

An Overview of Vadalog: a System for Reasoning over Large Knowledge Graphs

Luigi Bellomarini¹, Davide Benedetto² and Emanuel Sallinger^{3,4}

¹Banca d'Italia, Italy

²Università Roma Tre, Department of Computer Science and Engineering, Rome, Italy

³TU Wien, Faculty of Informatics, Vienna, Austria

⁴University of Oxford, Department of Computer Science, Oxford, UK

Abstract

As a main requirement, a knowledge representation and reasoning language must exhibit a good tradeoff between expressive power and computational complexity of reasoning. Warded Datalog+/-, a fragment from the Datalog+/- family, satisfies these requirements, by offering high expressive power and at the same time preserving tractability of query answering. The Vadalog system, a knowledge graph management system developed by the University of Oxford and Banca d'Italia, adopts Warded Datalog+/- as its core language. This enables Vadalog to solve many ontological reasoning tasks including those where complex graph traversal operations are involved. To control graph navigation, Vadalog combines different rules prioritization policies and join implementations. In this paper, a short version of our recent contribution appeared in the Journal of Information Systems, we describe Vadalog from an architectural point of view, focusing on the execution model, graph traversal strategies, and join algorithms. We also provide an experimental evaluation of the system.

Keywords

automated reasoning, knowledge representation, datalog, knowledge graph, vadalog

1. Introduction

We are witnessing a renewed attention in the use of logic-based languages for Knowledge Representation and Reasoning (KRR) on Knowledge Graphs (KG) from both the academia and the industry, with many companies such as LinkedIn [1] and Amazon [2] confirming the adoption of the Datalog language [3]. In the literature, there is no common agreement on a shared definition for KGs [4], but the following common trait emerges: a KG is a large network of entities, and instances for those entities, describing real-world objects and their interrelations with specific reference to a domain or to an organization [5]. From a technical standpoint, a knowledge graph is a semistructured data model characterized by an extensional, intensional and derived extensional component [6]. The intensional component is generally modeled with a KRR formalism, in the form of ontologies, which are evaluated over enterprise data sources (extensional data) to enrich the knowledge graph with new (derived) knowledge. The quest for an ideal KRR language that enables reasoning with large knowledge graphs spawned substantial

SEBD 2022: The 30th Italian Symposium on Advanced Database Systems, June 19-22, 2022, Tirrenia (PI), Italy

✉ luigi.bellomarini@bancaditalia.it (L. Bellomarini); davide.benedetto@uniroma3.it (D. Benedetto); sallinger@dbai.tuwien.ac.at (E. Sallinger)



© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

research in the database community, and many languages have been intensively investigated and proposed under a number of different names [7], which we commonly share here as the Datalog[±] class [8]. Datalog[±] extends Datalog with additional features, of which the most relevant is existential quantification (whence the “+”), while resorting to syntactic restrictions (“-”) to guarantee decidability and, in some cases, tractability of query answering. Many Datalog[±] fragments have been investigated [8, 9, 10, 11, 12] and each of them provides different expressive power and computational complexity. Among these, Warded Datalog[±] guarantees tractability [13], being a good KRR language candidate, since it exhibits high expressive power, being able to model complex real domains, for instance requiring full recursion and existential quantification, as well as low computational complexity, enabling scalability in practice.

The Vadalog system is a state-of-the-art knowledge graph management system (KGMS), that adopts Warded Datalog[±] as its core language. It has been widely presented in many works [5, 14, 15, 16], and largely used for solving many industrial tasks, particularly in [17, 18, 19].

Discussion Topics. In this paper we examine the system under the hood, by analyzing the reasoning algorithms and the architectural choices that make Vadalog a fully fledged KGMS. In particular, we discuss about the *reasoning termination strategy* based on the isomorphism check; the *system architecture* of the Vadalog system and its internals, with particular attention to the runtime model, the routing strategies and the cycle and termination management. We also propose an *experimental analysis* that evaluates the system in both real and synthetic settings.

Overview. The remainder of this paper is organized as follows: in the next section we describe the fundamental notions that address reasoning with Vadalog; in Section 3 we discuss the main architectural choices that make Vadalog a fully fledged KGMS; the experimental discussion is provided in Section 4; in Section 5 we draw our conclusions.

2. Reasoning with Vadalog

VADALOG is the reasoning language of the Vadalog system. It belongs to the Datalog[±] family of languages that generalizes Datalog rules by introducing existential quantification in rule heads and enriches it with additional features of practical utility, while posing syntactical restrictions to achieve decidability and data tractability. The logical core of VADALOG extends to Warded Datalog[±] [20], which captures plain Datalog as well as SPARQL queries under the entailment regime for OWL 2 QL [21] and is able to perform ontological reasoning tasks. A Vadalog program consists of a set of *existential rules*, or *tuple-generating dependencies* 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. Intuitively, the semantics of such a rule is the following: for each fact $\varphi(\bar{t}, \bar{t}')$ that occurs in an instance I , there exists a tuple \bar{t}'' of constants and nulls such that the facts $\psi(\bar{t}, \bar{t}'')$ are also in I .

The Chase Procedure. The semantics of a set of Datalog rules Σ over a database D , denoted $\Sigma(D)$, is usually specified in an operational way via the well known chase procedure [22]. It expands D with facts derived via the application of rules of Σ over D , into a new database *chase*(D, Σ), possibly containing fresh nulls as placeholders for the existentially quantified objects. A rule $\rho = \varphi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} \psi(\bar{x}, \bar{z})$ is *applicable* to $\Sigma(D)$ if there is a unifier (i.e a mapping from

variables to constants) θ_ρ such that $\varphi(\bar{x}\theta_\rho, \bar{y}\theta_\rho) \subseteq \Sigma(D)$ and $\psi(\bar{x}\theta_\rho)$ does not belong to $\Sigma(D)$. If ρ is applicable to $\Sigma(D)$ with a unifier θ_ρ , then it performs a *chase step* i.e., it *generates* new facts $\psi(\bar{x}\theta'_\rho)$ that are added to $\Sigma(D)$, where $\bar{x}\theta'_\rho = \bar{x}\theta_\rho$. The chase performs chase steps until no rule in Σ is applicable.

Example 1. Consider the following set of rules, which provide a simple description of how influence of individuals propagates in company networks.

- 1 : $Company(x) \rightarrow \exists p Ceo(p, x)$.
- 2 : $Ceo(p, x) \rightarrow Influences(p, x)$.
- 3 : $Control(x, y), Influences(p, x) \rightarrow Influences(p, y)$.
- 4 : $Influences(p, x), Influences(p, y), x \neq y \rightarrow Linked(x, y)$.

By Rule 1, for every company x , there is a CEO p . The CEO of a company exerts influence over it (Rule 2). By Rule 3, if a company x controls (i.e., controls the majority of voting rights) a company y , then a person exerting influence over x also exerts influence over y as well. Finally, by Rule 4, companies influenced by the same person are linked to each other.

Let us analyze how the rules are applied on $D = \{Company(a), Company(b), Ceo(Bob, a), Control(a, b), Influences(Bob, c)\}$, by considering the CHASE procedure. By the application of Rule 2 we obtain the influence of Bob on the company a . Then Rule 3 can be applied by unifying with the facts $Control(a, b)$ and $Influences(Bob, a)$, thus inferring that Bob influences the company b . Finally, by the application of rule 4 on the facts $Influences(Bob, a)$, $Influences(Bob, b)$ and $Influences(Bob, c)$ we can conclude that there is a link between the companies a and b , a and c , b and c .

Termination and Recursion Control. Reasoning tasks for Datalog with existential quantifiers are undecidable in general. In fact, the chase procedure for Datalog with existential quantification may in general not terminate due to the possibility of producing unboundedly many labelled nulls. However, we have seen that Warded Datalog[±] is decidable and reasoning with it is PTIME [13]. Intuitively, the key idea to guarantee chase termination is to tame the propagation on the labelled nulls. This is achievable by exploiting the so-called termination strategy, that blocks the generation of facts that have been produced in previous chase steps. The Vatalog system termination strategy relies on the isomorphism termination strategy. We say that two facts are isomorphic if they refer to the same predicate name, they have the same constants in the same positions and there is a bijection between the labelled nulls. Yet, theoretical tractability results are far away from a practical algorithm. Ideally, a fact produced by the chase that is isomorphic to a previous generated one can be skipped since it is not relevant for answering to the reasoning task. To obtain a terminating, correct algorithm it is sufficient to reformulate the notion of *applicable rule* in the chase step, introduced before. More details of the algorithm and the theoretical proof about its correctness can be found in [14]. We redefine the applicability as follows: we say $\rho = \varphi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} \psi(\bar{x}, \bar{z})$ is *applicable* to $\Sigma(D)$ if there is a unifier θ_ρ such that $\varphi(\bar{x}\theta_\rho, \bar{y}\theta_\rho) \subseteq \Sigma(D)$ and there are not facts in $\Sigma(D)$ isomorphic to $\psi(\bar{x}\theta'_\rho, \bar{z}\theta'_\rho)$, where $\bar{x}\theta'_\rho = \bar{x}\theta_\rho$ and $z_i\theta'_\rho$, for each $z_i \in \bar{z}$, is a fresh labeled null that does not occur in $\Sigma(D)$.

3. The Architecture of the Vadalog System

In this section we describe the architecture of the Vadalog system. Throughout the section we illustrate the main architectural choices that make the system a fully fledged KGMS. We give a description of i) the pipeline architecture, ii) the execution model, that is, how the rules are evaluated at runtime, iii) the reasoning routing strategies, i.e. different techniques to control rule prioritization applicability, iv) techniques for handling cycles and memory management. Overall, the Vadalog system is a memory-resident server exposing a *reasoning API* with the following interface: `reason(kg_ref, parameters)`. A client application issues calls to the reasoning API specifying a reference to a knowledge graph `kg_ref` to activate the reasoning process upon: the Vadalog system handles a repository of knowledge graphs, with unique identifiers `kg_ref`; technically the KG consists of a set of rules annotated with bindings of predicates to external data sources, a compact representation of the extensional and intensional components. The engine computes the answer to the reasoning task and returns a representation of the output facts. From now on, the terms scan, filter and rule are used interchangeably.

Pipeline Architecture. The Vadalog system implementation is based on the pipes and filters architectural style, a pattern used to process streaming data flows. In this pattern, data streams throughout a pipeline of filters, each of those receives data from the input flow, applies the required transformations, generates the output flow and passes it to the next filter through a connector called pipe. In our settings, this pipeline is build up by compiling a set of Vadalog rules where the input filters are represented by input rules (e.g. EDB predicates in a rule body) and the output filters are assumed by queried predicates of the reasoning task. The compilation process is divided in three distinct parts. At the first stage, the compilation process involves multiple rewriting actions, acted by a *logic optimizer*. At the second stage the optimized rules received from the parser are transformed into a reasoning access plan by the *logic compiler*. The logic compiler acts as a planner: it produces a static pipeline in the form of a predicate graph where each node (filter) is represented by an atom from the set of rules, and there is an edge (pipe) from a node `m` to a node `n` if there is a rule that unifies from `m` to `n`. At the last stage, a *query compiler* converts the logic pipeline into a reasoning query plan, where the nodes are translated into active data scans, connected by intermediate buffers. To enable the data flow consumption and production, input and output record managers are attached to the start and to the end of the pipeline, respectively. In the reasoning pipeline, input record managers are connected with virtual input scans, while output record managers receive data from output scans; in-between these two, we have three different scans, that symbolize the behaviour of the linear (one body atom), join TGDs (more than one body atom).

Execution Model. The Vadalog system does not directly adopt the chase procedure, but follows the architecture of traditional relational DBMSs and its execution model is a generalization of the *volcano iterator model* [23] extended to the entire reasoning query plan pipeline. The reasoning process of the Vadalog system follows a pull based (query driven) approach, where each filter reads facts from respective parent. The pull action starts from sink nodes, which asks for output facts and propagate the request down to its predecessors, triggering the invocations along the pipeline searching for available facts. Filters interact with each others using primitives `open()`, `next()`, `get()`, `close()`, which respectively open the parent stream, ask for the presence of a

fact to fetch, obtain it, and close the communication. The request is opened by the sink filter which issues a `next()` call, that is propagated to the parent filters until the input filters are reached. These access data from the external sources, converts it to ground facts and contribute to feed the reasoning process with new data. Facts stream through the pipeline up to the requestors and are stored into local buffer caches to be available for multiple consumers.

Reasoning Routing Strategies. The common strategy adopted by the chase procedure satisfies the filters applicability in a breadth first fashion. This behaviour is in general a good compromise when a priori order of rule applicability is not defined, since it guarantees a good balance among the filters. In [15] we studied how the chase can be leveraged in multiple non-deterministic choices, analyzing several scenarios where the breadth-first policy for the applicability of the rules is not always an adequate solution. We formalize this as the rule routing problem, that is the problem of filters having to decide which source to invoke, when multiple choices are possible. For instance, consider the set of Vadalog rules in the example 2.

Example 2. Given the following set of rules, the desired output are facts for q

- 1 : $edb_1(x,y) \rightarrow f(x,y)$
- 2 : $edb_2(x,y) \rightarrow h(y)$
- 3 : $h(x) \rightarrow \exists z p(x,z)$
- 4 : $f(x,y), p(x,k) \rightarrow q(x,y,k)$
- 5 : $q(x,y,z) \rightarrow f(x,x)$
- 6 : $edb_3(x,y), p(y,z) \rightarrow p(x,z)$

The filters 3, 4 and 5 can fetch data from alternative parent filters for their body predicates. For instance, 3 can fetch facts from 2; 4 can fetch facts from 1 and 5 or 3 and 6; then 6 can fetch facts from 3 and itself. Consider now the first activation of Rule 6, for example triggered by a `next()` call from Rule 4, which contains the desired output q . The filter 6 has to take a non-deterministic routing choice, about issuing a `next()` call to alternatively itself (recursively) or 3. It is intuitive that if the first alternative is issued before, no result would be fetched and an extra call to 3 and 6 would be necessary. It is easy to see that if filter 5 would take this sub-optimal sequence of filter invocations each time it is activated, the system would issue a huge number of useless calls, with clear performance loss. Many routing strategies exist, each implementing a diverse behaviour when issuing rule applicability order. We now briefly present the four strategies mostly relevant for our discussion.

Round-Robin. (RR) It is the default and simplest compile time ordering strategy. For each filter, an arbitrary order of source filters is established and fixed. Intuitively, this strategy fosters a breadth-first expansion of the chase.

Recursive-last. (RL) It a compile time ordering strategy. For each filter, the set of source filters is sorted by ascending the non-recursive ones. Clearly, by recursive filters we mean rules where at least one body atom is mutually recursive with the head. The rest of the rules are the non-recursive ones. This strategy reduces the number of failure calls, since it allows to feed the pipeline with facts available for recursive calls with invocations to base case rules.

Shortest EDB Path (SEP). This compile time ordering strategy is a refinement of RR. Let us define the relative depth $\mu_\Sigma(\rho)$ of a rule $\rho : \varphi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} \psi(\bar{x}, \bar{z})$, in a set of Vadalog rules Σ as follows. Let P be the reasoning query plan built from Σ . We define $\mu_\Sigma(\rho)$ as $\max\{\lambda_1, \dots, \lambda_n\}$, where λ_i

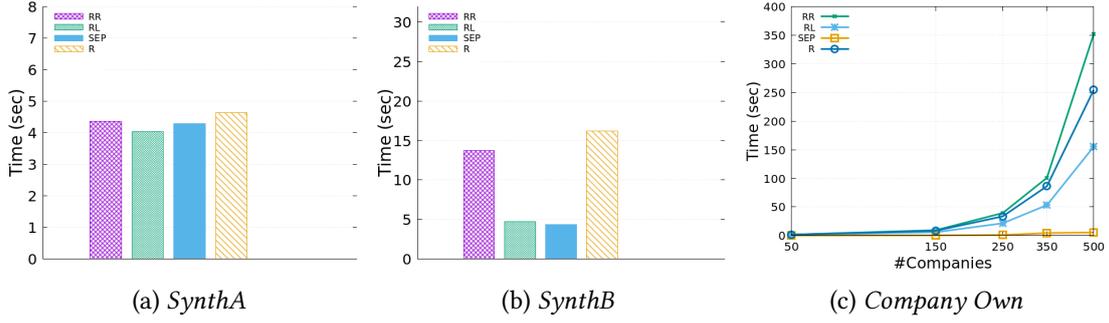


Figure 1: Evaluation of the routing strategies described in Section 3 with respect to the two iWARDED synthetic and the DBPedia real world scenarios. RR: round-robin; RL: recursive last; SEP: shortest EDB path; R: random.

is the length of the shortest path in P from ρ to the nearest rule ρ' that produces facts for the atom x_i in φ , such that ρ' contains only EDB in the body; intuitively, μ measures the distance of ρ from the nearest EDB. If ρ contains multiple body atoms, then it considers only the atom having the highest distance to the nearest EDB.

Random (R). This runtime ordering strategy adopts a random policy to select the candidate filter. It is particularly useful as a baseline to evaluate other strategies.

Cycle Management In the runtime execution of a reasoning task, we deal with two kinds of cycles: *runtime invocation cycles* and *non-terminating sequences*. Runtime cycles are recursive invocation paths of the next () primitive. As an example, let us consider the following sequence of filter invocations $a \leftarrow b \leftarrow c \leftarrow d \leftarrow b$. If b cannot fulfil the recursion with any fact from other recursive or base cases, we are still not allowed to interrupt the iteration, as the required facts for b may derive from other recursive cases to be explored first.

We distinguish the absence of facts in cyclic cases (*cyclic miss*) from the actual absence of further facts (*real miss*). The first case denotes that a specific routing resulted in a circular invocation of the next () primitive: it results in a failure, but the source is not discarded, because a following invocation may instead produce new facts for it. On the other hand, a real miss is permanent and denotes that a filter does not have available facts anymore and will not have them in the future, thus must be discarded. In case of cyclic misses, the invoked filter notifies the caller with a notifyCycle() primitive, that flows along the pipeline from called filters to caller. It is now clear that the specific routing policy is essential to balance the exploration of all the possible recursive and base cases. Non-terminating sequences directly derive from the presence of recursion (and hence cycles) and a recursive pipeline may generate infinite facts. To cope with this, the Vadalog system features a *termination strategy wrappers*, a component that prevents the generation of isomorphic facts, as described in Section 2.

4. Experimental Evaluation

The Vadalog system has been largely evaluated in previous works [14, 15]. In Section 3, we described the many alternative routing strategies for reasoning. This raises the question how

these different strategies affect the performance of the system. Thus in this section our discussion concentrates on showing how the four routing strategies impacts the performance of Vadalog in synthetic and a real world settings.

Test Setup. We invoked Vadalog via its REST interface and used CSV data to make tests independent of host-side optimizations. We ran each experiment ten times, averaging the elapsed times. All the experimental evaluations are run with a local installation of the Vadalog system in a MacBook Pro i7 – 2.5 GHz and 16 GB of RAM. The engine is compiled with JDK 9 and reasoning tasks are executed without any use of concurrency or distribution techniques.

Synthetic Scenarios. We analyzed the behaviour of the four routing strategies in two scenarios coming from the set of synthetic benchmarks proposed in [15], created with *IWARDDED* [24], a configurable tool for generating Warded Datalog[±] programs. Each scenario comprises 100 rules, with an alternating number of recursive and non-recursive rules. *SynthA* is made of 70 non-recursive and 30 recursive rules. *SynthB* is made of 55 recursive and 45 non-recursive rules. The results are depicted in figures 1(a) and 1(b). We can observe that in presence of intensive recursion SEP and RL perform better than the other two strategies.

Real Scenario. We analyzed the behaviour of the four routing strategies at scale. We ran a basic reachability task in dense networks of 50, 150, 250, 350, 500 companies, built as scale-free networks with parameters (controlling distribution, density) fitted from the European graphs.

Example 3. *We used the following set of Vadalog rules to compute reachability.*

$$\begin{aligned} 1 & : \text{Own}(x, y, w) \rightarrow \text{Path}(x, y). \\ 2 & : \text{Path}(x, y), \text{Own}(y, z, w) \rightarrow \text{Path}(x, z). \\ 3 & : \text{Path}(x, y) \rightarrow \text{Path}(y, x). \end{aligned}$$

The results in Figure 1(c) confirm the trend of RR and R shown in the synthetic experiments, demonstrating that these two strategies are not ideal in recursive settings. In fact, the rule applicability order adopted by these two strategy is only sub-optimal, because they are not able to prioritize base over the recursive cases thus leading to performance loss. In this case SEP enables best performance. This is addressable to the inability of RL to distinguish which recursive filter to opt for, whereas SEP applies a global heuristic.

5. Conclusion

In this paper we gave an overview of the Vadalog system, a modern architecture for automated reasoning in knowledge graphs. We discussed the main problems in implementing a fully fledged KGMS, with particular attention to the strategies for handling termination and cycle management. We also opened a discussion about the rule routing problem, the problem of filters having to decide which source to invoke, when multiple choices are possible and we described a set of routing strategies which adopt diverse heuristic to handle recursive cases and we experimented the impact on the performance of these strategies in the Vadalog system.

References

- [1] W. E. Moustafa, V. Papavasileiou, K. Yocum, A. Deutsch, Datalography: Scaling datalog graph analytics on graph processing systems, in: *BigData*, IEEE, 2016, pp. 56–65.
- [2] D. Zhao, P. Subotic, B. Scholz, Debugging large-scale datalog: A scalable provenance evaluation strategy, *ACM Trans. Program. Lang. Syst.* 42 (2020) 7:1–7:35.
- [3] S. Abiteboul, R. Hull, V. Vianu, *Foundations of databases*, volume 8, Addison-Wesley Reading, 1995.
- [4] L. Ehrlinger, W. Wöß, Towards a definition of knowledge graphs, in: *SEMANTiCS (Posters, Demos, SuCCESS)*, 2016.
- [5] L. Bellomarini, D. Fakhoury, G. Gottlob, E. Sallinger, Knowledge graphs and enterprise ai: The promise of an enabling technology, in: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019, pp. 26–37. doi:10.1109/ICDE.2019.00011.
- [6] L. Bellomarini, E. Sallinger, S. Vahdati, Knowledge graphs: The layered perspective, in: *Knowledge Graphs and Big Data Processing*, Springer, Cham, 2020, pp. 20–34.
- [7] J. Baget, M. Leclère, M. Mugnier, Walking the decidability line for rules with existential variables, in: *KR*, AAAI Press, 2010.
- [8] A. Cali, G. Gottlob, T. Lukasiewicz, A general datalog-based framework for tractable query answering over ontologies, in: *PODS*, 2009, pp. 77–86.
- [9] A. Cali, G. Gottlob, A. Pieris, Towards more expressive ontology languages: The query answering problem, *Artificial Intelligence* 193 (2012) 87–128.
- [10] J. Baget, M. Mugnier, S. Rudolph, M. Thomazo, Walking the complexity lines for generalized guarded existential rules, in: *IJCAI*, 2011, pp. 712–717.
- [11] A. Cali, G. Gottlob, M. Kifer, Taming the infinite chase: Query answering under expressive relational constraints, *J. Artif. Intell. Res.* 48 (2013) 115–174.
- [12] N. Leone, M. Manna, G. Terracina, P. Veltri, Efficiently computable datalog \exists programs, in: G. Brewka, T. Eiter, S. A. McIlraith (Eds.), *KR 2012*, AAAI Press, 2012.
- [13] G. Gottlob, T. Lukasiewicz, B. Marnette, A. Pieris, *Datalog+/-: A family of logical knowledge representation and query languages for new applications* (2010).
- [14] L. Bellomarini, E. Sallinger, G. Gottlob, The vadalog system: Datalog-based reasoning for knowledge graphs, *PVLDB* 11 (2018) 975–987.
- [15] L. Bellomarini, D. Benedetto, G. Gottlob, E. Sallinger, Vadalog: A modern architecture for automated reasoning with large knowledge graphs, *Information Systems* (2020) 101528.
- [16] L. Bellomarini, G. Gottlob, A. Pieris, E. Sallinger, Swift logic for big data and knowledge graphs, in: *IJCAI*, 2017, pp. 2–10.
- [17] P. Atzeni, L. Bellomarini, D. Benedetto, E. Sallinger, Traversing knowledge graphs with good old (and new) joins (2021).
- [18] P. Atzeni, L. Bellomarini, M. Iezzi, E. Sallinger, A. Vlad, Weaving enterprise knowledge graphs: The case of company ownership graphs., in: *EDBT*, 2020, pp. 555–566.
- [19] L. Bellomarini, M. Benedetti, S. Ceri, A. Gentili, R. Laurendi, D. Magnanimiti, M. Nissl, E. Sallinger, Reasoning on company takeovers during the covid-19 crisis with knowledge graphs, in: *RuleML+ RR (Supplement)*, 2020.
- [20] G. Gottlob, A. Pieris, Beyond SPARQL under OWL 2 QL entailment regime: Rules to the rescue, in: *IJCAI*, 2015, pp. 2999–3007.

- [21] B. Glimm, C. Ogbuji, S. Hawke, I. Herman, B. Parsia, A. Polleres, A. Seaborne, SPARQL 1.1 entailment regimes, 2013. W3C Recommendation 21 March 2013, 2013.
- [22] R. Fagin, P. Kolaitis, R. Miller, L. Popa, Data exchange: Semantics and query answering, in: ICDT, 2003, pp. 207–224.
- [23] G. Graefe, W. J. McKenna, The volcano optimizer generator: Extensibility and efficient search, in: ICDE, 1993, pp. 209–218.
- [24] T. Baldazzi, L. Bellomarini, E. Sallinger, P. Atzeni, iwarded: A system for benchmarking datalog+/-reasoning (technical report), arXiv preprint arXiv:2103.08588 (2021).