QL entailment regime: Rules to the rescue. In *IJCAI*, pages 2999–3007, 2015.

[24] G. Gottlob, S. Rudolph, and M. Simkus. Expressiveness of guarded existential rule languages. In *SIGMOD*, pages 27–38, 2014.

[25] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.

[26] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[27] M. Krötzsch. Efficient rule-based inferencing for OWL EL. In *IJCAI 2011*, pages 2668–2673, 2011.

[28] M. Marx, M. Krötzsch, and V. Thost. Logic on MARS: ontologies for generalised property graphs. In *IJCAI 2017*, pages 1188–1194, 2017.

[29] G. Mecca, P. Papotti, and D. Santoro. IQ-METER - an evaluation tool for data-transformation systems. In *ICDE*, pages 1218–1221, 2014.

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

[31] B. Motik, Y. Nenov, R. Piro, I. Horrocks, and D. Olteanu. Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In *AAAI*, pages 129–137, 2014.

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

[33] R. Pichler and V. Savenkov. Demo: Data exchange modeling tool. *PVLDB*, 2(2):1606–1609, 2009.

[34] C. A. H. R. and A. Barabási. Scale-free networks. *Scholarpedia*, 3(1):1716, 2008.

[35] S. Schulz. System description: E 1.8. In *LPAR*, volume 8312 of *Lecture Notes in Computer Science*, pages 735–743. Springer, 2013.

[36] J. Urbani, C. J. H. Jacobs, and M. Krötzsch. Column-oriented datalog materialization for large knowledge graphs. In *AAAI 2016*, pages 258–264, 2016.

[37] VADA. Project. `http://vada.org.uk/`, 2016.