OpenRuleBench: An Analysis of the Performance of Rule Engines

Senlin Liang Paul Fodor Hui Wan Michael Kifer
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794, USA
{sliang,pfodor,hwan,kifer}@cs.stonybrook.edu

ABSTRACT

The Semantic Web initiative has led to an upsurge of the interest in rules as a general and powerful way of processing, combining, and analyzing semantic information. Since several of the technologies underlying rule-based systems are already quite mature, it is important to understand how such systems might perform on the Web scale. OpenRuleBench is a suite of benchmarks for analyzing the performance and scalability of different rule engines. Currently the study spans five different technologies and eleven systems, but OpenRuleBench is an open community resource, and contributions from the community are welcome. In this paper, we describe the tested systems and technologies, the methodology used in testing, and analyze the results.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Performance Evaluation; H.2.4 [**Database Management**]: Rule-Based Databases; D.1.6 [**Logic Programming**]

General Terms

Experimentation, Performance, Languages

Keywords

OpenRuleBench, Semantic Web, Rule Systems, Benchmark

1. INTRODUCTION

Rule systems have seen an upsurge of interest in the past few years, as many in the academia and industry started to regard the Semantic Web as a vast playing field for rules. In response, W3C created the Rule Interchange Format working group and tasked it with creation of a set of standards for facilitating the exchange of rules among different rule systems.[1] As this group moves towards producing several candidate recommendations, developers and researchers can greatly benefit from a better understanding of the state of the art in the rule systems technology, the overall landscape of the available systems, their performance, and scalability.

Prolog has long had benchmarks for comparing the performance of the different implementations [21, 8]. More recently, test suites have been developed for OWL [14, 19]. In contrast, rule engines do not enjoy the benefit of a carefully constructed set of performance benchmarks. Instead, comparative analysis of the different engines is largely the domain of hearsay and hastily constructed tests, often employing wrong syntax for competing systems [23, 5].

This paper is a step towards filling in the void. Over the last eight months we have been studying and experimenting with eleven different systems—academic and commercial—and created *OpenRuleBench*, a set of diverse benchmarks for comparing and analyzing the performance of these systems. The study touches upon five different technologies: *Prolog-based*, *deductive databases*, *production rules*, *triple engines*, and *general knowledge bases*. It examines how the different systems scale for a number of common problem sets and, by interpolation, how they or their successors might perform on the Web scale.

Unlike Prolog and OWL whose syntax and features have been standardized, our job was much harder. Not only the syntax of one system often does not resemble that of another, the features and the capabilities of the different systems are often completely different. While the data can usually be generated programmatically, the rule sets almost always had to be hand-crafted after careful study of the manuals. Given the differences in the capabilities of the systems, some benchmarks are applicable only to some systems.

To gain better insight into the workings of the different systems, we conducted extensive correspondence with the developers and users of each system, as evidenced by the acknowledgments. In several cases our tests prompted the developers to fix hitherto unknown bugs, and we accepted all patches that were given to us. We have made OpenRuleBench, including the various rule sets, real-world data and data generators, and scripts, freely available [20] to provide the community with a reference point for further investigation into the scalability of the various systems and to ensure the verifiability of the results. To the best of our knowledge, OpenRuleBench is the largest study of performance of rule systems, and this paper reports only a subset of the results, due to space limitations.

This paper is organized as follows. Section 2 describes the systems used in this evaluation. Section 3 describes the methodology used in the evaluation. Section 4 presents the results of the study, and Section 5 concludes the paper.

2. SYSTEMS TESTED

The rule engines included in this study are quite different in their concepts, implementation, and intended use. To introduce some order into this diversity and to give the reader a better understanding of the nature of each system, we classify the systems into five categories, based on their tech-

[1] `http://www.w3.org/2005/rules/`

WWW 2009, April 20–24, 2009, Madrid, Spain.
ACM 978-1-60558-487-4/09/04.

nologies, and attempt to mitigate the inevitable flaws of such a categorization with short descriptions of each system. We stress, however, that this paper is *not* a survey of the different rule systems and of their underlying technologies. For the latter, the reader is referred to [6].

We should also remark that the snapshots of the systems used in this study were produced at different times. For instance, in case of CYC, DLV, Ontobroker, and XSB we used unreleased betas, which were kindly provided to us. In all other cases we used the versions that were the latest releases at the time of testing.

Prolog-based systems.

XSB[2] is an open-source Prolog engine with a number of important enhancements. Being a Prolog system, it is a top-down inference engine at heart, and it provides a complete environment for building applications. However, it can also function as a deductive database through the mechanism of *tabling* or *memoing* [30], which endows XSB with certain elements of bottom-up inferencing. Memoing has the effect that knowledge base programming becomes much more declarative. In great many cases, where Prolog normally goes into an infinite loop, XSB will terminate. In particular, it will always terminate for *Datalog*, i.e., in case of function-free Horn rules. Another important effect of tabling is that it may reduce the computational complexity of query answering—sometimes from exponential to polynomial [31].

Many Prolog systems get their efficiency from the underlying highly optimized virtual machine, called *WAM* (Warren's Abstract Machine) [1]. XSB has pioneered a modified version of this abstract machine, SLG-WAM [24], which provides efficient support for memoing.

One other important feature of XSB is that it supports a declarative form of negation known as the *well-founded negation* [29] (as opposed to the non-logical form of negation known as *negation-as-failure* [9], which is standard in Prolog systems). Through a plug-in, XSB also supports another declarative semantics for negation, the *stable-model semantics* [13], but we did not include these tests in OpenRuleBench, as only one other system (DLV) among the ones we tested supports that form of negation.

XSB is implemented in the C language.

Yap[3] is a highly optimized Prolog system, very similar to XSB. Like XSB, it provides tabling and is based on SLG-WAM. Its major limitation is that, unlike XSB, it supports neither the well-founded nor stable-model negation. Yap has many innovative features that are not directly related to our tests, but we will mention one, which does: demand-driven indexing. This means that the system creates indices on-the-fly when it judges that a certain index can speed up access to large amounts of data. We shall see that this makes a big difference for some of our tests.

Like XSB, Yap is an open-source system and it is written in the C language.

Deductive databases.

DLV[4] is a bottom-up rule system, which is unique among the tested systems in that it is based on *answer-set programming* [12]. Roughly this means that rules can have disjunction in the consequent and the semantics for negation is based on stable models. The well-founded negation is also supported, however. Unlike XSB and Yap, DLV is a pure query answering system: it does not provide operators for updating the knowledge base, and one cannot build a complete application using DLV alone. Instead, applications are to be built using a procedural language, such as Java, where DLV plays the role of a knowledge base component.

DLV employs important optimizations, such as Magic Sets [4], which endow it with certain benefits of a top-down system. It also employs a heuristic that reorders rule premises in order to achieve better performance. DLV is implemented in C++; it is free in binary form for non-commercial use, but is not distributed with an open-source license.

IRIS[5] is an open-source, bottom-up rule inference engine. It is based on the well-founded semantics for negation and provides the Magic Sets optimization. Like DLV, IRIS cannot be used as a standalone system, but only as a knowledge base component of an application written in a host language. Since IRIS is written in Java, IRIS-based applications are intended to be written in Java as well.

IRIS might be a system to watch, but it has not reached the level of maturity that would allow a meaningful comparison with other systems. Therefore, we omit further discussion of IRIS in this paper.

Ontobroker[6] is a bottom-up rule engine, which is similar in many respects to DLV and IRIS. Like those systems, it supports the well-founded negation, Magic Sets, and is intended to be used as a knowledge base component within a host language. Like IRIS, Ontobroker is written in Java and is integrated with it. In addition, Ontobroker has a number of features that differentiate it from other systems.

First, in addition to the Magic Sets optimization, Ontobroker supports several others, including *filtering* [17] and cost-based optimization akin to database management systems [15, 11]. To evaluate a query, it first builds a *cost model* and then decides which optimizations to use, what order to choose for joining the predicates in rule premises, and which methods to use for each individual join (e.g., nested loops, sort-merge). When necessary, Ontobroker builds the appropriate indices to speed up query evaluation, and, when multiple CPUs are available, it parallelizes the computation. Finally, unlike the other tested systems, Ontobroker's knowledge base language is based on F-logic [16].

Ontobroker has a commercial, non-open-source license, but binary copies for research purposes can be obtained from Ontoprise, Inc.

Like XSB and Yap, IRIS and Ontoprise support the use of function symbols, which makes them Turing-complete. DLV provides only a limited support for function symbols, but it is being constantly enhanced [7].

Rule engines for triples.

Jena[7] is a Java-based framework for building Semantic Web applications. Among other things, it includes two rule engines: a bottom-up engine and a top-down one. These inference modes can be used in tandem, albeit in a limited way. Since the top-down engine fared much better in our tests, we do not discuss the bottom-up engine any further.

[2] `http://xsb.sourceforge.net/`
[3] `http://www.dcc.fc.up.pt/~vsc/Yap/`
[4] `http://www.dbai.tuwien.ac.at/proj/dlv/`
[5] `http://sourceforge.net/projects/iris-reasoner`
[6] `http://www.ontoprise.de/de/en/home/products.html`
[7] `http://jena.sourceforge.net/`

The top-down engine roughly works the same way as in XSB and Yap. In particular, to ensure termination, Jena employs tabling. However, unlike XSB and Yap, Jena is an interpreted system; it does not have an underlying virtual machine optimized specifically for rule-based query processing.[8] Jena is distributed under an open-source license.

SwiftOWLIM and **BigOWLIM**[9] are bottom-up rule engines for RDF [18] and a certain subset of OWL, called OWL-Horst [25]. SwiftOWLIM supports a limited repertoire of query optimizations. For instance, caching of intermediate results can be selectively turned off, which is analogous to selective tabling of predicates in XSB and Yap.

BigOWLIM is a commercial brother of SwiftOWLIM. The main difference between the two engines is that the former has much better performance when it is used to process disk-bound data. BigOWLIM can also perform database-style cost-based optimization similarly to Ontobroker (but Ontobroker is a much more general and sophisticated system). We did not test BigOWLIM because OpenRuleBench focuses on main memory performance.

Apart from the above differences, Jena and SwiftOWLIM have much in common. They are both implemented in Java, are intended to be used as knowledge base components of applications written in Java, and they are designed to mainly process RDF triples. Since both systems focus on ternary relations,[10] they can afford full indexing on each of the arguments and thus improve the performance of queries. Jena and OWLIM also share a number of limitations. Being triple-based, they support only unary and ternary predicates. Neither system supports the well-founded negation or even its simpler forms (like predicate-stratified negation [2]), and the use of function symbols is not permitted. As a result, several of our tests (e.g., those that use predicates of higher arities) could not be used with these systems. These tests could, in principle, be encoded using triples and used with Jena and SwiftOWLIM. However, we decided that such a comparison would be questionable: encoding n-ary relations using triples saddles triple-based systems with an overhead that other systems do not have to deal with.

Both Jena and SwiftOWLIM are distributed under open-source licenses, while BigOWLIM has a commercial license.

Production and reactive rule systems.

Drools[11] is a *production rule* system—a bottom-up engine where rules have actions in the consequent. When the actions are insertions of new facts, this is analogous to other bottom-up systems for Datalog. However, actions can also be deletions of facts or even calls to arbitrary Java methods. In the last two cases, production rules have no logical semantics and cannot be directly compared to other systems, which we tested. Production rules can also have negation in the rule premises. If used carefully, such negation can be given logical semantics based on *stratification*—a restricted form of negation that lies in the intersection of the well-founded and stable-model semantics.

One other feature that is common to all production rule systems is that they are all based on (a version of) the Rete algorithm [10]—a combination of the semi-naive bottom-up computation [27] commonly used in Datalog systems with a certain heuristic for common expression elimination [6].

Jess[12] is also a production rule system. Like Drools, it is mainly a bottom-up system, based on an enhanced version of the Rete algorithm, but it also has certain top-down inference capabilities.

To improve performance, both Drools and Jess use adaptive indexing techniques, which profitably select the arguments on which to build indices (full indexing can be too expensive). For instance, join arguments are always indexed and so are the intermediate results derived by the rules.

Prova[13] Prova is a top-down *reactive* rule system, which has special support for event-based processing and actions. Operationally, Prova works similarly to Prolog, but it is an interpreted system and is not based on a WAM. However, unlike XSB and Yap, it does not have support for tabling and it uses the non-logical version of negation-as-failure.

Among these systems, only Prova supports function symbols. Although Prova appears to be quite different from Drools and Jess, these three systems are classified together because their technologies revolve around various kinds of actions. Incidentally, they are also all Java-based and are intended as rule components inside Java applications.[14] Drools and Prova are distributed under open-source licenses. Jess is free for research, but is not open-source.

We do not include Prova in subsequent comparisons, as it appears to have been designed for event processing, not data processing. As a result, it was not able to execute any of the data-intensive benchmarks included in OpenRuleBench.

Knowledge-base systems.

CYC[15] is an inference engine equiped with a vast body of domain knowledge and heuristics for solving many complex problems in classical and common-sense reasoning. Since CYC is capable of much of the rule-based inferencing of the kinds we benchmarked, we included it in our study. However, CYC's reasoning capabilities go well beyond rules, and the very optimizations that enable such reasoning put it at a disadvantage when pure rule-based reasoning is involved. As a consequence, CYC could not run most of our tests, and we decided to omit this system from the comparative analysis reported here.

3. METHODOLOGY

All tests were performed on a dual core 3GHz Dell Optiplex 755 with 4 gigabytes of main memory (of which only 3G are visible to applications). The machine was running Ubuntu 7.10, a distribution of Linux with kernel 2.6.22.

The test suite itself consists of different, carefully selected rule sets, which are intended to test several well-known tasks that rule systems are known to be good at. Since the tested systems have very different syntax, semantics, and capabilities, the rules had to be manually (and often non-trivially) adapted for each system. On the other hand, data sets were

[8] Jena programs are executed in JVM, but this is a general purpose virtual machine, which is not optimized for rules.

[9] `http://www.ontotext.com/owlim/index.html`

[10] Or binary relations—depending on the point of view.

[11] `http://www.jboss.org/drools/`

[12] `http://herzberg.ca.sandia.gov/`

[13] `http://www.prova.ws/`

[14] Although Prova stands for Prolog+Java, it does not belong in the same category as Prolog-based systems, since it is vastly different from them operationally, semantically, and architecturally.

[15] `http://www.cyc.com/`

created with the help of data generators. Typically OpenRuleBench uses data sets with sizes 50000, 100000, 250000, 500000, and 1000000 facts, but here we report the results for only some of these data sets. The suite also includes "real-world" benchmarks: the wine ontology, Mondial, WordNet, and the DBLP database. Here the data sets are fixed and range from a few hundreds in the wine ontology to several millions in the DBLP database. Finally, the test suite includes scripts for running the different systems in batch, measuring their run time, collecting the results, and creating LaTeX tables such as those included in Section 4.

In this paper, we describe and analyze the results obtained from several different categories of tests: *large join tests*, *Datalog recursion*, and *default negation*.[16] Every one of the systems has met a foe it could not beat—tests that it failed because of a timed-out, or due to a crash. Three systems, Yap, XSB, and Ontobroker, stood out both as the most feature-full and also as (by far) the best-performing in the entire bench. However, variation in the results among these three for different tests was significant.

A few words about our methodology. As mentioned earlier, the tested systems vary greatly in their capabilities, degrees of sophistication, and in what they do with their input rule sets before cranking an inference engine. As our goal was to compare *technologies*, not to run a beauty pageant among systems, we tried to look into the potential of the tested systems and often applied manual optimizations when algorithms for such optimizations were known. Here are some of the considerations that were factored into the test.

Indexing. A well-chosen set of indices may mean the difference between many hours and a few seconds.

Some systems, e.g., DLV, Ontobroker, Yap, and Jess analyze the rule sets and data and create useful indices on-the-fly. Others, like Jena and OWLIM, index on every argument. Yet others, like XSB, offer a repertoire of indexing techniques, which must be specified manually. However, XSB's default indexing is rather simple-minded, and a naive user might see wide variations in performance depending on the characteristics of the data. Yet other systems always index on the first argument and do not provide any other options. In some cases we had to significantly rewrite the tests when automatic indexing failed to do adequate job (for instance, the LUBM-derived tests in case of Jess).

To reduce variability, we decided to use the most advantageous indices for each system. For instance, when we knew that XSB would perform best with certain indices present then we would declare such indices manually. For Yap and some others, on the other hand, we had to do nothing.

Optimization knobs. Some systems provide various options for optimization. For instance, in Ontobroker one can enable or disable the parallel engine or choose specialized inference engines. In XSB and Yap, predicates can be declared as tabled in order to improve performance or ensure termination, and there are different tabling modes that one can choose (XSB has four, for example). In SwiftOWLIM, tabling can be turned off in some cases, although this did not matter in our tests. Another example: IRIS, Ontobroker, and DLV take advantage of the Magic Sets optimization. In all cases, we tried to use the optimizations that we were aware of, if we found them beneficial. In several cases the suggestions came from the system maintainers themselves.

[16]The full report [20] includes additional tests.

Query-based systems versus production rules. Most of the tested systems are capable of taking queries and avoiding the computation of intermediate results that are irrelevant to the given query. However, production rule systems, such as Jess and Drools, are designed to compute all possible derivations and do not take queries into account. In such cases, we changed the rule sets to remove the irrelevant rules and thus to compensate for the lack of optimization.

Cost-based optimizers versus none. We have already mentioned that Ontobroker and BigOWLIM can perform cost-based optimization which, in particular, may reorder predicates in the rule premises. Sometimes such reordering may drastically reduce the computation time and, in some cases, even reduce the computational complexity of the query. As with all the previous performance factors, we tried to find the most beneficial order of the predicates for each system and rewrite the rules manually.

Loading versus inference. Our main interest was in measuring the time to do *inference* rather than *loading* of the data sets. Most systems had clearly separated loading and inference phases, so measuring the speed of inference alone was easy to perform. However, some systems (for example, DLV, IRIS, and OWLIM) combine loading and much of the inference in one step. This made comparison with other systems harder, since it forced us to run additional tests to estimate the time of loading alone. This made our time measurements for smaller tests less precise than we liked.

Choice of tests. Our tests are representative of database and knowledge representation problems. They do not include actions, such as those found in production rule systems and Prolog. Although half of the tested systems supports actions, they use different paradigms. There is no obvious way to translate between these paradigms in a way that would be meaningful for performance tests.

We will now describe the tests in each category separately.

Large join tests.

These include database joins (Join1, Join2), LUBM-derived tests, the Mondial, and the DBLP tests.

Join1 has a form of a non-recursive tree of binary joins, which is expressed using these inference rules:[17]

```
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 base relations, `c2`, `c3`, `c4`, `d1`, and `d2`, were randomly generated. We used two data sets: one with 50000 facts and the other with 250000 facts. The queries are based on the derived predicates, `a`, `b1`, `b2`, with different bindings for the variables: free-free, free-bound, and bound-free.

The test **5*Join1** was created by making five copies of the above rule set with the predicate `a(X,Y)` renamed to `a1`, ..., `a5`, and then unioning the results using the rule

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

Join2 is another handcrafted pattern of joins, borrowed from [5]. It produces a large intermediate result, but only a small set of answers.

[17]All our examples use Prolog syntax.

```
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 content of the base relation is `p(a0)`, ..., `p(a18)`, and the query seeks all the facts for the predicate `q`.

LUBM-derived tests include three rule sets (Query1, Query2, and Query9) that are adapted from the original Lehigh benchmark, LUBM [14]. It is a university database where the number of universities, departments, and students can vary. We tested two data sets: one for 10 universities, which has over 1000000 tuples and one for 50 universities, which includes over 6000000 tuples.

Query1 retrieves students who take a particular graduate course. It is a join of two relations on an attribute that has *high selectivity* (i.e., each tuple in one relation joins with only a small number of tuples in the other relation). As a result, although the input database is large, the query answer is very small. *Query2* seeks those triples <s, d, u> where s is a graduate student in department d, s has an undergraduate degree from university u, and d is a department in u. This query joins three unary and three binary relations, but, again, the selectivity of the join attributes is high, so the final answer is small (around 100 tuples for the 50-university data set). *Query9* seeks those triples of the form <s, f, c> where f is a faculty advisor of student s, and s takes course c from f. This query joins three binary and three unary relations, but the join attributes have lower selectivity than in Queries 1 and 2. So, the query answer is relatively large (more than 10000 for the 50-university data set).

Due to space limitations, we do not show the helper predicates, which were introduced by the translation from the corresponding LUBM tests. The interested reader can find the details in [20].

```
query1(X) :- takesCourse(X,graduateCourse0),
             graduateStudent(X).
query2(X,Y,Z) :- graduateStudent(X), memberOf(X,Z),
                 undergraduateDegreeFrom(X,Y),
                 university(Y), department(Z),
                 subOrganizationOf_0(Z,Y).
query9(X,Y,Z) :- advisor(X,Y), teacherOf(Y,Z),
                 takesCourse(X,Z), student(X),
                 faculty(Y), course(Z).
```

Mondial[18] is a geographical database derived from the CIA Factbook.[19] The base relations contain around 60000 facts, and each fact provides information about cities, provinces, and countries around the world. The only query in this test seeks certain statistical information about provinces in China. The query is interesting because it produces an intermediate result (1676942 facts) that is orders of magnitude larger than the final results (888 facts). So, the query offers opportunities for optimization.

[18] `http://www.dbis.informatik.uni-goettingen.de/Mondial/`

[19] `https://www.cia.gov/library/publications/the-world-factbook/`

DBLP is a database of publications in the fields of databases and logic programming derived from the well-known Web-based bibliography under the same name.[20] The test contains a single relation that has close to 2500000 facts about more than 200000 publications. The only query in this test is a 4-way join of parts of the same database relation:

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

Datalog recursion.

Recursion is perhaps the most important feature that distinguishes rule-based systems from traditional database management systems (SQL:1999 extensions notwithstanding). Development of efficient techniques for computing recursive queries was a large subarea within the database and logic programming research in the 80's and 90's. The recursive tests in OpenRuleBench are intended to evaluate the performance of such queries.

The tests include the classical transitive closure, the well-known same-generation siblings problem, applications from natural language processing based on WordNet,[21] and a rule representation of the well-known wine ontology.[22]

The **transitive closure** of a binary relation, `par`, is the smallest transitive relation that contains `par`. It is written in Prolog notation as follows:

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

Traditional prologs have trouble computing the derived relation `tc`, if the predicates in the second rule switch places. However, even with the favorable order of the predicates Prolog might go into an infinite loop if the relation `par` represents a cyclic graph.

The tuples for the base relation `par` were randomly generated, and two types of data sets were generated: one with cycles and the other without. In each case we ran tests for two data sizes 50000 and 500000.

The **same-generation** problem seeks to find all siblings in the same generation, i.e., siblings that are equally remoted from a common ancestor. Of course, the relation `par` can represent an arbitrary graph, so the human analogy should not be taken too far.

```
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. As before, we considered both cyclic and acyclic data, and for each type two data sizes were used: 6000 and 24000.

The **WordNet tests** include common queries from natural language processing based on WordNet, a semantic lexicon for the English language. These queries seek to find all *hypernyms* (words more general than a given word), *hyponyms* (words more specific than a given word), *meronyms* (words related by the part-of-a-whole semantic relation), *holonyms* (words related by the composed-of relation), *troponyms*, *same-synset*, *glosses*, *antonyms*, and *adjective-clusters*. The base facts were extracted from WordNet Version 3.0 and

[20] `http://www.informatik.uni-trier.de/~ley/db/`

[21] `http://wordnet.princeton.edu`

[22] `http://www.w3.org/TR/2004/REC-owl-guide-20040210/#WinePortal`

converted to the specific syntaxes of our test systems. The database consists of about 115000 *synsets* (i.e., groups of semantically equivalent words) containing over 150000 words in total. In most tests, the number of solutions is over 400000, and in some cases exceed 2000000. Due to space limitation, we show only the hypernyms test here—see [20] for a complete description of the tests. In these tests, choosing a good index is of paramount importance, and we tried to adjust the rules and set up indices to benefit each individual system.

```
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).
```

The **wine ontology** is a rule-based representation of the well-known OWL wine ontology.[23] It has 815 rules and 654 facts. Since a ruleset this large cannot be included here, we refer the reader to the sources on the OpenRuleBench Web site [20]. Perhaps the most important feature of this test is that many predicates are recursively dependent on each other through chains of rules. This creates large cliques of predicates with respect to the depends-on relationship. Such cliques present special problems for top-down engines, and we had to use special optimizations in order for the best of our systems, Yap and XSB, to pass this test.

Default negation.

These tests include rules with default negation, `not`, in the body. Three different patterns of recursion through negation are considered: predicate-stratified negation [2], locally stratified [22], and negation under the well-founded semantics [29] for rule sets that are not even locally stratified. The predicate-stratified negation is represented by a *modified same generation* problem. Locally stratified negation is represented by the well-known *win-not-win* example [29] applied to acyclic data. The non-stratified case is obtained from the same rule set when the data has cycles. In addition, we include a rule set, referred to as *MS*, obtained from stratified rules by applying the Magic Sets transformation. The resulting rule set is not predicate-stratified and most likely not locally stratified either (this depends on the randomly generated data).

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).
```

The base relations of `par` and `sib` were randomly generated, and there are cycles in the data. We ran the tests for two data sizes: 6000 and 24000.

The **win-not-win** test consists of a single rule, where `move` is the base relation:

```
win(X) :- move(X,Y), not win(Y).
```

When the base facts form a cycle: $\{\texttt{move}(1,2), \ldots, \texttt{move}(\texttt{i}, \texttt{i}+1), \ldots, \texttt{move}(\texttt{n}-1,\texttt{n}), \texttt{move}(\texttt{n},1)\}$, the rule set (together with the data) is not locally stratified. When the data is tree-structured: $\{\texttt{move}(\texttt{i}, 2*\texttt{i}), \texttt{move}(\texttt{i}, 2*\texttt{i}+1) \mid 1 \leq \texttt{i} \leq n\}$, the rule set is locally stratified. We used three cyclic and three tree-structured data sets with $n = 100000$, 250000, and 1000000.

The **MS** test was borrowed from [3]; it is a typical example of when the magic transformation turns stratified negation into a non-stratified one.

```
fb(X) :- magicfb(X),d(X),not ab(X),h(X,Y),ab(Y).
ab(X) :- magicab(X),g(X).
ab(X) :- magicab(X), b(X,Y),ab(Y).
magicab(Y) :- magicab(X),b(X,Y).
magicab(Y) :- magicfb(X),d(X),not ab(X),h(X,Y).
magicab(X) :- magicfb(X),d(X).
```

We used two data sets: one with 24000 facts for each of the base relations `b` and `h`, and the other with 504000 facts.

Miscellaneous tests.

OpenRuleBench includes two other suits of tests: *dynamic indexing* tests and *database interface* tests.

Dynamic indexing tests measure the time it takes to insert and delete facts one-by-one in main memory. Some indexing methods, such as the hash-based methods, are very sensitive to skewed data patterns which cause excessive collisions. Prolog semantics may also affect the performance because it is sensitive to the order of in which facts have been inserted. Our tests show that XSB is particularly sensitive to data skews and the order of insertions and deletions. Details can be found in [20].

We also developed tests for evaluating *interfaces to database management systems*, such as MySQL. However, in the end we abandoned this effort for two reasons. First, our aim was to evaluate the different technologies for performing rule-based inference per se. Database interfaces are important, but they are not inherent to the rules technology. Second, the problem of pushing data to and from disk is not unique to rule systems. The only issue that is truly unique is the need to store and retrieve tree-structured data—facts that include function symbols.[24] However, only one system, XSB, has support for storing this kind of facts, and even that support is implemented in a naive way. We also note that systems such as BigOWLIM and DLV^{DB} [26] (which we did not test) can deal with applications that do not fit in main memory.

4. RESULTS

Three systems, Yap, XSB, and Ontobroker (sometimes referred to as OB), were clear leaders in most of the tests with one system holding an edge in some tests and another in others. DLV was also a good performer overall, and in some cases it did on a par or even better than the above systems. Since a short summary cannot adequately describe our results, we will present them in a tabular form with a brief analysis following each table. In these tables, *size* means the total size of all base relations; *error* means that the system gave an error during the evaluation; *timeout* indicates that evaluation did not finish within a set time limit of 30 minutes. All times are given in seconds and *exclude* the loading time. We note that Ontobroker and XSB are significantly slower in loading large data sets than Yap and DLV. So, counting the loading time, XSB and Ontobroker fall behind DLV and Yap on DBLP and LUBM tests.

[23] `http://www.w3.org/TR/2004/REC-owl-guide-20040210/#WinePortal`

[24] Some might argue that XML database do something similar already.

query	a(X,Y)		b1(X,Y)		b2(X,Y)	
size	50000	250000	50000	250000	50000	250000
ontobroker	4.089	28.385	0.213	4.806	0.019	0.168
xsb	12.774	timeout	0.122	14.920	0.013	0.269
yap	10.534	timeout	0.109	12.123	0.013	0.269
dlv	85.459	838.781	7.177	60.239	0.820	9.392
jena	149.918	timeout	2.910	174.135	0.799	6.269
owlim	104.166	1484.126	1.999	83.820	0.038	1.192
drools	error	error	27.414	error	2.111	94.474
jess	310.000	timeout	12.000	317.000	1.000	26.000

Table 1: Join1, no query bindings

Large join tests.

The execution times are shown in Tables 1, 2, and 3 (Join1), Table 4 (5*Join1), Table 5 (Join2), Table 6 (LUBM-derived queries), Table 7 (Mondial), and Table 8 (DBLP). Once again, XSB, Yap, and Ontobroker contested the crown, outperforming each other on some tests and falling behind on others. DLV was sometimes close and sometimes even did better than XSB or Yap. Other systems were usually one or two orders of magnitude slower or could not run the tests at all. Note that Join2, Mondial, and the DBLP tests are not applicable to Jena and OWLIM because of the restrictions on the predicate arity in those systems.

Interestingly, both XSB and Yap time out for query `a(X,Y)` on a large dataset (50000 tuples in each base relation and 250000 tuples total—see the description of this test in Section 3) in tests shown in Tables 1 and 4. The likely explanation is that this is due to the fact that both XSB and Yap are limited to only one kind of a join method, indexed nested loops [15], while Ontobroker automatically chooses the most appropriate method from a small repertoire of algorithms. Here it uses the sort-merge join [15], which scales better than nested loops. Interestingly, in Table 1 XSB and Yap do a little better than Ontobroker for queries `b1(X,Y)` and `b2(X,Y)` when smaller data sets are used, because of the initial overhead (of sorting the relations) associated with sort-merge joins. For larger datasets, this overhead gets amortized and Ontobroker comes out on top. DLV scales better for `a(X,Y)` than XSB and Yap, but it trails far behind Ontobroker.

For the tests in Tables 2 and 3 where one of the query arguments is bound to a constant (=1), Jess, Drools, and OWLIM are omitted, as they cannot take advantage of query bindings and so their times are comparable to Table 1. For these tests, the comparative results are essentially the same as before, with Ontobroker doing the best overall and DLV scaling better than XSB and Yap for queries `a(1,Y)` and `a(X,1)`. For the bound-free case in Table 2, XSB and Yap do surprisingly well for queries `b1(1,Y)` and `b2(1,Y)`. This might be due to the overhead of the Magic Sets method, which Ontobroker uses in order to take advantage of the bindings in the query. For larger datasets this overhead gets amortized.

Most interesting, however, is the observation that, for Ontobroker, the times for the query `a(1,Y)` in Table 2 and for the queries `a(X,1)` and `b1(X,1)` in Table 3 do not change substantially when the dataset size increases from 50000 to 250000 tuples. And the time for `b2(1,Y)` and `b2(X,1)` even goes down significantly. All other systems report drastic time increases or even time out! This unexpected behavior is due to the fact that Ontobroker performs rule analysis and determines that rule *unfolding* (compile-time substitution of rule heads by rule bodies) followed by cost-based premise reordering may be called for. The other systems, if they do premise reordering at all, do not apparently precede this step with unfolding, so reordering does not yield substantial benefits for them. The decrease in the execution time for `b2` is due to a data skew (but Ontobroker is the only system that takes advantage of this).

query	a(1,Y)		b1(1,Y)		b2(1,Y)	
size	50000	250000	50000	250000	50000	250000
ontobroker	0.035	0.038	0.013	0.051	0.070	0.012
xsb	0.013	35.990	0.000	0.016	0.000	0.001
yap	0.021	30.233	0.007	0.050	0.004	0.025
dlv	0.287	6.014	0.014	0.112	0.008	0.066
jena	0.381	342.059	0.026	0.350	0.014	0.035

Table 2: Join1 with 1st argument bound.

query	a(X,1)		b1(X,1)		b2(X,1)	
size	50000	250000	50000	250000	50000	250000
ontobroker	0.030	0.036	0.011	0.010	0.028	0.009
xsb	5.033	timeout	0.048	5.105	0.004	0.095
yap	0.814	539.710	0.014	0.246	0.004	0.025
dlv	0.665	54.653	0.023	0.338	0.008	0.065
jena	74.741	timeout	1.179	71.482	0.333	1.741

Table 3: Join1 with 2nd argument bound.

The 5*Join1 test would have been unremarkable if not for another surprise from Ontobroker. This test directs the systems to compute the same query, Join1, five times. It does so by renaming the derived predicates and fooling the systems into thinking that these are five different queries. If we now compare Tables 1 and 4, we see that all systems increased their running times by the factor of about five, but Ontobroker's times stay roughly the same! It turns out that, after unfolding (the same optimization that gave Ontobroker an edge in the Join1 tests) the system determines that all five queries have common subexpressions (4-way joins), and factors them out. So, for Ontobroker, Join1 and 5*Join1 are essentially the same query.

For Join2, XSB and Yap take about the same time, while Ontobroker comes in third. A possible explanation for Ontobroker's performance in this test (four times slower than XSB and Yap) is that Join2 starts by producing large relations using the Cartesian product operation, and database technology, on which many of Ontobroker's optimizations are based, does not have a good strategy for Cartesian products. In this situation, the well-optimized tabled SLG-WAM platform gives XSB and Yap an advantage.

In the LUBM tests, Drools and Jess run out of memory while loading the larger dataset (the 50univs data set has over 6000000 tuples, claiming more than 600MB of memory). XSB is the overall winner with Ontobroker coming close second. Yap is slow on Query 1, but it bests Ontobroker and sometimes XSB on other queries. Jena, and OWLIM do very well on queries 1 and 9, but poorly on Query2. DLV, on the other hand, does well overall: slightly behind OWLIM on queries 1 and 9, but far ahead of both Jena and OWLIM on Query 2.

The better performance of XSB may be explained by the fact that we manually requested indexing on all arguments (recall that XSB does not create suitable indices without being asked), while Yap, Ontobroker, and others use heuris-

query	a(X,Y)		b1(X,Y)		b2(X,Y)	
size	50000	250000	50000	250000	50000	250000
ontobroker	3.647	28.497	0.163	4.601	0.029	0.189
xsb	63.841	timeout	0.634	73.731	0.060	1.378
yap	52.202	timeout	0.522	61.650	0.054	1.246
dlv	415.237	timeout	28.276	297.341	2.582	35.595
jena	709.440	timeout	9.794	831.204	1.212	22.403
owlim	671.996	timeout	15.803	590.309	0.820	38.215
drools	error	error	error	error	16.059	error
jess	error	error	65.000	error	6.000	error

Table 4: 5*Join1, no query bindings

system	yap	xsb	OB	dlv	jess	drools
time	2.087	2.092	11.935	44.692	timeout	error

Table 5: Times for Join2.

query	Query1		Query2		Query9	
size	10univs	50univs	10univs	50univs	10univs	50univs
ontobroker	0.013	0.025	0.596	2.704	1.243	8.799
xsb	0.000	0.001	0.142	0.968	0.162	1.124
yap	0.106	0.607	0.139	0.938	0.417	3.300
dlv	0.251	3.188	1.041	7.388	1.260	7.068
jena	0.017	0.018	5.282	63.782	2.390	14.889
owlim	0.110	1.050	4.420	195.480	1.000	3.130
drools	0.001	error	13.537	error	22.483	error
jess	0.000	error	1.000	error	2.000	error

Table 6: Times for LUBM tests

tics to determine which indices to create. We conjecture that these heuristics might have overlooked better alternatives.

The Mondial test shows that XSB outperforms the other systems by an order of magnitude. Again, this is likely the effect of a better manual choice of indexing versus heuristic indexing of Yap, Ontobroker, and others.

The DBLP test is a 4-way self-join of a very large relation (see Section 3), which yields a small answer set. Good indexing is a must in order to pass this test, and the usual suspects, Ontobroker, XSB, DLV, and Yap, do well. But the true surprise is that the best time (by an order of magnitude!) belongs to none other than Drools—a system that was lagging in other tests. On close examination, it appears that here Drools uses the same query plan as a well-optimized database system would: first it selects the relation `att` on the second argument (thus reducing the size of the relation ten-fold), then builds index on the join attribute, and then computes the join. Other systems appear to be computing the 4-way join of `att` with itself *directly*, albeit using different join algorithms and different indices.

At this point it is instructive to revisit the issue of optimization in order to better appreciate its impact on performance. The Mondial test and XSB offer a striking example. In this test, XSB tables some predicates in order to avoid recomputation and indices are used for faster access to data. With these optimizations, Table 7 shows that XSB finishes in just 0.004 seconds. However, if tabling is turned off, the time jumps to 1.713 seconds, a 400-fold increase! With no indexing, the time jumps further to 129.89 seconds—yet another 76-fold increase. Altogether, these optimizations yield a better than 30000 times speed-up!

Datalog recursion.

We examined the runtime for the query `tc` (transitive closure) and `sg` (same generation) with different variable bindings: free-free, bound-free and free-bound. Again, Yap, XSB, and Ontobroker did the best job, followed by OWLIM, Jena, DLV, Jess, and Drools with performance lags ranging from 30% to an order of magnitude.[25] In addition, we looked at the cases when the base relation `par` is an acyclic graph

[25]OWLIM, Jess, and Drools are omitted for queries with argument bindings, as these systems cannot benefit from such information.

system	xsb	yap	OB	jess	dlv	drools
time	0.004	0.037	0.042	2.000	1.045	3.476

Table 7: Times for Mondial

system	OB	xsb	yap	drools	dlv	jess
time	1.602	1.752	2.447	0.186	2.201	error

Table 8: Times for DBLP

size	50000	50000	500000	500000
cyclic data	no	yes	no	yes
ontobroker	6.129	19.145	49.722	182.633
xsb	2.725	7.081	35.036	88.028
yap	2.066	13.026	33.128	82.900
dlv	19.655	73.837	148.740	900.773
drools	120.416	error	error	error
jess	55.000	184.000	411.000	timeout
jena	16.332	47.674	108.940	451.590
owlim	7.656	38.457	75.847	395.226

Table 9: Transitive closure, no query bindings.

as well as when it is cyclic. These two sets of tests reveal one interesting phenomenon, which is yet to be explained: on the `tc` tests, XSB is typically 2-3 times faster and scales better than Ontobroker, while on the `sg` tests Ontobroker is both faster and scales better.

Yap is the best overall performer in the WordNet tests, followed by XSB and Ontobroker. Ontobroker does marginally better than XSB on the *meronyms*, *holonyms*, and *troponyms* tests, while loosing by a wider margin on other tests.

For the Wine ontology, XSB is a notch ahead of Ontobroker. Yap takes twice the time, but is far ahead of the rest. However, both XSB and Yap run out of memory under their default settings. Special optimizations (subsumptive and batched tabling) and some reordering of rule premises are required in order for these systems to pass the test. Ontobroker's all-around good performance (with no need for manual intervention) is likely due to its premise-reordering heuristic and its parallel engine, which yields a 20% speedup.

Default negation.

One can see from Table 17 that, for stratified rule sets, Yap works best, followed by XSB then Ontobroker and DLV in different orders, depending on the test. Yap slips behind Ontobroker on the modified same generation test with a large dataset. The table indicates that all systems have scalability problems in one case or another: Ontobroker might time-out, XSB and DLV might take comparatively too long to execute the second query, while Yap takes more than 10 times longer when going from 6000 to 24000 facts in case of the modified same generation test (the Ontobroker's test shows that such a large jump is not inevitable).

Drools and Jess do well for the stratified win-not-win test on smaller datasets, but Drools has a problem with larger sets. Both fail for the modified same generation test.

Yap does not support rule sets that are not stratified locally, so it is not included in Table 18. The test shows that XSB is significantly faster and scales better than the other

size	50000	50000	500000	500000
cyclic data	no	yes	no	yes
ontobroker	5.708	21.787	47.804	180.758
xsb	1.437	5.563	18.532	55.447
yap	0.722	5.666	9.026	31.454
dlv	16.188	63.089	107.629	111.990
jena	3.884	11.593	35.383	120.466

Table 10: Transitive closure, first argument bound.

size	50000	50000	500000	500000
cyclic data	no	yes	no	yes
ontobroker	0.015	0.054	0.042	0.271
xsb	0.016	0.015	0.180	0.157
yap	0.036	0.048	0.575	0.689
dlv	0.025	0.068	0.489	0.830
jena	0.792	0.884	5.314	5.032

Table 11: Transitive closure, 2nd argument bound.

size	6000	6000	24000	24000
cyclic data	no	yes	no	yes
ontobroker	1.402	1.926	5.424	5.309
xsb	2.359	3.408	42.824	44.487
yap	1.875	3.148	43.510	43.452
dlv	20.274	31.346	365.136	438.008
drools	104.884	error	error	error
jess	64.000	error	1517.000	error
jena	21.007	37.692	387.268	415.376
owlim	8.666	13.314	174.968	195.825

Table 12: Same generation, no query bindings.

size	6000	6000	24000	24000
cyclic data	no	yes	no	yes
ontobroker	0.857	1.835	4.421	4.726
xsb	1.338	3.245	35.823	42.297
yap	0.619	1.398	18.238	18.866
dlv	11.976	29.887	300.403	419.099
jena	6.827	15.959	155.512	191.842

Table 13: Same generation, 1st argument bound.

size	6000	6000	24000	24000
cyclic data	no	yes	no	yes
ontobroker	0.849	1.873	4.874	5.339
xsb	2.859	3.954	55.855	51.406
yap	0.845	1.539	19.178	19.450
dlv	14.472	38.913	373.920	506.422
jena	15.784	21.275	251.355	254.322

Table 14: Same generation, 2nd argument bound.

Test	hypernyms	hyponyms	meronyms	holonyms	troponyms
OB	4.968	4.862	0.512	0.603	0.227
yap	0.244	0.236	0.216	0.224	0.216
xsb	1.524	1.868	0.688	0.612	0.340
dlv	33.65	35.920	7.460	7.200	1.390
drools	error	error	error	error	0.911
jess	error	error	4.060	4.050	0.094
jena	7.613	14.291	2.023	2.234	0.059
owlim	5.713	5.164	1.158	1.485	0.123

Table 15: The WordNet tests.

System	OB	xsb	yap	dlv	jess	owlim	jena	drools
time	5.23	4.47	12.05	20.19	140	error	error	error

Table 16: The Wine ontology test.

test	win-not-win			modified same gen	
size	100000	500000	2000000	6000	24000
ontobroker	1.327	9.988	timeout	14.883	24.963
xsb	0.231	1.218	5.081	7.265	90.928
yap	0.103	0.654	2.866	3.339	44.605
dlv	0.691	3.554	15.224	36.827	444.873
drools	1.228	7.466	error	error	error
jess	1.000	3.000	15.000	error	error

Table 17: Locally- and predicate-stratified negation.

test	win-not-win			MS	
size	50000	250000	1000000	24000	504000
ontobroker	0.419	3.754	17.237	0.236	8.409
xsb	0.339	1.416	5.647	1.381	1.663
dlv	0.344	1.879	8.361	0.189	2.043

Table 18: Times for locally non-stratified rule sets.

two systems, while DLV does better than Ontobroker. We should note that DLV and Ontobroker are both bottom-up systems, and they use similar algorithms for the well-founded negation, based on the alternating fixpoint computation [28]. In contrast, XSB is a top-down system, and its algorithm for the well-founded negation is radically different: it is based on SLG-resolution, and this might account for the difference in performance.

5. CONCLUSIONS

We presented the results of a comprehensive study of the performance and scalability for several well-known engines for rule-based reasoning. Our objective was to compare existing technologies and evaluate their potential as semantic engines for the Web. Our data, generators, and scripts are available as part of OpenRuleBench [20], a benchmarking suite provided as an open community resource for further investigation into the performance of various rule engines.

Our results show that two technologies hold great promise: the tabling Prolog technology, such as the one used in XSB and Yap, and the deductive database technology used in Ontobroker and DLV. Three systems, XSB, Yap, and Ontobroker were clear winners in most of the tests. DLV also did well in most of the tests.

The significant variability in the test results among the top-performing systems draws attention to three issues that are critical for engine scalability and performance: indexing, memory management, and heuristic query optimization. Indexing seems to have been more or less adequately addressed in Ontobroker and Yap (and to a lesser extent in XSB), but when it comes to memory management all of the tested systems have met their challenge in one benchmark or another (on larger data sets available in OpenRuleBench). The third problem, query planning, has been addressed in some bottom-up engines (DLV, Ontobroker), but further research is needed to improve the algorithms.

Future work on OpenRuleBench might include additional engines and tests. For instance, we would like to add tests that involve function symbols in the data.

Acknowledgments. This work was supported, in part, by a contract with Vulcan, Inc. We would like to thank Benjamin Grosof for his encouragement, discussions, and many helpful suggestions during the course of this project. We are also grateful to Juergen Angele, Juergen Baier, and Ulrich Siebald for suggesting the Mondial, DBLP, and wine benchmarks, and for the help with setting up and running Ontobroker. Many thanks to: David Warren and Terrance Swift—for assistance with the use of XSB; Vitor Santos Costa and Michel Ferreira—for the help with Yap; Keith Goolsbey, Douglas Miles, Douglas Lenat, Larry Lefkowitz, Vinay Chaudhri, Ken Murray, and Bill Jarrold—for helping out with CYC; Wolfgang Faber, Gerald Pfeifer, Nicola Leone—for many insights into the use of DLV; Edson Tirelli, Geoffrey De Smet, Jaroslaw Kijanowski, Mark Proctor, Anstis Michael, and Ojwang Wilson — for assistance with Drools;

Barry Bishop — for assistance with Iris; Andy Seaborne, James Howison, Damian Steer, Dave Reynolds, and Adrian Walker — for the help with Jena; Ernest Friedman-Hill, Wolfgang Laun, Dane Wyrick, and Erick Wang—for assistance with Jess; Damyan Ognyanoff and Atanas Kiryakov — for helping with OWLIM; Adrian Paschke and Alex Kozlenkov — for the help with Prova.

6. REFERENCES

[1] H. Aït-Kaci. *Warren's abstract machine: a tutorial reconstruction*. MIT Press, Cambridge, MA, 1991. `http://www.freetechbooks.com/warren-s-abstract-machine-a-tutorial-reconstruction-t397.html`.

[2] K.R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Los Altos, CA, 1988.

[3] I. Balbin, G. S. Port, K. Ramamohanarao, and K. Meenakshi. Efficient bottom-up computation of queries on stratified databases. *J. Log. Program.*, 11(3-4):295–344, 1991.

[4] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10:255–300, April 1991.

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

[6] F. Bry, N. Eisinger, T. Eiter, T. Furche, G. Gottlob, C. Ley, B. Linse, R. Pichler, and F. Wei. Foundations of Rule-Based Query Answering. In *Proceedings of Summer School Reasoning Web 2007, Dresden, Germany (3rd–7th September 2007)*, volume 4634 of *LNCS*, pages 1–153. REWERSE, 2007.

[7] F. Calimeri, S. Cozza, G. Ianni, and N. Leone. Computable functions in ASP: Theory and implementation. In *Int'l Conference on Logic Programming*, pages 407–424, December 2008.

[8] The China benchmark suite, 2001. `http://www.cs.unipr.it/China/Benchmarks/`.

[9] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 292–322. Plenum Press, 1978.

[10] C.L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Atrificial Intelligence*, 19(1):17–37, September 1982.

[11] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice-Hall, 2008.

[12] M. Gelfond and N. Leone. Logic programming and knowledge representation — the A-Prolog perspective. *Artificial Intelligence*, 138(1–2):3–38, 2002.

[13] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the Fifth Conference and Symposium*, pages 1070–1080, 1988.

[14] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 2005.

[15] M. Kifer, A. Bernstein, and P.M. Lewis. *Database Systems: An Application Oriented Approach, Complete Version (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2005.

[16] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, 42:741–843, July 1995.

[17] M. Kifer and E.L. Lozinskii. Implementing logic programs as a database system. In *IEEE 3-d Int'l Conference on Data Engineering*, pages 375–385, February 1987.

[18] O. Lassila and R.R. Swick (editors). Resource description framework (RDF) model and syntax specification. Technical report, W3C, February 1999. `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`.

[19] L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, and S. Liu. Towards a complete OWL ontology benchmark. In *European Semantic Web Conference*, pages 125–139, 2006.

[20] Openrulebench web site, 2008. `http://rulebench.projects.semwebcentral.org`.

[21] Prolog benchmarking, 1985. `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/code/bench/0.html`.

[22] T.C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, Los Altos, CA, 1988.

[23] Y Sure, S Staab, and J Angele. Ontoedit: Guiding ontology development by methodology and inferencing. In *1st International Conf. on Ontologies, Databases, and Applications of Semantics*, 2002.

[24] T. Swift and D.S. Warren. An abstract machine for SLG resolution: Definite programs. In *Int'l Logic Programming Symposium*, Cambridge, MA, November 1994. MIT Press.

[25] H.J. ter Horst. Combining RDF and part of OWL with rules: Semantics, decidability, complexity. In *International Semantic Web Conference (ISWC)*, pages 668–684, November 2005.

[26] G. Terracina, N. Leone, V. Lio, and C. Panetta. Experimenting with recursive queries in database and logic programming systems. *Journal of the Theory and Practice of Logic Programming*, 8(2):129–165, 2008.

[27] J.F. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 1*. Computer Science Press, Rockville, MD, 1988.

[28] A. Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993.

[29] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[30] D.S. Warren. Memoing for logic programming. *Communications of ACM*, 35(3):93–111, March 1992.

[31] D.S. Warren. Programming in tabled prolog. Manuscript. `http://www.cs.sunysb.edu/~warren/xsbbook/`, 1999.