VLog: A Rule Engine for Knowledge Graphs

David Carral[1], Irina Dragoste[1], Larry González[1], Ceriel Jacobs[2], Markus Krötzsch[1], and Jacopo Urbani[2(✉)]

[1] TU Dresden, Dresden, Germany
{david.carral,irina.dragoste,larry.gonzalez,
markus.kroetzsch}@tu-dresden.de
[2] Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
{ceriel,jacopo}@cs.vu.nl

Abstract. Knowledge graphs are crucial assets for tasks like query answering or data integration. These tasks can be viewed as reasoning problems, which in turn require efficient reasoning systems to be implemented. To this end, we present VLog, a rule-based reasoner designed to satisfy the requirements of modern use cases, with a focus on performance and adaptability to different scenarios. We address the former with a novel vertical storage layout, and the latter by abstracting the access to data sources and providing a platform-independent Java API. Features of VLog include fast Datalog materialisation, support for reasoning with existential rules, stratified negation, and data integration from a variety of sources, such as high-performance RDF stores, relational databases, CSV files, OWL ontologies, and remote SPARQL endpoints.

1 Introduction

Semantic web research covers a wide range of topics from knowledge representation, over information integration, to query answering and data analysis. Only a few concepts are important throughout all of these areas. One is the *Knowledge Graph* (KG) concept, that is, a knowledge base that can be represented as an entity-relationship graph. Another one is the *rule* concept, used to derive implicit consequences from given inputs: combinations of rules and (OWL) ontologies have a long tradition [22,28], and recent works introduce rules as ontology languages in their own right [3,12]. Moreover, rules play a key role in many reasoning algorithms [20,21,40]; database dependencies are rules used in data access and information integration [13]; and rules are also the basis of expressive query languages [1] used in graph analysis [34]. It is therefore not surprising that many new rule engines have been created in recent years [4,5,7,14,29,37].

These rule engines are used to solve many different use cases. For instance, the engine Llunatic [14] is tailored to solve data integration issues [13]; that is, to translate data from one or more sources into a single target database. The system RDFox [29] has been used to perform sophisticated data analysis for

C. Ghidini et al. (Eds.): ISWC 2019, LNCS 11779, pp. 19–35, 2019.
https://doi.org/10.1007/978-3-030-30796-7_2

the healthcare provider Kaiser Permanente in [31] (more RDFox use cases are described at https://www.oxfordsemantic.tech/usecases). Furthermore, using acyclicity notions [8,12] or consequence preserving DL-to-Datalog translations in [9–11], one can effectively employ rule engines to solve reasoning tasks over a large subset of OWL ontologies. Note that when it comes to reasoning over ontologies with large amounts of assertions, rule engines are much faster and scalable than state-of-the-art DL reasoners (see the evaluations in [9–11]).

We have recently extended our own rule engine *VLog* [37] with a highly efficient bottom-up computation strategy for existential rules (i.e, rules that allow for existential quantifiers in the head), and showed that it can outperform efficient rule engines such as RDFox [29] in a range of widely common benchmarks [38]. This performance enables rule-based reasoning over KGs with hundreds of millions of facts on a regular laptop, making this system valuable for semantic web applications that involve large KGs such as Wikidata [39].

In spite of these technical achievements, the research prototype used in our previous evaluations was hardly a polished software product, and deployment and practical usage was challenging. Moreover, VLog could originally only be controlled from the command line, making it difficult to interface with it from software applications – arguably one of the main uses of a knowledge representation and data analysis platform. To overcome these obstacles, we have developed VLog from a research prototype into a re-usable software package that bundles many new functionalities:

- Existential rule reasoning support using an optimised version of the *restricted* and *skolem chase* algorithms.
- Support for *stratified negation* [1], allowing negated atoms in rule bodies.
- Translation of OWL and RDFS ontologies into equivalent rule and fact sets.
- Integration with the Graal rule library [4] and its data structures (e.g., existential rules, facts, and queries). This includes support for loading rules in Graal's *DLGP* syntax.
- Methods for *static analysis* of rule sets, e.g., to verify the termination of reasoning over sets of existential rules using *acyclicity notions* [8,12].
- A *data federation layer* to integrate – seamlessly and on demand – data from many sources, including various database management systems, file formats, SPARQL endpoints, and data provided from Java programs.
- All these features are accessible through the Java library *VLog4j*, which provides a full-fledged API for rule representation and reasoning.

VLog (C++) and VLog4j (Java) are free and open source, and use public repositories for development, issue tracking, and continuous integration.[1] This paper is based on VLog v1.2.0 and Vlog4j v0.3.0. Packages for simple installation are distributed via Maven.

We present VLog(4j) through a practical example (Sect. 2) and then give a detailed system overview (Sect. 3). Further sections include a performance evaluation (Sect. 4), a detailed discussion of related tools (Sect. 5), and practical hints on how to obtain VLog (Sect. 6).

[1] https://github.com/karmaresearch/vlog and https://github.com/knowsys/vlog4j.

$$\mathsf{subClHier}(X, Y) \text{ :- } \mathsf{doidRdf}(X, \mathsf{rdfs{:}subClassOf}, Y). \quad (1)$$
$$\mathsf{subClHier}(X, Z) \text{ :- } \mathsf{subClHier}(X, Y), \mathsf{doidRdf}(Y, \mathsf{rdfs{:}subClassOf}, Z). \quad (2)$$
$$\mathsf{doid}(X, Y) \text{ :- } \mathsf{doidRdf}(X, \mathsf{geneon{:}id}, Y). \quad (3)$$
$$\mathsf{cancerDisease}(Z) \text{ :- } \mathsf{subClHier}(X, Y), \mathsf{doid}(Y, \text{"DOID:162"}), \mathsf{doid}(X, Z). \quad (4)$$
$$\mathsf{diedOfCancer}(X) \text{ :- } \mathsf{deathCause}(X, Y), \mathsf{diseaseId}(Y, Z), \mathsf{cancerDisease}(Z). \quad (5)$$
$$\mathsf{diedOfNonCancer}(X) \text{ :- } \mathsf{deathCause}(X, Y), \mathsf{diseaseId}(Y, Z), \sim \mathsf{cancerDisease}(Z). \quad (6)$$
$$\mathsf{hasDoid}(X) \text{ :- } \mathsf{diseaseId}(X, Y). \quad (7)$$
$$\mathsf{diedOfNonCancer}(X) \text{ :- } \mathsf{deathCause}(X, Y), \sim \mathsf{hasDoid}(Y). \quad (8)$$
$$\mathsf{deathCause}(X, Z) \text{ :- } \mathsf{recentDeathsCause}(X, Z). \quad (9)$$
$$\mathsf{deathCause}(X, V) \text{ :- } \mathsf{recentDeaths}(X). \quad (10)$$

Fig. 1. Example for rule reasoning and data integration; geneon:id and rdfs:subClassOf are shortcuts for <http://www.geneontology.org/formats/oboInOwl#id> and <http://www.w3.org/2000/01/rdf-schema#subClassOf>, respectively

2 Functionality Overview

In this section we present an example that illustrates the use of VLog for data integration and reasoning, which allows us to explain VLog's main features in an intuitive way. We use two data sources: the Disease Ontology (DOID),[2] which contains information about human diseases and their relationships, and Wikidata [39], from which we retrieve information about recent fatalities attributed to certain diseases. This data will be integrated and reasoned over using the rules shown in Fig. 1, which we will explain step by step. Rules are written as in logic programming, with premise (body) on the right and conclusion (head) on the left. The overall code for running the example is available as part of VLog4j.[3]

Basic Rule Reasoning. We first configure VLog to use DOID as the only data source. Triples from the RDF serialisation of this ontology are mapped to facts of the form $\mathsf{doidRdf}(s, p, o)$. Then we can use rules (1) and (2) to compute the subclass hierarchy of diseases. Rule engines can capture much more complex OWL inferences [9], but RDFS reasoning suffices for this simple example. Rule (3) now extracts a string identifier for each disease IRI, and rule (4) combines this with the disease hierarchy to find all types of cancer (id DOID:162).

Combining Facts from Different Input Sources. VLog can load data from many different sources, including files of various formats and databases. In this example, we add data that is fetched from the live SPARQL endpoint of Wikidata [26]. For example, we can query for humans who died in 2018 as follows:

```
SELECT ?human WHERE { ?human wdt:P31 wd:Q5; wdt:P570 ?deathDate .
                      FILTER (YEAR(?deathDate)=2018)}
```

[2] More information about the disease ontology at http://disease-ontology.org/.
[3] See file DoidExample.java in the *vlog4j-examples* module (VLog4j repository).

where we use Wikidata IRI such as `wdt:P570` (date of death) or `wd:Q5` (human). The result of this query is mapped to VLog facts recentDeaths(hum). We further define SPARQL-based facts recentDeathsCause(hum, cau) (recent deaths with known cause of death) and diseaseId($dis, doid$) (diseases in Wikidata with a DOID identifier). We can now find all people who died of cancer in 2018, using rule (5). For the moment, let's assume that deathCause in the body holds just the data from recentDeathsCause, as inferred from rule (9). Using VLog, we find 562 cancer-related deaths in 2018.

Negation. VLog supports *stratified* negation, which relies on a simple syntactic check to ensure that no inference can depend recursively on its own negation [1]. Using $\sim$ for negation, rule (6) finds all recently deceased humans who died of a cause that was not cancer. However, there are also people whose cause of death cannot be found in DOID. To include these, we use rule (8), where hasDoid defines Wikidata diseases with a DOID (7). Overall, we thus find 1849 non-cancer casualties in Wikidata.

Existentials and Incomplete Information. These result could lead us to believe that 23% of recent deaths in Wikidata were due to cancer. However, many deceased have no cause of death stated, and are therefore not counted. We can state that every death must have some (possibly unknown) cause using rule existential quantifiers: rule (10) uses a variable Y that occurs only in the head to denote that some such Y must exist, i.e., the rule corresponds to the logical formula $\forall x.\exists y.\mathsf{deathCause}(x, y) \leftarrow \mathsf{recentDeaths}(x)$. This rule allows us to apply (8) even in cases where no cause was specified, leading to a total of 16,173 deaths that are not known to be caused by cancer.

Rule Syntax. Figure 1 uses a common logic programming syntax for illustration. In practice, VLog uses the Graal rule library for Java to read rules from files [4]. This library uses the *DLGP* format, which supports most of Fig. 1 as shown. Only negation is not supported by Graal yet, and our example program therefore constructs rules (6) and (8) directly in Java code.

OWL Support. Another way of defining rules is to load them from OWL ontologies. VLog has built-in methods for converting a (disjunction-free) subset of OWL into rules. In this transformation, OWL classes and properties become unary and binary predicates in VLog, which is different from our example, where classes (diseases) were represented as individual constants to achieve data integration with diseases from Wikidata. In practice, it is important to chose the right perspective on ontological data, and VLog provides this flexibility.

Reasoning Implementation. VLog's main approach for fast inference computation is bottom-up materialisation of consequences. The *standard* (a.k.a. *restricted*) *chase* is used as the main algorithm, but the *skolem* (a.k.a. *semi-oblivious*) *chase* is also supported [38]. In addition, VLog implements some

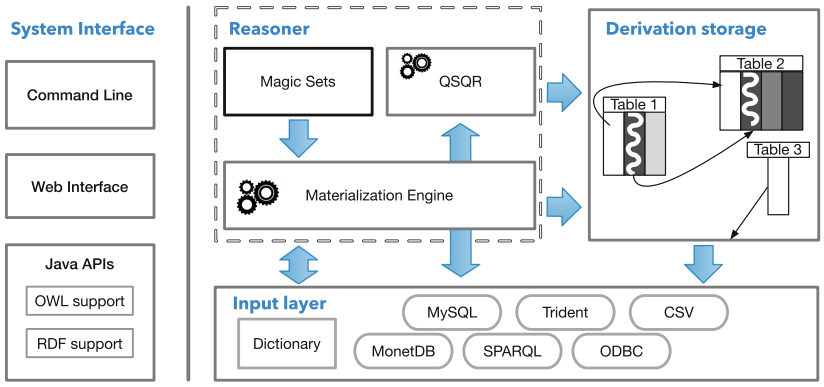

Fig. 2. Overview of the system architecture of VLog

heuristic optimisations based on goal-directed approaches such as *QSQR* and *Magic Sets* [37].

Since existential rules can entail new (unknown) values, reasoning may produce an unbounded number of new facts and thus fail to terminate. Detecting this is undecidable in general, but VLog supports several recently proposed checks that were found to determine chase termination in many practical cases [8].

3 System Overview

In this section we provide a high-level view of our design and overall architecture before elaborating on the details on individual components in the following sections. The design of VLog has been driven by five main requirements: *performance*, *efficiency*, *expressiveness*, *portability*, and the *ability of interfacing with existing technologies*.

Performance and efficiency, i.e., the ability to solve tasks quickly and with a minimum of resources, are obviously central to any reasoner. Performance is important because reasoning can be a time-consuming operation and some use cases introduce time constraints, e.g., to guarantee an interactive usage of the system. Efficiency is crucial to apply our solution also to platforms where the hardware is limited, e.g., IoT devices [35]. Expressiveness broadly refers to the system's ability to use rules that can describe the conceptual relationships of many relevant use cases. There is a well-known trade-off between expressive power and complexity of related computational tasks, so one has to balance this requirement with our considerations for performance.

Portability of a tool refers to its applicability on many different platforms, and as such is well-appreciated in general, and in the particularly diverse application scenarios encountered in the semantic web in particular. It can be challenging to provide portability without compromising performance. Our related requirement of interfacing with existing technologies is a natural consequence of the intention to use our rule engine as a key component for integrating and analysing knowledge from a variety of data sources, including legacy sources and sources that are not under the full control of the user.

In order to achieve good performance and efficiency, VLog takes the distinctive approach of using a *vertical* storage layout that stores derivations column-by-column rather than row-by-row (this approach has been described in more detail in [37]). This strategy is beneficial because it allows memory savings due to data-structure sharing, and is able to avoid much unnecessary computation. Expressiveness is addressed in several ways. Already on the level of the basic Datalog rule language, VLog supports predicates of arbitrary arity. Even in the world of triples, predicates with more than three parameters can be crucial for performing certain computations [21] and they have applications in utilising less strongly normalised data models, as, e.g., in modern knowledge graphs [39]. In addition, VLog supports *existential rules* that extend significantly beyond standard Datalog. Finally, portability and the ability of interfacing to existing sources are addressed at the system level by reducing the external dependencies to the minimum, and by imposing a strict separation between the underlying databases and the set of derivations. This leads to an architecture that can make use of many different data sources during reasoning.

VLog is a complex system where four major components are responsible for different tasks. The components and their interactions are illustrated in Fig. 2. They comprise: the *input layer*, which provides access to the underlying databases; the *derivation storage*, which stores the derivations in main memory; the *reasoner*, which is responsible for the computation of the derivations; and the *system interface*, which provides access to the functionalities to the system.

The components on the right of Fig. 2 are integral parts of the backend of VLog, which is implemented in C++. The system interface involves the Java API VLog4j, which is software project that uses VLog's backend as a dependency and comprises further sub-modules. Each of these components is described in more detail in the following sections.

3.1 Backend Components: Input and Derivation Storage, Reasoning

Input Layer. VLog keeps a strict distinction between data that is available in some external sources and data that is inferred by the rules. To enable a seamless integration with different data structures, we abstracted the access to these sources into a small API. We implemented this API so that our engine can read information from sources like RDF Triple stores, MySQL, ODBC (standard relational database API), remote SPARQL endpoints, and CSV tables. Extending the support to other sources is an operation that does not require a deep knowledge of the system. Note that internally VLog uses numerical IDs to compress the storage of strings. The conversion between strings into IDs (dictionary encoding) is not trivial if the data comes from multiple independent sources. In VLog, we addressed this challenge implementing a sophisticated mechanism to translates on-the-fly terms that are read from multiple sources to shared IDs.

Derivation Storage. A characteristic design choice of VLog is its optimised, "vertical" derivation storage that represents all facts that are computed during reasoning. These are stored in a series of in-memory data structures following the

distinctive columnar layout [37]. Moreover, the derivation storage also provides access to derivations in a similar way as the input layer.

Internally, columns of terms can be stored with different data structures. The most commonly used data structure is a plain in-memory array, but other representations are also possible to save memory. For instance, a special representation is used if the column consists of a list of the same repeated term. Another special data structure is used in case the column is a projection of a column of an input predicate. To illustrate this last case, consider as example the Datalog rule $\mathsf{H}(Y, X) \text{ :- } B(X, Y)$ where B is a predicate that maps to an underlying data source. In this case, the column that represents the first field of the H predicate (i.e., Y) is equivalent to the column X in B (assuming that no H-facts have been previously derived, which might require duplicate elimination). To save space, the column Y used in H does not contain a physical copy of all values retrieved from the input layer, but simply stores a query that will allow VLog to retrieve them as needed. This is possible because columns are immutable objects, and in practice results in large memory savings.

Reasoner. VLog supports two types of reasoning: *full materialisation* (i.e., the bottom-up computation of derived facts) and *query-driven reasoning* (i.e., the top-down search for answers to a given conjunctive query). Computing the full materialisation is perhaps the most common reasoning task in the Semantic Web community while query-driven reasoning is useful whenever full materialisation is not possible. The algorithm for performing full materialisation is conceptually simple as it can be seen as a single-threaded loop where all rules are executed one-by-one until saturation. VLog implements the usual "semi-naive" optimisation that largely reduces the amount of duplicates that are inferred, with slight modifications to account for the more fine-grained columnar data structures [37].

When dealing with existential rules, the process becomes significantly more complicated. A blind application of rules would almost always lead to the creation of unbounded numbers of new objects, and the process would not terminate. We therefore implement an additional restriction that checks if existing objects can be re-used to satisfy the conclusion of rules before creating any new objects. In detail, our approach is a variant of the *1-parallel restricted chase* in the terminology of Benedikt et al. [6]. We further refine this approach by ensuring that non-existential (plain Datalog) rules are always saturated before considering an existential rule, which achieves termination in additional cases that occur in real-world knowledge bases [8]. As an optional setting, we also implement the *skolem chase*, which uses a simpler check for deciding on rule applications and terminates in fewer cases. However, experiments suggest that this approach leads to lower performance and higher memory usage across all common benchmarks [38], so this algorithm is not used by default.

In contrast, query-driven reasoning considers an input query and only returns derivations that match it. Two well-known procedures are supported for query-driven reasoning: Magic Sets and QSQR [1]. The first is a rewriting technique which rewrites the rules so that the derivations produced by the rewritten rules are relevant for the input query while the second procedure is a set-based

variant of the well-known SLD procedure [1]. Since Magic Sets is a rewriting procedure, it does not perform any reasoning in itself but instead offloads it to the materialisation engine. In contrast, the QSQR algorithm has a dedicated implementation which uses in-memory lightweight data structures to store the intermediate derivations. This makes it suitable for answering queries which do not trigger substantial reasoning due to its small overhead. Magic sets, in contrast, exploits the efficient full materialisation engine so it is able to handle the remaining cases. VLog implements both procedures, and it is the user who can choose the method to use. The query-driven methods can optionally be enabled to heuristically increase reasoning performance even when using materialisation [37]. However, the methods are not applicable to rules with existential quantifiers in their common form, so we do not invoke them in such cases.

3.2 System Interface: Java Integration and Stand-Alone Programs

The system interface component of VLog comprises several independent modules for invoking the reasoner in a variety of application contexts. Concretely, VLog ships with two stand-alone programs – a command-line client and an interactive Web interface –, and is integrated into the Java library VLog4j, which allows the engine to be used within larger applications.

The Java API VLog4j. We have developed a new API for tight integration with Java, which is a popular language in the Semantic Web community. The purpose of this interface is not only to control VLog from Java, but also to provide a complete framework for working with rules and facts. We have therefore designed an object model for representing such data, and provided classes for configuring the reasoning process. Through several extension modules, the Java library can be used to obtain facts from RDF files and to extract rules and facts from OWL. Besides loading facts and rules directly from objects in memory, this library can also configure VLog to use multiple possible data sources, including SPARQL federation, and the results of the materialisation are streamed back using iterators.

This interface also includes some functionalities to simplify the use of the underlying rule engine. In particular, it supports *punning*, i.e., the use of the same predicate name for predicates of different arity. This is not currently allowed in VLog, but it is enabled by the Java interface by renaming predicates before passing them on to the backend. This library also provides methods for transforming rules and sets of rules; more specifically it can ensure that predicates that map to input sources are distinct from all predicates used in rule heads. Further algorithms for transformation and analysis of rule sets are planned for future development.

Conceptually, VLog4j includes some aspects of a data format representation library, making it more similar to Graal [4] than to RDFox in this respect. The successful OWL API [19] is an example of a similar project for OWL ontologies, and indeed has been a model for some of our design. When comparing VLog4j to Graal, we can see that the latter currently provides a larger set of transformation

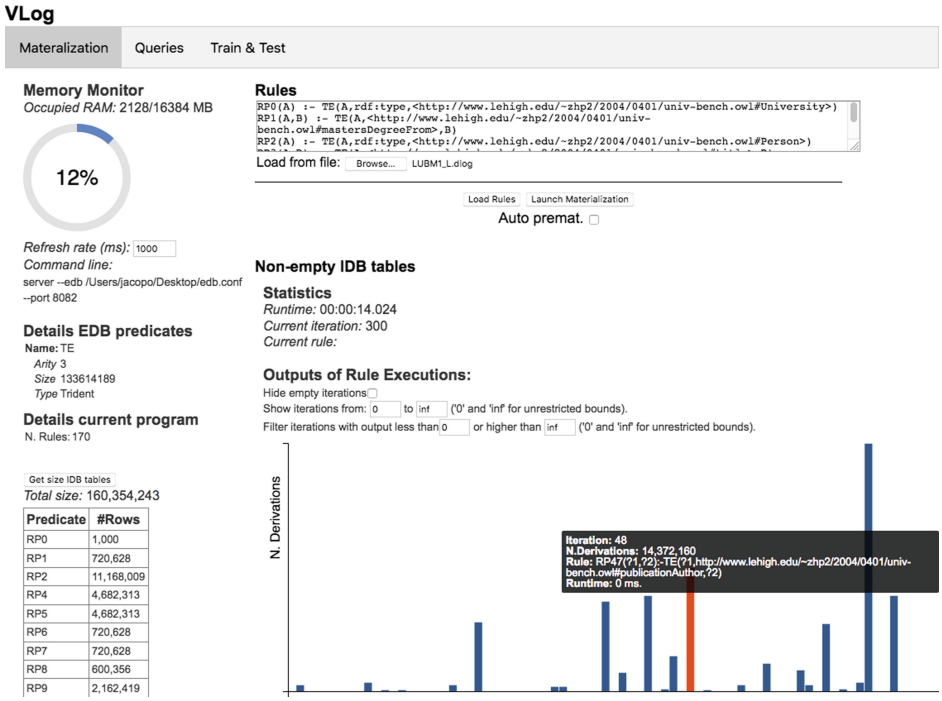

Fig. 3. VLog's Web interface during full materialisation

algorithms, whereas VLog4j comes with a significantly faster reasoning engine [6, 38]. We plan to interface with some components of Graal in upcoming releases so as to establish interoperability between the two projects – unfortunately, no standard for representing rules is widely accepted today, so rule representation APIs often have subtle structural or syntactic differences.

An important goal of VLog4j is to simplify usage, and we take several steps to support this. The online repository includes a Javadoc code documentation and a set of simple example programs to illustrate how to use VLog4j in several scenarios. The Java API is released as a multi-module project through Maven Central to ease its integration into existing projects.

Stand-Alone Programs. Two stand-alone executables are available to run VLog services without the additional Java layer.

Web Interface. We built a web interface to offer the user the ability to specify the rules without using any programming language, and for inspecting the results of the materialisation in a convenient way. The first reason is especially useful for educational purposes, while the second can ease the debugging of the system. A screenshot of the Web interface in action is shown in Fig. 3. On the left side, it reports some useful statistics about the resource consumption and other details about the input layer while the right side allows the user to specify the rules and inspect statistics which are shown as the materialisation progresses. Further information about how to use this interface can be found online[4].

[4] https://github.com/karmaresearch/vlog/wiki/Web-Interface.

Command Line. From the command line, the user can launch reasoning (both full materialisation and query-driven procedures) and export the results into a number of different formats. For instance, the user can request that all derivations are being exported as RDF triples or simply as CSV files. Moreover, if Trident is used as only input backend, then the reasoner can add back derivations to the original database to enable SPARQL queries on both original and derived triples.

4 Evaluation

A comparison between the performance of VLog and other state-of-the-art systems in computing the materialisation of KBs with large ABoxes is available at [37,38]. In this section, we evaluate the practical feasibility of solving conjunctive query (CQ) answering over data-intensive OWL ontologies using VLog.

Efficient DL reasoning support is highly relevant for our tool, as known rule engines are significantly faster than DL reasoners for solving standard reasoning tasks over ontologies with large data [9–11]. Moreover, (CQ) answering is a non-standard reasoning task that cannot be solved by DL reasoners [15,36].

To solve CQ answering, we use our implementation of the Datalog-first restricted chase (see Sect. 2). All test ontologies, queries, and result tables considered in this section are available online.[5] All experiments were conducted on a Mac Book Pro with 16 GB of RAM, and a 2,2 GHz Intel Core i7 processor.

We consider three real-world OWL ontologies and a benchmark. Each of these ontologies consists of a TBox (a terminological axiom set) and an ABox (a fact set).

- **ChEMBL**, **Reactome**, and **Uniprot** are real-world ontologies available from the European Bioinformatics Institute (EBI) online platform.[6] In order to test scalability on these large datasets, we make use of a data sampling algorithm based on random walks [25], and compute ABox subsets of increasing size. This algorithm was reimplemented for RDF-based data and used in [40].
- **LUBM** is a widely used ontology benchmark [18] modelling universities. The TBox in these ontologies has been manually created and is fixed, whilst an arbitrarily large ABox can be instantiated using an automatic generator.

For simplicity, we filter all axioms containing annotations, data properties, or datatypes. Since VLog does not support non-deterministic rules, we also remove (1) non-Horn axioms that cannot directly be transformed into deterministic existential rules (e.g., "subclass of" axioms containing a disjunction of class names in the superclass). Moreover, we ignore (2) all axioms that, if transformed into rules, would require the use of equality or inequality (e.g., functionality restrictions, or axioms featuring "at most" restrictions or "at least" restrictions with

[5] Evaluation materials at https://github.com/knowsys/eval-2019-ISWC-VLog.
[6] https://www.ebi.ac.uk.

Table 1. Statistics for TBoxes and translated rule sets: the columns report the number of classes and properties in the TBoxes, and the number of existential, Datalog, and non-Datalog rules in the translated rule sets in that order

	#Classes	#Properties	#Rules	#∀-Rules	#∃-Rules
Uniprot	161	52	245	242	3
Reactome	68	55	210	209	1
ChEMBL	134	55	200	200	0
LUBM	43	25	97	89	8

Table 2. Number of atoms and answers per query; in each cell of the table we include the values corresponding to each of the 3 queries considered for each ontology

Ont.	#Atoms	#Answers for samples 1–4			
Chem.	5/7/6	123/738K/60	1K/5.4M/129	7K/26.1M/241	21K/90.2M/339
React.	2/6/6	338K/24/64K	1M/90/123K	2M/319/170K	2.5M/1K/185K
Unip.	2/5/7	9K/5K/15K	20K/10K/32K	30K/16K/50K	39K/23K/68K
LUBM	3/3/2	647K/738K/507K	1.3M/1.5M/1M	2M/2.2M/1.5M	2.6M/2.9M/2M

cardinality strictly larger than (1) because VLog only supports reasoning over equality via axiomatisation and this might be too slow in practice. All axioms removed in steps (1) and (2) were simply commented in the ontology files and can be consulted if desired.

Then, we transform the TBoxes into equivalent rules using the transformation implemented by VLog (see Sect. 2). We include statistics for the ontologies and translated rule sets in Table 1. Finally, we use the acyclicity checks implemented in VLog to determine that the chase does terminate for the translated rule sets (see Sect. 2). Since this is the case, our implementation of the chase can be effectively used to solve CQ answering over the output rules sets (and thus, over the considered ontologies).

For each ontology, we consider three example queries and four ABox samples with an increasing number of facts. The queries are manually designed for each ontology to retrieve significant numbers of answers. Table 2 reports the number of atoms composing each query, and the number of query answers obtained for each of the four samples of facts. Figure 4 reports the execution times of the queries on each of the ontologies. The reported times include performing materialisation and returning all query answers to Java by the C++ reasoner. We exclude time needed to parse the CSV-files that contained the facts.

We find that VLog can efficiently compute answers in all cases, even if the ABox is relatively large. We consider all query answering times to be practically feasible, since they are well within the usual timeouts of, e.g., SPARQL endpoints. When interpreting the times, it must be taken into account that ontological reasoning has a major performance impact in this case as compared to plain query answering on SPARQL.

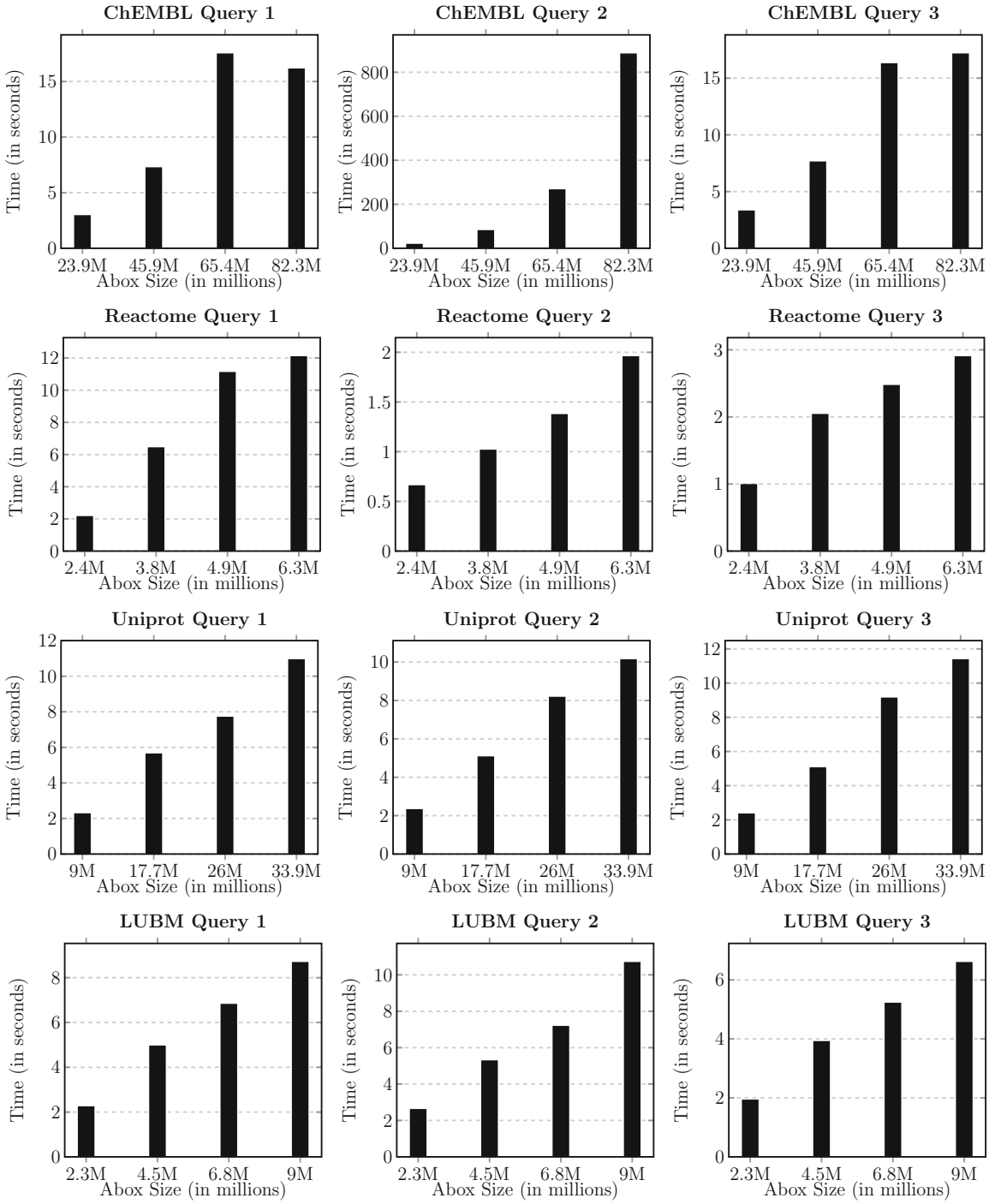

Fig. 4. OWL query answering evaluation results; each table includes results for each of the four different samples considered, one ontology (ChEMBL, Reactome, Uniprot, and LUBM) and one query

We observe an interesting result for answering Query 1 on ChEMBL ontology: the smaller third sample took more than one second longer than the larger fourth sample. This may be due to the fact that VLog uses some heuristics to decide between several join algorithms at runtime, based on cardinalities.

The introduction of an additional Java layer did not seem to hamper performance, and indeed the times needed to convert ontologies to rules and to transfer results back to Java were negligible. Our experiments demonstrate that the use of VLog for CQ answering over data-heavy DL ontologies is feasible.

5 Related Work

To better compare VLog against other state-of-the-art, recursive rule engines, we separate these systems into two broad categories.

1. *RDBMS-based Systems* [7,14,30], which use existing database technologies to implement the chase. This category includes systems such as Demo [30], Llunatic [14], and PDQ [7] which run on top of PostgreSQL.
2. *In-memory Systems*, which rely on the use of RAM memory to compute the chase. This category includes systems such as Graal [4], DLV 2 [2], RDFox [29], and Vadalog [5] as well as our own tool, VLog.

This classification is not perfect. Systems in the second category, such as Graal or VLog, rely on database technologies to store and query input data. Furthermore, systems such as Bash Datalog [33] cannot be categorised as either.

Even if we restrict our focus to "in-memory" tools, it is difficult to compare VLog with the other systems in (2) as these support very distinct features. For instance, DLV 2 supports disjunctions in the head of the rules, Graal can recognise specific logic fragments and use this knowledge to apply specific optimised algorithms, Vadalog can reason over a non-acyclic fragment of existential rules [16], and RDFox is optimised for parallel [27] and even distributed [32] computation. Nevertheless, unlike the other systems, VLog can ingest data from a great variety of heterogenous formats. Furthermore, VLog implements the *Datalog-first restricted chase* [38], a variant of the chase that terminates more often than Skolem and restricted, and has been conjectured to be more computationally powerful [23]. Table 3 compares different features of these Datalog reasoners, based on publications and software released as of June 2019.

In recent work [38], we conduct an extensive evaluation to compare the performance of our tool in comparison with that of RDFox, repeating experiments from [6] and adding several more based on 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. Note that RDFox greatly outperforms both Graal and DLV [24] in the evaluation presented in [6] (note that DLV is different from DLV 2, which was not considered in [6]). We re-ran our earlier experiments with the current version of VLog, but the results were largely similar (with an average speed-up of 12%), so we do not restate them here.

6 Accessing VLog

VLog is written in C++11, has only very few external dependencies, and compiles with GNU GCC, CLang, and Microsoft's Visual C++ compilers. Binaries

Table 3. Features of in-memory Datalog reasoners: *Inputs* (1: RDBMS, 2: RDF files, 3: CSV files, 4: SPARQL endpoints); *Neg.* (negation semantics); *Eq.* (optimised equality reasoning); *Incr.* (incremental updates); *Mult.* (integrating data from multiple sources)

Engine	Inputs	Neg.	Eq.	Incr.	Mult.	Free license
DLV 2 [2,24]	1	+ (ASP)	+	+	–	–
Graal [4]	1, 2	–	–	–	+	+ (CeCILL)
RDFox [29]	2	–	+	+	–	–
Vadalog [5,17]	1, 2, 3	–	+	–	+	–
VLog	1, 2, 3, 4	+ (strat.)	–	–	+	+ (Apache2)

are available for Linux, MacOS, and Windows. The codebase uses CMake in order to simplify and automate the compilation and in most of the tested scenarios this process reduced to the execution of two commands.

VLog and VLog4j are available on *github* (see Footnote 1). Both projects are free and open-source. They have been released under Apache License 2.0, are available via Maven under artifact id *org.semanticweb.vlog4j*, and their development is monitored by Travis CI to ensure compliance with unit tests.

Furthermore, VLog is also available as Docker image in the Docker repository `karmaresearch/vlog`. Docker images are automatically built when the master branch is updated to ensure the availability of the latest version. The Docker images are useful because they allow the user to either launch the Web interface or use the command line without any prior manual installation. Moreover, they enable a easy deployment of VLog in a cloud environment.

7 Conclusion

We presented VLog, an efficient rule engine that is suitable for scenarios that require expressive reasoning on large KGs. Moreover, the Java API VLog4j allows its usage in complex pipelines, while the ability of the system to interface with existing data sources opens the door to the application of reasoning to novel scenarios (e.g., federated reasoning). VLog and VLog4j support a range of semantic web technologies, including RDF, OWL, and SPARQL, and integrate with other relevant software components, such as Graal. To facilitate the adoption, all the code and documentation is freely available and the development process is open to contributors in the spirit of collaborative open source projects.

The project is under active development and we are considering several new features for implementation. Important directions for extensions of the expressive power will support equality and incremental reasoning, and introduce support for datatypes, especially numbers. We are also considering new optimisations that take advantage of the high level of control that we have on the execution order of rules in VLog. While these are definitely enough to keep us busy, we are also looking forward to inputs from users in the semantic web community, who might encounter completely unforeseen needs in their rule-based applications.

Acknowledgements. This work is partly supported by DFG in projects 389792660 (TRR 248, Center for Perspicuous Systems) and KR 4381/1-1 (DIAMOND).

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley, Boston (1994)
2. Alviano, M., et al.: The ASP system DLV2. In: Balduccini, M., Janhunen, T. (eds.) LPNMR 2017. LNCS (LNAI), vol. 10377, pp. 215–221. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61660-5_19
3. Baget, J.F., Leclère, M., Mugnier, M.L., Salvat, E.: On rules with existential variables: walking the decidability line. J. Artif. Intell. Res. **175**, 1620–1654 (2011)
4. Baget, J.-F., Leclère, M., Mugnier, M.-L., Rocher, S., Sipieter, C.: Graal: a toolkit for query answering with existential rules. In: Bassiliades, N., Gottlob, G., Sadri, F., Paschke, A., Roman, D. (eds.) RuleML 2015. LNCS, vol. 9202, pp. 328–344. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21542-6_21
5. Bellomarini, L., Sallinger, E., Gottlob, G.: The vadalog system: datalog-based reasoning for knowledge graphs. J. PVLDB **11**(9), 975–987 (2018)
6. Benedikt, M., et al.: Benchmarking the chase. In: Proceedings of the 36th Symposium on Principles of Database Systems (PODS) (2017)
7. Benedikt, M., Leblay, J., Tsamoura, E.: PDQ: proof-driven query answering over web-based data. J. PVLDB **7**, 1553–1556 (2014)
8. Carral, D., Dragoste, I., Krötzsch, M.: Restricted chase (non)termination for existential rules with disjunctions. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI) (2017)
9. Carral, D., Dragoste, I., Krötzsch, M.: The combined approach to query answering in Horn-$\mathcal{ALCHOIQ}$. In: Proceedings of the 16th International Conference on Principles of Knowledge Representation and Reasoning (KR) (2018)
10. Carral, D., Feier, C., Hitzler, P.: A practical acyclicity notion for query answering over Horn-$\mathcal{SRIQ}$ ontologies. In: Groth, P., et al. (eds.) ISWC 2016. LNCS, vol. 9981, pp. 70–85. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46523-4_5
11. Carral, D., González, L., Koopmann, P.: From Horn-$\mathcal{SRIQ}$ to datalog: a data-independent transformation that preserves assertion entailment. In: Proceedings of the 33rd Conference on Artificial Intelligence (AAAI) (2019)
12. Cuenca Grau, B., et al.: Acyclicity notions for existential rules and their application to query answering in ontologies. J. Artif. Intell. Res. **47**, 741–808 (2013)
13. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data exchange: semantics and query answering. J. Theor. Comput. Sci. **336**, 89–124 (2005)
14. Geerts, F., Mecca, G., Papotti, P., Santoro, D.: That's all folks! LLUNATIC goes open source. J. PVLDB **7**(13), 1565–1568 (2014)
15. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z.: HermiT: an OWL 2 reasoner. J. Autom. Reason. **53**(3), 245–269 (2014)
16. Gottlob, G., Pieris, A.: Beyond SPARQL under OWL 2 QL entailment regime: rules to the rescue. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI) (2015)
17. Gottlob, G., Pieris, A., Sallinger, E.: Vadalog: recent advances and applications. In: Calimeri, F., Leone, N., Manna, M. (eds.) JELIA 2019. LNCS (LNAI), vol. 11468, pp. 21–37. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-19570-0_2

18. Guo, Y., Pan, Z., Heflin, J.: LUBM: a benchmark for OWL knowledge base systems. J. Web Semant. **3**, 158–182 (2005)
19. Horridge, M., Bechhofer, S.: The OWL API: a Java API for OWL ontologies. J. Semant. Web **2**, 11–21 (2011)
20. Kazakov, Y.: Consequence-driven reasoning for Horn-$\mathcal{SHIQ}$ ontologies. In: Proceedings of the 21st International Joint Conferences on Artificial Intelligence (IJCAI) (2009)
21. Krötzsch, M.: Efficient rule-based inferencing for OWL EL. In: Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI) (2011)
22. Krötzsch, M., Maier, F., Krisnadhi, A.A., Hitzler, P.: A better uncle for OWL: nominal schemas for integrating rules and ontologies. In: Proceedings of the 20th International Conference on World Wide Web (WWW) (2011)
23. Krötzsch, M., Marx, M., Rudolph, S.: The power of the terminating chase (invited talk). In: Proceedings of the 22nd International Conference on Database Theory (ICDT) (2019)
24. Leone, N., et al.: The DLV system for knowledge representation and reasoning. J. ACM Trans. Comput. Log. **7**, 499–562 (2006)
25. Leskovec, J., Faloutsos, C.: Sampling from large graphs. In: Proceedings of the 12th International Conference on Knowledge Discovery and Data Mining (ACM SIGKDD) (2006)
26. Malyshev, S., Krötzsch, M., González, L., Gonsior, J., Bielefeldt, A.: Getting the most out of Wikidata: semantic technology usage in Wikipedia's knowledge graph. In: Vrandečić, D., et al. (eds.) ISWC 2018. LNCS, vol. 11137, pp. 376–394. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00668-6_23
27. Motik, B., Nenov, Y., Piro, R., Horrocks, I., Olteanu, D.: Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In: Proceedings of the 28th Conference on Artificial Intelligence (AAAI) (2014)
28. Motik, B., Sattler, U., Studer, R.: Query answering for OWL DL with rules. J. Web Semant. **3**, 41–60 (2005)
29. Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: RDFox: a highly-scalable RDF store. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 3–20. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25010-6_1
30. Pichler, R., Savenkov, V.: Demo: data exchange modeling tool. J. PVLDB **2**, 1606–1609 (2009)
31. Piro, R., et al.: Semantic technologies for data analysis in health care. In: Groth, P., et al. (eds.) ISWC 2016. LNCS, vol. 9982, pp. 400–417. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46547-0_34
32. Potter, A., Motik, B., Nenov, Y., Horrocks, I.: Dynamic data exchange in distributed RDF stores. J. IEEE Trans. Knowl. Data Eng. **30**, 2312–2325 (2018)
33. Rebele, T., Tanon, T.P., Suchanek, F.: Bash datalog: answering datalog queries with unix shell commands. In: Vrandečić, D., et al. (eds.) ISWC 2018. LNCS, vol. 11136, pp. 566–582. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00671-6_33
34. Seo, J., Guo, S., Lam, M.S.: SociaLite: an efficient graph query language based on datalog. J. IEEE Trans. Knowl. Data Eng. **27**, 1824–1837 (2015)
35. Siow, E., Tiropanis, T., Hall, W.: SPARQL-to-SQL on internet of things databases and streams. In: Groth, P., et al. (eds.) ISWC 2016. LNCS, vol. 9981, pp. 515–531. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46523-4_31
36. Steigmiller, A., Liebig, T., Glimm, B.: Konclude: system description. J. Web Semant. **27**, 78–85 (2014)

37. Urbani, J., Jacobs, C., Krötzsch, M.: Column-oriented datalog materialization for large knowledge graphs. In: Proceedings of the 30th Conference on Artificial Intelligence (AAAI) (2016)
38. Urbani, J., Krötzsch, M., Jacobs, C., Dragoste, I., Carral, D.: Efficient model construction for horn logic with VLog: system description. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 680–688. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_44
39. Vrandečić, D., Krötzsch, M.: Wikidata: a free collaborative knowledge base. J. Commun. ACM **57**, 78–85 (2014)
40. Zhou, Y., Cuenca Grau, B., Nenov, Y., Kaminski, M., Horrocks, I.: PAGOdA: pay-as-you-go ontology query answering using a datalog reasoner. J. Artif. Intell. Res. **54**, 309–367 (2015)