A New Benchmark Database and An Analysis of Transitive Closure Runtimes

Stefan Brass and Mario Wenzel

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
brass@informatik.uni-halle.de, mario.wenzel@informatik.uni-halle.de

Abstract. In this paper, we compare the performance of logic programming systems such as XSB with the performance of relational database systems such as SQLite and RDF stores such as Apache Jena.
There are three main contributions compared to the OpenRuleBench [6]: (1) We created a database for storing and evaluating the runtimes of the benchmarks. In particular, this manages several executions of the same benchmark configuration, which is missing in the OpenRuleBench scripts. We also measure more parameters. (2) We identified and solved a problem with the OpenRuleBench test graphs for the transitive closure benchmark. We also added a relatively large number of additional test graphs of different structure. (3) We propose a simple cost measure for the transitive closure benchmark based on parameters of the graph, and compare the measured run times with this cost measure.
Currently, we did measurements and analysis only for the transitive closure benchmark. The OpenRuleBench collection is much larger. Our database is general enough to store all benchmark results, but the other work is just the beginning of a larger project.

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 [6] 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. The original paper [6] contains 18 tables with benchmark results.

However, the OpenRuleBench scripts run each benchmark only once, and do not support comparing benchmarks run with 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. 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 report results for only one of the OpenRuleBench benchmark problems, namely the transitive closure, but we look at it in much more detail. In particular, we define a cost measure, namely the number of applicable rule instances, and compare the actual runtimes with this cost measure. 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.

We checked not only logic programming systems such as XSB [9], YAP [5], and SWI-Prolog, but also relational databases (currently only SQLite3, but PostgreSQL and HSQLDB are coming soon), and RDF triple stores/graph databases (Apache Jena) as well as DataScript (a Datalog query engine for JavaScript). Of course, we also included our own abstract machine for Push bottom-up evaluation of Datalog [4]. Our own system development project also made it necessary to develop a database for our performance measurements that permits to evaluate the effects of updates and different implementation alternatives.

The runtimes of transitive closure in relational databases have already been investigated in [7], 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 much less important now.

In addition, we did some measurements on main memory consumption, also giving a cost measure for that resource.

2 A Benchmark Database

The OpenRuleBench scripts [6] 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. For instance, the total runtime of XSB for the `tcff` benchmark with the largest tested random graph (`f_2k_1m`) in 10 runs varied between 64.1 s and 75.9 s (with average 69.7 s and standard deviation 5.1 s). 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 has the following tables:

- MACHINE(MACHINE, SEQ_NO, STATUS, MODEL, YEAR, MONTH, PRICE,
  NUM_CPUS, CPU, CPU_CORES, CPU_THREADS, GHZ, ARCH, RAM,
  NUM_DISKS, DISK, SSD, OS, OS_CORE)
  These are some data on the test machines. We underline primary key attributes, in this case the machine name. There is a single "standard machine" marked with STATUS='S', and many views for evaluating the data are restricted to this machine. However, the database permits to store and evaluate benchmark measurements for different machines with different architectures and operating systems. The number in column SEQ_NO can be used for ordering the list of machines.
- BENCHMARK(BENCH, SEQ_NO, HEADLINE)
  This is the list of all benchmarks, e.g. tcff is a benchmark identifier in column BENCH. It corresponds to the computation of the transitive closure with query tc(X,Y), i.e. with both arguments "free". Another benchmark, tcbf, is the computation of transitive closure with query tc(1,X), i.e. the first argument "bound" (to a given value), and the second argument "free". The column SEQ_NO is for ordering benchmarks in the output, e.g. when a web page with all benchmark results should be generated. This would also need a HEADLINE for each benchmark.
- PROGRAM(PROG, SEQ_NO, FULL_NAME, URL, REMARK)
  This table lists the tested programs, such as xsb.
- PROG_VER(PROG→PROGRAM, VER, MACHINE→MACHINE, VER_NAME,
  VER_ADDTEXT, VER_DATE, COMPILER, STATUS, REMARK)
  This table lists program versions that are or were installed on a test machine. They are identified here by a number that is unique together with the program name and the machine. These three elements (Program, Version number, and machine) are also encoded in the file names with the test results. We use a simple sequential number VER for the version in the key, not the real version number (which can be an arbitrary string). The real version number is available in column VER_NAME and can be used for output.

  This table also defines a current version of a program on a machine (by STATUS='C'). Some views show only results for the current version. It is also possible to define a previous version (STATUS='P') as reference, if the result of a program update should be evaluated.
- BENCH_IMPL(PROG→PROGRAM, BENCH→BENCHMARK, IMPL, STATUS,
  DESCRIPTION)
  In general, there are different option settings (e.g. index selections) that should be tested for the same program to execute a given benchmark as fast as possible. We call such a combination of benchmark, program and option settings a "benchmark implementation". STATUS='B' marks the best implementation that is used in some views to compare different programs. This is selected manually, but there are also views that help to find the best implementation.

- `DATA_FILE(FILE_ID, SEQ_NO, LINES, BYTES, FILENAME)`
  The filename is stored without extension, since we need the data files in different formats for different programs. The number of bytes is the size of the "Datalog Facts" version. There is one fact per line, i.e. `LINES` is the number of facts (tuples in the relation).
- `GRAPH_DATA(FILE_ID→DATA_FILE, NUM_NODES, NUM_EDGES, ..., TC_SIZE, COST)`
  For data files that are standard directed graphs, more information is available. This helps to evaluate the results for the transitive closure benchmark.
- `BENCH_INPUT(BENCH→BENCHMARKS, FILE_ID→DATA_FILE)`
  This is the many-to-many relation that states which input files should be used for which benchmark.
- `BENCH_RUN(RUN_DATE, RUN_NO, PROG, VER, MACHINE, BENCH, IMPL, FILE_ID, WITH_OUTPUT, LOAD_TIME, EXEC_TIME, TOTAL_TIME, USER_TIME, SYS_TIME, REAL_TIME, MEM, STATUS, REMARK)`
  This is the main table that contains the benchmark results. There are three foreign keys:
  - `PROG`, `BENCH`, `IMPL` → `PROG_IMPL`.
  - `BENCH`, `FILE_ID` → `BENCH_INPUT`.
  - `PROG`, `VER`, `MACHINE` → `PROG_VER`.

  There are two sources for timing information: Measurements done by the program itself, and external measurements done with `/usr/bin/time`. The following values are what the program that runs the benchmark reports (if available):
  - `LOAD_TIME`: This is the time for loading the input data. (All time values are stored in milliseconds.)
  - `EXEC_TIME`: This is the execution time for the benchmark query. If the column `WITH_OUTPUT` is `'Y'`, it contains the time for writing the query result. However, all normal measurements are done without writing the result. But we generated a query result file at least once for each system and benchmark in order to check whether the result is correct. Also, a very large time difference to the normal benchmark execution would indicate that the optimizer of the system is too clever and uses the fact that the result values are not really read.
  - `TOTAL_TIME`: This is the total runtime for executing the benchmark as reported by the system.

  If the system can report CPU time and real time ("wallclock time"), all three values are CPU time. If the two differ a lot, the `STATUS` column contains a warning. If only wallclock time is available, we take that. We use the external measurements for the main comparison on the benchmarks, so these three values are only interesting if one wants to see what fraction of the time is spent on loading the data. Furthermore, startup time and time for compiling the benchmark is not included in these data.

The following values are measured externally and should be comparable for all systems that run as a single process:

- USER_TIME: This is the CPU time the program spent in user mode.
- SYS_TIME: This is the CPU time the program spent in kernel mode, i.e. while executing operating system calls. Usually, this is small. The sum of USER_TIME and SYS_TIME is the total CPU time the program used for executing the benchmark.
- REAL_TIME: This is the total wallclock time that the execution of the benchmark took. For single-threaded programs, it is slightly longer than the total CPU time used (e.g. when waiting for I/O). For multi-threaded programs, it can be significantly less than the CPU time used (which is added up over all threads).
- MEM: This is the "maximum resident set size" (in kilobytes), i.e. the "high water mark" for the amount of memory a process uses and that is present in real RAM (e.g., not swapped out). It includes shared libraries (as far as actually used). In our previous paper [2] we checked that the amount of memory used for only starting and immediately stopping the tested systems is small compared to the memory used for actually executing the benchmarks.

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 Problem on Different Systems

In this section we describe the systems that we used in this work.

For comparability we decided on the transitive closure as our first problem to benchmark the database systems on. The problem of finding the transitive closure of a graph has been extensively studied and easily formulated for the different database systems. The transitive closure is also one benchmark in the OpenRuleBench collection [6]. There, the standard tail-recursive formulation is chosen:

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

"par" stands for "parent", but it is simply the edge relation of the graph. The nodes in the graph are integers, thus are large set of facts of the form par(1,2) are given. We will discuss the test graphs in Section 4.

3.1 Prolog-Systems with Tabling

Since many of the test graphs are cyclic, a standard Prolog system would not terminate with the above program. We need systems that support tabling to detect repeated calls to the tc predicate.

XSB Prolog version 3.8.0
XSB Prolog is a well-known deductive system [9] with a long experience in efficiently implementing Prolog evaluation with tabling. We ran our tests loading the data with `load_dync` and enabled subsumtive tabling, and a trie index on the `par` relation.

YAP Prolog version 6.2.2
YAP is a high performance Prolog compiler [5] that can be configured in a version with tabling.

SWI Prolog version 7.7.15
SWI Prolog is a robust and scalable implementation of the Prolog language. However, the support for tabling is very new, and declared as merely a first prototype. As mentioned on the SWI Prolog web page, it cannot be considered yet as a serious competitor to the above systems.

For the three systems above, we used the query "`tc(_,_), fail.`". I.e. we backtrack over all solutions, but the (quite large) output is not actually written. Also in OpenRuleBench, it is done in this way. However, our scripts also support measuring the time with writing of the result in order to check that the correct relation is computed, and the systems are not doing too big optimizations in the main benchmark because the query result is not used.

3.2 Bottom-Up Evaluation, Datalog systems

Push-Method with Abstract Machine (BAM)
We developed the Push method for bottom-up evaluation of Datalog [1,3]. It uses each derived fact immediately and can be seen as an extreme form of seminaive evaluation that dates back to the PhD thesis of Heribert Schütz [10]. Whereas we defined the method originally as a translation from Datalog to C++, we recently developed an abstract machine "BAM" for bottom-up evaluation based on the push method [4]. For the performance comparison, this is better, because the other systems also interpret code of some abstract machine. The translation to C++ would result in native machine code.

DataScript 0.16.6
DataScript is a Datalog query engine for JavaScript. We ran it on node.js 10.5.0. Even though DataScript's performance is severely lacking, it's an example for a benchmark on newer Datalog query engines akin to Datomic or Mozilla Mentat (formerly Datomish).

DataScript accepts queries in extensible data notation (EDN). We had to define the `par` relation as a relation with the multiplicity "many" and allowed for an index on this relation. In EDN the query and the rules are defined in the following way (`$` is a forward-reference to the database and `%` is a forward-reference to the rules):

```
[:find ?e1 ?e2 :in $ % :where (tc ?e1 ?e2)]
[ [(tc ?e1 ?e2) [?e1 "par" ?e2] ]
  [(tc ?e1 ?e2) [?e1 "par" ?ex] (tc ?ex ?e2) ] ]
```

3.3 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.

SQLite3 version 3.24.0
SQLite is a self-contained, public-domain, SQL database engine.

To improve performance, we used a temporary in-memory table for the `par`-relation. We added an index on the second column.

```
PRAGMA temp_store = MEMORY;
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);
```

Without the `temp_store` pragma, SQLite would use temporary files to evaluate the recursive query, even though the input data is in a transient in-memory database.

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;
```

3.4 Graph Databases / RDF Triple Stores

Apache Jena version 3.7.0
Apache Jena is a Java Framework for linked data that supports querying RDF data via SPARQL. We ran Jena on OpenJDK 1.8 (Java 8) using Property Paths for the queries. To allow for deeper recursions we increased the JVM's thread stack size to 16MB using the environment variable `JVM_ARGS=-Xss16m` .

Graph databases and and RDF Triple Stores accept queries in the SPARQL format. SPARQL does not allow for general recursion [8] but SPARQL 1.1 added Property Paths [11] where non-variable predicates could be combined in a similar fashion to regular expressions. This allows us to write this query in a concise way:

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

3.5 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. However, the current version of our program "BAM" does not use multiple threads (we are currently working on this). A Java program (such as Jena) always uses multithreading at least for garbage collection. The machine has 64 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 was measured with the Linux `/usr/bin/time` program. The time for loading the data and for executing the query are measured by functions of each system (as far as possible).

Most tests were run ten times and the time average values were calculated. We also checked for outliers, where the real time was much higher than the average. A few times in more than 5000 measurements, it seemed that the system was locked up for two minutes. The CPU time was normal, but the real time was unusally high. This occurred all on the same day. We repeated the measurements of that day.

In the comparison table (Fig. 2 below), we use real time. We believe that parallel execution should be honoured (there, CPU time is higher than real time). Apache Jena used on average 215% CPU (i.e. two parallel threads), DataScript used 133% CPU. For all other systems, the average CPU utilization was 97% to 100%, i.e. there was no big difference between real time and CPU time. However, there were a few measurements with only 50% CPU for XSB. These were very small benchmarks (waiting on the disk might explain the low CPU utilization).

4 Test Graphs

4.1 Random Graphs from the OpenRuleBench collection

The OpenRuleBench collection [6] uses a particular algorithm to generate random graphs that avoids a duplicate check for already selected edges. The domain $d = \{1, \ldots, n\}$ is copied into separate lists a and b which are both extended with a special symbol $\circ$ so that their lengths are coprime, and then shuffled. A random graph contains edges from $(a_{(i \mod |a|)}, b_{(i \mod |b|)})$ where neither component is $\circ$. Selecting e tuples in order with increasing i (no larger than $|a| * |b|$) leads to all vertexes having nearly identical degrees, compared to just selecting edges at random. Cyclic graphs generated in this manner are usually strongly connected, even with low average node degree (about 1.5).

Additionally, the OpenRuleBench scripts generate graphs that are marked "nocyc" (non-cyclic). This is done by ordering the edges towards the greater node. Since this happens after the edges have already been selected, it leads to duplicate edges in the generated non-cyclic graphs. Furthermore, reflexive edges (loops) were not excluded in the settings of the distributed scripts.

We have rewritten the graph generation in the same spirit but for acyclic graphs we skipped reflexive edges and added a duplicate check in order to generate the correct amount of unique edges.

The cyclic and non-cyclic graphs are named $U_{n,e}$ and $F_{n,e}$ respectively (where n is the number of nodes, and e is the number of edges). The motivation for the letter U is the quite "uniform" degree, and F is the same with only "forward" edges. The parameter values for the test graphs are listed in Figure 1.

4.2 Additional Deterministic Graphs

Additionally, we created generators for graphs with properties that are easily analyzed in the context of the transitive closure problem. The number of edges in the graph is the input size of the problem. The transitive closure than computes the connected node pairs, i.e. the "reachability" relation. The diameter of the connected subgraphs is the maximum over the length of the shortest paths between any two connected nodes. This is the number of iterations until the fixed point of the T_p operator is found. We label nodes incrementally starting from 1.

- The complete graph K_n has n vertices and the diameter 1. The graphs are their own transitive closure with n^2 edges.
- The maximum acyclic graph A_n has n vertices and is a subset of K_n without the edges (a, b) with $a \geq b$. The family of graphs have the diameter 1 and the graphs themselves are their own transitive closure with $\frac{n(n-1)}{2}$ edges.
- The cycle graph C_n has n vertices and the diameter n. Its transitive closure is the complete graph K_n.
- The directed short-circuited cycle graph is a directed graph $S_{n,m}$, where n is divisible by $m + 1$, that is a superset of the directed cycle graph C_n where $m > 0$ additional edges per vertex are added, skipping $\frac{n}{m+1}$ nodes in the cycle. The diameter of the graph is $\frac{n}{m+1}$. The transitive closure of that family of graphs is the complete graph K_n. We give the set of edges $E = \{(i, (i+\frac{nt}{m+1}) \mod n) | t = 1, \ldots, m \wedge i = 1, \ldots, n\} \cup C_n$. One additional edge per vertex allows skipping $\frac{1}{2}$ of the circle, while two additional edges would allow skipping $\frac{1}{3}$ or $\frac{2}{3}$ of the circle.
- The graph P_n is a directed graph with n vertices that is the path containing the edges $E = \{(i, i + 1)|i = 1, \ldots, n - 1\}$. The diameter is $n - 1$. The transitive closure of the path P_n is A_n.
- The multi-path graph $M_{n,m}$ is a directed graph that contains n vertices and m pairwise vertex-disjoint paths of length $\frac{n}{m} - 1$. This is also the diameter of the connected subgraphs. The set of edges is $E = \{(i, i+m)|i = 1, \ldots, n-m\}$. The transitive closure of that family of graphs has $\frac{n}{2}(\frac{n}{m} - 1)$ edges.
- 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. The diameter of that graph is $h - 1$. The transitive closure of that family of graphs has $\sum_{i=1}^{h-1} i2^i = (h - 2) * 2^h + 2$ edges. The set of edges is $E = \{(i, 2i), (i, 2i + 1)|i = 1, \ldots, 2^{h-2} - 1\}$.

4.3 Cost Measure for Processing Time

Calculating the transitive closure of a graph takes time. We want to associate a given problem instance with some sort of cost in order to estimate how much

Graph	Nodes	Edges	In-Degree	Out-Deg.	Cyc.	TC Size	Iter.	Comp. Cost
k_1k	1000	1000000	1000–1000	1000–1000	yes	1000000	1	1001000000
k_2k	2000	4000000	2000–2000	2000–2000	yes	4000000	1	8004000000
a_1k	1000	499500	0–999	0–999	no	499500	1	166666500
a_2k	2000	1999000	0–1999	0–1999	no	1999000	1	1333333000
c_1k	1000	1000	1–1	1–1	yes	1000000	1000	1001000
c_2k	2000	2000	1–1	1–1	yes	4000000	2000	4002000
c_4k	4000	4000	1–1	1–1	yes	16000000	4000	16004000
s_2k_1	2000	4000	2–2	2–2	yes	4000000	1000	8004000
s_2k_2	2000	6000	3–3	3–3	yes	4000000	288	12006000
s_2k_3	2000	8000	4–4	4–4	yes	4000000	500	16008000
s_2k_4	2000	10000	5–5	5–5	yes	4000000	400	20010000
p_1k	1000	999	0–1	0–1	no	499500	999	499500
p_2k	2000	1999	0–1	0–1	no	1999000	1999	1999000
p_4k	4000	3999	0–1	0–1	no	7998000	3999	7998000
m_4ki_2	8192	8190	0–1	0–1	no	16773120	4095	16773120
m_1ki_8	8192	8184	0–1	0–1	no	4190208	1023	4190208
m_256_32	8192	8160	0–1	0–1	no	1044480	255	1044480
m_64_128	8192	8064	0–1	0–1	no	258048	63	258048
m_16_512	8192	7680	0–1	0–1	no	61440	15	61440
m_4_2ki	8192	6144	0–1	0–1	no	12288	3	12288
b_17	131071	131070	0–1	0–2	no	1966082	16	1966082
b_18	262143	262142	0–1	0–2	no	4194306	17	4194306
b_19	524287	524286	0–1	0–2	no	8912898	18	8912898
u_1k_50k	1000	50000	47–51	46–51	yes	1000000	3	50050000
u_1k_125k	1000	125000	122–127	122–127	yes	1000000	2	125125000
u_1k_250k	1000	250000	248–251	248–251	yes	1000000	2	250250000
u_2k_250k	2000	200000	98–101	98–101	yes	4000000	3	400200000
u_2k_500k	2000	500000	248–251	248–251	yes	4000000	2	1000500000
u_2k_1m	2000	1000000	499–502	499–502	yes	4000000	2	2001000000
f_1k_50k	1000	50000	0–101	0–101	no	472863	8	15421338
f_1k_125k	1000	125000	0–251	0–251	no	492170	5	40705777
f_1k_250k	1000	250000	0–500	0–501	no	497810	5	83073458
f_2k_250k	2000	200000	0–201	0–201	no	1946015	8	128081155
f_2k_500k	2000	500000	0–500	0–500	no	1985377	6	330379789
f_2k_1m	2000	1000000	0–1000	0–1001	no	1995412	4	665127921

Fig. 1. Data of the Test Graphs

time solving the problem will roughly take. The cost measure is independent of any actual implementation but corresponds to an abstract evaluation scheme.

Such a cost measure is an independent reference point for the problem size. Without such a cost measure, the runtimes of a system for different graphs would just be single numbers.

As a simple first try, we take the number of applicable rule instances, i.e. the number of rule instances where all body literals are contained in the minimal Herbrand model. This corresponds to the number of rule instances that the T_P operator will apply. Seminaive evaluation requires that each such rule instance is considered only once.

As an example, consider the complete graph K_{1000} with 1000 vertices and 1000 000 edges. The (non-recursive) starting rule

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

is applicable 1000 000 times (once for each of the input facts). For the rule

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

there are 1000 000 facts in the `par` relation and each one has 1000 join partners in the `tc` relation. Thus the total size of the join `par(X,Y), tc(Y,Z)` is 1000 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 final cost of 1001 000 000.

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`-relation in the minimal model.

This simple cost measure does not include non-successful join operations, or the actual number or size of arguments. Also the reading of the input data files is not explicitly considered, but for the transitive closure program, the cost of the first rule is the input size. However, one could consider different weights. Furthermore, many data structures for looking up tuples have runtime $O(\log(n))$.

For the practical approach we used PostgreSQL to check the test graphs by means of SQL queries on the `par` relation. The results are shown in Figure 1. The associated computation cost measure for the problem of the given size is listed in the last column.

Contrary to previous work [7], the number of iterations seems to have little or even no impact on the cost compared to the other factors in our cost measure. As we can see from the examples, a single large join can be much more costly than a large number of smaller joins. This is also reflected in the actual runtimes for most systems (comparing C_{1k} to K_{1k}).

In Section 5, we will investigate the relationship between this cost measure and the actual runtimes of various systems.

4.4 Cost Measure for Memory Usage

For memory usage, we simply take the number of facts in the minimal model as cost measure. For the transitive closure, this is the sum of the size of the `par`

relation and the size of the tc relation. For the complete graph K_{1000}, the result is 2000 000. Simple solutions for duplicate elimination will keep all derived facts in memory, so this cost measure seems quite reasonable.

5 Analysis of the Results

5.1 Runtime Results

The average runtimes of the main systems for the various test graphs are shown in Fig. 2. We also give the factor compared to our own BAM prototype. E.g. a factor of 2 would mean that the system needs double as much time for computing the transitive closure of the graph.

Figure 3 shows the relation between the runtime and the cost measure for the graphs. Basically, for many systems the relationship between our cost measure and the runtime seems to be more or less linear. We were very positively surprised when we saw the graphs e.g. for XSB for the first time. At least, this shows that the cost measure is quite well correlated with the runtime.

However, in pure numbers it does not look quite as nice. We define the average processing speed of a system as the average processed cost over time. If a database system solves a problem that has an associated cost of 1000 in 10 seconds, the speed is $100 \frac{cost}{s}$ (i.e. rule instances per second). To estimate runtimes we would like our cost measure to have a linear relation to the actual runtimes. For a perfect cost measure, the ratio of cost and runtime would be constant. However, we see already from Figure 2 that the factors between different systems vary between the graphs. So there cannot be a cost measure that uses only properties of the graph and completely accurately predicts the runtime with a fixed "processing speed" for each system. Figure 4 shows the resulting speed value for a selection of the input graphs.

For lower runtimes and cost measures the results vary widely, as might be expected. The speed is in general lower because it takes some time to start up, analyze and optimize the query, create caches, and so on. In particular, for Jena even a small graph takes 2 seconds. For larger inputs, the speed values do not vary so much. If one wants to reduce all the measurements to a single number for each system, an average speed still might be useful first indicator. Of course, this is a "lossy compression". The average over all graphs is shown in Figure 5.

From the speed and the cost measure of the input graph, one can compute an estimate for the runtime (cost divided by speed). If one tries to minimize the number of cases where the error is more than 2 s and more than 20% of the estimate, the average time might not be the best choice. We used the speed values shown in the second row of Figure 5 (the result of "manual optimization" with the total error of the limit violations as secondary optimization goal). For XSB, the estimate was outside the limits only for 4 (out of 35) graphs (and not very far outside). Thus, the XSB runtime is quite well predictable from our cost measure. For BAM, there are 6 errors. For other programs, the estimation does not work well. In case of Jena, the estimation formula probably should consider the large startup time of the JVM.

Graph	BAM	XSB [Factor]		YAP [Factor]		Jena [Factor]		SQLite [Factor]	
k_1k	32.291	103.074	3.2	70.120	2.2	180.557	5.6	595.849	18.5
k_2k	363.309	810.679	2.2	552.711	1.5	1477.700	4.1	4983.400	13.7
a_1k	10.595	16.272	1.5	12.660	1.2	34.982	3.3	95.443	9.0
a_2k	130.801	127.694	1.0	122.537	0.9	255.690	2.0	811.345	6.2
c_1k	0.137	0.574	4.2	0.496	3.6	3.738	27.3	1.999	14.6
c_2k	0.449	1.514	3.4	2.110	4.7	8.098	18.0	7.816	17.4
c_4k	2.948	5.511	1.9	9.628	3.3	23.023	7.8	31.396	10.6
s_2k_1	0.554	1.765	3.2	3.090	5.6	9.347	16.9	9.900	17.9
s_2k_2	0.660	2.054	3.1	4.137	6.3	10.071	15.3	12.311	18.7
s_2k_3	0.804	2.337	2.9	4.607	5.7	10.615	13.2	14.030	17.5
s_2k_4	0.947	2.595	2.7	6.108	6.4	11.367	12.0	15.947	16.8
p_1k	0.081	0.344	4.2	0.184	2.3	2.885	35.6	0.998	12.3
p_2k	0.188	0.939	5.0	1.015	5.4	5.370	28.6	3.740	19.9
p_4k	0.758	3.055	4.0	4.341	5.7	13.797	18.2	15.016	19.8
m_4ki_2	3.656	6.167	1.7	9.240	2.5	24.628	6.7	32.829	9.0
m_1ki_8	0.331	1.833	5.5	2.149	6.5	8.954	27.1	7.688	23.2
m_256_32	0.126	0.726	5.8	0.691	5.5	3.978	31.6	1.920	15.2
m_64_128	0.110	0.240	2.2	0.193	1.8	2.663	24.2	0.629	5.7
m_16_512	0.010	0.130	13.0	0.057	5.7	2.290	229.0	0.206	20.6
m_4_2ki	0.000	0.106		0.047		2.065		0.113	
b_17	0.471	1.144	2.4	1.058	2.2	8.082	17.2	3.957	8.4
b_18	1.832	1.989	1.1	2.496	1.4	14.817	8.1	8.119	4.4
b_19	8.590	4.016	0.5	4.587	0.5	31.808	3.7	17.402	2.0
u_1k_50k	1.914	5.230	2.7	12.394	6.5	13.711	7.2	33.188	17.3
u_1k_125k	5.284	12.440	2.4	30.228	5.7	26.976	5.1	79.150	15.0
u_1k_250k	11.025	24.609	2.2	53.183	4.8	50.019	4.5	155.578	14.1
u_2k_250k	17.155	40.852	2.4	114.367	6.7	85.051	5.0	286.996	16.7
u_2k_500k	39.972	98.343	2.5	291.672	7.3	197.237	4.9	676.669	16.9
u_2k_1m	83.816	194.181	2.3	515.968	6.2	371.938	4.4	1303.429	15.6
f_1k_50k	0.710	1.825	2.6	2.442	3.4	6.631	9.3	9.727	13.7
f_1k_125k	1.867	4.118	2.2	7.084	3.8	11.791	6.3	24.343	13.0
f_1k_250k	3.623	8.390	2.3	17.066	4.7	20.468	5.6	48.579	13.4
f_2k_250k	6.536	12.854	2.0	17.031	2.6	31.899	4.9	85.702	13.1
f_2k_500k	16.727	32.402	1.9	55.599	3.3	72.170	4.3	213.121	12.7
f_2k_1m	34.841	65.082	1.9	142.702	4.1	138.465	4.0	417.273	12.0

Fig. 2. Run Time (Real Time in s) for TCFF Benchmark, Factor compared with BAM

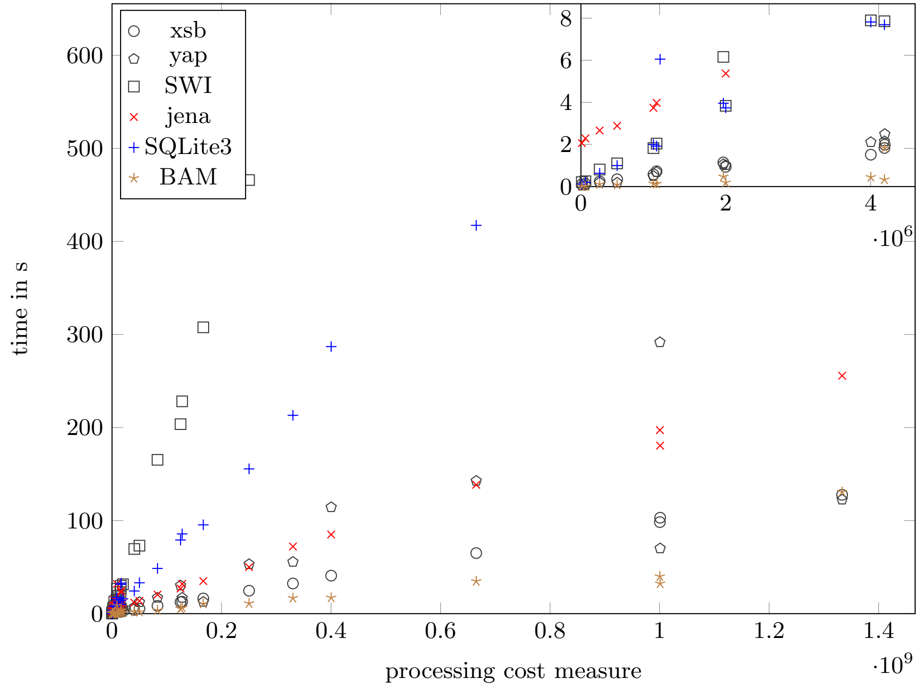

Fig. 3. Time vs cost measure

instance	p_2k	c_2k	b_18	s_2k_2	m_4ki_2	u_1k_50k	f_2k_250k	a_1k	f_2k_500k	u_2k_250k	f_2k_1m	k_1k	k_2k
cost in 10^6	2	4	4	12	17	50	128	167	330	400	665	1001	8004
BAM	10.63	8.91	2.29	18.19	4.59	26.15	19.60	15.73	19.75	23.33	19.09	31.00	22.03
XSB	2.13	2.64	2.11	5.85	2.72	9.57	9.96	10.24	10.20	9.80	10.22	9.71	9.87
YAP	1.97	1.90	1.68	2.90	1.82	4.04	7.52	13.17	5.94	3.50	4.66	14.28	14.48
SWI	0.52	0.51	0.32	0.62		0.69	0.56	0.54	0.47	0.59	0.47	0.51	0.50
Jena	0.37	0.49	0.28	1.19	0.68	3.65	4.02	4.76	4.58	4.71	4.80	5.54	5.42
SQLite3	0.53	0.51	0.52	0.98	0.51	1.51	1.49	1.75	1.55	1.39	1.59	1.68	1.61
DataScript	0.0004	0.0004	0.0259										

Fig. 4. System speed for selected graphs (10^6 rule instances/s)

System	BAM	XSB	YAP	Jena	SQLite
Avg. Speed (10^6 rule inst./s)	15.08	6.04	4.27	2.37	1.05
Speed for runtime estimation	19.90	8.31	4.13	4.41	1.39
Estimation errors (from 35)	6	4	11	22	12

Fig. 5. Average speed over all measured graphs, Runtime estimation

5.2 Memory Estimation

As we can see from Figure 6, our memory cost measure (number of facts) seems to have a relatively good correlation to the actually used memory for some systems.

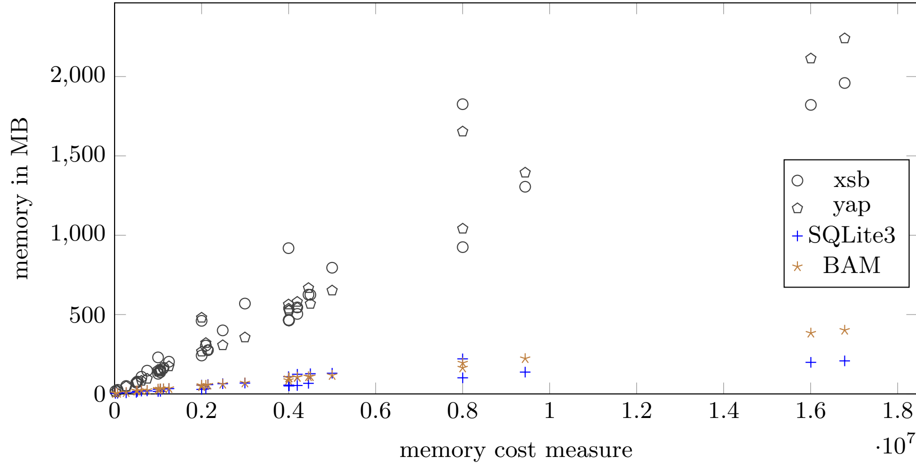

Fig. 6. Memory vs Memory Cost Measure

For Jena and SWI-Prolog, the memory cost measure does not seem to work well, therefore we removed them from the graph. In case of Jena, the memory management of the JVM with a garbage collector running at undefined times could explain the problems.

5.3 Additional Observations

The two problems K_{1k} (complete graph with 1000 edges) and $U_{2k,500k}$ (random graph with 2000 vertices and 500000 edges) have nearly identical cost (about 10^9) but for all systems the complete graph, with fewer vertices but more edges, was a bit faster to calculate than the cost measure would have suggested.

While the SPARQL query for Jena was quite expressive, the implementation for Property Paths was a problem for us. With the default settings we quickly had a stack overflow in the Java Virtual Machine for graphs that had a diameter of more than 2000. We changed some memory settings of the JVM to try to mitigate this problem.

DataScript was incredibly slow and often ran out of memory. Some benchmarks ran up to 40 hours when we terminated them. In the future we want to try smaller graphs to see whether our cost measure holds for problem sizes that are easier to handle for DataScript.

6 Conclusions

The source code, data files and single benchmarking results are available at

[http://dbs.informatik.uni-halle.de/rbench/]

We are working on integrating other systems (e.g., PostgreSQL and Soufflé) and more benchmarks into our scripts.

References

1. Brass, S., Stephan, H.: Bottom-up evaluation of Datalog: Preliminary report. In: Schwarz, S., Voigtländer, J. (eds.) Proc. WLP'15/'16/WFLP'16. pp. 13–26. No. 234 in EPTCS, Open Publishing Association (2017), https://arxiv.org/abs/1701.00623
2. Brass, S., Stephan, H.: Experiences with some benchmarks for deductive databases and implementations of bottom-up evaluation. In: Schwarz, S., Voigtländer, J. (eds.) Proc. WLP'15/'16/WFLP'16. pp. 57–72. No. 234 in EPTCS, Open Publishing Association (2017), https://arxiv.org/abs/1701.00627
3. 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
4. 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)
5. Costa, V.S., Rocha, R., Damas, L.: The YAP Prolog system. Theory and Practice of Logic Programming 12(1–2), 5–34 (2012), https://www.dcc.fc.up.pt/~ricroc/homepage/publications/2012-TPLP.pdf
6. Liang, S., Fodor, P., Wan, H., Kifer, M.: OpenRuleBench: An analysis of the performance of rule engines. In: Proceedings of the 18th International Conference on World Wide Web (WWW'09). pp. 601–610. ACM (2009), http://rulebench.projects.semwebcentral.org/
7. Przymus, P., Boniewicz, A., Burzańska, M., Stencel, K.: Recursive query facilities in relational databases: A survey. In: Zhang, Y., Cuzzocrea, A., Ma, J., Chung, K., Arslan, T., Song, X. (eds.) 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
8. Reutter, J.L., Soto, A., Vrgoč, D.: Recursion in SPARQL. In: Arenas, M., et al. (eds.) The Semantic Web — ISWC 2015, 14th International Semantic Web Conference, Proceedings, Part I. LNCS, vol. 9366, pp. 19–35. Springer (2015)
9. 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
10. Schütz, H.: Tupelweise Bottom-up-Auswertung von Logikprogrammen (Tuple-wise bottom-up evaluation of logic programs). Ph.D. thesis, TU München (1993)
11. SPARQL 1.1 Query Language, W3C Recommendation (March 2013), http://www.w3.org/TR/sparql11-query/