RUBEN: A Rule Engine Benchmarking Framework

Kevin Angele[1,3], Jürgen Angele[2], Umutcan Şimşek[1] and Dieter Fensel[1]

[1]Semantic Technology Institute, University of Innsbruck, Technikerstrasse 21a, 6020 Innsbruck, Austria
[2]adesso, Competence Center Artificial Intelligence
[3]Onlim GmbH, Weintraubengasse 22, 1020 Vienna, Austria

Abstract

Knowledge graphs have become an essential technology for powering intelligent applications. Enriching the knowledge within knowledge graphs based on use case-specific requirements can be achieved using inference rules. Applying rules on knowledge graphs requires performant and scalable rule engines. Analyzing rule engines based on test cases covering various characteristics is crucial for identifying the optimal rule engine for a given use case. To this end, we present *RUBEN: A Rule Engine Benchmarking Framework* providing interfaces to benchmark rule engines based on given test cases. Besides a description of RUBEN's interfaces, we present a selection of test cases adopted from the *OpenRuleBench*, and an evaluation of four rule engines. In the future, we aim to benchmark existing rule engines regularly and encourage the community to propose new test cases and include other rule engines.

Keywords

RUBEN, Rules, Rule Engines, Benchmark

1. Introduction

Knowledge graphs have become an important technology for powering intelligent applications integrating data from heterogeneous (often incomplete) sources. Parts of the missing knowledge can be inferred by using inference rules. Besides, rules can be used for data integration or information extraction. In recent years, many new rule engines [1, 2, 3] were developed targeting knowledge graphs. Performance and scalability are eminent requirements for rule engines operating on knowledge graphs to deliver fast responses for intelligent applications. Analyzing rule engines based on test cases covering various characteristics is crucial for an overview of the available engines, their performance, and scalability.

In 2009 OpenRuleBench [4] providing a set of performance benchmarks for comparing and analyzing rule engines was published. OpenRuleBench included systems relying on different technologies, including *Prolog-based, deductive databases, production rules, triple engines*, and *general knowledge bases*. The main issues when comparing various academic and commercial systems are the different syntaxes and supported features. Therefore, manually generating the rules for the various systems was necessary. At least the data for those test cases were generated programmatically. Due to the differences in the capabilities, not all test cases are applicable for all the systems. OpenRuleBench is freely available and encourages the community to contribute. Unfortunately, OpenRuleBench is not a benchmarking framework but a collection of rule sets

RuleML+RR'22: 16th International Rule Challenge and 6th Doctoral Consortium, September 26–28, 2022, Virtual
kevin.angele@sti2.at (K. Angele)

representing different test cases and corresponding datasets. Each tested system has its scripts for running the test cases. This makes running the evaluation quite cumbersome. Besides, the last assessment was conducted in 2011, and nothing seemed to have happened since then.

Therefore, we present *RUBEN: A **Ru**le Engine **Ben**chmarking Framework* providing a simple interface for including rule engines into a given set of test cases. The main aim of this framework is to provide an easy way to extend the collection of engines to be evaluated and execute the test cases without running multiple scripts. In the end, the output of all engines is combined into a single result file.

As a basis for this framework, we rely on the data provided by the OpenRuleBench [4]. The test cases were completely adopted, and the test data was adapted to the new versions of the engines. For the first version of RUBEN, we used a selection of the engines of the different categories:

- **Deductive database** - Stardog[1], VLog [3]
- **Production and reactive rule systems** - Drools[2]
- **Rule engines for triples** - Jena[3]

In the future, we plan to extend this list of engines and also encourage the community to add new engines.

In this paper, we give an overview of RUBEN's implementation (Section 2), introduce the test cases (Section 3) and present a small subset of the evaluation results (Section 4). Afterward, related work in this area is presented. Finally, Section 6 concludes the paper and gives an outlook on future work.

2. RUBEN

RUBEN[4] is a rule engine benchmarking framework written in Java, bundling the evaluation of various engines into a single framework that is easy to configure and execute. This chapter presents RUBEN's architecture and the interface to be implemented for rule engines that should be included in the evaluation.

Figure 1 presents the components RUBEN is composed of. The main component called *Ruben* loads the evaluation configuration and triggers the execution of the test cases via the *BenchmarkExecutor*. Rule engines and test cases are configurable for the evaluation.

Table 1 presents the general configuration options, namely *name* and *testDataPath*, for RUBEN. While the *name* is used as a label for the result file, the *testDataPath* specifies the location of the test data. The test data needs to include the data needed for each test case and the corresponding rules for each engine. So far, the data and rule files need to be provided in the format supported by the rule engine. In the future, we aim to represent the test cases in a rule engine-independent format. Then, the independent format needs to be interpreted by each rule engine and transformed into their format for loading the data and rules. The properties *engines* and *testCases* embed the rule engine and test case configurations.

[1] https://www.stardog.com/
[2] https://www.drools.org/
[3] https://jena.apache.org/
[4] Check https://github.com/kev-ang/RUBEN for the source code.

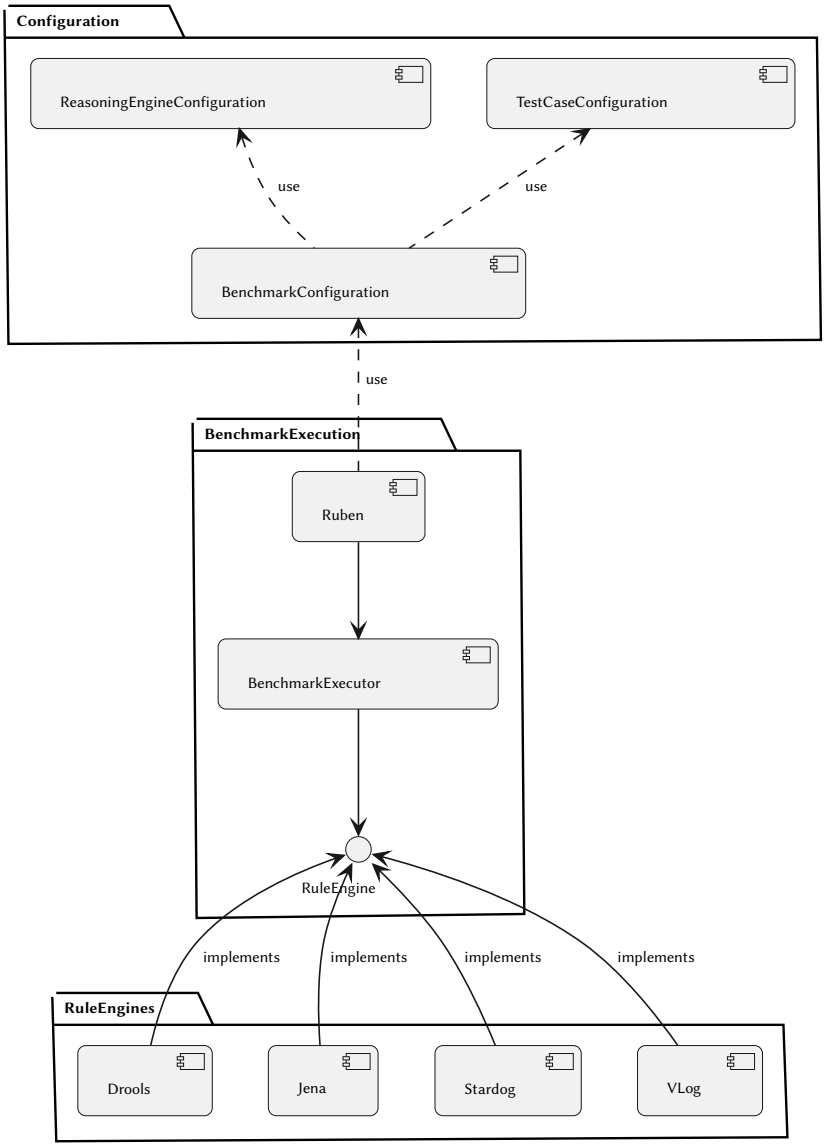

Figure 1: UML-Component diagram of RUBEN

Each rule engine can be configured by using the properties *name, classpath*, and *settings* (see Table 2). The *name* of the rule engine is used to identify the corresponding test data within the test data folder. Besides, the *classpath* refers to the class within the evaluation framework used to execute the evaluation. *settings* can be used to provide rule engine-specific settings (optional). Including the same rule engine with a different name and settings allows evaluating multiple configurations for the same rule engine.

Table 3 presents the configuration of test cases and properties that need to be specified. A test case consists of a *testCategory, testCaseIdentifier*, and *testName*. The *testCategory* is used to categorize tests. Within those categories the *testName* identifies the name of the test. Each test

Table 1
General configuration

Property	Description
name	Specify a name for the configuration.
engines	Engines to be used for the evaluation. For further details on how to configure the engines, see Table 2.
testCases	Configure the test cases to be included in the evaluation. For further details on how to configure the test cases, see Table 3.
testDataPath	Path to the folder containing the data required for the evaluation. The structure of the folder containing the test data must follow a predefined pattern. For each rule engine to be evaluated a folder is needed (the folder name must be equal to the *name* field in Table 2. Inside this folder there must be a folder for each test category (see *testCategory* in Table 3). Within the category folder a folder with the name equal to the *testName* (see Table 3) needs to be included. The files within the test folder need to be named according to the *testCaseIdentifier* values (see Table 3).

Table 2
Configuration of a rule engine

Property	Description
name	Name of the rule engine to be evaluated. This name must be used as name for the folder within the test data folder.
classpath	Refers to the implementation of the rule engine within the framework.
settings (optional)	Define additional settings for the rule engine. Those settings are provided as a map consisting of key values.

can have multiple test cases identified by the *testCaseIdentifier*.

The path for loading the test data for each test case and rule engine is composed of different information in the configuration and has the following structure:

```
{testDataPath}/{engine_name}/{testCase_testCategory}/
{testCase_testName}/{testCase_testCaseIdentifier}
```

Including a rule engine into the evaluation framework requires the implementation of the *RuleEngine* interface. Table 4 presents the methods to be implemented for a rule engine.

For an overview of the flow through the framework, we will present the steps taken to

Table 3
Configuration of a test case

Property	Description
testCategory	Category the test belongs to. The value of this property is used to distinguish test categories for each rule engine.
testCaseIdentifier	Unique identifier for the given test case. The value of this property needs to be used for naming the test case files.
testName	Each test category can have various tests. The value of this property is used to distinguish tests within a category.

Table 4
Rule Engine Interface

Method	Description
cleanUp()	This method is used to clean up the rule engine after the evaluation of a test case. Caches need to be invalidated and the data removed from the rule engine.
executeQuery(query)	Executes a single query. As a return value the number of results need to be returned.
getEngineName()	Returns the name of the engine as defined in the configuration.
prepare(testDataPath, testCase)	Based on the given test data and the current test case, the rule engine needs to load the relevant data and prepare everything for the execution of queries.
setEngineName(engineName)	Allows to set the name of the rule engine.
setSettings(settings)	Rule engine specific settings are provided via this method. The rule engine gets those settings in a map consisting of key-value pairs.
shutDown()	Stop all processes initiated by the rule engine. All temporary data that was created during the evaluation needs to be cleaned up.

load the configuration and execute the test cases. Initially, the main component (*Ruben*) loads the evaluation configuration. Afterward, the framework iterates through the provided rule engine configurations to execute all test cases for each rule engine. To evaluate the test cases, *Ruben* forwards the required information about the test data, the rule engine, and the test cases to the *BenchmarkExecutor* component. In the first step, the *BenchmarkExecutor* calls the *prepare* method of the current rule engine to load the relevant test data and rules required for the given test case. Afterward, the queries of the provided test case are executed using the *executeQuery* method, and the results in the form of the number of results are stored in a report

together with the execution time. After executing all queries for a given test case, the *cleanUp* method is called to prepare for the next test case. Finally, when all test cases are evaluated, the *BenchmarkExecutor* calls the *shutDown* method of the rule engine to stop all processes and clean up all temporary files. The results are collected, and the framework continues with the following rule engine.

3. Test Cases

For the initial version of RUBEN, we rely on the test cases provided by the OpenRuleBench [4]. OpenRuleBench's main aim is to test several tasks rule engines are known to be good at. Therefore, the authors in [4] used datasets of different sizes ranging from 50,000 to 1,000,000 facts. Alongside the generated test cases, OpenRuleBench includes four real-world benchmarks: *DBLP database*[5], *Mondial*[6], *wine ontology*[7], and *WordNet*[8]. The selected tests are representative of database and knowledge representation problems. Although all the rule engines in the original OpenRuleBench support actions, such as those in production rule systems and Prolog, they are not included due to the different paradigms of the engines.

The authors in [4] introduce several test categories with dedicated test cases:

- Large join tests
- Datalog recursion
- Default negation

The following briefly introduces the tests for the different test categories. The descriptions are taken from [4].

3.1. Large Join Tests

This test category contains two database join tests *Join1* and *Join2*, *LUBM-derived* tests, the *Mondial*, and *DBLP* tests.

Join1 is about a non-recursive tree of binary joins. Those recursions are represented by the following rules[9]:

```
a(X,Y) :- b1(X,Z), b2(Z,Y).
b1(X,Y) :- c1(X,Z), c2(Z,Y).
b2(X,Y) :- c3(X,Z), c4(Z,Y).
c1(X,Y) :- d1(X,Z), d2(Z,Y).
```

The facts for the base relations *c2, c3, c4, d1*, and *d2* are randomly generated, resulting in two datasets with 50,000 facts and 250,000 facts. Based on the derived predicates *a, b1*, and *b2* the

[5] http://www.informatik.uni-trier.de/~ley/db/
[6] http://www.dbis.informatik.uni-goettingen.de/Mondial/
[7] http://www.w3.org/TR/2004/REC-owl-guide-20040210/#WinePortal
[8] http://wordnet.princeton.edu
[9] The rules within this section are represented using the Prolog syntax.

test queries are formulated using different bindings for the variables: *free-free*, *free-bound*, and *bound-free*.

Additionally, a test called *5*Join1* was created out of five copies of the previous rule set. Here, the predicate *a(X,Y)* was renamed to *a1, ..., a5*. Then, a new rule unioning the results of the previous five rule set was introduced.

```
a(X,Y) :- a1(X,Y); a2(X,Y); a3(X,Y); a4(X,Y); a5(X,Y).
```

Join2 consists of patterns of joins borrowed from [5]. Those joins produce a large intermediate result but a small set of answers.

```
ra(A,B,C,D,E) :- p(A),p(B),p(C),p(D),p(E).
rb(A,B,C,D,E) :- p(A),p(B),p(C),p(D),p(E).
r(A,B,C,D,E) :- ra(A,B,C,D,E),rb(A,B,C,D,E).
q(A) :- r(A,_,_,_,_).
q(B) :- r(_,B,_,_,_).
q(C) :- r(_,_,C,_,_).
q(D) :- r(_,_,_,D,_).
q(E) :- r(_,_,_,_,E).
```

The query in this test case determines all facts for predicate *q* and the content of the base relation is *p(a0), ..., p(a18)*.

LUBM-derived Tests include three rule sets adapted from the original Lehigh Benchmark, LUBM [6]. LUBM is a dataset generator for a synthetical university dataset consisting of universities (the amount can be specified for the generation), courses, departments, professors, and students. For the OpenRuleBench, the authors generated two datasets: one for ten universities resulting in more than 1,000,000 tuples; one for 50 universities resulting in over 6,000,000 tuples. The original LUBM benchmark provides 14 queries, of which three (Query1, Query2, and Query9) were selected and adapted.

```
query1(X) :- takesCourse(X,graduateCourse0), graduateStudent(X).
```

Query1 joins two relations on an attribute with high selectivity (i.e., each tuple in one relation joins with a small number of tuples in the other relation).

```
query2(X,Y,Z) :- graduateStudent(X), memberOf(X,Z),
↪ undergraduateDegreeFrom(X,Y), university(Y), department(Z),
↪ subOrganizationOf_0(Z,Y).
```

Query2 joins three unary and three binary relations with high selectivity. Therefore, the final answer, even for the most extensive dataset consisting of 50 universities, contains only around 100 tuples.

```
query9(X,Y,Z) :- advisor(X,Y), teacherOf(Y,Z), takesCourse(X,Z), student(X),
↪ faculty(Y), course(Z).
```

Query9 joins three binary and three unary relations with a lower selectivity than Query1 and Query2. The answer to this query is rather large (for the large dataset, around 10,000 tuples).

Mondial is a database consisting of geographical information derived from the CIA Factbook[10]. It contains around 60,000 facts providing information about cities, provinces, and countries worldwide. Only one query is used for this test case delivering statistical information about provinces in China. The particularity of the query is the large intermediate result (1,676,942 facts) compared to the small final result (888 facts).

DBLP is a database representing publications about databases and logic programming. The database is derived from the Web-based bibliography DBLP[11]. A single relation with nearly 2,500,000 facts about more than 200,000 publications builds up the test.

```
q(Id,T,A,Y,M) :- att(Id,title,T), att(Id,year,Y), att(Id,author,A),
 ↪  att(Id,month,M).
```

The query within this test case is a 4-way join of parts of the same database relation.

3.2. Datalog Recursion

This category includes test cases using recursion, a significant feature for distinguishing rule-based systems from traditional database management systems. The authors of the OpenRuleBench intended to evaluate the performance of such queries by providing recursive tests. The tests in this category include:

- Classical transitive closure
- The same-generation siblings problem
- WordNet (application from natural language processing)
- The wine ontology in a rule-based representation

Transitive Closure of a binary relation (*par*) is the smalles transitive relation containing *par*.

```
tc(X,Y) :- par(X,Y).
tc(X,Y) :- par(X,Z), tc(Z,Y).
```

Deriving the relation *tc* causes trouble in traditional prologs if the predicates in the second rule switch places. Also, when the *par* relation represents a cyclic graph, Prolog might run into an infinite loop. Four datasets were randomly generated for this test: with cycles with 50,000 facts and 500,000 facts and without cycles (also 50,000 facts and 500,000 facts).

Same-Generation Problem tries to find all siblings in the same generation. The same generation refers to the equal distance from a common ancestor.

```
sg(X,Y) :- sib(X,Y).
sg(X,Y) :- par(X,Z), sg(Z,Z1), par(Y,Z1).
```

The base relations of *par* and *sib* were randomly generated, and again two types of datasets (cyclic and acyclic) are used for this test. The smaller datasets consist of 6000 and the larger of 24000 facts.

[10] https://www.cia.gov/the-world-factbook/
[11] http://www.informatik.uni-trier.de/~ley/db/

WordNet includes common queries from natural language processing. The queries of the provided test case seek to find all:

- **hypernyms** - words more general than the given word
- **hyponyms** - words more specific than the given word
- **meronyms** - words related by the part-of-a-whole semantic relation
- **holonyms** - words related by the composed-of relation
- **troponyms** - words more precise than the given word
- **same-synset** - a word that is in the same set of synonyms as the given word
- **glosses**
- **antonyms** - a word with the opposite meaning of the given word
- **adjective-clusters**

As the basis for the test data, WordNet Version 3.0 was used. The database consists of around 115,000 synsets containing over 150,000 words in total. Most tests deliver more than 400,000 facts, and some cases exceed 2,000,000 facts. In this paper, we only show the hypernyms test. For a complete description of the tests, check the GitHub-Repository[12]

```
hypernyms(W1,W2) :- s(S1,_,W1,_,_,_), hypernymSynsets(S1,S2),
 ↪  s(S2,_,W2,_,_,_).

hypernymSynsets(S1,S2) :- hypernym(S1,S2).

hypernymSynsets(S1,S2) :- hypernym(S1,S3), hypernymSynsets(S3,S2).
```

Wine Ontology is a rule-based representation of the OWL wine ontology, consisting of 815 rules and 654 facts. A characteristic of this test is the recursive dependency of many predicates on each other through chains of rules, resulting in large groups of predicates connected via the depends-on relationship. Those large groups are especially critical for top-down engines.

3.3. Default Negation

In this category, the tests include default negation in the body of the rules. In contrast to the OpenRuleBench, we focus only on *predicate-stratified negation*[7], so far. A modified same-generation problem is used for the predicate-stratified negation.

The modified same-generation test is as follows:

```
nonsg(X,Y) :- tc(X,Y).
nonsg(X,Y) :- tc(Y,X).
sg2(X,Y) :- sg(X,Y), not nonsg(X,Y).
```

Again, as for the original same-generation problem, the base relations of *par* and *sib* were randomly generated, with cycles in the data. This test is executed for the two data sets of 6000 and 24000 facts.

[12] https://github.com/kev-ang/RUBEN

4. Evaluation and Results

This section presents a subset of the evaluation results produced by RUBEN. We focused on the test cases and a subset of the tools evaluated in OpenRuleBench [4]. This section first presents the experimental setup, then the methodology, and finally, the evaluation results.

4.1. Experimental Setup

The evaluation framework was hosted on a server with an *Intel XEON E5-v3, 6 Cores / 12 Threads, 3.50 GHz Base Frequency, 3.80 GHz Max Turbo Frequency* processor and 256GB RAM. Out of the 256GB RAM, only 32GB were made available to the framework and, therefore, to the engines. The operating system on the machine is Debian GNU / Linux 11.

Similar to the OpenRuleBench, we evaluate engines from different categories. In total, we evaluate four engines out of three categories. The engines to be evaluated are assigned to the following categories:

- **Deductive Databases**
  - Stardog - *is implemented in Java providing extensive reasoning capabilities, including datalog evaluation.*
  - VLog - *is implemented in C++ and provides an efficient Datalog engine for large knowledge graphs supporting RDF, OWL, and SPARQL. The efficiency (memory usage and speed) is a result of the combination of a column-based layout with novel optimization methods. VLog's code is open source and freely available.*
- **Production rule system**
  - Drools - *a bottom-up engine based on the Rete algorithm [8]. The Rete algorithm combines semi-naive bottom-up computation [9] with a certain heuristic for common expression elimination [10].*
- **Rule engines for triples**
  - Apache Jena - *a Java-based framework including two rule engines (bottom-up and top-down).*

Apache Jena, Drools, and Stardog are rule engines not materializing the rules. Materializing implies calculating all the implicit facts on load time and storing them. In contrast, VLog materializes the rules, and for evaluating a query, VLog can directly access the materialized facts[13]. Therefore, the results are not directly comparable.

4.2. Methodology

The used data sets ranging from 50,000 to 1,000,000 facts are generated with the help of data generators. For running the various tests OpenRuleBench provides, we adopted the test cases. We provided a data file containing the facts, a rule file, and a file containing the queries to be evaluated. This allows providing the files in a format optimal for the rule engine. Additionally,

[13]The paper mentions a query-driven reasoning mode, which we did not find so far in the examples.

rules can be optimized for specific test cases and for each engine. Each query is evaluated two times, and the response time of the second value is taken as the result. The first execution is used to initialize all indices and fill the caches. The second query then represents the optimized query response time. The query response time includes the rule evaluation and the result counting. The results for each test case are collected and returned in a file.

Due to the limited space, we present a small subset of the evaluation results:

- **Large join tests** - include database joins (50,000 facts).
- **Datalog recursion** - include classical transitive closure and the well-known same-generation siblings problem.

The selection of those test cases was based on the ability of the given rule engines. Since Drools and Jena participated in the OpenRuleBench, we knew upfront the intersection between those. Those test case candidates were also checked for the other rule engines.

4.3. Results

Focusing on the number of test cases the engines evaluated, Drools and VLog supported the large join and datalog recursion tests. Jena and Stardog could not evaluate tests like *Join2* and *Same Generation* for negation. The first requires predicates like `p("abcd0")` and *Same Generation* requires negation which is not supported by both engines in their rule language. Tables 5, 6, and 7 present the results of those tests that were supported by the selected engines.

Special values in cells specify an unexpected behavior: *error* implies that the query evaluation did not finish as expected, *timeout* indicates that the evaluation was not finished within 15 minutes. All times within the table are given in seconds and contain only the rule evaluation and result counting. Loading time is excluded from those numbers. For VLog, we discuss the results separately, as it materializes all the rules before evaluating the queries. The materialization time for the different test cases for VLog is given in brackets in the table cell.

Table 5
Large joins, join1, no query bindings (time in seconds)

query	a(X,Y)	b1(X,Y)	b2(X,Y)
Size	50000	50000	50000
Drools	error	error	error
Jena	timeout	104.9	2.7
Stardog	exception	37.2	1.4
VLog	0.122 (356)	0.114 (356)	0.110 (356)

The results in Table 5 show that Stardog is the fastest rule-based engine for the large join test that does not materialize the rules upfront. The large join test uses the same dataset for all three queries. Therefore, the materialization time shown for VLog is relevant only once for the whole test case. As you can see, the materialization time for this test case for VLog is quite high, at nearly 6 minutes. However, the query times are then a tiny fraction of the query time of Stardog.

Table 6
Datalog recursion, same generation, no query bindings (time in seconds)

size	6000	6000	24000	24000
Cyclic data	no	yes	no	yes
Drools	error	error	error	error
Jena	53.8	61	189.1	239.1
VLog	0.027 (5)	0.030 (6)	0.030 (21)	0.030 (24)

Table 7
Datalog recursion, transitive closure, no query bindings (time in seconds)

size	50000	50000	500000	500000
Cyclic data	no	yes	no	yes
Drools	error	error	error	error
Jena	8.9	28.6	76.7	346.5
Stardog	10	27.8	66.6	262.5
VLog	0.058 (1)	0.126 (5)	0.064 (12)	0.124 (51)

Jena is the fastest engine for the *Datalog recursion same generation* test in Table 6, as Drools only throws errors. Here, VLog would be faster than Jena even if the materialization time is added to the query times. Stardog was not able to execute this test case and was therefore left out.

Stardog is the fastest rule engine for the Datalog recursion transitive closure test in Table 7 for the engines without materialization. As for the test case before, VLog would be faster than Stardog even if each query's materialization times are included.

Drools threw an exception in all test cases with an `OutOfMemoryError`. When evaluating the rules in Drools, Java objects are generated. This number of objects proliferates, causing an out-of-memory error. For the next run (next year), optimizing and adapting the rules is necessary to avoid out-of-memory errors.

5. Related Work

This section presents related work about rule engine benchmarking frameworks. Besides, in the second part of this section, we present some benchmarking datasets used by various rule engine implementations for the evaluation.

5.1. Benchmarking Frameworks

One framework already known is the OpenRuleBench [4]. OpenRuleBench analyzes the performance and scalability of various rule engines by providing a set of test cases covering different functionalities. OpenRuleBench is open source and welcomes contributions from the community. However, the last evaluation report dates 2011. Besides, running OpenRuleBench is cumbersome due to customized scripts for each supported rule engine. Integrating a new rule

engine into the framework requires writing shell scripts and adopting the given test cases to introduce the rule engine.

A more up-to-date framework for benchmarking rule-based inference engines is [11]. The authors provide a fully automated framework for benchmarking rule-based reasoning engines. Therefore, a configuration for a generation tool for generating a meta-model needs to be provided. Then, the generated meta-model is translated for the different rule models of the rule engines to be benchmarked. The translated model is then fed into the respective rule engine, and reasoning time and memory consumption is measured. The different parts of the framework are implemented in Java. However, when integrating a new rule engine into the framework, a translation class needs to be implemented to translate the meta-model into the corresponding rule model. Afterward, a shell script needs to be provided to run the test cases on the new rule engine. In the end, there is an overall shell script responsible for running the whole benchmark. This framework supports only generated test cases which is a drawback compared to OpenRuleBench, for example.

In contrast to OpenRuleBench and the other framework, RUBEN is not based on shell scripts and is fully implemented in Java. RUBEN provides interfaces implemented by rule engines that should be included in the benchmark, allowing seamless integration.

5.2. Benchmarking Datasets

Two popular rule engines published in recent years are RDFox [2] and Vlog [3]. The authors did not use one of the previously introduced frameworks for their evaluation. They used existing datasets and adapted them to measure the performance. In the following, we first present datasets of rules followed by datasets for benchmarking RDF and OWL systems.

Datasets including rules are, e.g., Manners or WaltzDB[14]. These datasets focus on evaluating pure matching algorithms (selecting rules to be evaluated) or testing a subset of available reasoning engines. However, they can not be used for other systems than those for which they are defined.

RDFox and Vlog rely more or less on the same set of test data[15]. One of those datasets is *Claros*, a cultural database catalog representing archaeologic artifacts. This dataset does not come with rules. Therefore, the authors added some manually generated rules. Another dataset is *DBpedia*, representing structured data extracted from the Wikipedia info boxes. Same as for Claros, the authors extended the dataset with manually generated rules. Both previous datasets are real-world datasets. In the following we briefly introduce two synthetical datasets called *LUBM* [6] and an extension of LUBM called *UOBM* [12]. LUBM is widely used for RDF systems with an ontology describing universities, departments, professors, and students. The data is generated by specifying the number of universities and comes with 14 well-designed test queries. Those queries cover the features of traditional reasoning systems. However, rules are not included, and the authors of Vlog and RDFox added them manually. UOBM extends LUBM by addressing the sparsity of the data. Therefore, external links between members in different university instances are added, resulting in exponential growth of the complexity for scalability testing.

[14] https://www.cs.utexas.edu/ftp/ops5-benchmark-suite/
[15] http://www.cs.ox.ac.uk/isg/tools/RDFox/2014/AAAI/

6. Conclusion and Future Work

In this paper, we presented a rule engine benchmarking framework called RUBEN, implemented in Java. RUBEN evaluates the performance and scalability of rule engines; essential to select a rule engine best fitting a given use case. In contrast to existing benchmarking frameworks like OpenRuleBench [4], RUBEN provides a simple interface a rule engine needs to implement to be included in the benchmark. Unfortunately, the first version of RUBEN requires adapting the test cases to the newly introduced rule engine. So far, RUBEN provides most of the tests initially introduced in the OpenRuleBench suite. The rule engines supported by RUBEN are a subset of the engines introduced by OpenRuleBench. To this end, we welcome the community to extend the rule engines and test cases list.

In the future, we plan to add a general description of test cases by introducing a common rule format. Then, each rule engine can include a translation method for converting the common rule format into the required format. Still, keeping the possibility to provide optimized rule sets for the cases. Besides, the list of rule engines will be extended by including RDFox [2]. Finally, (maybe) with the help of the community, we will extend the test cases to fully support and extend the list of test cases provided by OpenRuleBench. Further, it is planned to repeat the benchmark every year.

References

[1] J.-F. Baget, M. Leclère, M.-L. Mugnier, S. Rocher, C. Sipieter, Graal: A toolkit for query answering with existential rules, in: International Symposium on Rules and Rule Markup Languages for the Semantic Web, Springer, 2015, pp. 328–344.

[2] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, J. Banerjee, Rdfox: A highly-scalable rdf store, in: International Semantic Web Conference, Springer, 2015, pp. 3–20.

[3] D. Carral, I. Dragoste, L. González, C. Jacobs, M. Krötzsch, J. Urbani, Vlog: A rule engine for knowledge graphs, in: International Semantic Web Conference, Springer, 2019, pp. 19–35.

[4] S. Liang, P. Fodor, H. Wan, M. Kifer, Openrulebench: An analysis of the performance of rule engines, in: Proceedings of the 18th international conference on World wide web, 2009, pp. 601–610.

[5] B. Bishop, F. Fischer, Iris-integrated rule inference system, in: International Workshop on Advancing Reasoning on the Web: Scalability and Commonsense (ARea 2008), sn, 2008.

[6] Y. Guo, Z. Pan, J. Heflin, Lubm: A benchmark for owl knowledge base systems, Journal of Web Semantics 3 (2005) 158–182.

[7] K. R. Apt, H. A. Blair, A. Walker, Towards a theory of declarative knowledge, in: Foundations of deductive databases and logic programming, Elsevier, 1988, pp. 89–148.

[8] C. L. Forgy, Rete: A fast algorithm for the many pattern/many object pattern match problem, in: Readings in Artificial Intelligence and Databases, Elsevier, 1989, pp. 547–559.

[9] J. D. Ullman, Principles of Database and Knowledge-Base Systems – Volume I: Classical Database Systems, Computer Science Press, New York, Oxford, 1988. URL: http://dl.acm.org/citation.cfm?id=42790, 1995 wurde der 8. Nachdruck des Buches publiziert.

[10] F. Bry, N. Eisinger, T. Eiter, T. Furche, G. Gottlob, C. Ley, B. Linse, R. Pichler, F. Wei, Foundations of rule-based query answering, Reasoning Web International Summer School (2007) 1–153.

[11] S. Bobek, P. Misiak, Framework for benchmarking rule-based inference engines, in: International Conference on Artificial Intelligence and Soft Computing, Springer, 2017, pp. 399–410.

[12] L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, S. Liu, Towards a complete owl ontology benchmark, in: European Semantic Web Conference, Springer, 2006, pp. 125–139.