Efficient Model Construction for Horn Logic with VLog
System Description[⋆]

Jacopo Urbani[1], Markus Krötzsch[2], Ceriel Jacobs[1], Irina Dragoste[2], and David Carral[2]

[1] Vrije Universiteit Amsterdam, The Netherlands, `firstname@cs.vu.nl`
[2] cfaed, TU Dresden, Germany, `firstname.lastname@tu-dresden.de`

Abstract. We extend the Datalog engine VLog to develop a column-oriented implementation of the *skolem* and the *restricted chase* – two variants of a sound and complete algorithm used for model construction over theories of existential rules. We conduct an extensive evaluation over several data-intensive theories with millions of facts and thousands of rules, and show that VLog can compete with the state of the art, regarding runtime, scalability, and memory efficiency.

1 Introduction

Rules of inference are a fundamental building block of many important algorithms in automated reasoning, and in related fields, such as artificial intelligence, data analytics, information integration, and knowledge management. They are at the core of leading tools and methods in many areas, ranging from logic programming, tableaux-based model construction, and "consequence-driven" approaches to ontological reasoning [13,14], over data integration [11] and query answering under constraints [7], to reasoning over knowledge graphs [16], and even social network analysis [18]. The optimisation of rule-based inferencing is therefore of crucial interest to automated reasoning.

In the recent past, there has been significant progress in this area, and many new rule-based systems have been presented [2,3,5,6,12,17,20]. At the core of these implementations is the most basic rule language *Datalog*, which syntactically corresponds to Horn logic without functions or existential quantifiers, while semantically it might be viewed either as a query language (reasoning = second-order model checking) or as a knowledge representation language (reasoning = first-order entailment checking) [1]. Systems nevertheless may exhibit strong differences due to the different use cases they have been designed for, which often also leads to different extensions and limitations.

One of the most important such extensions is support for *value invention*, manifested in the ability to handle either existential quantifiers or function terms in the consequences of rules. Equivalent formalisms are *existential rules* in ontological modelling, *tuple-generating dependencies* in database query answering, and Horn logic programs with function symbols in logic programming. The ability to create new terms during reasoning is crucial in many applications, e.g., for capturing incomplete information in databases [1], or for creating auxiliary structures in knowledge modelling [15]. But it is also

⋆ This work was supported by the DFG within the cfaed Cluster of Excellence, CRC 912 (HAEC), and Emmy Noether grant KR 4381/1-1.

much harder to implement since the resulting logic may no longer admit finite universal models, and reasoning becomes undecidable [10].

In this system description, we present our recent implementation of existential rule reasoning support in the Datalog engine VLog [20]. VLog differs from many other systems because of its column-based ("vertical") approach for storing inferred facts. This leads to high memory efficiency and competitive runtimes, but also requires specific implementation strategies and data structures. To the best of our knowledge, existential rule reasoning has never been implemented or studied in such an architecture.

Rule engines typically implement the so-called *chase* procedure – a saturation-based bottom-up model construction akin to a Horn logic tableau procedure. We implement it in two variants, the *skolem chase* and the *restricted chase*, of which the latter is more complicated but can produce smaller models in many cases. Indeed, it has recently been demonstrated that the restricted chase can compute models for many real-world ontologies where the skolem chase fails to terminate altogether [8]. This often requires Datalog rules to be preferred over existential rules, which we ensure in VLog.

We conduct an extensive evaluation to gauge the performance of our tool in comparison to the state of the art. In a recent evaluation of several chase implementations, Benedikt et al. found RDFox to be the most efficient tool in many contexts [4]. RDFox is also similar to VLog in that both conduct most of their computation in memory. We therefore compare VLog against RDFox, repeating many experiments of Benedikt et al. and adding several more using further real-world datasets. We find that, for reasoning with plain existential rules on a reasonably powerful laptop, VLog can often deliver comparable or even better performance than RDFox, while consistently needing much less memory. The former came as a surprise, since RDFox could take full advantage of its highly parallel algorithms, whereas VLog ran on a single thread on one CPU.

2 Preliminaries

We give a brief account of the relevant basic definitions and notation. Existential rules are based on a standard predicate logic vocabulary consisting of infinite, mutually disjoint sets of *predicates* $\mathbf{P}$ (each with a fixed arity), *constants* $\mathbf{C}$, and *variables* $\mathbf{V}$. A *term* is a variable $x \in \mathbf{V}$ or a constant $c \in \mathbf{C}$. An *atom* is a formula of the form $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms, and $p \in \mathbf{P}$ is a predicate of arity n. An *existential rule* (or simply *rule* in the context of this paper) is a formula of the form

$$\forall \boldsymbol{x} \forall \boldsymbol{y}.(B_1 \wedge \ldots \wedge B_k \rightarrow \exists \boldsymbol{v}.H_1 \wedge \ldots \wedge H_l), \tag{1}$$

where $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{v}$ are mutually disjoint lists of variables, and $B_1, \ldots, B_k$ are atoms with variables from $\boldsymbol{x}$ and $\boldsymbol{y}$, $H_1, \ldots, H_l$ are atoms with all variables from $\boldsymbol{y}$ and $\boldsymbol{v}$, $l \geq 1$, and all variables in $\boldsymbol{y}$ occur in $B_1, \ldots, B_k$. The premise of a rule is called the *body*, while its conclusion is called the *head*. A *Datalog rule* is a rule without existential quantifiers, and a rule with $k = 0$ is called a *fact* (a conclusion that is unconditionally true). A finite set of facts is a *database*. Since all variables in rules are quantified, we often omit the explicit preceding universal quantifiers.

Example 1. The following rules capture basic part-whole relationships (meronomy), and are a typical pattern in many ontologies.

$$Bicycle(x) \to \exists v.hasPart(x, v) \wedge Wheel(v) \tag{2}$$
$$Wheel(x) \to \exists w.properPartOf(x, w) \wedge Bicycle(w) \tag{3}$$
$$properPartOf(x, y) \to partOf(x, y) \tag{4}$$
$$hasPart(x, y) \to partOf(y, x) \tag{5}$$
$$partOf(x, y) \to hasPart(y, x) \tag{6}$$

A major reasoning task of rule engines is (conjunctive) query answering. A *conjunctive query* (CQ) is a formula $\exists \boldsymbol{v}.B_1 \wedge \ldots \wedge B_k$, where B_i are atoms. Free variables (not in $\boldsymbol{v}$) are called *answer variables*. A *substitution* is a partial mapping $\sigma : \mathbf{V} \to \mathbf{V} \cup \mathbf{C}$. It is *ground* if it only maps to constants. Its application to terms and formulae is defined as usual. An *answer* to a CQ q over a set of rules $\mathcal{R}$ and database $\mathcal{D}$ is a ground substitution σ defined on the answer variables of q such that $\mathcal{R}, \mathcal{D} \models q\sigma$ under the usual semantics of first-order logic. Existential variables can be replaced by function terms. The *skolemisation* of a rule ρ as in (1) is obtained by replacing each variable $v \in \boldsymbol{v}$ by the term $f_{\rho,v}(\boldsymbol{y})$, where $f_{\rho,v}$ is a fresh skolem function symbol specific to ρ and v.

3 The Chase

The *chase* is a class of sound and complete reasoning algorithms that are widely used to implement query answering [4]. Rules are applied bottom-up until saturation, resulting in a *universal model*, which matches exactly those queries that are entailed by the original rules (and given data). For existential rules, the chase may fail to terminate (approximating an infinite universal model instead), and detecting termination is undecidable [1]. However, many decidable criteria that are sufficient for termination have been proposed and shown to be applicable in many practical cases [9]. There are many variants of the chase, depending, e.g., on which conditions are checked to determine whether the consequence of an applicable rule should be added. In this section, we explain the *restricted* and *skolem chase* since these are among the most studied variants.

Any chase produces a sequence of databases $\mathcal{D}^0, \mathcal{D}^1, \ldots$, beginning from the initially given database. In the cases we consider, we have $\mathcal{D}^{i+1} = \mathcal{D}^i \cup \Delta^{i+1}$, for the set Δ^{i+1} of facts derived in step $i + 1$. We use abbreviations $\Delta^{[i,j]} = \bigcup_{k=i}^{j} \Delta^k$, $\Delta^0 = \mathcal{D}$ (the initial database), and $\Delta^{-1} = \emptyset$. In the chase variants we consider, only one rule is applied in each chase step, and consecutive chase steps consider different rules. We therefore store, for each rule ρ, the index prev_ρ of the chase step when it was last applied.

Algorithm 1 shows how one rule ρ is applied during the chase to compute Δ^{i+1}. Line 1.2 iterates over all *matches* of ρ: a *match* of a rule $\forall \boldsymbol{x}, \boldsymbol{y}.\varphi \to \exists \boldsymbol{v}.\psi$ over a database $\mathcal{D}$ is a ground substitution σ defined on $\boldsymbol{x} \cup \boldsymbol{y}$ such that $\mathcal{D} \models \varphi\sigma$. The additional requirement $\varphi\sigma \cap \Delta^{[\ell,i]} \neq \emptyset$ ensures that we only consider matches that were not found up to the previous application of ρ. This corresponds to a *semi-naive* materialisation strategy; we omit the details of how the matches σ can be found in practice [20]. Line 1.3 verifies that the entailments under a given match are logically relevant. Line 1.4 selects

Algorithm 1: applyRule(rule $\rho = \forall \boldsymbol{x}, \boldsymbol{y}.\varphi \to \exists \boldsymbol{v}.\psi$)

```
Global variables : index $i$, index $\mathsf{prev}_\rho$, previous derivations $(\Delta^k)_{k \leq i}$, bool changed
1.1 $\Delta^{i+1} = \emptyset$    $\ell = \mathsf{prev}_\rho$
1.2 foreach match $\sigma$ of $\rho$ over $\Delta^{[0,i]}$ with $\varphi\sigma \cap \Delta^{[\ell,i]} \neq \emptyset$ do
1.3     if $\Delta^{[0,i+1]} \not\models \exists \boldsymbol{v}.\psi\sigma$ then
1.4         $\sigma' = \sigma \cup \{\boldsymbol{v} \mapsto \boldsymbol{n}\}$ where $\boldsymbol{n} \subseteq \mathbf{Nulls}$ is fresh
1.5         $\Delta^{i+1} = \Delta^{i+1} \cup \{\psi\sigma'\} \setminus \Delta^{[0,i]}$
1.6 $\mathsf{prev}_\rho = i + 1$
1.7 $i = i + 1$
1.8 if $\Delta^i \neq \emptyset$ then changed = true
```

Algorithm 2: restrictedChase(rule set $\mathcal{R}$, database $\mathcal{D}$)

```
2.1 $i = 0$    $\Delta^0 = \mathcal{D}$    changed = true    $\mathsf{prev}_\rho = -1$ for all rules $\rho \in \mathcal{R}$
2.2 while changed do
2.3     changed = false
2.4     foreach $\rho \in \mathcal{R}$ do applyRule($\rho$)
2.5 return $\Delta^{[0,i]}$                // final result: union of all derived facts
```

fresh labelled nulls for instantiating the newly derived fact(s), which then get(s) added. After finishing, we update ρ's step counter (Line 1.6) and global chase step (Line 1.7). Global variable changed records if any fact was derived (Line 1.8).

Algorithm 2 now shows the overall *restricted chase* procedure. It is named after the check in Line 1.3, which restricts the application of rules – when omitting this check, one obtains the *oblivious chase* instead. The restricted chase can reduce the number of derived facts, which may allow it to terminate in more cases than the oblivious chase.

Example 2. Consider the restricted chase over the rules from Example 1 with database $\mathcal{D} = \{Bicycle(c)\}$. Applying rules in the given order, the first iteration of Line 2.4 yields $\Delta^0 = \mathcal{D}$, $\Delta^1 = \{hasPart(c, n_1), Wheel(n_1)\}$, $\Delta^2 = \{properPartOf(n_1, n_2), Bicycle(n_2)\}$, $\Delta^3 = \{partOf(n_1, n_2)\}$, $\Delta^4 = \{partOf(n_1, c)\}$, and $\Delta^5 = \{hasPart(n_2, n_1)\}$. Note that, when computing Δ^2, the check in Line 1.3 finds that $\Delta^{[0,2]} \not\models \exists w.partOf(n_1, w) \wedge Bicycle(w)$. No further derivations are produced thereafter; specifically the previous inferences already entail $\exists v.hasPart(n_2, v), Wheel(v)$. In contrast, the oblivious chase in this case would not terminate, since it would continue to apply rule (2) to new nulls.

Example 3. In contrast to the oblivious chase, the restricted chase is sensitive to the order of rules. For Example 2, if we apply rules in order (2), (3), (5), (6), (4), then we obtain $\Delta^0 = \mathcal{D}$, $\Delta^1 = \{hasPart(c, n_1), Wheel(n_1)\}$, $\Delta^2 = \{properPartOf(n_1, n_2), Bicycle(n_2)\}$, $\Delta^3 = \{partOf(n_1, c)\}$, $\Delta^4 = \emptyset$, and $\Delta^5 = \{partOf(n_1, n_2)\}$. Rule (3) can then be applied to match $\{x \mapsto n_2\}$ before $hasPart(n_2, n_1)$ gets inferred. The chase does not terminate.

Finally, the *skolem chase* is obtained by initially applying skolemisation to the rules in $\mathcal{R}$. This eliminates all existential variables, so that we have $\sigma = \sigma'$ in Line 1.4.

Algorithm 3: restrictedOrderedChase(rule set $\mathcal{R}$, database $\mathcal{D}$)

```
3.1 $i = 0$    $\Delta^0 = \mathcal{D}$    changed = **true**    $\text{prev}_\rho = -1$ for all rules $\rho \in \mathcal{R}$
3.2 while changed do
3.3     changed = false
3.4     foreach Datalog rule $\rho \in \mathcal{R}$ do applyRule($\rho$)
3.5     if ¬changed then
3.6         foreach Non-Datalog rule $\rho \in \mathcal{R}$ do applyRule($\rho$)
3.7 return $\Delta^{[0,i]}$                // final result: union of all derived facts
```

Moreover, Line 1.3 in this case is merely a syntactic check for duplicates: since $\psi\sigma$ is ground, $\Delta^{[0,i+1]} \models \psi\sigma$ holds only if $\psi\sigma \subseteq \Delta^{[0,i+1]}$. The skolem chase terminates in significantly more cases than the oblivious chase, but it is still inferior to the restricted chase in this respect.

4 Chasing in VLog

VLog adopts the distinctive approach of computing each set Δ^i in bulk using an efficient "set-at-a-time" processing, storing the set of derivations column-by-column rather than row-by-row. Recent literature on columnar databases has shown that columnar data structures are very memory efficient and enable fast data access, but cannot be updated easily [20]. To avoid this problem, VLog works in an append-only mode and stores each set Δ^i into a dedicated data structure. This strategy avoids the problem of updates altogether, and in practice has resulted in significantly shorter runtimes and lower memory consumption than the state-of-the-art – sometimes up to an order of magnitude.

The rest of this section sums up some of our main insights on implementing the restricted and the skolem chase efficiently in VLog. For the restricted chase, we make two further adjustments. First, we do not consider facts that were derived in the current (ongoing) chase step for checking if a rule application is restricted. Line 1.3 therefore checks if $\Delta^{[0,i]} \not\models \exists \boldsymbol{v}.\psi\sigma$. This leads to what is called the *1-parallel restricted chase* [4].

Second, we ensure that Datalog rules are applied exhaustively before considering existential rules, as shown in Algorithm 3. This is motivated by recent studies of Carral et al., who proposed a criterion that uses this order to detect chase termination in more cases than previous works [8]. In fact, Example 1 shows a case for which VLog's restricted chase terminates, while other restricted chase implementations (e.g., of RDFox) do not.

From an implementation perspective, the execution of a rule can be split into the computation of all matches of the rule and the consequent computation of instantiations of the head. The first operation is the same regardless whether the rule contains existential quantifiers or not. Thus, we can reuse the same efficient algorithms developed for non-existential rule execution. The second operation, in contrast, requires ad-hoc operations due to the existence of unbound variables.

The exact operations differ depending on whether a restricted or a skolem chase is being computed. In the first case, we perform a series of merge joins between the set of matches and the columnar data structures that store the existing facts to remove

Table 1. Rules and databases used in benchmarks (*MA* is maximal predicate arity)

Dataset	Number of rules	Number of facts	MA
Uniprot-005 / 010	531	4,713,207 / 9,252,708	2
Reactome-040 / 060 / 080	601	3,144,962 / 4,400,913 / 5,604,133	2
UOBM-10 / 20 / 40	426	1,926,879 / 3,980,967 / 7,843,543	2
STB-128	198	1,109,037	10
Ontology-256	529	2,146,490	11
doctors-10K / 1M	16	10,837 / 951,500	6
LUBM-010 / 100 / 1K	136	1,272,575 / 13,405,381 / 133,573,854	2
deep-100 / 200 / 300	1,100 / 1,200 / 1,300	1,000	4

partly instantiated matches. A merge join is very efficient here because the columnar data structures are already sorted [20]. Notice that if the head of the rule is a conjunction of multiple atoms, this procedure must be repeated for each head atom. Whenever the merge join finds a substitution to remove, it adds an entry into a positional index and use this index to skip to-be-removed matches. This strategy is adopted to avoid costly in-place removals. In the second case, we do not need to remove matches but we must retrieve the correct skolem terms. To support this operation, the system maintains a series of hash maps in main memory (one per rule/variable) with the arguments of the function and use it to return fresh IDs with average constant time.

5 Evaluation

We conducted an evaluation to gauge the performance and correctness of VLog.[3] We compared to RDFox, which emerged as a leading tool in [4]. Experiments were conducted in a laptop system (2.2 GHz Intel Core i7 (4 CPUs), 16GB 1600MHz DDR3, 512GB SSD, MacOS High Sierra v10.13.3). The benchmark inputs we use are shown in Table 1. UOBM, Reactome, and Uniprot are based on data-intensive OWL ontologies,[4] which we converted to rules after removing non-deterministic axioms that do not correspond to Horn logic rules. The remaining benchmarks are as given by Benedikt et al., where we omitted the rules with equality, which are not supported by VLog [4].

For all tests, we measured the time and peak memory used for computing (a) the restricted chase and (b) the skolem chase. We also verified that the size of the skolem chases was the same for VLog and RDFox in all cases (the restricted chase shows minor fluctuations, as expected for the different implementations). The results are shown in Fig. 1. VLog could finish deep-300 on our laptop, but using some OS swap space. Since we cannot measure this reliably, we only report a lower bound.

VLog generally used much less memory, on average 40% of what was used by RDFox in either chase. This is expected since VLog uses highly optimised compressed data structures. In cases where only VLog finished (LUBM-1K and deep-300), RDFox ran out of memory. The times taken by VLog ranged from 5.8% (deep-100, rest.) to 137.5% (doctors-1M, rest.) of what was needed by RDFox. This is surprising, since VLog used

[3] All files used in this section are available at https://github.com/karmaresearch/Chasing-VLog.

[4] Source http://www.cs.ox.ac.uk/isg/tools/PAGOdA/2015/jair/, accessed 2 Feb 2018

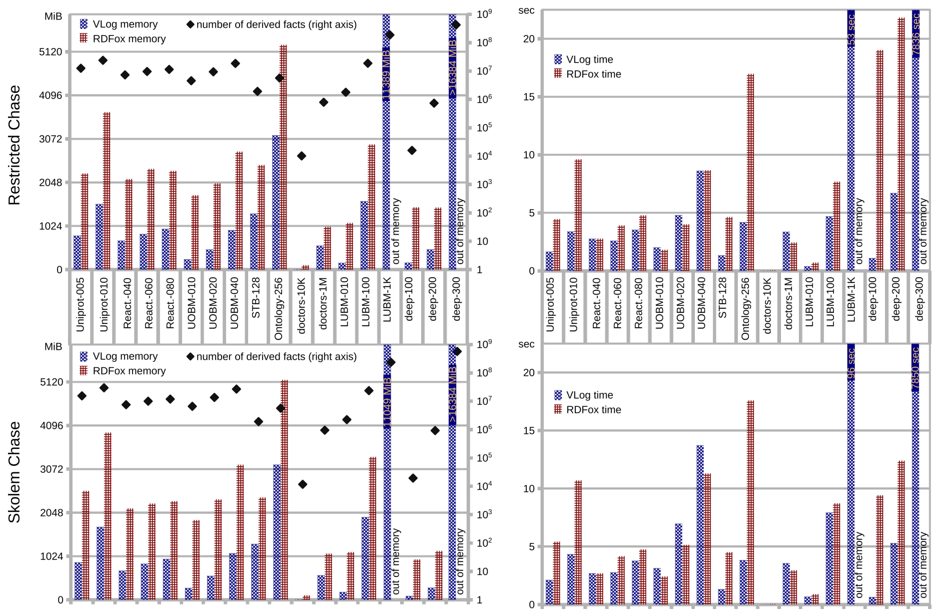

Fig. 1. Memory usage (left) and materialisation time (right) for VLog and RDFox

only a single thread, whereas RDFox used maximal parallelism and often achieved above 700% CPU utilisation. Comparing the chase variants, VLog used significantly less time and memory for the restricted chase, except on deep-100, deep-200, and Ontology-256. RDFox shows similar behaviour, though the additional cost on deep is more pronounced. Nevertheless, the restricted chase seems to be the more efficient algorithm in general.

6 Conclusions

VLog is a fast and memory-efficient system for constructing models for Horn Logic. We extended its set-at-a-time and columnar approach to handle existential rules and discussed our implementation of the chase, which exhibits excellent performance.

The system is free and open source,[5] with only few dependencies for optional database connectors. Pre-compiled Docker images enable quick installation on major platforms (Docker repository *karmaresearch/vlog*). Users can control VLog through a command-line tool, a web interface (useful for demonstrating the system), and though the Java bindings of the companion project *VLog4j*.[6] The latter is available as a Maven package that includes the necessary binaries for major operating systems. In the future, we plan to add further expressive features, such as equality, negation, or aggregation. This can make VLog useful in even more scenarios, and thereby further advance our understanding of the potential of this architecture for automated reasoning in general.

[5] C++ source code and documentation: https://github.com/karmaresearch/vlog
[6] Java source code and documentation: https://github.com/mkroetzsch/vlog4j

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley (1994)
2. Aref, M., ten Cate, B., Green, T.J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T.L., Washburn, G.: Design and implementation of the LogicBlox system. In: Proc. 2015 ACM SIGMOD Int. Conf. on Management of Data. pp. 1371–1382. ACM (2015)
3. Baget, J., Leclère, M., Mugnier, M., Rocher, S., Sipieter, C.: Graal: A toolkit for query answering with existential rules. In: Proc. 9th Int. Web Rule Symposium (RuleML'15). LNCS, vol. 9202, pp. 328–344. Springer (2015)
4. Benedikt, M., Konstantinidis, G., Mecca, G., Motik, B., Papotti, P., Santoro, D., Tsamoura, E.: Benchmarking the chase. In: Proc. 36th Symposium on Principles of Database Systems (PODS'17). pp. 37–52. ACM (2017)
5. Benedikt, M., Leblay, J., Tsamoura, E.: PDQ: proof-driven query answering over web-based data. PVLDB 7(13), 1553–1556 (2014)
6. Bonifati, A., Ileana, I., Linardi, M.: Functional dependencies unleashed for scalable data exchange. In: Proc. 28th Int. Conf. on Scientific and Statistical Database Management (SS-DBM'16). pp. 2:1–2:12. ACM (2016)
7. Calì, A., Gottlob, G., Kifer, M.: Taming the infinite chase: Query answering under expressive relational constraints. In: Proc. 11th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'08). pp. 70–80. AAAI Press (2008)
8. Carral, D., Dragoste, I., Krötzsch, M.: Restricted chase (non)termination for existential rules with disjunctions. In: Sierra [19], pp. 922–928
9. Cuenca Grau, B., Horrocks, I., Krötzsch, M., Kupke, C., Magka, D., Motik, B., Wang, Z.: Acyclicity notions for existential rules and their application to query answering in ontologies. J. of Artificial Intelligence Research 47, 741–808 (2013)
10. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Computing Surveys 33(3), 374–425 (2001)
11. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data exchange: semantics and query answering. Theoretical Computer Science 336(1), 89–124 (2005)
12. Geerts, F., Mecca, G., Papotti, P., Santoro, D.: That's all folks! LLUNATIC goes open source. PVLDB 7(13), 1565–1568 (2014)
13. Kazakov, Y.: Consequence-driven reasoning for Horn $\mathcal{SHIQ}$ ontologies. In: Proc. 21st Int. Joint Conf. on Artificial Intelligence (IJCAI'09). pp. 2040–2045. IJCAI (2009)
14. Kazakov, Y., Krötzsch, M., Simančík, F.: The incredible ELK: From polynomial procedures to efficient reasoning with $\mathcal{EL}$ ontologies. J. of Automated Reasoning 53, 1–61 (2013)
15. Krötzsch, M., Thost, V.: Ontologies for knowledge graphs: Breaking the rules. In: Proc. 15th Int. Semantic Web Conf. (ISWC'16). LNCS, vol. 9981, pp. 376–392 (2016)
16. Marx, M., Krötzsch, M., Thost, V.: Logic on MARS: Ontologies for generalised property graphs. In: Sierra [19], pp. 1188–1194
17. Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: RDFox: A highly-scalable RDF store. In: et al., M.A. (ed.) Proc. 14th Int. Semantic Web Conf. (ISWC'15), Part II. LNCS, vol. 9367, pp. 3–20. Springer (2015)
18. Seo, J., Guo, S., Lam, M.S.: SociaLite: an efficient graph query language based on Datalog. IEEE Trans. Knowl. Data Eng. 27(7), 1824–1837 (2015)
19. Sierra, C. (ed.): Proc. 26th Int. Joint Conf. on Artificial Intelligence (IJCAI'17). IJCAI (2017)
20. Urbani, J., Jacobs, C., Krötzsch, M.: Column-oriented Datalog materialization for large knowledge graphs. In: Proc. 30th AAAI Conf. on Artificial Intelligence (AAAI'16). pp. 258–264. AAAI Press (2016)