Performance Analysis and Comparison of Deductive Systems and SQL Databases

Stefan Brass and Mario Wenzel

University of Halle, Germany

Datalog 2.0 2019 (June 4, 2019)

Contents

This is NOT a Talk about . . .

- Our Push method for bottom-up evaluation of Datalog.
  - Although that was our motivation for the work presented here: We wanted to check the performance of our approach before investing a lot of work to implement it.

    The implementation is not finished: We can execute some benchmarks, but not yet arbitrary Datalog programs.
  - (The first performance results look nice.)
- New Benchmarks for deductive databases.
  - The benchmarks we have implemented are all from the OpenRuleBench (Liang, Fodor, Wan, Kifer, 2009–2011).

    Currently, we implemented only a few of the OpenRuleBench benchmarks in our system (3+2 of 12/18), but we analyze them in much more detail. We are working on more benchmarks.

This is a Talk about . . .

- Doing benchmark measurements.
- A database for storing benchmark results.
- A collection of graphs for checking the performance of transitive closure, same generation, win-not-win.

  Also different versions with respect to input and output arguments.
- Making sense of all the numbers.
- Checking recursive views in SQL databases.

Benchmarks are important

- When selecting a system for a project, there are many aspects: Language features, development tools, support.
- But this all becomes important only when the performance is at least acceptable for the task.

  Declarative systems do not have an especially good reputation with regard to performance. Maybe such doubts are inherent in the declarative approach, because there is no prescribed evaluation algorithm. We aim at a simple declarative model to predict the approximate runtime. If runtime would suddenly explode for certain inputs, the entire system would be unreliable.
- "Stress tests" with benchmarks might also help to discover limitations in a system.
- Benchmarks also motivate the developers ("competition" in the end helps to improve all systems).

Contents

Benchmark Database (1)

- Runtimes depend on:
  - Benchmark (Logic Program/SQL Query)
  - System to Test
  - Input File (Facts)
  - Version of System, Installation Settings, Compiler Options
  - Settings for System and Benchmark (e.g. Index Selection)

    To be fair, one should invest some time to find good settings.

  - Machine
  - Some small random influences on the machine

    Therefore it is common practice to measure the same runtime multiple times, and take the average. OpenRuleBench has no support for this.

- The OpenRuleBench scripts only record first three.

Benchmark Database (2)

- Data measured for each run (as far as possible):
  - Time to load the input (CPU time and real time)

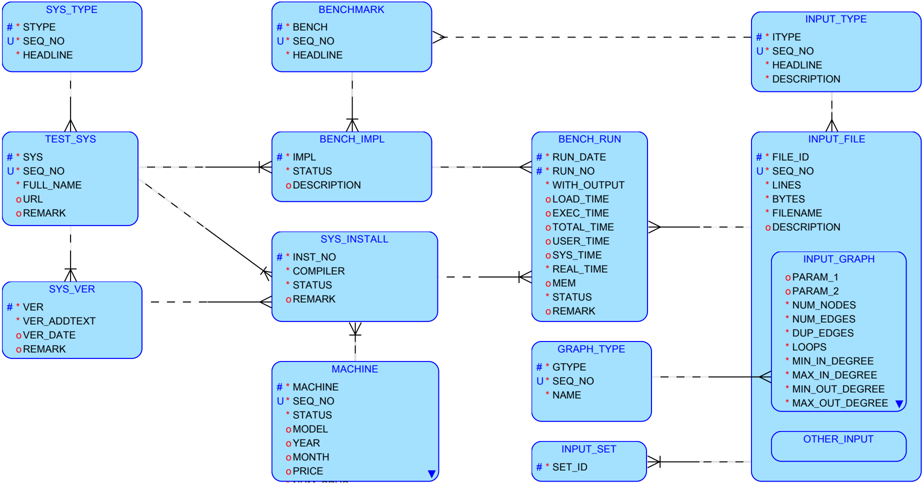
    This is based on timing functions within the system.
  - Time to process the query (CPU time and real time)

    Also measured internally (within the system) (if supported).
  - Total time for benchmark execution.

    This is measured externally with `/usr/bin/time` or the process information file system (see `man 5 proc`). For server-based systems like PostgreSQL, it does not include the time for starting and stopping the server, but it includes everything from `CREATE TABLE` to `DROP TABLE`.
  - Memory usage (maximum resident set size)

    Memory allocated and actively used, thus in RAM.
- OpenRuleBench measures only the first two.

Benchmark Database (3)



Benchmark Database (4)

- We defined a relatively large number of views, e.g. for:
  - Outlier detection
  - Comparing different settings for the same system and benchmark (to find the best)
  - Generating HTML and LaTeX-Tables with the results

    Also with data of the input files (graphs).
  - Analyzing runtimes compared to input size measures

Contents

The TCFF Benchmark

- The main benchmark, which we investigated very thoroughly, is the transitive closure:

```
tc(X, Y)  ←  par(X, Y).
tc(X, Z)  ←  par(X, Y), tc(Y, Z).
```

- The relation `par` can be understood as defining edges in a directed graph.
- Then `tc` contains pairs of nodes `X` and `Y`, such that there is a path from `X` to `Y` in the graph.
- The benchmark is called "TCFF" because all such pairs should be computed, i.e. the predicate `tc` is queried with both arguments "free".

  OpenRuleBench also contains TCBF and TCFB.

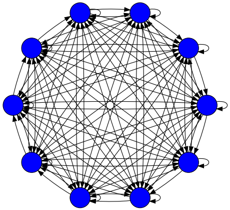
Input Graphs (1)

- OpenRuleBench used ten random graphs of different size.

  However, due to the construction algorithm, the all nodes in the graph have more or less the same degree.

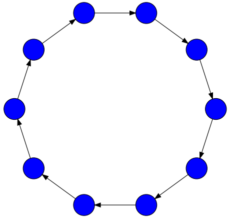
- We added a collection of non-random graphs of different structure, which permit a better analysis.

- E.g. complete graph K_n:



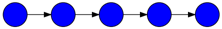
Input Graphs (2)

- Cycle-Graph C_n:



We also studied a cycle with shortcuts $S_{n,k}$.

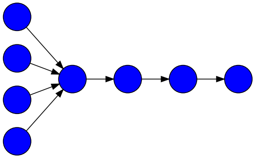
- Path P_n:



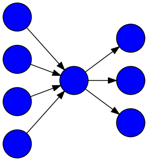
We also studied a "Multipath" $M_{n,k}$ with several disjoint paths.

Input Graphs (3)

- Y-Graph $Y_{n,k}$ (n nodes pointing to central node, which starts path P_k):



- X-Graph $X_{n,k}$ (n nodes pointing to central node, from there to k):



Contents

Compared Systems

- Prolog Systems with Tabling:
  - XSB (Stony Brook)
  - YAProlog (Universidade do Porto)
- New Datalog Systems:
  - Soufflé (University of Sydney, Oracle Labs)
  - Our Bottom-Up Abstract Machine (BAM)
- SQL Databases
  - PostgreSQL
  - SQLite
  - MariaDB (fork of MySQL)
- RDF Graph Store:
  - Apache Jena

TCFF Benchmark Implementation

- Datalog as shown above (with tc tabled)
- SQL:

```
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;
```

- SPARQL query using property paths (for Apache Jena):

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

Results: TCFF (1)

Graph	XSB (s)	BAM	YAP	SQLite	PG	Maria	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
$Y_{1k,8k}$	14.084	1.41	1.33	5.66	3.12	67.51	3.78	1.75
X_{10k}	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

Results: TCFF (2)

- The bottom line shows the average factor of the runtime in that system compared with XSB (over 50 graphs).

  Of course, this depends on the selection of graphs: For each system, there are input graphs, where the system is slower than XSB. Maybe the maximum would be more interesting if one wants no surprises and believes that XSB delivers a predictable performance (which is plausible, see below).

- PostgreSQL has is on average 3.4 times slower than XSB on the TCFF problem, and 7.0 times in the worst case.

  Among the measured 50 graphs. SQLite: AVG=4.88, MAX=9.7.

- MariaDB has a very young implementation of recursive views, and was $>$ 100 times slower than XSB for five graphs.

  Maybe we did not find the best settings. Performances goes severely down for large graphs. I.e. in the current state, without better ideas, it should not be used for problems similar to TCFF.

Results: SGFF

Graph	XSB (s)	BAM	YAP	SQLite	PG	Maria	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
$Y_{500,4k}$	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

Contents

A Simple Cost Measure: Applicable Rule Instances

- A first try to estimate the work a deductive system has to do for a given program and input facts is the number of applicable rule instances.

  I.e. rule instances, the body of which is true in the minimal model.

  E.g. seminaive evaluation would apply each such rule instance exactly once.

- Consider the transitive closure program:

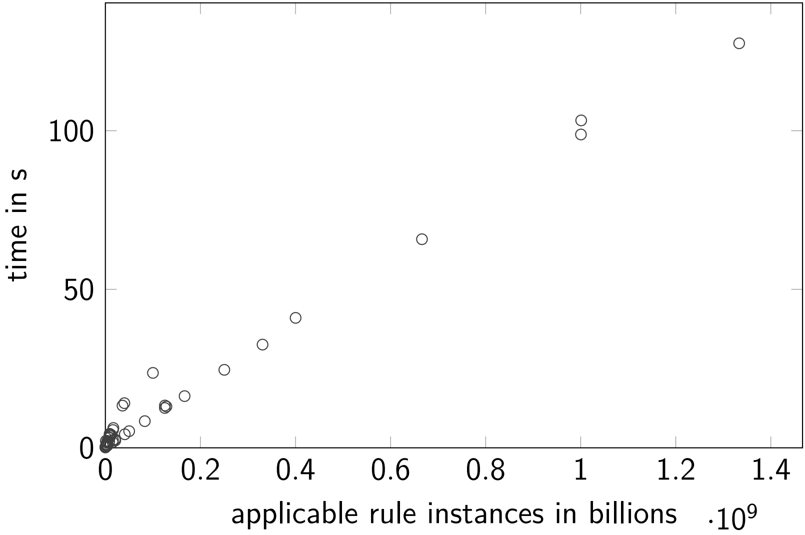
$$\texttt{tc(X, Y)} \leftarrow \texttt{par(X, Y).}$$
$$\texttt{tc(X, Z)} \leftarrow \texttt{par(X, Y), tc(Y, Z).}$$

- Let the input be the K_{100}, i.e.

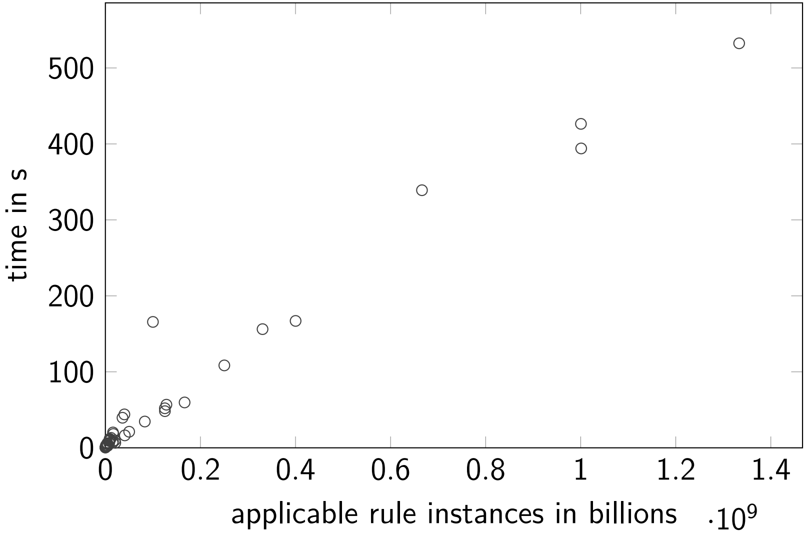
$$\texttt{par} := \{(i, j) \mid 1 \leq i \leq 100,\ 1 \leq j \leq 100\}.$$

- The first rule has 100^2 instances (the size of the par-relation), and the second has 100^3 instances (each par-fact has 100 join partners in tc), in total $1\,010\,000$ rule instances.

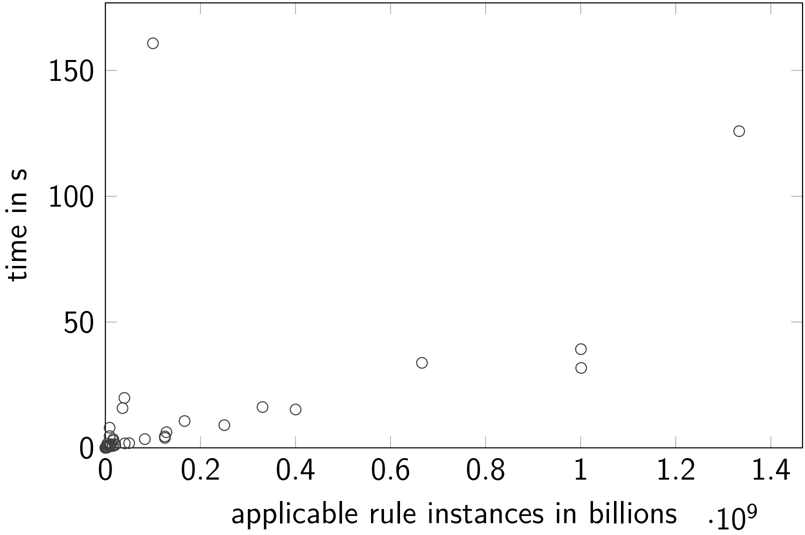
Runtime vs. Rule Instances: XSB



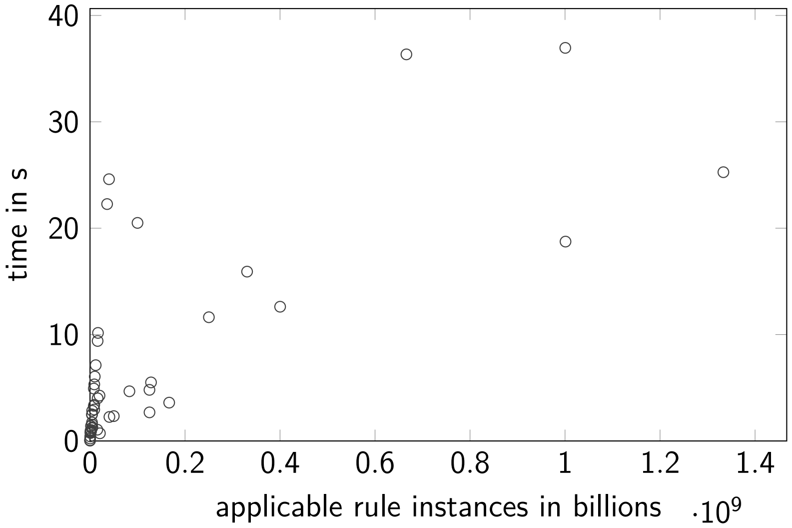
Runtime vs. Rule Instances: PostgreSQL



Runtime vs. Rule Instances: BAM



Runtime vs. Rule Instances: Soufflé



Contents

Predicting TCFF Runtimes (1)

- Goal: Predict the runtime (first for the TCFF program) from simple features of the graph and the program rules.
- Runtime estimation is done in databases for a long time, but this is a kind of "black box approach" that does not assume knowledge of internal data structures and algorithms of the system.

  Of course, runtime depends e.g. on the chosen indexes, but we do our estimation for reasonable settings that a somewhat knowlegable user would have chosen (the best settings for the system we could find).
- The parameters of the prediction formula for a system can be seen as the result of "compressing" all the benchmark measurements to just a few numbers that characterize the performance of a system.

  Of course, this is not a lossless compression.

Predicting TCFF Runtimes (2)

- Four components (with system-dependent parameters):
  - Startup time (independent of input): I_S.

    Since we consider only one logic program/query, this contains the time for parsing the rules, and query optimization. I_S is measured in ms.
  - Load time (depends on input size): $L_S * e(G) * \log(e(G))$.

    $e(G)$ is the input size (number of edges). L_S is measured in ms per 10^6 $n * \log_2(n)$ units of edges, e.g. ms for approx. 63.000 edges.
  - Deduction time (depends on rule instances): $D_S * R(G)$.

    $R(G)$ is the number of applicable rule instances. D_S: ms/10^6 rule inst.
  - Answer time (depends on result size): $A_S * T(G)$.

    $T(G)$ is the result size (transitive closure of G). A_S: ms/10^6 derived `tc`-tuples (not counting duplicates). Note that $R(G) - T(G)$ is the number of duplicates. One can see D_S as the cost of deriving a duplicate and $D_S + A_S$ as the cost of deriving a new result tuple.

Predicting TCFF Runtimes (3)

Program	I_S	L_S	D_S	A_S	±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%

Contents

Conclusions

- XSB has beaten older systems based on bottom-up evaluation (such as Coral). Now it seems that bottom-up evaluation can be implemented in a competitive way.
- We are developing a "Bottom-Up Abstract Machine" (BAM). System development needs systematic runtime measurements.
- We developed a database for runtime measurements.

  Including scripts for doing the benchmarks and SQL views for generating HTML and LaTeX tables, and analyzing the data.
- There are first results for using the collected data to predict runtimes (based amongst others on # applicable rule inst.).
- Even SQL databases such as PostgreSQL and SQLite have acceptable preformance on the tested benchmarks.

Web Pages and Downloads

- New benchmark page:
  [http://dbs.informatik.uni-halle.de/rbench/]
- Old benchmark page (some additional benchmarks&systems):
  [http://users.informatik.uni-halle.de/~brass/push/bench.html]
- Git repository for benchmark project:
  [https://gitlab.informatik.uni-halle.de/brass/rbench]

  The subdirectory `db` contains the SQL scripts to create and fill the database.
- Git repository for Graph generator:
  [https://gitlab.informatik.uni-halle.de/mwenzel/graphgen]

  This was used for generating the graphs. Alternative: `rbench/graph`.
- Benchmarking and build platform (used for some systems):
  [https://gitlab.informatik.uni-halle.de/mwenzel/benchF]