

OpenRuleBench: Detailed Report

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`

March 31, 2010

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 twelve systems, but OpenRuleBench is an open community resource, and contributions from the community are welcome. In this report, we describe the tested systems and technologies, the methodology used in testing, and analyze the results.

Contents

1	Introduction	3
2	Systems Tested	4
2.1	Prolog-Based Systems	4
2.2	Deductive Databases Systems	6
2.3	Rule Engines for Triples	7
2.4	Production and Reactive Rule Systems	8
2.5	Knowledge-Base Systems	9
3	Methodology	9
3.1	Performance Related Factors	10
3.2	Tests	12
3.2.1	Large Join Tests	12
3.2.2	Datalog Recursion	16
3.2.3	Default Negation	23
3.2.4	Tests Involving Function Symbols and Database Updates	25
3.2.5	Dynamic Indexing	25
3.2.6	Interfaces to Database Management Systems	26
4	Results	26
4.1	Large Join Tests	27
4.2	Datalog Recursion	32
4.3	Default Negation	37
4.4	Tests Involving Function Symbols and Database Updates . . .	38
4.5	Dynamic Indexing	39
5	Conclusions	41

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 report is a step towards filling in the void. Over the last eight months we have been studying and experimenting with twelve 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

¹ <http://www.w3.org/2005/rules/>

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.

This report 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 presents the conclusions.

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 technologies, and attempt to mitigate the inevitable flaws of such a categorization with short descriptions of each system. However, this report is *not* intended to be a survey of the different rule systems and of their underlying technologies. For the latter, the reader is referred to [6].

We should 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.

2.1 Prolog-Based Systems

XSB² 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

² <http://xsb.sourceforge.net/>

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.

Yap³ 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.

SWI⁴ is a popular open-source Prolog system. Like XSB and Yap, it has a *WAM*-based implementation, but unlike these systems it does not support tabling. As a result, this system cannot run most of the recursive tests in the OpenRuleBench suite. However, enough interesting tests can be performed using SWI, and we included it in our study. SWI is written in the C language.

³ <http://www.dcc.fc.up.pt/~vsc/Yap/>

⁴ <http://www.swi-prolog.org>

2.2 Deductive Databases Systems

DLV⁵ 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 [13]. The well-founded negation is also supported. 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.

IRIS⁶ 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.

Ontobroker⁷ 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,

⁵ <http://www.dbai.tuwien.ac.at/proj/dlv/>

⁶ <http://sourceforge.net/projects/iris-reasoner>

⁷ <http://www.ontoprise.de/de/en/home/products.html>

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].

2.3 Rule Engines for Triples

Jena⁸ 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.

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.⁹ Jena is distributed under an open-source license.

SwiftOWLIM and **BigOWLIM**¹⁰ 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.

⁸ <http://jena.sourceforge.net/>

⁹ Jena programs are executed in JVM, but this is a general purpose virtual machine, which is not optimized for rules.

¹⁰ <http://www.ontotext.com/owlim/index.html>

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,¹¹ 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.

2.4 Production and Reactive Rule Systems

Drools¹² 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-naïve bottom-up computation [27] commonly used in Datalog systems with a certain heuristic for common expression elimination [6].

Jess¹³ is also a production rule system. Like Drools, it is mainly a bottom-up

¹¹ Or binary relations—depending on the point of view.

¹² <http://www.jboss.org/drools/>

¹³ <http://herzberg.ca.sandia.gov/>

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.

Prova¹⁴ 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.¹⁵ Drools and Prova are distributed under open-source licenses. Jess is free for research, but is not open-source.

2.5 Knowledge-Base Systems

CYC¹⁶ is an inference engine equipped 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, we should note that CYC's reasoning capabilities go well beyond rules, and this puts it at a disadvantage when pure rule-based reasoning is involved.

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).

¹⁴ <http://www.prova.ws/>

¹⁵ 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.

¹⁶ <http://www.cyc.com/>

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 created with the help of data generators. Typically OpenRuleBench uses data sets with sizes 50000, 100000, 250000, 500000, and 1000000 facts. 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.

This report describes and analyzes the results. 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.

In this section, we discuss performance related factors and describe the test programs.

3.1 Performance Related Factors

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.

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.

3.2 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.

3.2.1 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 three total data sizes: 50000, 250000 and 1250000. 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

¹⁷ All our examples use Prolog syntax.

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

here we used the same base relations as for the Join1 test.

Join2 is another handcrafted pattern of joins, borrowed from [5]. It produces a large intermediate result, but only 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 content of the base relation is $p(a0), \dots, p(a18)$, 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 $\langle s, d, u \rangle$ 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 $\langle s, f, c \rangle$ 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).

$query1(X) :- takesCourse(X,graduateCourse0),$

```

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).

```

LUBM Data Generator (UBA)¹⁸ generates data (in owl format), which is then transformed by our tools into data formats required by different systems.

Rules are translated manually from the LUBM ontology,¹⁹ and relevant rules are selected to help answer the above queries. *Query1* can be answered directly with input data. *Query2* is formulated in terms of a number of derived predicates, which are defined using the following rules:

```

memberOf(X,Y):- worksFor(X,Y).
worksFor(X,Y):- headOf(X,Y).
university(Y):- undergraduateDegreeFrom(_,Y).
university(Y):- degreeFrom(_,Y).
university(X):- hasAlumnus(X,_).
university(Y):- doctoralDegreeFrom(_,Y).
university(Y):- mastersDegreeFrom(_,Y).
degreeFrom(X,Y):- doctoralDegreeFrom(X,Y).
degreeFrom(X,Y):- mastersDegreeFrom(X,Y).
degreeFrom(X,Y):- undergraduateDegreeFrom(X,Y).
hasAlumnus(X,Y):- degreeFrom(Y,X).
subOrganizationOf_0(X,Y):- subOrganizationOf(X,Y).
subOrganizationOf_0(X,Z):- subOrganizationOf(X,Y),
                           subOrganizationOf_0(Y,Z).

```

Query9 uses derived predicates, which are defined by the following rules:

```

student(X):- researchassistant(X).
student(X):- takesCourse(X,Y), course(Y).
student(X):- undergraduatestudent(X).
course(X):- graduatecourse(X).

```

¹⁸<http://projects.semwebcentral.org/frs/download.php/26/uba1.7.zip>

¹⁹<http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl>

```

course(Y):- listedCourse(_,Y).
course(Y):- teacherOf(_,Y).
course(Y):- teachingAssistantOf(_,Y).
faculty(X):- lecturer(X).
faculty(X):- postdoc(X).
faculty(X):- professor(X).
faculty(X):- teacherOf(X,_).
professor(X):- assistantprofessor(X).
professor(X):- associateprofessor(X).
professor(X):- fullprofessor(X).
professor(X):- visitingprofessor(X).
professor(X):- chair(X).
professor(X):- dean(X).
professor(Y):- advisor(_,Y).
professor(X):- tenured(X,_).
chair(X):- headOf(X,Y), department(Y).
dean(X):- headOf(X,Y), college(Y).

```

Mondial²⁰ is a geographical database derived from the CIA Factbook.²¹ The base relations, `province` and `city`, 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.

```

isa(prov(Y,X),provi) :- p(_,_,X,Y).
att(prov(Y,X),number,A) :- p(A,_,X,Y).
att(prov(Y,X),number2,D) :- p(_,D,X,Y).
p(A,D,X,Y) :- province(Y,X,D,_,_,_), city(_,X,Y,A,_,_).
query(X,A,D) :- isa(prov(X,'cn'),provi),
                  att(prov(X,'cn'),number,A),
                  att(prov(X,'cn'),number2,D).

```

DBLP is a database of publications in the fields of databases and logic programming derived from the well-known Web-based bibliography under the

²⁰ <http://www.dbis.informatik.uni-goettingen.de/Mondial/>

²¹ <https://www.cia.gov/library/publications/the-world-factbook/>

same name.²² 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, which seeks the ID, title, authors, year and month for all publications:

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

3.2.2 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,²³ and a rule representation of the well-known wine ontology.²⁴

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 generate data from a domain **D**, and different data sizes are used: 50000, 250000, 500000, and 1000000.

The **same-generation** problem seeks to find all siblings in the same generation, i.e., siblings that are equally remotd from a common ancestor.

²² <http://www.informatik.uni-trier.de/~ley/db/>

²³ <http://wordnet.princeton.edu>

²⁴ <http://www.w3.org/TR/2004/REC-owl-guide-20040210/#WinePortal>

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. In each case, data are generated from a domain D and the size of `par` is five times of the size of `sib`. Different sizes are used: 6000, 12000, and 24000.

The **WordNet benchmarks** contain applications from natural language processing based on the WordNet semantic lexicon²⁵ in the OpenRuleBench benchmark. The WordNet benchmarks are real-life tests dealing with common tasks from natural language processing with WordNet, such as, detection of synonyms (i.e., search of words in the same synset with a given word), hypernyms (i.e., given a word, find all words that are more generic than the given word), hyponyms (i.e., given a word, find all words that are more specific than the given word), meronyms (part-of a whole, for instance, “brim” and “crown” are meronyms of “hat”), holonyms (composed-of), antonyms, and similar adjectives.

The input data was extracted from the version 3.0 of the WordNet ontology and contains the following relations:

1. The predicate $s(\text{synset_id}, \text{w_num}, \text{'word'}, \text{ss_type}, \text{sense_number}, \text{tag_count})$ stipulates the synset, word number, sense and tag count for every word sense in WordNet, where the synset identifier is a numeric value, the word number is the sequence number of the current word in the synset list, the sense type ss_type is a one character code indicating the synset type: **n** for noun, **v** for verb, **a** for adjective, **s** for adjective satellite, and **r** for adverb. Size 212158 tuples; over 7.2M in total.
2. The predicate $g(\text{synset_id}, \text{'gloss'})$ provides the gloss for a synset, i.e., a detailed description in English of the meaning of the words in the synset. This description typically includes examples of the use of the most frequently used word in the synset in a sentence. Size: 121280 tuples; over 11M in total.

²⁵ <http://wordnet.princeton.edu>

3. The predicate *hyp(synset_id,synset_id)* stipulates that the second synset is a hypernym of the first synset. This relation holds for nouns and verbs. Size: 89089 tuples; over 2M in total.
4. The predicate *mm(synset_id,synset_id)* stipulates that the second synset is a member meronym (i.e., "part-of") of the first synset. This relation only holds for nouns. Size: 12293 tuples; about 300K in total.
5. The predicate *sim(synset_id,synset_id)* is a relation between adjective synsets. It says that the second synset is a satellite to the first synset. The adjective synsets are organized in clusters, where one synset is the head of the cluster and other synsets are satellites of the head. Size: 21386 tuples; over 500K in total.
6. The predicate *ant(synset_id,w_num,synset_id,w_num)* stipulates antonymous words. This is a lexical relation that holds for all syntactic categories. Size: 8689 tuples; 250K in total.
7. The predicate *ent(synset_id,synset_id)* stipulates that the second synset is an entailment of first synset. This relation only holds for verbs. Size: 408; over 10K in total.

Our benchmarks contain the following queries, which are often used natural language processing applications and question-answering systems, including parsers, question analysis, and answer selection.

1. The query **sameSynset** finds all synonyms for every word sense in WordNet. This query is a large join of the relation s/6 with itself on the first argument (*synsetId*), followed by projection on the third arguments (the actual words).

```
sameSynset(W1,W2):-
    s(S,_,W1,_,_,_),
    s(S,_,W2,_,_,_).
?- findall(sameSynset(W1,W2),sameSynset(W1,W2),L).
```

2. The query **wordGloss** finds all the glosses (i.e., longer descriptions) for each word in WordNet. This query is a join of the relations s/6 with g/2 on the first arguments (the synset), followed by projection of the third argument of s/6 (the actual word) and the second argument of g/2 (the gloss).

```

wordGloss(W,G):-
    s(S,_,W,_,_,_),
    gloss(S,G).
?- findall(wordGloss(W,G),wordGloss(W,G),L).

```

3. The queries `directHypernym` and `hypernyms` finds all hypernyms for every word sense in WordNet. The first query joins s/6 with hyp/2 and s/6 again, while computing direct hypernyms (just one level part-of relations). The second query computes multiple levels of hypernyms using recursion (essentially the transitive closure of the relation hyp/2 followed by a join).

```

directHypernym(W1,W2):-
    s(S1,_,W1,_,_,_),
    hyp(S1,S2),
    s(S2,_,W2,_,_,_).
?- findall(directHypernym(W1,W2),directHypernym(W1,W2),L).

```

```

hypernymSynsets(S1,S2):-
    hyp(S1,S2). % direct hypernym sets
hypernymSynsets(S1,S2):-
    hyp(S1,S3),
    hypernymSynsets(S3,S2). % multiple-levels hypernym sets

```

```

hypernym(W1,W2):-
    s(S1,_,W1,_,_,_),
    hypernymSynsets(S1,S2),
    s(S2,_,W2,_,_,_).
?- findall(hypernym(W1,W2),hypernym(W1,W2),L).

```

4. The queries `directHyponym` and `hyponyms` find all hyponyms (direct and all) for every word in WordNet. These tests are the reverse of the hypernym tests above; they are sensitive to the choice of indexing for hyp/2. For XSB, the default index is on the first argument, so the `directHyponym` test is very slow (491.125 seconds), while using an index defined for the second argument is much faster (0.391 seconds).

```

directHyponym(W1,W2):-

```

```

s(S1,_,W1,_,_,_),
hyp(S2,S1),
s(S2,_,W2,_,_,_) .
?- findall(directHyponym(W1,W2),directHyponym(W1,W2),L) .

```

```

hyponymSynsets(S2,S1):-
hyp(S2,S1). % direct hyponym sets
hyponymSynsets(S2,S1):-
hyp(S3,S1),
hyponymSynsets(S2,S3). % multiple-levels hyponym sets

```

```

hyponym(W1,W2):-
s(S1,_,W1,_,_,_),
hyponymSynsets(S2,S1),
s(S2,_,W2,_,_,_) .
?- findall(hyponym(W1,W2),hyponym(W1,W2),L) .

```

5. The queries `directMeronyms` and `meronyms` find direct meronyms and all meronyms for every word in WordNet. This test can also be sped up by a proper choice of indices: from 67.546 seconds to 0.172 seconds.

```

directMeronym(W1,W2):-
s(S1,_,W1,_,_,_),
mm(S2,S1),
s(S2,_,W2,_,_,_) .
?- findall(directMeronym(W1,W2),directMeronym(W1,W2),L) .

```

```

meronymSynsets(S2,S1):-
mm(S2,S1). % direct meronym sets
meronymSynsets(S2,S1):-
mm(S3,S1),
meronymSynsets(S2,S3). % multiple-levels meronym sets

```

```

meronym(W1,W2):-
s(S1,_,W1,_,_,_),
meronymSynsets(S2,S1),
s(S2,_,W2,_,_,_) .
?- findall(meronym(W1,W2),meronym(W1,W2),L) .

```

6. The queries `allDirectHolonyms` and `allHolonyms` find all direct holonyms and all holonyms, respectively, for every word in WordNet.

```
directHolonym(W1,W2):-
    s(S1,_,W1,_,_,_),
    mm(S2,S1),
    s(S2,_,W2,_,_,_).
?- findall(directHolonym(W1,W2),directHolonym(W1,W2),L).
```

```
holonymSynsets(S1,S2):-
    mm(S1,S2). % direct holonym sets
holonymSynsets(S1,S2):-
    mm(S1,S3),
    holonymSynsets(S3,S2). % multiple-levels holonym sets
```

```
holonym(W1,W2):-
    s(S1,_,W1,_,_,_),
    holonymSynsets(S2,S1),
    s(S2,_,W2,_,_,_).
?- findall(holonym(W1,W2),holonym(W1,W2),L).
```

7. The queries `directTroponyms` and *troponyms* find all direct troponyms and all troponyms for every word in WordNet.

```
directTroponym(W1,W2):-
    s(S1,_,W1,_,_,_),
    ent(S1,S2),
    s(S2,_,W2,_,_,_).
?- findall(directTroponym(W1,W2),directTroponym(W1,W2),L).
```

```
troponymSynsets(S1,S2):-
    ent(S1,S2). % direct troponym sets
troponymSynsets(S1,S2):-
    ent(S1,S3),
    troponymSynsets(S3,S2). % multiple-levels troponym sets
```

```
troponym(W1,W2):-
    s(S1,_,W1,_,_,_),
```

```

troponymSynsets(S1,S2),
s(S2,_,W2,_,_,_),
?- findall(troponym(W1,W2),troponym(W1,W2),L).

```

8. The queries `directAdjectiveClusters` and `adjectiveClusters` find all direct similar adjectives and all (direct or indirect) similar adjectives for every word in WordNet.

```

directAdjectiveCluster(W1,W2):-
s(S1,_,W1,_,_,_),
sim(S1,S2),
s(S2,_,W2,_,_,_),
?- findall(directAdjectiveCluster(W1,W2),directAdjectiveCluster(W1,W2),L).

```

```

adjectiveClusterSynsets(S1,S2):-
sim(S1,S2). % direct adjectiveCluster sets
adjectiveClusterSynsets(S1,S2):-
sim(S1,S3),
adjectiveClusterSynsets(S3,S2). % multiple-level adjectiveCluster sets

```

```

adjectiveCluster(W1,W2):-
s(S1,_,W1,_,_,_),
adjectiveClusterSynsets(S1,S2),
s(S2,_,W2,_,_,_),
?- findall(adjectiveCluster(W1,W2),adjectiveCluster(W1,W2),L).

```

9. The query `antonyms` find all antonyms for every adjective in WordNet.

```

antonym(W1,W2):-
s(S1,_,W1,_,_,_),
ant(S1,_,S2,_),
s(S2,_,W2,_,_,_),
?- findall(antonym(W1,W2),antonym(W1,W2),L).

```

The following table shows the number of solutions for each of these WordNet tests.

The **wine ontology** is a rule-based representation of the well-known OWL wine ontology.²⁶ It has 815 rules and 654 facts. Since a ruleset this

²⁶ <http://www.w3.org/TR/2004/REC-owl-guide-20040210/#WinePortal>

Test	Number of Solutions
sameSynset (join)	558996
Glosses (join)	223716
directHypernyms (join)	329402
hypernyms (recursion)	2323413
directHyponyms (join)	329402
hyponyms (recursion)	2323413
directMeronyms (join)	60486
meronyms (recursion)	487501
directHolonyms (join)	60486
holonyms (recursion)	487501
directTroponyms (join)	2352
troponyms (recursion)	2653
directAdjectiveClusters (join)	50746
adjectiveClusters (recursion)	688452
antonyms (join)	12949

Table 1: Number of results for the WordNet experiments

large cannot be included here, we refer the reader to the source files on this Web site [20]. The query seeks all Californian wines. Perhaps the most important feature of this test is that a large number of predicates are recursively dependent on each other through several 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.

3.2.3 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 apply-

ing 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 derived predicates `tc` and `sg` were defined in Section 3.2.2. The base relations of `par` and `sib` were randomly generated, and there are cycles in the data. The base relations for `par` and `sib` were randomly generated. As before, we considered both cyclic and acyclic data. In each case, data are generated from a domain D and the size of `par` is five times of the size of `sib`. Different sizes are used: 6000, 12000, 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: $\{\text{move}(1,2), \dots, \text{move}(i, i+1), \dots, \text{move}(n-1,n), \text{move}(n,1)\}$, the rule set (together with the data) is not locally stratified. When the data is tree-structured: $\{\text{move}(i, 2*i), \text{move}(i, 2*i+1) \mid 1 \leq i \leq n\}$, the rule set is locally stratified. We used three cyclic and three tree-structured data sets with $n = 50000, 100000, 250000, 500000$, 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).
```

Base relations `b` and `h` are randomly generated, and we used different data sizes: 10000, 50000, 100000, 250000, and 500000.

3.2.4 Tests Involving Function Symbols and Database Updates

These tests include three programs described below. Only Yap, XSB, DLV, Jess, Ontobroker, and Prova could represent the corresponding rule sets. The rule sets for Yap and XSB are essentially identical. For DLV, we rewrote the tests to allow this system take advantage of its answer set programming capabilities. Jess is less expressive: it can represent only the BitRev test. Ontobroker and Prova can express all three programs but, as we shall see in Section 4.4, they cannot run most of these tests.

The **16-Puzzle** is a sliding puzzle that consists of a 4×4 grid of numbered square blocks (from 1 to 15) with one square empty. A *move* slides a block into the adjacent empty space. Given an initial configuration I and a final configuration F of the grid, the goal is to find a sequence of moves that transforms I into F . Our tests limit the maximal number of moves to 14, 16, 18, 20, and 22.

N-Queens is a problem of putting n queens on a chessboard of the size $n \times n$ so that none of them will be attacking any other. Our tests were performed for $n = 8, 10, 12, 14$, and 16.

Bitrev is taken from the 2007 Prolog Programming Contest.²⁷ $bitrev(n, i)$ is a binary number obtained from the first n bits of the binary representation of i and reversing them. For example, $bitrev(4, 12) = reverse(1100_2) = 0011_2 = 3$. The predicate `bitrev(L, P)` accepts a list L of length 2^n , and unifies P with a permutation of L such that the i^{th} element of L is the $bitrev(n, i)^{th}$ element of P . In our tests, we used $n = 16, 17, 18, 19$, and 20.

3.2.5 Dynamic Indexing

These tests include insertion, querying, and deletion of facts based on patterns that are meant to measure the performances of default indexing strategies for the various systems. Each test inserts a sequence of facts one-by-one, queries all the inserted facts, and then deletes the facts one-by-one. Some indexing methods, such as the hash-based methods, are very sensitive to skewed data patterns, since this causes excessive collisions. Prolog semantics may also affect the performance because it is sensitive to the order in which facts have been inserted. Our tests show that XSB is particularly sensitive to data skews and the order of insertions and deletions.

²⁷ <http://www.cs.kuleuven.be/~bmd/PrologProgrammingContests/Contest2007.html>

The fact-sequences have one of the following three patterns: $p(1,i)$, $p(i,1)$, and $p(i,i)$, where p is a predicate name and $1 \leq i \leq n$. For each pattern, the sequences had the sizes 50000, 100000, 250000, 500000, and 1000000.

3.2.6 Interfaces to Database Management Systems

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.²⁸ 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, *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; they measure the total elapsed time since the start of a test till its end (known as *walltime*).

The reported times *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.

²⁸ Some might argue that XML database do something similar already.

4.1 Large Join Tests

The execution times are shown in Tables 2, 3, and 4 (Join1), Table 5 (5*Join1) (Join1 and 5*Join1 test datasets are described in Table 6), Table 7 (Join2), Table 8 (LUBM-derived queries), Table 9 (Mondial), and Table 10 (DBLP). XSB, Yap, SWI, and Ontobroker contested the crown, outperforming each other on some tests and falling behind on others. DLV was sometimes close and sometimes did even 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, all Prolog systems, SWI, XSB, and Yap time out for query $a(X,Y)$ on a large data set (see data1 and data2 in Table 6) in tests shown in Tables 2 and 5. The likely explanation is that this is due to the fact that Prolog-based systems 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. In this case, it uses the sort-merge join [15], which scales better than nested loops. Interestingly, in Table 2 SWI, 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.

system	a			b1			b2		
	data0	data1	data2	data0	data1	data2	data0	data1	data2
ontoBroker	4.089	28.385	155.510	0.213	4.806	38.332	0.019	0.168	3.479
xsb	15.906	timeout	timeout	0.310	20.699	timeout	0.034	0.860	15.772
yap	10.534	timeout	timeout	0.109	12.123	timeout	0.013	0.269	8.325
dlv	85.698	850.719	1160.265	7.235	60.873	290.874	0.677	9.260	69.704
jena	139.570	timeout	timeout	3.095	169.655	timeout	0.485	5.756	97.416
drools	error	error	error	25.703	error	error	2.406	41.912	error
jess	310.000	timeout	error	12.000	317.000	error	1.000	26.000	error
owlim	105.426	1521.430	timeout	2.133	85.361	550.693	0.142	5.022	101.633
iris	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
swi	10.530	timeout	timeout	0.100	10.040	timeout	0.010	0.180	14.890

Table 2: Join1, no query bindings

For the tests where one of the arguments in the query is bound to a constant ($=1$), in Tables 3, and 4, Jess, Drools, and OWLIM are omitted, as they cannot take advantage of query bindings and so their times are comparable

to Table 2. 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 3, XSB, Yap, and SWI 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 3 and for the queries $a(X,1)$ and $b1(X,1)$ in Table 4 do not change substantially when the dataset size increases from 50000 (data0) to 250000 tuples (data1). 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).

system	a			b1			b2		
	data0	data1	data2	data0	data1	data2	data0	data1	data2
ontoBroker	0.035	0.038	0.175	0.013	0.051	0.149	0.070	0.012	0.034
xsb	0.015	40.156	timeout	0.001	0.018	3.073	0.000	0.000	0.012
yap	0.021	30.233	timeout	0.007	0.050	2.310	0.004	0.025	0.189
dlv	0.275	5.916	68.643	0.014	0.140	0.776	0.006	0.062	0.496
jena	0.492	332.292	timeout	0.026	0.349	17.739	0.013	0.036	0.247
iris	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
swi	0.010	26.160	timeout	0.000	0.020	3.120	0.010	0.000	0.050

Table 3: Join1 with 1st 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 2 and 5, 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

system	a			b1			b2		
	data0	data1	data2	data0	data1	data2	data0	data1	data2
ontoBroker	0.030	0.036	0.229	0.011	0.010	0.045	0.028	0.009	0.030
xsb	4.997	timeout	timeout	0.049	4.754	1586.839	0.005	0.092	6.404
yap	0.814	539.710	timeout	0.014	0.246	8.557	0.004	0.025	0.165
dlv	0.540	59.642	288.908	0.021	0.328	15.943	0.006	0.063	0.504
jena	73.849	timeout	timeout	1.168	73.834	timeout	0.331	1.857	44.197
iris	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
swi	4.330	timeout	timeout	0.040	6.340	timeout	0.010	0.110	10.370

Table 4: Join1 with 2nd argument bound

queries have common subexpressions (4-way joins), and factors them out. So, for Ontobroker, Join1 and 5*Join1 are essentially the same query.

system	a			b1			b2		
	data0	data1	data2	data0	data1	data2	data0	data1	data2
ontoBroker	3.647	28.497	156.138	0.163	4.601	38.556	0.029	0.189	3.533
xsb	61.879	timeout	timeout	0.601	71.419	timeout	0.061	1.392	54.956
yap	52.202	timeout	timeout	0.522	61.650	timeout	0.054	1.246	39.287
dlv	418.555	timeout	timeout	28.313	292.042	1443.781	2.471	35.380	339.457
jena	734.951	timeout	timeout	9.682	867.672	timeout	1.428	21.542	486.712
drools	error	error	error	error	error	error	14.462	error	error
jess	error	error	error	65.000	error	error	6.000	error	error
owlim	688.103	timeout	timeout	16.279	590.559	timeout	0.901	42.844	810.811
iris	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
swi	48.060	timeout	timeout	0.410	63.050	timeout	0.040	1.120	63.140

Table 5: 5*Join1, no query binding

For Join2, XSB and Yap take about the same time, while SWI and Ontobroker come in third and fourth. SWI does worse than XSB and Yap because the latter take advantage of tabling and avoid repeated computation, while SWI does not have this option. 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 Prolog engines an advantage.

system	ontoBroker	xsb	yap	dlv	jess	drools	iris	swi
time	11.935	2.021	2.087	49.540	timeout	error	timeout	5.500

Table 7: Times for Join2

query size	Query1		Query2		Query9	
	10univs	50univs	10univs	50univs	10univs	50univs
ontobroker	0.014	0.024	0.333	2.615	1.022	8.493
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
swi	0.070	0.399	362.000	timeout	589.000	timeout

Table 8: Times for LUBM tests

data files	domain size	base relation size
data0	1000	10000
data1	1000	50000
data2	1000	250000

Table 6: Join1 and 5*Join1 test datafiles

In the LUBM tests, Drools and Jess ran out of memory while loading the larger of the two data sets (for 50 universities this test has over six million tuples and takes 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 heuristics 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, Drools, and Jess.

system	xsb	yap	ontobroker	jess	dlv	drools	cyc	swi	iris
time	0.004	0.037	0.042	2.000	0.890	3.476	timeout	118.020	timeout

Table 9: Times for Mondial

system	ontobroker	xsb	yap	drools	dlv	jess	cyc	swi	iris
time	1.602	1.752	2.447	0.186	2.041	error	timeout	0.660	timeout

Table 10: Times for DBLP

SWI’s poor performance is due to the fact that it does not support tabling and is forced to perform repeated identical computations.

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, SWI, 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. For comparison, the same query takes 18.966 seconds for MySQL if indexing is built on the first argument.

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 re-computation and indices are used for faster access to data. With these optimizations, Table 9 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!

4.2 Datalog Recursion

We examined the runtime for the transitive closure query **tc** (the results are described in Tables 11, 12 and 13, and the test datasets in Table 14) and the same generation query **sg** (see the results in Tables 15, 16 and 17, and the test datasets in Table 18) 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.²⁹ In addition, we looked at the cases when the base relation **par** is an acyclic graph 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. We did not include SWI in these tests, since, being a pure Prolog system, it runs into an infinite loop for many of the tests.

system	data0	data1	data2	data3	data4	data5	data6	data7	data8	data9
ontoBroker	6.129	19.145	29.172	95.970	49.722	182.633	156.202	443.891	277.840	911.730
xsb	2.725	7.081	15.322	38.205	35.036	88.028	61.042	148.582	136.213	323.492
yap	2.066	13.026	13.690	48.934	33.128	82.900	48.979	269.419	128.128	428.680
dlv	21.673	74.512	89.084	331.483	128.215	880.706	380.292	1521.722	651.966	timeout
jena	16.332	47.674	66.291	222.092	108.940	451.590	325.269	1024.108	584.158	timeout
drools	120.416	error	error	error	error	error	error	error	error	error
jess	55.000	184.000	242.000	1102.000	411.000	timeout	1350.000	timeout	timeout	timeout
owlim	8.549	35.717	42.176	189.332	73.486	393.178	603.174	timeout	1165.760	timeout

Table 11: Transitive closure, no query bindings

system	data0	data1	data2	data3	data4	data5	data6	data7	data8	data9
ontoBroker	5.708	21.787	25.145	108.835	47.804	180.758	224.979	510.653	361.114	899.843
xsb	1.437	5.563	8.922	27.921	18.532	55.447	35.473	112.490	73.373	226.064
yap	0.722	5.666	4.327	19.675	9.026	31.454	16.945	106.590	35.259	169.589
dlv	17.230	64.031	74.162	285.374	109.321	507.360	362.113	1282.090	638.588	timeout
jena	3.884	11.593	19.694	60.170	35.383	122.466	84.581	239.950	156.934	479.196

Table 12: Transitive closure, 1st argument bound.

²⁹ OWLIM, Jess, and Drools are omitted for queries with argument bindings, as these systems cannot benefit from such information.

system	data0	data1	data2	data3	data4	data5	data6	data7	data8	data9
ontoBroker	0.015	0.054	0.033	0.148	0.042	0.271	0.016	0.255	0.017	0.566
xsb	0.016	0.015	0.088	0.079	0.180	0.157	0.186	0.167	0.371	0.332
yap	0.036	0.048	0.243	0.300	0.575	0.689	0.572	0.692	1.270	1.800
dlv	0.020	0.062	0.165	0.414	0.293	0.840	0.394	0.902	0.729	1.837
jena	0.792	0.884	2.552	2.433	5.314	5.032	6.720	5.341	16.887	11.925

Table 13: Transitive closure, 2nd argument bound.

data files	domain size	par/2 size	cycles in data
data0	1000	50000	no
data1	1000	50000	yes
data2	1000	250000	no
data3	1000	250000	yes
data4	1000	500000	no
data5	1000	500000	yes
data6	2000	500000	no
data7	2000	500000	yes
data8	2000	1000000	no
data9	2000	1000000	yes

Table 14: Transitive closure test datafiles

system	data0	data1	data2	data3	data4	data5	data6	data7	data8	data9
ontoBroker	1.499	1.923	2.691	3.274	5.234	5.548	2.863	6.943	7.224	10.150
xsb	2.359	3.408	10.197	11.905	42.824	44.487	1.410	4.671	9.673	14.342
yap	1.875	3.148	8.929	11.339	43.510	43.452	0.858	4.042	7.292	13.224
dlv	19.901	31.419	86.164	109.201	396.417	435.069	11.074	45.673	73.487	141.776
jena	21.007	37.692	85.526	106.461	387.268	415.376	12.647	43.570	78.708	136.261
drools	104.884	error	error	error	error	error	59.623	error	error	error
jess	64.000	error	324.000	error	1517.000	error	31.000	error	312.000	error
owlim	5.279	10.541	29.181	40.254	130.378	156.194	4.211	36.498	52.369	134.917

Table 15: Same generation, no query bindings.

system	data0	data1	data2	data3	data4	data5	data6	data7	data8	data9
ontoBroker	0.918	1.720	1.707	2.772	4.122	5.166	0.481	7.197	4.075	10.094
xsb	1.338	3.245	7.954	11.305	35.823	42.297	0.045	4.329	5.514	13.727
yap	0.619	1.398	3.120	4.993	18.238	18.866	0.066	1.791	1.186	5.916
dlv	12.322	30.044	57.496	106.105	328.246	420.572	2.309	39.475	33.636	131.668
jena	6.827	15.959	33.615	54.404	155.512	191.842	2.007	14.233	17.116	59.023

Table 16: Same generation, 1st argument bound.

system	data0	data1	data2	data3	data4	data5	data6	data7	data8	data9
ontoBroker	0.947	1.748	1.958	2.920	4.562	5.380	0.449	6.934	4.440	10.167
xsb	2.859	3.954	13.307	14.488	55.855	51.406	1.690	5.673	12.828	18.478
yap	0.845	1.539	3.779	5.253	19.178	19.450	0.380	2.125	3.309	6.550
dlv	14.987	39.339	71.902	125.865	402.087	502.053	3.108	84.025	53.535	280.852
jena	15.784	21.275	65.195	68.963	251.355	254.322	8.729	22.858	52.894	86.166

Table 17: Same generation, 2nd argument bound.

data files	domain size	par/2 size	sib/2 size	cycles in data
data0	500	5000	1000	no
data1	500	5000	1000	yes
data2	500	10000	2000	no
data3	500	10000	2000	yes
data4	500	20000	4000	no
data5	500	20000	4000	yes
data6	1000	5000	1000	no
data7	1000	5000	1000	yes
data8	1000	10000	2000	no
data9	1000	10000	2000	yes

Table 18: Same Generation test datafiles

Yap is the best overall performer in the WordNet tests, followed by XSB then Ontobroker. Ontobroker does better than XSB for the *meronyms*, *holonyms* and *troponyms* tests. We conjecture that this is due to Ontobroker’s cost-based optimizer, which may have found a better query execution plan than what we could do manually for XSB.

system	allSameSynset	allGlosses	allDirectHypernyms	allHypernyms	allDirectHyponyms
cyc	error	error	error	error	error
dlv	5.61 (7.04)	3.15 (5.6)	4.67 (6.58)	33.65 (35.48)	4.67 (6.62)
drools	33.502	28.933	109.906	timeout	111.765
iris	1498.546	1463.559	timeout	timeout	timeout
jena	1.570	1.090	1.350	7.613	1.340
jess	2.730	1.370	1.960	timeout	1.960
ontoBroker	0.283	0.198	0.543	6.888	0.605
owlim	0.828 (5.05)	0.184 (5.937)	0.693 (5.44)	5.713 (10.429)	0.93 (5.579)
prova	timeout	timeout	timeout	timeout	timeout
swi	0.2	0.15	0.2	N/A	203.86
xsb	0.2200	0.1600	0.3800	1.5240	0.3760
yap	0.496	0.224	0.156	0.244	0.168

Table 19: Computing Time for WordNet tests 1

system	allSameSynset	allGlosses	allDirectHypernyms	allHypernyms	allDirectHyponyms
cyc	error	error	error	error	error
dlv	1.43	2.45	1.91	1.83	1.95
drools	N/A	N/A	N/A	N/A	N/A
iris	N/A	N/A	N/A	N/A	N/A
jena	N/A	N/A	N/A	N/A	N/A
jess	N/A	N/A	N/A	N/A	N/A
ontoBroker	19.222	28.637	23.926	24.163	24.06
owlim	4.222	5.753	4.747	4.716	4.649
prova	N/A	N/A	N/A	N/A	N/A
swi	1.9	3.12	2.49	N/A	2.42
xsb	0.8080	1.4600	1.0400	1.0370	1.0410
yap	1.44	2.46	1.912	2.02	2.06

Table 20: Loading Time for WordNet tests 1

system	allHyponyms	allDirectMeronyms	allMeronyms	allDirectHolonyms	allHolonyms
cyc	error	error	error	error	error
dlv	35.92 (37.9)	1.86 (3.44)	7.46 (9.11)	1.46 (3.15)	7.2 (8.88)
drools	timeout	9.969	timeout	12.305	timeout
iris	timeout	1692.542	timeout	1685.602	timeout
jena	14.291	0.830	2.023	0.770	2.234
jess	timeout	0.510	4.060	0.520	4.050
ontoBroker	6.611	0.086	1.074	0.923	0.964
owlim	5.164 (9.987)	0.008 (4.343)	1.158 (5.373)	0.247 (4.413)	1.485 (5.465)
prova	timeout	timeout	timeout	timeout	timeout
swi	N/A	14.44	N/A	0.09	N/A
xsb	1.8680	0.1440	0.6880	0.1240	0.6120
yap	0.236	0.148	0.216	0.164	0.224

Table 21: Computing Time for WordNet tests 2

system	allHyponyms	allDirectMeronyms	allMeronyms	allDirectHolonyms	allHolonyms
cyc	error	error	error	error	error
dlv	1.98	1.58	1.65	1.69	1.68
drools	N/A	N/A	N/A	N/A	N/A
iris	N/A	N/A	N/A	N/A	N/A
jena	N/A	N/A	N/A	N/A	N/A
jess	N/A	N/A	N/A	N/A	N/A
ontoBroker	24.166	20.319	19.914	20.231	19.942
owlim	4.823	4.335	4.215	4.166	3.98
prova	N/A	N/A	N/A	N/A	N/A
swi	N/A	2.02	N/A	1.98	N/A
xsb	1.0130	0.8530	0.8320	0.8410	0.8330
yap	2.072	1.516	1.548	1.54	1.604

Table 22: Loading Time for WordNet tests 2

System	ontobroker	xsb	yap	dlv	jess	owlim	jena	drools
time	5.23	4.47	12.05	21.067	140	error	error	error

Table 25: The Wine ontology test.

system	allDirectTroponyms	allTroponyms	allDirectAdjectiveClusters	allAdjectiveClusters	allAntonyms
cyc	error	error	error	error	error
dlv	1.18 (2.7)	1.39 (2.86)	1.75 (3.63)	10.35 (11.97)	1.41 (2.89)
drools	0.816	0.911	7.636	9.187	2.368
iris	1476.519	timeout	1852.489	timeout	1640.491
jena	0.550	0.059	0.800	2.892	1.060
jess	0.079	0.094	0.053	6.510	0.290
ontoBroker	0.031	0.042	0.267	1.23	0.206
owlim	-0.017 (4.14)	0.123 (4.102)	0.857 (5.043)	1.264 (5.731)	0.284 (4.32)
prova	timeout	timeout	timeout	timeout	timeout
swi	0.08	N/A	0.1	N/A	0.1
xsb	0.0680	0.3400	0.1680	0.7440	0.1240
yap	0.152	0.216	0.128	0.176	0.356

Table 23: Computing Time for WordNet tests 3

system	allDirectTroponyms	allTroponyms	allDirectAdjectiveClusters	allAdjectiveClusters	allAntonyms
cyc	error	error	error	error	error
dlv	1.52	1.47	1.88	1.62	1.48
drools	N/A	N/A	N/A	N/A	N/A
iris	N/A	N/A	N/A	N/A	N/A
jena	N/A	N/A	N/A	N/A	N/A
jess	N/A	N/A	N/A	N/A	N/A
ontoBroker	19.28	20.103	21.005	21.875	20.266
owlim	4.157	3.979	4.186	4.467	4.036
prova	N/A	N/A	N/A	N/A	N/A
swi	1.96	N/A	2.08	N/A	1.92
xsb	0.8040	0.8170	0.8880	0.8810	0.8130
yap	1.516	1.5	1.604	1.664	1.564

Table 24: Loading Time for WordNet tests 3

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.

4.3 Default Negation

One can see from Table 26 that for stratified rule sets (whose corresponding test datasets are described in Table 27), 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-non-win test on smaller datasets, but Drools has a problem with larger sets. Both fail for the modified same generation test. We did not include SWI, since it runs indefinitely for most of the tests. Again, this is due to the lack of support for tabling.

system	win-non-win					same generation				
	data0	data1	data2	data3	data4	data0	data1	data2	data3	data4
ontoBroker	1.327	3.175	9.988	17.482	timeout	14.883	24.963	timeout	51.957	timeout
xsb	0.231	0.470	1.218	2.475	5.081	7.265	24.215	90.928	10.947	30.052
yap	0.103	0.239	0.654	1.401	2.866	3.339	11.673	44.605	4.689	13.866
dlv	0.674	1.406	3.558	7.232	14.917	37.116	118.545	447.397	62.552	165.182
jess	1.000	1.000	3.000	7.000	15.000	error	error	error	error	error
drools	1.288	2.333	7.466	14.331	error	error	error	error	error	error
prova	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout

Table 26: Locally- and predicate-stratified negation.

data files	upper bound
data0	50000
data1	100000
data2	250000
data3	500000
data4	1000000

Table 27: Win-non-win test datafiles

Yap does not support rule sets that are not stratified locally, so it is not included in Table 28 (the datasets used in the tests are described in Table 29). The tests show that XSB is significantly faster and scales better than the other 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.

system	win-non-win					magic set				
	data0	data1	data2	data3	data4	data0	data1	data2	data3	data4
ontoBroker	0.419	1.224	3.754	8.226	17.237	0.236	1.426	8.409	3.236	21.502
xsb	0.339	0.698	1.416	2.857	5.647	1.381	1.476	1.663	error	error
dlv	0.348	0.753	1.892	3.995	8.223	0.084	0.386	2.041	0.963	5.009

Table 28: Times for rule sets that are not locally stratified.

data files	domain size	h/2 and b/2 relation size
data0	1000	10000
data1	1000	50000
data2	1000	250000
data3	5000	100000
data4	5000	500000

Table 29: Magic Sets transformation test datafiles

4.4 Tests Involving Function Symbols and Database Updates

For the 16 puzzle problem, DLV moved to the first place, followed by Yap, SWI and XSB. DLV was more than three times faster than Yap, about ten times faster than SWI, and twenty times faster than XSB when the limit on the number of steps in the game is 20 or less. When the limit is 22 steps, DLV is still about ten times faster than SWI and XSB, and slightly faster than Yap. We believe that this is due to a different algorithm used by DLV: this test was rewritten for DLV so it could take advantage of its answer-set programming capabilities. In case of Ontobroker, although it can express all the rule sets, these tests favor top-down processing for which Ontobroker is ill-suited; it runs out of memory and is not shown in the tables.

Yap is the best performer in the N-Queens test. XSB takes almost twice the time, but is still far ahead of SWI, DLV and Prova. The reason behind the better performance of Yap and XSB compared to DLV might be that the former uses heuristic search, while DLV uses answer-set programming. In

this particular test, answer-set programming results in a brute-force search through all possible configurations.

The BitRev test does not use function symbols, but it requires databases updates. Yap performed best followed by XSB and then by SWI and Jess.

system	14	16	18	20	22
xsb	5.307	9.115	17.646	30.498	66.530
yap	0.769	1.428	2.602	4.513	9.816
ontoBroker	timeout	timeout	timeout	timeout	timeout
dlv	0.209	0.304	0.507	1.607	7.468
swi	3.330	5.710	10.990	18.770	42.150

Table 30: Times for 16 Puzzle

system	8	10	12	14	16
xsb	0.005	0.135	3.822	147.430	timeout
yap	0.002	0.054	1.507	58.152	timeout
dlv	0.093	1.301	33.225	1120.251	timeout
prova	timeout	timeout	timeout	timeout	timeout
swi	0.030	0.840	26.520	1062.650	error

Table 31: Times for N-Queens

system	16	17	18	19	20
xsb	6.812	14.903	33.949	75.328	error
yap	1.506	4.383	17.032	54.893	141.161
jess	18.000	37.000	78.000	163.000	error
swi	9.460	error	error	error	error

Table 32: Times for BitRev

4.5 Dynamic Indexing

For dynamic indexing (see Tables 33, 34, 35, and 36), Yap and SWI are the best performers, followed by Jena, DLV, OWLIM, and then Drools, Jess, Ontobroker in another batch. Prova failed to retrieve and delete all inserted facts. The default indexing of XSB (hash index on the first argument) was the worst choice for inserting and deleting data with the pattern $p(1, i)$. This caused the creation of an overflow chain, and each time a new fact was inserted, the entire chain had to be traversed. DLV also has hash indexing on

the first argument, but its performance was good for the same pattern. The reason for this difference is the Prolog semantics used by XSB, since Prolog’s *assert* puts the newly inserted facts at the end of the fact list (this is the reason for traversing the entire overflow chain). If we used *asserta* instead, the result would have been comparable to DLV.

system	data0	data1	data2	data3	data4	data5	data6	data7	data8	data9	data10	data11	data12	data13	data14
dlv	0.221	0.219	0.211	0.440	0.433	0.429	1.125	1.102	1.109	2.272	2.218	2.238	4.629	4.526	4.590
drools	0.611	0.184	0.184	1.209	0.650	0.694	3.203	2.822	2.501	7.861	7.320	6.063	16.108	13.303	13.033
jena	0.415	0.202	0.136	0.228	0.274	0.408	0.896	1.209	1.204	2.281	1.772	3.053	4.701	4.995	5.843
jess	1.000	1.000	2.000	2.000	1.000	2.000	4.000	4.000	4.000	9.000	10.000	10.000	20.000	20.000	19.000
ontobroker	1.015	1.009	0.998	1.565	1.557	1.540	3.837	3.703	3.738	8.551	8.929	7.201	18.239	18.896	16.760
owlim	0.712	0.703	0.726	0.894	0.852	0.827	1.296	1.298	1.358	1.914	2.229	2.319	3.951	4.006	3.904
prova	2.343	2.313	2.297	4.484	4.594	4.344	10.969	10.922	10.985	21.703	22.031	22.015	48.234	48.938	47.877
xsb	6.208	0.232	0.238	370.058	0.491	0.476	491.984	1.222	1.210	timeout	2.476	2.471	timeout	22.253	0.442
yap	0.044	0.045	0.044	0.064	0.059	0.062	0.162	0.172	0.164	0.308	0.353	0.324	0.733	0.708	0.786
swi	0.050	0.050	0.020	0.070	0.080	0.040	0.260	0.200	0.210	error	0.440	0.440	0.770	0.780	0.880

Table 33: Times for insertion

system	data0	data1	data2	data3	data4	data5	data6	data7	data8	data9	data10	data11	data12	data13	data14
dlv	0.141	0.119	0.138	0.305	0.272	0.293	0.831	0.710	0.720	1.924	1.610	1.554	3.671	3.325	3.229
drools	0.246	0.154	0.149	0.304	0.327	0.328	0.835	0.791	0.806	1.714	1.710	1.733	3.611	3.544	3.600
jena	3.541	0.001	0.000	5.760	0.001	0.000	14.550	0.001	0.000	28.164	0.000	0.000	57.927	0.000	0.000
jess	1.000	1.000	0.000	1.000	1.000	0.000	2.000	1.000	2.000	2.000	2.000	2.000	4.000	5.000	6.000
ontobroker	0.319	0.326	0.347	0.297	0.299	0.329	0.294	0.297	0.330	0.323	0.295	0.294	0.312	0.296	0.340
owlim	0.124	0.128	0.131	0.139	0.154	0.219	0.213	0.213	0.178	0.941	0.339	0.321	1.267	1.056	1.252
xsb	0.002	0.002	0.002	0.004	0.004	0.004	0.010	0.010	0.010	timeout	0.020	0.019	timeout	0.091	0.039
yap	0.007	0.006	0.007	0.017	0.016	0.016	0.040	0.039	0.039	0.082	0.083	0.082	0.160	0.156	0.155
swi	0.000	0.000	0.000	0.010	0.000	0.010	0.030	0.030	0.030	error	0.080	0.080	0.160	0.180	0.160

Table 34: Times for querying

system	data0	data1	data2	data3	data4	data5	data6	data7	data8	data9	data10	data11	data12	data13	data14
drools	0.829	0.596	0.804	1.180	1.613	1.609	4.098	4.241	4.235	6.953	8.133	6.844	14.642	14.206	14.543
jena	0.294	0.183	0.271	0.270	0.455	0.494	0.661	1.633	1.509	2.379	2.477	2.340	5.332	5.806	6.906
jess	1.000	1.000	1.000	2.000	2.000	3.000	4.000	5.000	4.000	8.000	8.000	9.000	16.000	16.000	17.000
ontobroker	0.570	0.558	0.556	1.174	1.201	1.149	2.956	3.008	2.999	5.198	5.775	5.650	12.626	11.863	12.288
owlim	0.096	0.100	0.112	0.133	0.123	0.120	0.286	0.155	0.274	0.304	0.410	0.448	1.121	0.477	0.538
prova	timeout	8.453	8.437	timeout	27.610	27.578	timeout	152.953	152.453	timeout	642.062	638.485	timeout	timeout	timeout
xsb	1641.781	0.196	0.195	timeout	0.388	0.390	timeout	0.977	0.971	timeout	2.160	2.167	timeout	15.418	15.218
yap	0.029	0.024	0.024	0.069	0.057	0.057	0.162	0.132	0.132	0.442	0.360	0.363	0.944	0.782	0.784
swi	error	0.040	0.040	error	0.080	0.090	error	0.200	0.210	error	0.410	0.380	error	0.760	0.760

Table 35: Times for deletion

data files	upper bound	pattern
data0	50000	(1,i)
data1	50000	(i,1)
data2	50000	(i,i)
data3	100000	(1,i)
data4	100000	(i,1)
data5	100000	(i,i)
data6	250000	(1,i)
data7	250000	(i,1)
data8	250000	(i,i)
data9	500000	(1,i)
data10	500000	(i,1)
data11	500000	(i,i)
data12	1000000	(1,i)
data13	1000000	(i,1)
data14	1000000	(i,i)

Table 36: Data files for testing dynamic indexing

5 Conclusions

This report presents 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. 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 Grosf 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.

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. *Artificial 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. Przymusiński. 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 *First International Conference 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.