Performance Analysis and Comparison of Deductive Systems and SQL Databases

Stefan Brass and Mario Wenzel

Universität Halle, Institut für Informatik, 06099 Halle (Saale),
brass@informatik.uni-halle.de, mario.wenzel@informatik.uni-halle.de

Abstract. We compare the performance of logic programming systems with tabling (XSB, YAP) and new Datalog implementations (Soufflé, our BAM) with the performance of relational database systems (SQLite, PostgreSQL, MariaDB). These databases support recursive views, and can compute, e.g. the transitive closure and same generation cousins. We also tried the RDF store Apache Jena and the graph database Neo4j. The main contributions compared to previous benchmarking approaches like OpenRuleBench [4] are: (1) We created a number of additional graphs of different structure when we identified a problem with the OpenRuleBench test graphs. (2) We developed a database to store and analyze benchmark measurements. It is much more general than the previous result data files, and supports detailed evaluations of the data. (3) We propose a formula to predict the runtime of the transitive closure benchmark based on parameters of the input graph. This work on runtime prediction is only the beginning of using machine learning techniques on runtime data.

Keywords: Performance, Benchmarks, Deductive Databases, Recursive Queries, SQL

1 Introduction

Benchmarks can be a useful tool to compare performance between systems and evaluate how well different kinds of systems and evaluation schemes perform against each other. Especially comparing different models of computation and evaluation schemes can be helpful for system developers in identifying shortcomings and furthering cross-pollination between disciplines.

In the area of deductive databases/rule-based systems, OpenRuleBench [4] is a well-known benchmark suite. It is a quite large collection of problems, basically 12 logic programs, but some with different queries and different data files. In contrast to database benchmarks, the OpenRuleBench benchmarks contain no updates. They assume that the input data is given in a file (in a format suitable for the tested system), and the task is to evaluate some query.

However, the OpenRuleBench scripts run each benchmark only once, and do not support comparing runtimes of the same benchmark for different option settings or program versions, or even on different machines. We have developed

a database and new benchmark execution scripts in order to overcome these restrictions. The scripts measure all runtimes externally, and collect also information about the used main memory (where possible). Furthermore, the test graphs used in OpenRuleBench for the transitive closure have a very specific (dense) structure, and even graphs declared as non-cyclic contain loops. We solved this problem and used a much larger collection of different test graphs.

In this paper, we show results for only two of the OpenRuleBench benchmark problems, namely the transitive closure, and the same generation cousins (our project web pages contain some more benchmark results). However, we analyze the benchmark results in detail. In particular, we investigate formulas to predict the runtime of a system based on characteristics of the graph. We believe that for declarative programming, it is important that one can get a rough estimate of the runtime on an abstract level (without need to understand implementation details). At least, one wants to be sure that for all inputs that might occur (even larger ones), the runtime will be acceptable. If the runtime would suddenly "explode" for specific inputs, the system would be unreliable.

Database optimizers have done runtime estimation for a long time. But this uses internal information of the system, such as the availability of indexes and sizes of data structures. We took a "black box" approach and measure runtime, and try to approximate the results with a formula based on characteristics of the given problem (input and output). If this works, it helps to understand the runtime behaviour of the system, and gives the user some trust into its reliable performance. Furthermore, it helps to "compress" the many single numbers for the runtime measurements into a few understandable parameters of the formula. This compression is not lossless, but it helps to see the "big picture".

The runtimes of transitive closure in relational databases have already been investigated in [6], but there has been progress in the implementations of relational database systems. For instance, the number of iterations, which used to be important eight years ago, has become less important now. Today, it is common that even open source database systems support "recursive common table expressions" (recursive views), whereas earlier this was restricted to the big players (e.g., DB2). So far, we used SQLite, PosgreSQL, and MariaDB for our performance tests.

We are especially interested in performance and performance measurements since we are developing an implementation of our "Push" method for bottom-up evaluation of Datalog [1, 2]. We recently defined an abstract machine "BAM" for the bottom-up evaluation of Datalog, and implemented a first prototype [3]. "Pushing" tuples through relational algebra expressions has also been done very successfully for standard databases.

There is a renewed interest in Datalog, probably mostly due to new applications. E.g. the Soufflé system [8] uses Datalog for the static analysis of Java programs. It is also a fast new Datalog implementation.

We also checked classical logic programming systems, in particular, XSB [7]. This supports full Prolog, but with tabling, such that termination is guaranteed for Datalog. It has beaten many older deductive database prototypes.

2 A Benchmark Database

The OpenRuleBench scripts [4] run each benchmark only once. However, it is common practice to execute benchmark programs several times. There is some variation in the runtimes of the same program on the same data on the same machine. Since we are developing our own system, we want a reliable indication of the effects of changes in the implementation, even if they are not dramatic.

When measuring other systems, we feel obliged to do some tests with different option settings (or different indexes) and choose the best settings we could find (with our limited expertise). Thus, the situation gets more complicated than in the OpenRuleBench, since we have different implementation variants of the same benchmark for the same system.

Furthermore, as developers we use different machines with different operating systems. In order to keep the overview of the different runtime measurements, we decided to develop a benchmark database. The database of measurements also allows us to easily run statistical analyses on any kind of dataset.

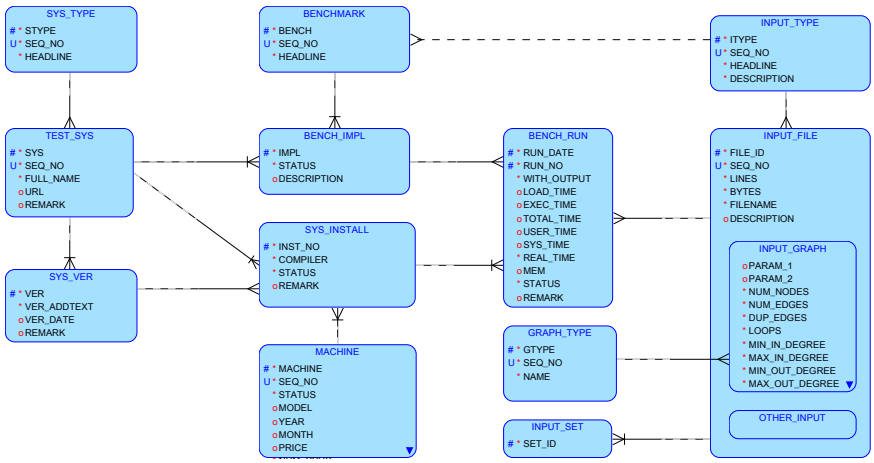

Fig. 1. ER-diagram of benchmark database

We also implemented a number of views to analyze the data. The generation of tables in LaTeX and HTML format is done by means of SQL queries. Our views also help to check for outliers to get more confidence in the data. One can also query for the best implementation variant of a benchmark for a system.

3 Transitive Closure and Same Generation

For space reasons, we can discuss only two benchmark problems in this paper. The first is the well known task to compute the transitive closure of a binary

relation, i.e. the reachability relation between nodes in a graph, where the edges are given. The standard tail-recursive formulation in Datalog/Prolog is:

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

The name "`par`" stands for "parent" or "parameter" (of `tc`). It is the edge relation of a directed graph. The nodes in the test graphs (see Section 4) are integers, thus a large set of facts of the form `par(1,2)` is given. The benchmark is to compute the entire `tc` relation, i.e. the predicate `tc` is queried with both arguments "free". Therefore this benchmark is called "TCFF".

The second problem is the "same generation" problem ("SGFF"), which is also part of the OpenRuleBench collection [4]. It can be formulated based on the same edge relation "`par`": Two vertices are in the same generation if they each have a path of the same length connecting to a common vertex:

```
sg(Y,Y) :- par(_, Y).
sg(X,Y) :- par(X,A), sg(A,B), par(Y,B).
```

While this program first looks very similar to the transitive closure, performance is quite different, because the program is not tail-recursive.

Since many of the test graphs are cyclic, a standard Prolog system would not terminate with these programs. We need systems that support tabling to detect repeated calls to the recursively defined predicate. We used XSB version 3.8.0, and YAP version 6.2.2.

XSB has some tuning parameters, and we tried ten different settings for the same generation benchmark. The best we could get was using subsumptive tabling, `load_dync` for loading the data, and an index on `par(2,1)`. For this benchmark, the trie index gave bad results.

We also tried the Soufflé system [8]. It is intended for static analysis of Java programs, but it is also a fast Datalog evaluator. We used version 1.5.1. We ran our tests with only a single job enabled to have a better comparison to the other systems that do not parallelize the query evaluation. We used the version that compiles via C++ to native code. The interpreted version is much slower.

Of course, native code gives an advantage over systems based on an abstract machine, such as XSB, YAM, and our "Bottom-Up Abstract Machine" BAM. For Prolog systems, a rule of thumb is that native code gives on average a factor of 3 over emulators of the Warren abstract machine. Experiments we did with a native code implementation of our Push method vs. our BAM showed only a factor of 1.5–2.

3.1 Relational Databases

Recursive view definitions were introduced in the SQL-99 standard. Most modern relational database systems support them, and therefore can compute the transitive closure.

The query is (using a recursive "common table subexpression", i.e. a local view definition):

```
WITH RECURSIVE tc(a,b) AS (
  SELECT par.a, par.b from par
  UNION
  SELECT par.a, tc.b from par JOIN tc ON par.b = tc.a
) SELECT Count(*) FROM tc;
```

The query for the same generation problem is analogous. As a table definition for the `par`-table, we used the following definition, with slight alterations and additions depending on the used database system.

```
CREATE TEMP TABLE par (
  a INT NOT NULL, b INT NOT NULL,
  CONSTRAINT par_pk PRIMARY KEY (a, b)
) WITHOUT ROWID;
CREATE INDEX par_fb ON par (b);
```

For SQLite, we used a temporary in-memory table for the `par` relation. We added `PRAGMA temp_store = MEMORY;` in order to deactivate using temporary files. We used SQLite3 version 3.24.0.

PostgreSQL is a server-based database system. As CPU time, we record in our database the time the server process of the session used. However, the main comparison is done on real time, which is simpler for server-based systems (it also gives parallel implementations an advantage). The startup time of the server is not included in the time measurements. We used version 9.4.15.

MariaDB is the community fork of the server-based MySQL database system. We used both the `InnoDB` storage engine and the `Memory` storage engine for the `par` relation. For the latter we set `max_head_table_size` to $4*1024^3$. The support of recursive CTEs is new (since version 10.2.2). We used version 10.3.9.

3.2 Graph Databases / RDF Triple Stores

Transitive Closure and Same Generation are included in the OpenRuleBench suite for the graph database system Apache Jena using the Java API for the included OWL reasoner. Since SPARQL 1.1 added Property Paths [9] where non-variable predicates can be combined in a similar fashion to regular expressions, we can formulate the Transitive Closure now directly in SPARQL:

```
SELECT (count(*) as ?resultcount) WHERE {?a :par+ ?b}
```

This does not work for the same generation, because there we need two paths of the same length. For the transitive closure benchmark, we used Apache Jena version 3.8.0 on OpenJDK 1.8 (Java 8), and had to increase the JVM's thread stack size to 16MB (`JVM_ARGS=-Xss16m`).

Since the two benchmark problems are graph problems, we also tried the well-known graph database Neo4j. However, the following query in the graph query language Cypher did not give competitive performance:

```
MATCH (a)-[*]->(b)
RETURN DISTINCT a,b;
```

There are other possibilities to formulate this query (e.g., via shortest paths), but they do not work for arbitrary graphs.

It is possible to formulate the same generation problem in Cypher, but since one computes first arbitrary pairs of paths ending in the same node, and then selects paths of equal length, the performance is bad. One would need a clever query optimizer.

3.3 Execution Methodology and Hardware

We executed the benchmarks on a HP Blade server with two Intel Xeon CPUs E5-2630 v4@2.20GHz with 10 cores and 20 threads each. As far as we know, Jena is the only one of the tested systems that uses multiple threads for the test query. The machine has 128 GB of RAM. The operating system is Debian `x86_64` GNU/Linux 8.10 (3.16.0).

The overall execution time ("elapsed wall clock time" and CPU time) and the memory ("maximum resident set size") for each test of the standalone programs was measured with the Linux `/usr/bin/time` program.

For the server based systems we took a snapshot of the status information provided by `/proc/[pid]/stat` [5] at the start of the client program and when the query was finished. The difference between both snapshots is used to measure CPU time. This approach can not be used to measure memory usage. We made sure that for MariaDB only one user was connected, as we gathered the status of the server process. PostgreSQL starts a different process for every user session and we measured the stats for that specific process.

Most tests were run ten times and the time average values were calculated.

In the comparison table (Fig. 2 below), we use real time.

4 Test Graphs

The OpenRuleBench collection [4] uses a particular algorithm to generate random graphs that leads to all vertices having nearly identical degrees, compared to just selecting edges at random. Cycle-free graphs are created by ordering the edges towards the greater node. Since this happens after the edges have been selected, it leads to duplicate edges in the generated non-cyclic graphs. Furthermore, reflexive edges (loops) were not excluded for non-cyclic graphs.

Because of this, we developed generators for graphs with properties that are easily analyzed in the context of the given recursive problems. The number of edges in the graph is the input size of the problem. The maximum of the lengths of the shortest path between any two connected vertices is the number of iterations until the fixed point of the T_p operator is found for the transitive closure (N_{tc}). We label vertices incrementally starting from 1.

- The complete graph K_n has n vertices, n^2 edges and $N_{tc}(K_n) = 1$.
  The set of edges is $E = \{(i,j) \mid i = 1,\ldots,n,\ j = 1,\ldots,n\}$.
- The maximum acyclic graph T_n is a subset of K_n without the edges (a,b) with $a \geq b$. It has n vertices, $\frac{n(n-1)}{2}$ edges and $N_{tc}(T_n) = 1$.

- The cycle graph C_n has n vertices, n edges and $N_{tc}(C_n) = n$. $E = \{(i, i+1) \mid i = 1, \ldots, n-1\} \cup \{(n, 1)\}$.
- The cycle graph with shortcuts $S_{n,k}$ is a directed graph that is a superset of the cycle graph C_n where $n - 1 > k > 0$ additional edges per vertex are added, skipping $\frac{n}{k+1}$ vertices in the cycle. n should be divisible by $k+1$. One additional edge per vertex allows skipping $\frac{1}{2}$ of the cycle, while two edges would allow skipping $\frac{1}{3}$ or $\frac{2}{3}$ of the cycle. $N_{tc}(S_{n,k}) = \frac{n}{k+1}$. The set of edges is $E = \{(i,\ 1 + (i - 1 + \frac{nt}{k+1}) \bmod n) \mid t = 1, \ldots, k,\ i = 1, \ldots, n\} \cup C_n$.
- The path P_n has n vertices and $n-1$ edges: $E = \{(i, i+1) | i = 1, \ldots, n-1\}$. $N_{tc}(P_n) = n - 1$. The transitive closure of P_n is T_n.
- The multi-path graph $M_{n,k}$ is a directed graph that contains $n * k$ vertices that form k pairwise vertex-disjoint paths of length $n - 1$. The set of edges is $E = \{(i, i + k) | i = 1, \ldots, (n - 1) * k\}$. $N_{tc}(M_{n,k}) = n - 1$. $M_{n,1}$ is P_n.
- The binary tree graph B_h of height $h > 0$ is a directed graph that contains $2^h - 1$ vertices and $2^h - 2$ edges. $N_{tc}(B_h) = h - 1$. The set of edges is $E = \{(i, 2i) \mid i = 1, \ldots, 2^{h-1} - 1\} \cup \{(i, 2i + 1) \mid i = 1, \ldots, 2^{h-1} - 1\}$.
- The graph V_n is the reverse of B_n (all edges point towards the root).
- The Y-graph $Y_{n,k}$ is a directed graph with $n + k$ vertices. It consists of n vertices with edges to the center vertex $n+1$, which is attached to a path of k vertices: $E = \{(i,\ n+1) \mid i = 1, \ldots, n\} \cup \{(i-1,\ i) \mid i = n+2, \ldots, n+k\}$. The number of iterations for the `tc` computation is $N_{tc}(Y_{n,k}) = k$.
- The W-graph $W_{n,k}$ (with $k \leq n$) has a single level of edges, where each vertex in a set of n vertices points to k vertices from a disjoint set of another n vertices. The graph has $2n$ vertices, $n * k$ edges, and $N_{tc}(W_{n,k}) = 1$. The set of edges is $E = \{(i,\ n + 1 + (i + j - 1) \bmod n) \mid i = 1, \ldots, n,\ j = 1 \ldots, k\}$.
- The X-graph $X_{n,k}$ consists of n vertices pointing to a central vertex, which in turn points to k vertices (the default for k is n). The set of edges is $E = \{(i,\ n+1 \mid i = 1, \ldots, n\} \cup \{(n+1,\ n+1+j) \mid j = 1, \ldots, k\}$. The graph has $n + k + 1$ vertices, $n + k$ edges, and $N_{tc}(X_{n,k}) = 2$.

4.1 A Cost Measure: Applicable Rule Instances

As a first try to estimate the runtime, we look at the number of applicable rule instances in the given logic program. I.e. this is the total number of variable assignments that satisfy the body (condition) of the rules. The standard seminaive evaluation would look at each such rule instance exactly once.

As an example, consider the transitive closure program applied to the complete graph K_{100} (100 vertices, 10 000 edges). The (non-recursive) starting rule

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

is applicable 10 000 times (once for each of the input facts). For the rule

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

there are 10 000 facts in the `par` relation and each one has 100 join partners in the `tc` relation (the number of tuples with a given `Y` value). Thus the total size of

the join `par(X,Y), tc(Y,Z)` is 1 000 000. Of course, in this example, the second rule generates only duplicates, but they must be computed and eliminated. The associated costs for the rules are added and we get a total cost of 1 010 000.

Note that this cost measure is purely declarative, and does not look at the sequence in which facts are computed. For the applicable rule instances, we only need the final version of the `tc`/`sg`-relation in the minimal model.

Graph	`tc` size	`tc` instances	`sg` size	`sg` instances
K_n	n^2	n^2+n^3	n^2	n^2+n^4
T_n	$\frac{n(n-1)}{2}$	$\frac{1}{6}n(n-1)(n+1)$	$1+(n-1)^2$	$\frac{n^4-6n^3+19n^2-22n}{4}+2$
C_n	n^2	n^2+n	n	$2n$
$S_{n,k}$	n^2	$(k+1)(n+n^2)$	n^2 †	$n(k+1)+n^2(k+1)^2$ †
P_n	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	n	$2(n-1)$
$M_{n,k}$	$k\frac{n(n-1)}{2}$	$k\frac{n(n-1)}{2}$	$k*n$	$2k(n-1)$
B_h	$(h-2)*2^h+2$	$(h-2)*2^h+2$	2^h-1	$2^{h+1}-4$
V_h	$(h-2)*2^h+2$	$(h-2)*2^h+2$	$\sum_{i=0}^{h-1}2^{2i}$	$2^h-2+4*\sum_{i=0}^{h-2}2^{2i}$
$Y_{n,k}$	$n*k+\frac{k(k-1)}{2}$	$n*k+\frac{k(k-1)}{2}$	n^2+k	$n^2+n+2k-2$
$W_{n,k}$	$n*k$	$n*k$	$n*\min(2k,n+1)$	$n*(k+k^2)$
$X_{n,k}$	$n+k+n*k$	$n+k+n*k$	$k+1+n^2$	$2k+n+n^2$

† For $k=1 \wedge n \bmod 4 = 2$, `sg` size is $\frac{n^2}{2}$ and `sg` instances is $2(n^2+n)$.

5 Analysis of the Results

5.1 Results for the Transitive Closure Benchmark

The average runtimes of the main systems for the some test graphs are shown in Fig. 2. We give the runtime of XSB in seconds (real time), and for all other systems, we list the factor of the runtime of that system compared to XSB. Thus, an entry less than 1 means that the system is faster than XSB. The average factor at the bottom of the table is determined over all 50 graphs we measured, not only the ones shown in the table. All data are available on our web page.

Although the transitive closure is a typical problem for a logic programming system such as XSB, the relational databases PostgreSQL and SQLite are for most graphs only 3–5 times slower (for PostgreSQL, the worst factor in our TCFF tests was 7.0, for SQLite 9.7). For MariaDB there are five examples where it is more than 100 times slower than XSB. Performance seems to go down for larger problem instances. Soufflé is very fast on some problem instances, but slower than XSB on others (remember that we used Soufflé in single-threaded native-code mode). The numbers for our own prototype look promising, but there are a few problem instances that have potential for further optimization.

Figure 3 shows the relation between the runtime and the cost measure for the graphs. For some systems, such as XSB, the relationship between the cost measure (the number of applicable rule instances) and the runtime looks to be more or less linear. A first try for runtime prediction for XSB and the transitive closure program is to assume a speed of $5.3*10^6$ rule instances per second.

Graph	XSB (s)	BAM	YAP	SQLite	PostgreSQL	MariaDB	Jena	Soufflé
K_{500}	13.342	0.30	0.66	5.07	3.62	3.75	1.92	0.20
K_{1000}	103.266	0.31	0.67	5.56	3.82	3.76	1.82	0.18
T_{500}	2.301	0.62	0.76	4.52	2.84	4.58	3.09	0.31
C_{2000}	1.597	0.18	1.18	4.93	3.28	7.87	5.14	1.55
$S_{2000,1}$	1.844	0.28	1.47	5.43	3.39	9.02	5.03	1.79
P_{4000}	3.145	0.23	1.29	4.81	2.87	31.58	4.33	1.57
$M_{64,128}$	0.283	0.17	0.49	2.25	2.59	2.07	9.28	0.83
$M_{4096,2}$	6.252	0.55	1.44	5.16	2.91	50.63	4.07	1.62
B_{18}	2.012	0.82	1.03	3.50	2.49	8.85	7.33	0.76
$Y_{1000,8000}$	14.084	1.41	1.33	5.66	3.12	67.51	3.78	1.75
$W_{1000,1000}$	2.210	0.12	1.17	1.71	1.31	3.20	4.80	0.43
X_{10000}	23.630	6.80	0.36	9.70	7.01	243.93	4.30	0.87
AVG		0.56	1.21	4.88	3.42	28.27	4.96	0.95

Fig. 2. TCFF Run Time (Real Time for XSB, Factor compared with XSB for others)

For other systems, including our own BAM, there are graphs where the more or less linear relationship is severely violated. Of course, these inputs are interesting for the system developers. We also have to look at other cost factors besides the number of applicable rule instances. One such approach is the subject of the next subsection.

5.2 Predicting Runtimes

We assume that the main computation consists of the following four components:

- **Initialize** is the time it takes to initialize the program (e.g. for parsing and optimizing the query, and allocating memory). Since we consider only a single query, this time is a system-dependent constant I_S (for the system S).
- **Loading and preprocessing** is the process of reading input data from the file system and storing it into some other data structure. This includes creating any indexes. While reading from the file system is expected to be done in linear time, creating indexes may take log-linear time. Therefore, we assume that this phase costs $L_S * e(G) * \log(e(G))$, where $e(G)$ is the number of edges in the input graph G and L_S is a system-dependent constant.
- **Deduction** is the work of applying the rules to derive new facts, in this case for the `tc` predicate. This is where the previously discussed cost measure is used. We assume that the time needed is $D_S * R(G)$, where D_S is a system-dependent constant, and $R(G)$ is the number of applicable rule instances.
- **Answer processing** is the work of processing a computed answer to the query. We assume that the time needed is $A_S * T(G)$, where A_S is a system-dependent constant, and $T(G)$ is the size of the `tc`-relation (i.e. the answer). Note that $R(G) - T(G)$ is the number of non-successful rule applications,

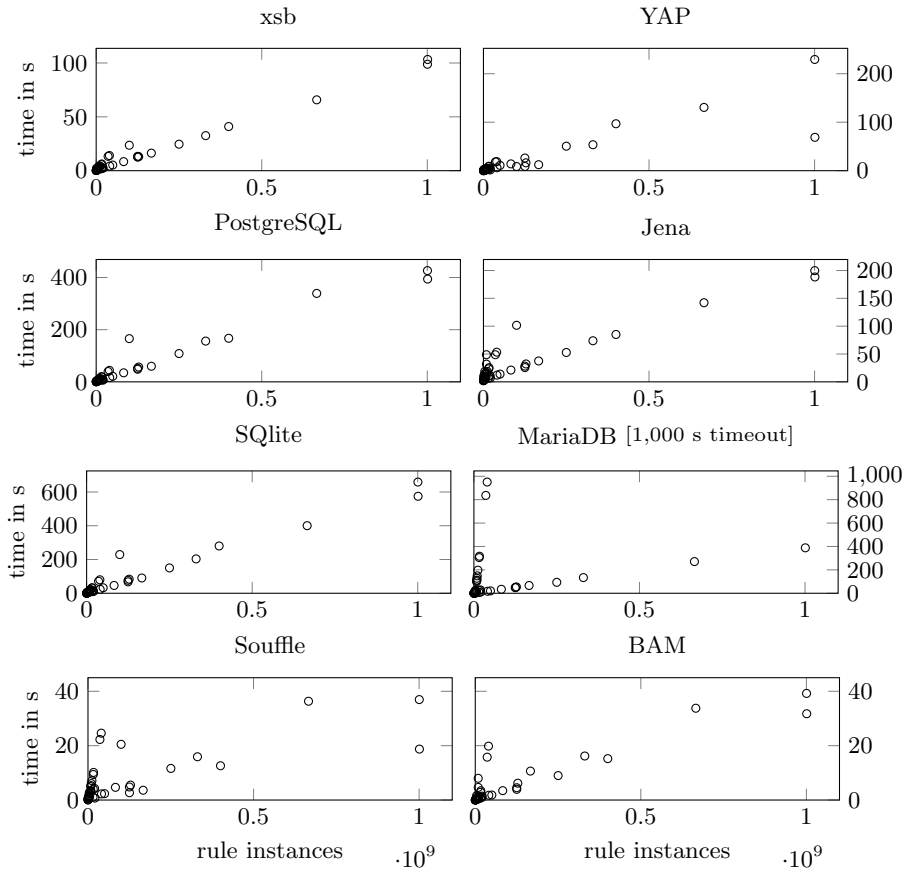

Fig. 3. Time vs. Rule Instances for Transitive Closure Benchmark

i.e. rule applications that computed only duplicates of already known answers. With the two parameters D_S and A_S, it is possible to treat successful and non-successful rule applications differently.

Given the partial costs above, we estimate the runtime $E_S(G)$ for a specific system S and for an input G as the sum of the components, i.e.

$$E_S(G) = S_I * 10^{-3} + \big(L_S * e(G) * \log(e(G)) + D_S * R(G) + A_S * T(G)\big) * 10^{-9}.$$

The factors mean that S_I is measured in milliseconds, whereas the other constants are in milliseconds per one million cost units. E.g. D_S are the milliseconds per one million rule applications.

Let us consider an estimate as acceptable if the real value differs less than 20% from the estimate, i.e. lies between $0.8 * E_S(G)$ and $1.2 * E_S(G)$. We used a simple optimization program to search for good values for the parameters. The average error was a secondary optimization criterion (of lower priority than

improving the number of graphs with acceptable prediction). Obviously, for some systems, we have not yet the right cost parameters. E.g., currently, non-successful joins are not counted at all. For MariaDB, it seems that we need an additional factor for large data (that presumably do not fit in the main memory buffer).

Program	Init: I_S	Load: L_S	Ded.: D_S	Ans.: A_S	Within ±20%	Avg. Err.
XSB	130	90	95	284	49/50	6%
SQLite	90	93	593	1317	49/50	7%
PostgreSQL	369	102	419	711	48/50	8%
Jena	2076	1599	166	1402	47/50	9%
YAP	20	4	200	221	35/50	30%
Soufflé	25	12	42	671	32/50	23%
BAM	0	373	32	54	31/50	81%
MariaDB	19	1026	438	1164	24/50	512%

5.3 Results for the Same Generation Benchmark

The runtimes for the same generation benchmark and some selected input graphs are shown in Fig. 4. Data for more graphs and the plots relating runtime to the number of applicable rule instances are available on the web page.

Graph	XSB (s)	BAM	YAP	SQLite	PostgreSQL	MariaDB	Soufflé
K_{500}	9277.250	0.31	0.41	2.31	1.79	3.51	0.14
T_{100}	2.137	0.61	1.18	3.12	2.70	4.97	0.43
C_{1000}	0.075	0.03	0.13	0.33	1.87	0.73	0.25
$S_{1000,1}$	1.233	0.32	0.86	3.64	2.83	3.93	1.35
P_{4000}	0.105	0.00	0.19	0.86	1.56	0.95	0.18
$M_{4,2048}$	0.125	0.00	0.32	1.05	1.36	1.04	0.24
$M_{4096,2}$	0.133	0.00	0.45	1.23	1.36	1.21	0.23
B_{18}	1.088	0.11	1.14	2.17	1.20	4.03	0.28
V_{12}	2.296	0.42	0.30	4.24	4.87	21.80	0.69
$Y_{500,4000}$	0.218	0.30	0.16	2.82	3.06	1.97	0.56
AVG		0.20	0.47	2.00	2.13	3.05	0.41

Fig. 4. SGFF Run Time (Real Time for XSB, Factor compared with XSB for others)

6 Conclusions

Since we are developing our own Datalog system, we of course need to do systematic runtime measurements. We started with some benchmarks of the OpenRuleBench [4], but we soon lost the overview on the many measurements we made on different systems with different versions — and of course, also running

exactly the same benchmark twice gives slightly different results. The OpenRuleBench scripts have no support for all this information, and therefore we developed the database that is described in this paper. Currently, the main table `BENCH_RUN` contains more than 30.000 rows. If one really wants to make sense of the collected data, one has to do some data mining. We found that the number of applicable rule instances can be used as a first try to estimate runtimes. In order to give better estimations, our current approach uses four parameters: The initialization time, the load time depended on the size of the input relation, the deduction time based on the applicable rule instances, and the answer (query result) time that uses the output size. In future, we plan to investigate more parameters, and to extend the approach to other benchmarks.

Further information (including source code, data files, SQL scripts to create the database, and benchmark results) is available on the web page of the project: `http://dbs.informatik.uni-halle.de/rbench/`.

References

1. Brass, S.: Implementation alternatives for bottom-up evaluation. In: Hermenegildo, M., Schaub, T. (eds.) Technical Comm. of the 26th Int. Conf. on Logic Programming (ICLP'10). LIPIcs, vol. 7, pp. 44–53. Schloss Dagstuhl (2010), `http://drops.dagstuhl.de/opus/volltexte/2010/2582`
2. Brass, S., Stephan, H.: Pipelined bottom-up evaluation of Datalog: The Push method. In: Petrenko, A.K., Voronkov, A. (eds.) Perspectives of System Informatics (PSI'17). LNCS, vol. 10742, pp. 43–58. Springer (2018), `http://www.informatik.uni-halle.de/~brass/push/publ/psi17.pdf`
3. Brass, S., Wenzel, M.: An abstract machine for Push bottom-up evaluation of Datalog. In: Hartmann, S., Ma, H., Hameurlain, A., Pernul, G., Wagner, R.R. (eds.) Database and Expert Systems Applications, 29th International Conference, DEXA 2018, Proceedings, Part II. LNCS, vol. 11030, pp. 270–280. Springer (2018), `http://www.informatik.uni-halle.de/~brass/push/publ/dexa18.pdf`
4. Liang, S., Fodor, P., Wan, H., Kifer, M.: OpenRuleBench: An analysis of the performance of rule engines. In: Proc. of the 18th Int. Conf. on World Wide Web. pp. 601–610. ACM (2009), `http://rulebench.projects.semwebcentral.org/`
5. Linux man-pages Project: proc(5) Linux User's Manual, 4.16 edn. (9 2017)
6. Przymus, P., Boniewicz, A., Burzańska, M., Stencel, K.: Recursive query facilities in relational databases: A survey. In: Database Theory and Application, Bio-Science and Bio-Technology (DTA/BSBT 2010). pp. 89–99. No. 118 in Communications in Computer and Information Science, Springer (2010), `http://www-users.mat.umk.pl/~eror/papers/dta-2010.pdf`
7. Sagonas, K., Swift, T., Warren, D.S.: XSB as an efficient deductive database engine. In: Snodgrass, R.T., Winslett, M. (eds.) Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94). pp. 442–453 (1994), `http://user.it.uu.se/~kostis/Papers/xsbddb.html`
8. Scholz, B., Jordan, H., Subotić, P., Westmann, T.: On fast large-scale program analysis in Datalog. In: Proceedings of the 25th International Conference on Compiler Construction (CC'2016). pp. 196–206. ACM (2016)
9. SPARQL 1.1 Query Language, W3C Recommendation (3 2013), `http://www.w3.org/TR/sparql11-query/`