Experiences with Some Benchmarks for Deductive Databases and Implementations of Bottom-Up Evaluation

Stefan Brass Heike Stephan
Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg, Germany

brass@informatik.uni-halle.de stephan@informatik.uni-halle.de

OpenRuleBench [14] is a large benchmark suite for rule engines, which includes deductive databases. We previously proposed a translation of Datalog to C++ based on a method that "pushes" derived tuples immediately to places where they are used. In this paper, we report preformance results of various implementation variants of this method compared to XSB, YAP and DLV. We study only a fraction of the OpenRuleBench problems, but we give a quite detailed analysis of each such task and the factors which influence performance. The results not only show the potential of our method and implementation approach, but could be valuable for anybody implementing systems which should be able to execute tasks of the discussed types.

1 Introduction

With deductive database technology, the range of tasks a database can do is increasing: It can not only fetch and compose data, as classical SQL queries, but a larger part of the application can be specified declaratively in a pure logic programming language. It is common that database systems offer stored procedures and triggers today, but this program code is written in an imperative language, not declaratively as queries in SQL. In addition, still other (non-declarative) languages are used for the application development itself. It is the claim of deductive databases like LogicBlox [1] to unify all this.

Making deductive databases a successful programming platform for future applications has many aspects, including the design of language features for declarative, rule-based output (see, e.g., [6]). But improving the performance of query evaluation / program execution is still an important task. Some older deductive database prototypes have been not very good from the performance viewpoint. Typically, people who are not yet convinced of the declarative approach ask about performance as one of their first questions. Furthermore, the amount of data to be processed is constantly increasing. New processor architectures offer new opportunities, which may not be utilized in old code.

Benchmarks can be a useful tool to evaluate and compare the performance of systems. Furthermore, they can be motivating for new system developers. In the area of deductive databases, OpenRuleBench [14] 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 [14] contains 18 tables with benchmark results. The logic programs used for the tests (with different syntax variants and settings for different systems), the data files and supporting shell scripts can be downloaded from

`http://www3.cs.stonybrook.edu/~pfodor/openrulebench/download.html`

The tests have been re-run in 2010 and 2011, the results are available at:

`http://rulebench.semwebcentral.org/`

Submitted to:
WLP 2016

© S. Brass & H. Stephan
This work is licensed under the Creative Commons
Attribution-Noncommercial-No Derivative Works License.

In this paper, we look at only five of the benchmark problems, but in significantly more detail than the original OpenRuleBench paper. We are currently developing an implementation of our "Push"-method for bottom-up evaluation [5, 7], and would like to document and to share our insights from many preformance tests with different variants. We compare our method with XSB [18], YAP [10], and dlv [13]. It is also of some value that we did these measurements again five years later with new versions of these systems on a different hardware. We also put more emphasis on the loading time and overall runtime than the authors of the original OpenRuleBench paper [14]. In addition, we did some measurements on main memory consumption.

As we understand the rules of OpenRuleBench, the options for each benchmark and system were manually selected in cooperation with the system developers. This means that a good automatic optimization does not "pay off" in the benchmark results, although it obviously is of importance for the usability of the system. However, this also helps to separate two different aspects: The basic performance of program execution and having an intelligent optimizer, which finds a good query execution plan and index structures. For systems that allow to influence query evaluation with many options, the optimizer becomes less important for the OpenRuleBench results.

Since our query execution code is in part manually created and we still try different data structures, this can be seen as an extreme case of the manual tuning. It is our goal to test and demonstrate the potential of our approach, but we are still far from a complete system. Naturally, before investing maybe years to develop a real deductive database system, it is good to see what can be reached, and whether it is worth to follow this path.

In contrast, each of the three systems we use as a comparison has been developed over more than a decade. They are mature systems which offer a lot of programming features which are important for usage in practical projects. Although our performance numbers are often significantly better, our prototype could certainly not be considered as a competitor for real applications.

Our approach is to translate Datalog (pure Prolog without function symbols) to C++, which is then compiled to native code by a standard compiler. We have an implementation of this translation (written in Prolog), and a library of data structures used in the translated code written in C++:

`http://www.informatik.uni-halle.de/~brass/push/`

At the moment, the result of the translation must still be manually copied to a C++ class frame, and compiler and linker are manually executed, but it should be an easy task to automize this. The tests reported in the current paper discuss different code structures, therefore it seemed better to first find the optimal structure before programming the complete translation. However, the current implementation of the Datalog-to-C++ translation already generates different variants of the main query execution code.

XSB compiles into machine code of an abstract machine (XWAM) and interprets that code. The same is true for YAP [10] (the abstract machine is called YAAM). DLV probably uses a fully interpreted approach. Since we compile to native code, this obviously gives us an advantage. If one compares the numbers given in [9] for Sicstus Prolog native code vs. emulated bytecode, the speedup for native code is between 1.5 and 5.6, with median value 2.6. Of course, the speedup also depends on the granularity of the operations that are executed. For instance, if many operations are index lookups or other large database operations, these are in native code in the virtual machine emulator anyway, so that the overhead of interpretation is not important in this case. But this means that the speedup that must be attributed to using native code might even be less than what is mentioned above.

One of the strengths of the "Push" method investigated in this paper is that it tries to avoid copying data values (as far as possible). In the standard implementation of bottom-up evaluation, the result of applying a rule is stored in a derived relation. In contrast, Prolog implementations do not copy variable values or materialize derived literals (except with tabling). It might be that the "Push" method

makes bottom-up evaluation competitive because it gives it something which Prolog systems already had. However, as we will see, good performance depends on many factors (including, e.g., efficient data structures). The message of this paper is certainly not that the "Push" method alone could make us win benchmarks.

2 Query Language, General Setting

In this paper, we consider the very basic case of Datalog, i.e. pure Prolog without negation and without function symbols (i.e. terms can only be variables or constants). A logic program is a set of rules, for example

```
answer(X) :- grandparent(sam, X).
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

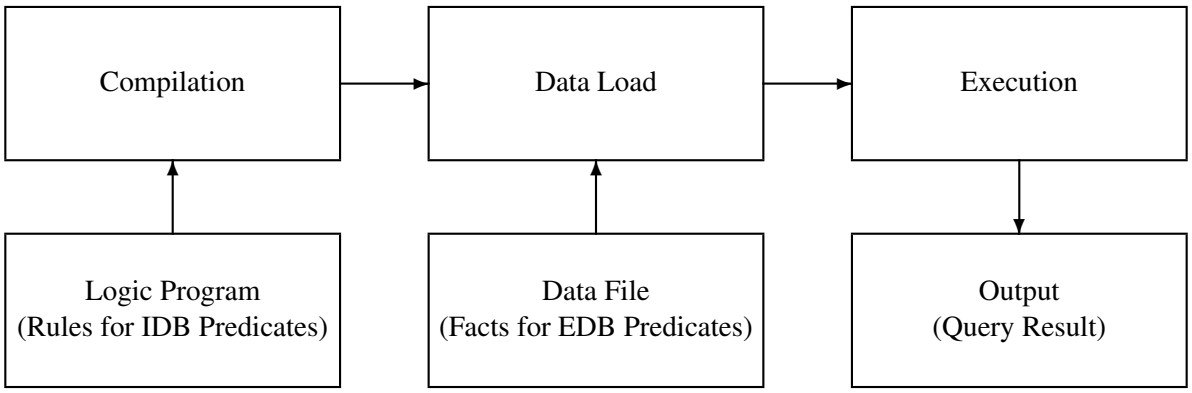
This computes the grandparents of `sam`, given a database relation `parent`. For our bottom-up method, we assume that there is a "main" predicate `answer`, for which we want to compute all derivable instances. For the other systems, the body of the `answer`-rule is directly posed as query/goal.

We require range-restriction (allowedness), i.e. all variables in the head of the rule must also appear in a body literal. Therefore, when rules are applied from right (body) to left (head), only facts are derived, i.e. variable-free atomic formulas like `grandparent(sam, john)`.

As usual in deductive databases, the predicates are classified into

- EDB-predicates ("extensional database"), which are defined only by facts (usually a large set of facts stored in a database, or maybe specially formatted files). In the above example, `parent` is an EDB predicate.
- IDB-predicates ("intensional database"), which are defined by rules. In the example, `grandparent` and `answer` are IDB predicates.

The execution of a logic program/query has three distinct phases:



The input to our transformation are the rules for the IDB-predicates, i.e. these are known at "compile time". The concrete data for the EDB-predicates are only known at "run time". Therefore, the C++ program resulting from our translation can be executed for different database states (sets of facts). In this way, optimizations or precomputations done at compile time can be amortized over many program executions. Actually, since the database is usually large and the program relatively small, even a single execution might suffice so that work invested during the transformation "pays off".

In the OpenRuleBench tests, the runtime is again split into the time required for loading the data file and the real execution time when the data is already in main memory (and useful data structures

have been built). In the OpenRuleBench paper [14], the authors are interested only in the time for the inference (i.e. the execution phase), and only this time is shown in the paper. We show the times for both, the data load and the execution, and the overall runtime measured externally. Only in case of DLV it was not possible to separate this (we did measure the time for only loading the data, but it is not clear how much of the data structure building is done if the data is not used).

Of course, handling the data load separately is interesting if many queries are executed later on the data in main memory, i.e. if one had a main memory DBMS server.

For the most part, we did not consider the compilation time which might be considered as giving us an unfair advantage. Systems which have no separate compilation phase cannot profit from this.

3 Data Structures

When the facts from a data file are loaded, they are stored in main-memory relations. A large part of the C++ library which is used in the translation result implements various data structures for main-memory relations. Most relations are C++ templates which are parameterized with the tuple type. However, there are a few exceptions (experimental/old implementations, or data structures for very special cases).

Actually, all arguments are currently mapped to integers. We have implemented a string table using a radix tree similar to [12] in order to assign unique, sequential numbers to strings. If we know that different arguments have different domains (which are never joined) we use different such "string tables" to keep the numbers dense (which is good for array based data structures).

There are basically three types of relation data structures:

- Lists, which support access to the tuples with binding pattern "`ff...f`", i.e. all arguments are free (output arguments). One can open a cursor over the list and iterate over all its elements (tuples). Lists are implemented as a tree of 4 KB pages, each containing an array of tuples (i.e. the tuples are stored in consecutive memory locations which helps to improve CPU cache utilization). The next level of the tree contains pages with pointers to the data pages (similar to the management of blocks in the original Unix file system).

- Sets, which support access to the tuples with binding pattern "`bb...b`", i.e. all arguments are bound (input arguments). One can insert a tuple and gets the information whether the insertion was successful (new tuple) or not (the tuple was contained in the set already). We are experimenting with different set implementations, currently: (a) a simple hash table of fixed size, (b) a dynamic hash table which doubles its size if it gets too full, and (c) an array of bitmaps (some of the benchmarks have only a small domain of integers from 0 to 1000).

- Maps (or really multimaps) for all other cases, e.g. a binding pattern like `bf`: Given values for the input arguments (in this case, the first one), one can open a cursor over the corresponding (tuples of) values for the output arguments. The current state is that we have only a flexible array implementation for a single integer input argument, but a general map will be implemented soon.

Temporary relations might be needed during execution of a logic program: If a rule has more than one IDB body literal, previously derived facts matching these literals must be stored (except possibly one of the literals, if we know that its facts arrive last and we can immediately process them). In this case it is important that when a cursor is opened, the set of tuples it loops over is not affected by future insertions.

Temporary sets are also needed for duplicate elimination during query evaluation. For some benchmarks, this has a big influence on speed.

4 The Push Method for Bottom-Up Evaluation

The push method has been introduced in [5], and a version with partial evaluation has been defined in [7]. The following is just a quick reminder in order to be able to understand different variants of the code.

The basic method only works with rules with at most one IDB body literal (as generated, e.g., by the SLDMagic method [4]). Rules with multiple IDB body literals are normalized by creating intermediate storage for previously derived tuples (similar to seminaive evaluation with a single tuple as "delta" relation, see [7]).

In the push method, a fact is represented by a position in the program code and values of C++ variables. A fact type is a positive IDB literal with constants and C++ variables as arguments, for instance `p(a, v_1, v_2)`. If execution reaches a code position corresponding to this fact type, the corresponding fact with the current values of the variables `v_1` and `v_2` has been derived. Now this derived fact is immediately "pushed" to rules with a matching body literal. A rule application consists of

- a rule of the given program,
- a fact type that is unifiable with the IDB body literal of this rule, or a special marker "`init`" if the rule has only EDB body literals,
- a fact type that is general enough to represent all facts which can be derived with the rule applied to the input fact type.

A rule application graph consists of the fact types as nodes (plus an "`init`" node), and the rule applications as edges.

One can use different sets of fact types. In [5] the fact types had the form `p(p_1, p_2, ...)`, i.e. there was one C++ variable for each argument position of each IDB predicate. This requires a lot of copying in a rule application. The approach of [7] did avoid this by computing a set of fact types as a fixpoint of an abstract T_P-operator working with C++ variables instead of real constants. These variables were introduced if a data value was needed from an EDB body literal (the real values are not known at compile time). This avoids copying, but the set of generated fact types can "explode" (as, e.g., in the wine ontology benchmark). Therefore, our current implementation supports both approaches, and we work on mixed variants which combine the advantages of both.

There is one code piece for each rule application. At the end of this code piece, a fact has been derived and the variables of the resulting fact type have been set.

Of course, a derived fact might match body literals of several rules. In that case control jumps to one such rule application, and the other possibilities are later visited via backtracking. If variables are changed when there might still be backtrack points on the stack which need the old value, these are saved on a stack, too. In both of the above variants of the push method, this happens only for recursive rule applications.

It is also possible that a rule application can produce several facts. Only one is computed at a time and immediately used, other facts are computed later via backtracking. Therefore, the code piece for a rule application has two entry points: One for computing the first fact ("`START`"), and one for computing an additional fact ("`CONT`"). Before control jumps to a place where the derived fact is used, the "`CONT`"-label is put on the backtrack stack. In case a rule application is unsuccessful or there is no further fact that can be derived, the main loop is continued, which takes the next rule application from the backtrack stack and executes its code piece.

5 About the Time and Memory Measurements

We compared our method with the following systems:

- XSB Prolog version 3.6 (Gazpatcho)
- YAP Prolog version 6.2.2
- DLV version x86-64-linux-elf-static

Both Prolog systems support tabling. The DLV system uses a variant of magic sets transformation and then does bottom-up evaluation.

We executed the benchmarks on a machine with two 6272 AMD Opteron 16 Core CPUs. However, the current version of our program does not use multiple threads (we will consider this in future research). The machine has 64 GB of RAM. This is quite a lot, but we also measure how much memory is really used by the systems. The operating system is Debian `x86_64` GNU/Linux 3.2.63.

The overall execution time ("elapsed wall clock time") and the memory ("maximum resident set size") for each test was measured with the Linux `time` program. The time for loading the data and for executing the query are measured by functions of each system.

Every test was run ten times and the time average values were calculated. Additionally, for the XSB and YAP systems the times for loading the data and program files and for executing the queries were separately measured.

For the DLV system the use of the `-stats` option is officially discouraged for benchmark purposes, so for estimating the loading time the loading of the data files was measured once using the `time` program; the value appears in parentheses in the tables.

XSB Prolog compiles the input files to a `xwam` file which can be loaded quite fast, but for better comparison with the other systems dynamic loading was used.

6 The DBLP Benchmark

For the DBLP benchmark [14], data from the DBLP bibliography (`dblp.uni-trier.de`) is stored as a large set of facts for an EAV-relation

```
att(DocID, Attribute, Value).
```

The file contains slightly more than 2.4 million facts of this type, for instance:

```
att('node12eo5mnsvx56023',title,'On the Power of Magic.').
```

It is about 122 MB large. The test query is

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

In this case, `att` is an EDB-predicate, and `answer` is the only IDB-predicate. The file contains data of about 215.000 documents (i.e. 11.3 facts per document).[1]

[1] The DBLP data are available as XML file from `http://dblp.uni-trier.de/xml/`. The DBLP has now data of 3.4 million documents, the XML-file is 1.8 GB large. We plan to transform this larger data file to the same type of Datalog facts as the official OpenRuleBench DBLP test file.

It seems that the key to success for this benchmark problem are the data structures to represent the facts. When the data is loaded, we must store it in relation data structures for later evaluation of the EDB body literals. Note that the program/query is known when the data is loaded, so we can try to create an optimal index structure for each EDB body literal. In particular, selections for constants known at compile time can already be done when the data is loaded. In the example, all body literals contain different constants, therefore we create a distinct relation for each of them.

E.g. there is a relation `att_title`, which represents `att`-facts that match the first body literal `att(Id, title, T)`. Of course, the constant is not explicitly stored, therefore the relation has only two columns. We use a nested loop/index join. The first occurrence of each variable binds that variable, and at all later occurrences, its value is known. Since this is the first body literal, both variables are still free, and the relation `att_title` is accessed with binding pattern `ff`, i.e. it is a list.

For the second body literal `att(Id, year, Y)`, there is a relation `att_year` which represents only `att`-facts with `year` as second argument. Again, this argument is not explicitly represented in the stored tuples. Since the value of the variable `Id` is known when this body literal is evaluated, the relation must support the binding pattern `bf`. Therefore it is a map (or really multimap, since we do not know that there is only one year for each document).

In the same way, the other two body literals are turned into exactly matching relations of type `bf`. This means that although the loader has to look at all facts in the data file, it stores only those facts which are relevant for the query (which are 35% of the facts in the data file).

The strings that occur in the data are mapped to unique, sequential integers. Since the three arguments of the predicate `att` are never compared, we use three distinct "string tables": This makes it simpler to implement the maps based on a kind of flexible array — for other map data structures, this would probably have no advantage. However, also bitmap implementations of sets profit from a small, dense domain. In summary, the main data structures are:

Data Structure	Rows/Strings	Memory (KB)
String table for Arg. 1	214905	91612
String table for Arg. 2	19	4
String table for Arg. 3	923881	284052
List for `att(Id, title, T)`	209943	1648
Map for `att(Id, year, Y)`	209944	844
Map for `att(Id, author, A)`	440934	3688
Map for `att(Id, month, A)`	2474	844

One might expect that creating these indexes slows down the loading of data (in effect, some time is moved from the execution phase to the load phase). However, our loader is quick. Here are the benchmark results:

System	Load (ms)	Execution (ms)	Total time (ms)	Factor	Memory (KB)	Mem. Diff.
Push	2565	22	2610	1.0	385172	383990
XSB	89045	2690	92390	35.4	415500	404535
XSB (trie)	90710	269	91275	35.0	380259	369294
YAP	24878	7370	32438	12.4	813760	808911
DLV	(20110)	—	25898	9.9	926864	926167

The total time is dominated by the loading time, however the pure execution time is also interesting. By the way, just reading the file character by character with the standard library takes about the same time

as our loader (which reads the file in larger chunks). In the OpenRuleBench programs, XSB did not use the trie index, but this dramatically improves execution time (the influence on the total time is not big).

The main memory consumption of our "Push" implementation is acceptable (383 MB, which is about the same as XSB, and less than half of YAP and XSB). However, since only one third of the facts are actually stored, and only with a specific binding pattern, memory could become an issue for other application scenarios.

In order to check how much memory was used by the data for the benchmark and how much is program code in the system (including possibly large libraries) we determined the memory used by just starting and stopping each system (without loading data or compiling rules). The result is:

System	Base Memory (KB)
Push	1 182
XSB	10 965
YAP	4 849
DLV	697

The column "Mem. Diff." in the benchmark result table contains the difference of the memory used in the benchmark minus this baseline.

Since this query is a standard database task, we also tried HSQLDB 2.3.4. It loads all data into main memory when the system starts, this took 32 471 ms. Executing the following SQL query took 3 863 ms:

```
SELECT COUNT(*)
FROM   ATT ATT_TITLE, ATT ATT_YEAR, ATT ATT_AUTHOR, ATT ATT_MONTH
WHERE  ATT_TITLE.ATTRIBUTE = 'title'
AND    ATT_YEAR.ATTRIBUTE = 'year'
AND    ATT_AUTHOR.ATTRIBUTE = 'author'
AND    ATT_MONTH.ATTRIBUTE = 'month'
AND    ATT_TITLE.DOC_ID = ATT_YEAR.DOC_ID
AND    ATT_TITLE.DOC_ID = ATT_AUTHOR.DOC_ID
AND    ATT_TITLE.DOC_ID = ATT_MONTH.DOC_ID
```

The total time for starting the database, executing the query and shutting down the database was 40 777 ms (the shutdown took 4 330 ms). HSQLDB did use three threads (300% CPU) and 3 GB of main memory (i.e. three times the system resources of the deductive systems above). HSQLDB is written in Java (and was executed in the OpenJDK 64-bit Server VM: IcedTea 2.6.6, Java 1.7.0). We used an index over `(ATTRIBUTE, DOC_ID)`. We tried also an index with the two attributes inversed, or all three attributes, but all this did not change much. Although there might be further options for tuning HSQLDB, it is at least not easy to get much better runtimes with a relational database.

7 The Join 1 Benchmark

The Join1 example from [14] contains the following rules:

```
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 EDB predicates are `c2`, `c3`, `c4`, `d1`, `d2`. There are three data files: One with 10 000 facts each (i.e. 50 000 facts in total), one with 50 000 facts each, and one with 250 000 facts each. The data values are randomly generated integers between 1 and 1000. Different queries are considered in [14], namely `a(X, Y)`, `b1(X, Y)`, `b2(X, Y)`, and the same with bound first or second argument, e.g. the query `a(1, Y)`. In our test, we only tried the query `a(X, Y)`.

The benchmark results for the small data file (10 000 facts per EDB-predicate, 624 KB) are:

System	Load (ms)	Execution (ms)	Total time (ms)	Factor	Memory (KB)
Push (Bitmap)	14	1 772	1 787	1.0	7 311
Push (Dyn.Hashtab)	11	10 372	10 383	5.8	45 688
xsb-btc	141	22 432	22 827	12.8	82 667
XSB	148	24 551	25 002	14.0	82 526
YAP	421	17 557	18 067	10.1	10 933
DLV			147 172	82.4	461 646

The benchmark scripts (from the authors of [14]) used a version of XSB with "batched scheduling" for this test. However, it gives only a relatively small improvement over the standard version (with "local scheduling").

The domain of values are in all cases integers from 1 to 1000, so many duplicates will be generated. Actually, this test is dominated by the time for duplicate elimination. With an "array of bitmaps" implementation, we are much faster than with a dynamic hash table (it doubles its size if the chains get too long). Here are the relations used in our Push implementation:

Table	Comment	Rows	Memory (KB)
`c2_bf`	EDB-Relation (Map)	10 000	88
`c3_ff`	EDB-Relation (List)	10 000	84
`c4_bf`	EDB-Relation (Map)	10 000	88
`d1_ff`	EDB-Relation (List)	10 000	84
`d2_bf`	EDB-Relation (Map)	10 000	88
`b1_fb`	Temporary IDB-Relation (Map)	634 088	4 976
`b1_bb`	Duplicate Check for IDB-Pred. (Set)	634 088	14 004
`b2_bb`	Duplicate Check for IDB-Pred. (Set)	95 954	2 524
`c1_bb`	Duplicate Check for IDB-Pred. (Set)	95 570	2 520
`a_bb`	Duplicate Check for IDB-Pred. (Set)	1 000 000	19 724

In the version with bitmap duplicate check, the last four sets need only 128 KB each. The temporary IDB-relation is needed because the first rule has two IDB-body literals:

a(X, Y) :- b1(X, Z), b2(Z, Y).

Therefore, we compute first the `b1`-facts and store them in `b1_fb`. Later, when we derive a `b2`-fact, we use it immediately to derive `a`-facts with this rule. Since in this case `Z` is bound from the `b2`-fact, we need the binding pattern `fb` for the intermediate storage of `b1`-facts.

If one does not do any duplicate elimination, one gets 99.9 million result tuples, i.e. on average every answer is computed 100 times. Our implementation of this needed 2 501 ms execution time, i.e. was quite fast. This shows again that this benchmark depends a lot on the efficiency of duplicate elimination.

Our results with the hash table are not impressive and could be fully explained with using native code instead of emulating an abstract machine. However, it scales better than XSB as the other data sets

show. XSB timeouts already on the middle data set with 50 000 rows per EDB predicate (timeout is set at 30 minutes). Our Push implementation (with hash table) needs 106 s. This means that it is better by at least the factor 17.

However, we found that tabling all IDB predicates for XSB and YAP and introducing indexes for the EDB relations in XSB dramatically improved the runtime performance of these systems (in the original program files, tabling was only done for the predicate `a/2` in XSB, and no indices were used). This led to the following results:

System	Load (ms)	Execution (ms)	Total time (ms)	Memory (KB)
XSB	414	11 302	11 927	123 885
YAP	437	6 719	7 256	133 622

Due to creating indexes, the loading time of XSB has increased, and both systems need more memory for the tables.

8 The Transitive Closure Benchmark with Both Arguments Free

An important test for a deductive database system is the transitive closure program:

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

Of course, this is one of the OpenRuleBench benchmark problems for recursion. It uses 10 data files of different size (see below), we first concentrated on the file `tc_d1000_parsize50000_xsb_cyc.P` where the arguments of the `par`-relation are taken from a domain of 1000 integers (1 to 1000) (i. e. the graph has 1000 nodes); there are 50 000 randomly generated `par`-facts (i. e. the graph has 50 000 edges); the graph is cyclic; and the file size is 673 KB. OpenRuleBench used three test queries: `tc(X, Y)`, `tc(1, Y)`, and `tc(X, 1)`[2]. Here are the benchmark results for the first query asking for the entire `tc` relation (i. e. with binding pattern `ff`):

System	Load (ms)	Execution (ms)	Total time (ms)	Factor	Memory (KB)
Push	11	2 171	2 190	1.0	21 832
Seminaive	10	4 598	4 613	2.1	29 587
XSB	1 095	11 357	12 641	5.8	134 024
YAP	424	19 407	19 867	9.1	145 566
DLV	(260)	—	109 673	50.1	404 737

Obviously, one must detect duplicates in order to ensure termination. But even with non-cyclic data, there are many duplicates: Each node is linked to 5% of the other nodes. So the probability is quite high that even after short paths the same node is reached multiple times. XSB uses tabling to solve this problem [18, 8, 22]. The relations we used for this problem are:

Table	Comment	Rows	Memory (KB)
`par_ff`	EDB-relation (as list), used in first rule	50000	396
`par_fb`	EDB-relation (as map), used in second rule	50000	400
`tc_bb`	Duplicate check for result (set)	1000000	19 536

[2] Actually, anonymous variables were used in the test, because the answers are not further processed.

The first rule produces initial `tc`-facts and simply loops over the entire `par`-relation, therefore it needs a version implemented as list. The second (recursive) rule is activated in the "Push" method when a new `tc`-fact is derived. Therefore, a value for variable `Z` is known and the EDB-relation is accessed with binding pattern `fb`.

We also implemented a standard seminaïve bottom-up evaluation for comparison. It used the same data structures as our Push implementation, it only needed an additional list `tc_ff` for the result. It is slower than the "Push" method. Probably, this is due to the different memory accesses: In each iteration, freshly derived tuples are first stored in the result list, and they are accessed again in the next iteration.

The following table lists the benchmark results for the full set of 10 data files of various size and characteristics. We compared only the execution times with XSB. We used here the original settings for XSB from the OpenRuleBench scripts. In the result shown above, a `trie`-index was added, which improved the performance (12.6 s instead of 15.5 s).

Rows	Dom	cyc	XSB	Push	Factor
50 000	1000	N	5.980 s	0.732 s	8.2
50 000	1000	Y	15.513 s	2.160 s	7.2
250 000	1000	N	33.274 s	4.350 s	7.6
250 000	1000	Y	82.497 s	12.440 s	6.6
500 000	1000	N	76.673 s	10.230 s	7.5
500 000	1000	Y	187.200 s	30.620 s	6.1
500 000	2000	N	139.881 s	23.870 s	5.9
500 000	2000	Y	329.873 s	61.460 s	5.4
1 000 000	2000	N	297.870 s	56.623 s	5.3
1 000 000	2000	Y	714.721 s	150.270 s	4.8

The first column gives the number of rows in the database, i. e. the size of the `par`-relation. The second column lists the size of the domain of values which occur in the `par` columns.[3] The last column shows the speed improvement factor of our Push implementation over XSB. For loading the last relation (1 million rows, 14.2 MB), XSB needs 90.2 s, our Push implementation needs 0.2 s.

9 The Transitive Closure Benchmark with the First Argument Bound

In this benchmark, one is not interested in all connected pairs in the transitive closure, but only in nodes reachable from node `1`. So the binding pattern for accessing the predicate is `bf`.

Different systems use different methods to pass bindings to called predicates. E.g. for XSB and YAP, this is a feature of SLD-resolution. For a system based on bottom-up evaluation, the magic set method is the standard solution [2, 3]. Probably DLV does this. However, since magic sets are known to have problems with tail recursions [17], we use SLDMagic [4] for our "Push" method instead (the "Push" method alone is a pure bottom-up method would not pass query bindings). The output of the transformation is:

[3] Some of the graphs which claim to be non-cyclic actually do contain cycles. The largest non-cyclic graph with 1000 nodes is $\{(i, j) \mid 1 \leq i < j \leq 1000\}$. This has only $999 * 1000/2 = 499\,500$ edges.

```
p1(A) :- p1(B), par(B,A).
p1(A) :- par(1,A).
p0(A) :- p1(B), par(B,A).
p0(A) :- par(1,A).
tc(1,A) :- p0(A).
```

The predicate `p0` is not really needed, but since the SLDMagic prototype produces this program, it would be unfair to improve it manually. Nevertheless, it can be evaluated extremely fast, since it reduced the arity of the recursive predicate. Here are the benchmark results:

System	Load (ms)	Execution (ms)	Total time (ms)	Factor	Memory (KB)
Push	10	14	30	1.0	9 716
XSB	1 098	7 142	8 418	280.6	86 936
YAP	440	8 743	9 246	308.2	91 325
DLV	(260)	—	110 779	3 692.6	404 736

At least the standard version of tabling has the same problem with tail recursions as magic sets: One cannot materialize the derived `tc`-facts in this case, since they are not only for the type `tc(1, Y)`, but all facts `tc(X, Y)` for every `X` reachable from `1`. Depending on the graph, this can make the difference between a linear number of facts and a quadratic number of facts. XSB and YAP might be affected by this problem.

It could be argued that the good results for the Push Method are mainly due to the SLDMagic program transformation. Therefore, in a further test, the SLDMagic-transformed program was also given as input to the other systems. Indeed, they did profit from the transformation (in runtime as well as in memory usage), but the Push Method is still in the first place:

System	Load (ms)	Execution (ms)	Total time (ms)	Factor	Memory (KB)
Push	10	14	30	1.0	9 716
XSB	1 093	10	1 292	43.1	14 298
YAP	449	36	562	18.7	16 696
DLV	(260)	—	389	13.0	11 997

For the other data files, we compared the execution times only with XSB (using SLDMagic for the Push method, but not for XSB):

Rows	Dom	cyc	XSB	Push	Factor
50 000	1000	N	1.296 s	0.010 s	130
50 000	1000	Y	6.912 s	0.010 s	691
250 000	1000	N	9.309 s	0.030 s	310
250 000	1000	Y	35.098 s	0.030 s	1170
500 000	1000	N	19.989 s	0.050 s	400
500 000	1000	Y	69.929 s	0.060 s	1165
500 000	2000	N	36.067 s	0.060 s	601
500 000	2000	Y	154.110 s	0.070 s	2202
1 000 000	2000	N	80.117 s	0.130 s	616
1 000 000	2000	Y	300.719 s	0.150 s	2005

The case with the second argument bound still has to be investigated. The SLDMagic method is no advantage in this case. So our Push method would need the same time as in the `tc(X, Y)` case, which means 2.171 s for execution. XSB needs only 0.020 s for execution, i.e. is better by a factor of 109! Since our data loader is quicker, the factor for total time is less than 2, but XSB is still better. Of course, we are working on improving the SLDMagic method. But that is a different topic.

10 The Wine Ontology Benchmark

The wine ontology benchmark consists of 961 rules with 225 IDB-predicates, of which all but one are recursive, and 113 EDB-predicates. The program is basically one big recursive component.[4]

System	Load (ms)	Execution (ms)	Total time (ms)	Factor	Memory (KB)
Push	1	2 255	2 260	1.0	9 236
XSB	106	8 548	8 851	3.9	322 894
YAP	52	10 793	10 899	4.8	334 761
DLV	(60)	—	31 743	14.0	42 452

This is an example where the Push method with partial evaluation as introduced in [7] "explodes": It produces a lot of different specializations of the rules, of which there are already many in the input program. Therefore, we used the version of the push method without partial evaluation from [5] here. Even with that, the resulting C++ program is large (34 294 lines). The detection of duplicates is essential for termination. We used the hash table here, probably a bitmap would further improve performance.

Standard implementations of bottom-up evaluation would iterate all rules in a recursive clique until one such iteration did not produce a new fact. In this example, this would be very inefficient because basically all rules form a single recursive clique, but in each iteration only a few rules actually "fire". In contrast, the "Push" method only looks at rules which are activated by a new fact for an IDB body literal.

11 Related Work

The push method has been studied in [5], and compared with "Pull' and "Materialization" methods of bottom-up evaluation, but only with artificial examples, no real data loaded from files, and no index structures. The paper [7] defines a different version of the push method, which does not create variables for every argument of each IDB predicate, but for variables occurring in EDB literals in rule bodies. This helps to reduce or nearly eliminate the copying of values, at the price of creating several specializations of the same rule (this reduces runtime, but increases code size, sometimes significantly). It is basically this version which was used in our benchmarks, with certain improvements. Furthermore, our transformation program can also fall back to the version of [5], if the generated program would otherwise become unacceptably large. Actually, both versions can be combined for a single input program.

The idea of immediately using derived facts to derive more facts is not new in those papers. For instance, variants of semi-naive evaluation have been studied which work in this way [19, 21]. It also seems to be related to the propagation of updates to materialized views. However, the representation of tuples at runtime and the code structure is different from [19] (and this is essential for the reduction of copying values). The paper [21] translates from a temporal Datalog extension to Prolog, which makes any efficiency comparison dependend on implementation details of the used Prolog compiler.

In relational databases, the benefits of not materializing intermediate results have been recognized for a long time; this is known as *pipelining*. Pushing data up a pipeline has been studied in [15]; however, they work with relational algebra expressions rather than Datalog rules.

[4] Actually, the wine ontology program does not do what it is supposed to do: The test query is for californian wines, but the result contains other objects, too. We checked that XSB produces the same output. Nevertheless, the program is a challenging test for a deductive system, no matter whether it is meaningful. The wine ontology was developed for the OWL guide `http://www.w3.org/TR/owl-guide/`. It links to a site no longer available, but the ontology is probably the one available here: `https://www.movesinstitute.org/exi/data/DAML/wines.daml`. Thanks to Boris Motik, who referred us to the paper [11]: This introduces an approximate translation from OWL DL TBoxes to Datalog. Although it is not completely clear yet that this translation was used, the approximation would explain that the result might contain wrong answers.

12 Conclusion

This is a paper about some experiences gained while implementing a deductive database system and testing its performance. Of course, the system is not yet finished, and more problems and insights are waiting along the road. Nevertheless, the understanding reached so far seems interesting and useful:

- A system based on a bottom-up method can compete with systems based on SLD-resolution with tabling. Older deductive database systems like Coral [16, 20] had no chance against XSB [18]. Avoiding the copying of data values and the materialization of derived tuples seems to be the main reason for the success of the Push method.
- Fast data structure implementations, especially for duplicate elimination, are very important. For duplicate tests, tuples were materialized in "set" data structures. We plan to use nested tables in future in order to keep the advantage of reducing the copying of values.
- The SLDMagic method proved very useful for the transitive closure with binding pattern `bf`. For the binding pattern `fb`, work is still needed.
- Some variants in the code that intuitively seemed important had no effect on performance. For instance, we first tried a static procedure which only counted the derived facts, but did not return them. When we later used an object with a cursor interface where one can fetch each result there was no difference (although there were many more procedure calls, and what had been local or static variables before had to become attributes). Furthermore, replacing a `switch` and `goto` (our standard translation) with `while`-loops (where possible) had no important influence on performance (but improves the readability of the generated code, of course).

We plan to define an abstract machine and translate alternatively into this machine. Then it would not always be necessary to use a C++ compiler.

We also work on a parallelized version of our method. At times where even small PCs have at least four cores, one should obviously make use of this resource. The easier parallelization is also one motivation for declarative programming.

Of course, also adding function symbols (term constructors), negation and aggregation functions are on the agenda, plus more exotic things like ordered predicates and declarative output.

References

[1] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen & Geoffrey Washburn (2015): *Design and Implementation of the LogicBlox System*. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ACM, pp. 1371–1382.

[2] Francois Bancilhon, David Maier, Yehoshua Sagiv & Jeffrey D. Ullman (1986): *Magic Sets and Other Strange Ways to Implement Logic Programs*. In: *Proc. of the 5th ACM Symp. on Principles of Database Systems (PODS'86)*, ACM Press, pp. 1–15.

[3] Catril Beeri & Raghu Ramakrishnan (1987): *On the Power of Magic*. In: *Proc. of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'87)*, ACM, pp. 269–284.

[4] Stefan Brass (2000): *SLDMagic — The Real Magic (with Applications to Web Queries)*. In W. Lloyd et al., editors: *First International Conference on Computational Logic (CL'2000/DOOD'2000)*, *LNCS* 1861, Springer, pp. 1063–1077. Available at `http://www.informatik.uni-halle.de/~brass/publ/Bra00a.ps.gz`.

[5] Stefan Brass (2010): *Implementation Alternatives for Bottom-Up Evaluation*. In Manuel Hermenegildo & Torsten Schaub, editors: *Technical Communications of the 26th International Conference on Logic Programming (ICLP'10)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 7, Schloss Dagstuhl, pp. 44–53. Available at `http://drops.dagstuhl.de/opus/volltexte/2010/2582`.

[6] Stefan Brass (2012): *Order in Datalog with Applications to Declarative Output*. In Pablo Barceló & Reinhard Pichler, editors: *Datalog in Academica and Industry, 2nd Int. Workshop, Datalog 2.0*, *LNCS* 7494, Springer-Verlag, pp. 56–67. Available at `http://users.informatik.uni-halle.de/~brass/order/`.

[7] Stefan Brass & Heike Stephan (2015): *Bottom-Up Evaluation of Datalog: Preliminary Report*. In Sibylle Schwarz & Steffen Hölldobler, editors: *29th Workshop on (Constraint) Logic Programming (WLP 2015)*, HTWK Leipzig, pp. 21–35. Available at `http://www.imn.htwk-leipzig.de/WLP2015/`.

[8] Weidong Chen & David S. Warren (1996): *Tabled Evaluation with Delaying for General Logic Programs*. *Journal of the ACM* 43(1), pp. 20–74.

[9] Vítor Santos Costa (1999): *Optimizing Bytecode Emulation for Prolog*. In Gopalan Nadathur, editor: *Principles and Practice of Declarative Programming, Internat. Conf. PPDP'99*, *LNCS* 1702, Springer, pp. 261–277.

[10] Vítor Santos Costa, Ricardo Rocha & Luís Damas (2012): *The YAP Prolog System*. *Theory and Practice of Logic Programming* 12(1–2), pp. 5–34. Available at `https://www.dcc.fc.up.pt/~ricroc/homepage/publications/2012-TPLP.pdf`.

[11] Pascal Hitzler, Markus Krötzsch, Sebastian Rudolph & Tuvshintur Tserendorj (2008): *Approximate OWL Instance Retrieval with SCREECH*. In Anthony G. Cohn, David C. Hogg, Möller. Ralf & Bernd Neumann, editors: *Logic and Probability for Scene Interpretation*, *Dagstuhl Seminar Proceedings* 08091, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany, pp. 1–8. Available at `http://drops.dagstuhl.de/opus/volltexte/2008/1615`.

[12] Viktor Leis, Alfons Kemper & Thomas Neumann (1997): *The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases*. In: *Proc. of the 2013 IEEE International Conference on Data Engineering (ICDE'2013)*, IEEE Computer Society, pp. 38–49. Available at `http://www3.informatik.tu-muenchen.de/~leis/papers/ART.pdf`.

[13] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri & Francesco Scarcello (2006): *The DLV system for knowledge representation and reasoning*. *ACM Trans. Comput. Logic* 7(3), pp. 499–562. Available at `https://arxiv.org/pdf/cs/0211004`.

[14] Senlin Liang, Paul Fodor, Hui Wan & Michael Kifer (2009): *OpenRuleBench: An Analysis of the Performance of Rule Engines*. In: *Proceedings of the 18th International Conference on World Wide Web (WWW'09)*, ACM, pp. 601–610. Available at `http://rulebench.projects.semwebcentral.org/`.

[15] Thomas Neumann (2011): *Efficiently Compiling Efficient Query Plans for Modern Hardware*. *Proceedings of the VLDB Endowment* 4(9), pp. 539–550. Available at `http://www.vldb.org/pvldb/vol4/p539-neumann.pdf`.

[16] Raghu Ramakrishnan, Divesh Srivastava & S. Sudarshan (1994): *Rule Ordering in Bottom-Up Fixpoint Evaluation of Logic Programs*. *IEEE Transactions on Knowledge and Data Engineering* 6(4), pp. 501–517.

[17] Kenneth A. Ross (1991): *Modular Acyclicity and Tail Recursion in Logic Programs*. In: *Proc. of the Tenth ACM SIGACT-SIGMOD-SIGART Symp. on Princ. of Database Systems (PODS'91)*, pp. 92–101.

[18] Konstantinos Sagonas, Terrance Swift & David S. Warren (1994): *XSB as an Efficient Deductive Database Engine*. In Richard T. Snodgrass & Marianne Winslett, editors: *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94)*, pp. 442–453.

[19] Heribert Schütz (1993): *Tupelweise Bottom-up-Auswertung von Logikprogrammen (Tuple-wise bottom-up evaluation of logic programs)*. Ph.D. thesis, TU München.

[20] Praveen Seshadri, Shaun Flisakowski & Seymour Hersh (1996): *CORAL: The Inside Story. Shocking Hacks Revealed*. Technical Report, Department of Computer Sciences, The University of Wisconsin-Madison. Available at `http://ftp.cs.wisc.edu/coral/doc/Inside.ps`.

[21] Donald A. Smith & Mark Utting (1999): *Pseudo-Naive Evaluation*. In: *Australasian Database Conf.* , pp. 211–223. `http://www.cs.waikato.ac.nz/research/jstar/1999-ADC-pseudo-naive-eval.pdf`.

[22] Terrance Swift (1999): *Tabling for non-monotonic programming*. *Annals of Mathematics and Artificial Intelligence* 25, pp. 201–240.