To appear in: A.K. Petrenko, A. Voronkov (Eds.):
Perspectives of System Informatics — PSI 2017
Springer, LNCS 10742, 2018
Final version available from https://doi.org/10.1007/978-3-319-74313-4_4

Pipelined Bottom-Up Evaluation of Datalog Programs: The Push Method

Stefan Brass and Heike Stephan

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

Abstract. In this paper, we present a method for bottom-up evaluation of Datalog programs in deductive databases that "pushes" derived facts immediately to other rules where they are used for deriving more facts. In this way, the materialization of derived relations is avoided as far as possible. Derived facts are represented by values in variables and a location in the program, and not as explicitly constructed tuples. This helps to avoid copying operations and to keep the actively used memory small to make better use of modern processors. The method can be quite easily explained by translating Datalog to functions in a standard programming language like C++ and then using optimizations of the C++ compiler like inlining. First performance tests with benchmarks from the OpenRuleBench collection give good results. This is interesting because systems based on SLD-resolution with tabling such as XSB have beaten older deductive database implementations based on bottom-up evaluation. Now it seems that bottom-up evaluation can be done in a very competitive way.

1 Introduction

Database application programs usually consist of queries, written in a declarative language (typically SQL), and surrounding program code, written in a procedural language (like PHP or Java). The goal of deductive databases is to write a larger part of the application, ideally the entire program, in a declarative language based on Prolog/Datalog.

This has many aspects, e.g. the invention of new language constructs for declarative output [4]. However, the classic task of improving the speed of query evaluation is still important. The amount of data to be processed is ever growing. We might also be able to make better use of new processor architectures. For our approach, which is implemented by translating Datalog to C++, also the compiler technology is very important. The optimization in a compiler has become more powerful, so approaches that were too slow earlier can be very competitive now.

In this paper, we further develop the "Push Method" for bottom-up evaluation introduced in [3,6]. It applies the rules from body to head (right to left) as any form of bottom-up evaluation, but it immediately "pushes" a derived fact to other rules with matching body literals. In contrast, the classic approach would

first apply a rule completely before using the derived facts (which also requires intermediate storage of these tuples). Our method has the following advantages:

- We try to avoid the materialization and storage of facts as far as possible, and on a lower level the copying of values. For instance, consider the rule

  $$p(X, Z) \leftarrow q(X, Y) \wedge r(Y, Z).$$

  Suppose that we have derived a new fact $q(a, b)$ and let r be a database relation (defined by facts). Now if there are many facts $r(b, Z)$, there is no reason to copy the value a of X for each such fact. Actually, standard implementations of SLD-resolution/Prolog would also not touch the value of X while they backtrack over different solutions for the second body literal, whereas standard implementations of bottom-up evaluation do this.
- By immediately using a derived fact, we can keep the data values near to the CPU (in registers or the cache), and thereby speed up the computation. Furthermore, memory is used for a shorter time and can be recycled earlier.
- Another feature of our method is that it does partial evaluation and rule specialization at "compile time". The method translates a given Datalog program to C++ (which is then compiled by a standard compiler). We assume that the rules defining derived predicates are known in this step, whereas the facts for the database predicates (including user input) are known only at runtime. Time invested in the compilation phase can later be redeemed over many executions of the resulting program (with different database states). Actually, it might pay off even in a single execution, because often there are many more database facts than rules.
- Finally, if there is large number of mutually recursive rules, classic bottom-up evaluation would iteratively apply all these rules until none produces a new fact. If only a small number of these rules can actually fire in an iteration (because only they have new matches for body literals), this is obviously inefficient. Our "Push Method" does not look at rules unless there really is a new matching fact for a body literal.

The main contribution of the current paper, compared with our older papers, is that we now generate a set of recursive procedures and rely on the optimizations of the compiler (in particular, inlining and copy propagation). Instead, the version of [6] used C++ only as a portable assembler and generated a single large procedure with a big `switch`. This implementation managed its own stack, but only for recursive rules. We thought that the procedure call overhead would be too large. However, because the compiler unfolds procedure calls ("inlining"), this is not necessarily the case. We did compare the runtime on a number of benchmarks from the OpenRuleBench collection [11], and the runtime was about the same, sometimes even slightly faster. One reason for improved runtime might be that the older version produced code for each specialized version of a predicate only once, while a compiler that does aggressive inlining is stopped only by recursions — it may inline several calls to the same procedure if the body is not large. This saves jumps and stack usage. The new version of the method is also much simpler to understand than the old one.

2 Query Language

We consider basic Datalog, i.e. pure Prolog without negation and function symbols. Thus, a logic program is a finite set of rules of the form $A \leftarrow B_1 \wedge \cdots \wedge B_n$ with atomic formulas A (the head literal of the rule) and $B_1, \ldots, B_n$ (the body literals of the rule). The atomic formulas have the form $p(t_1, \ldots, t_m)$ with a predicate p and argument terms t_i, which are variables or constants. As usual in Prolog, $\leftarrow$ is written "`:-`", $\wedge$ is written "`,`", and variables start with an uppercase letter to distinguish them from constants. As an example, consider

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

This logic program computes the grandparents of `sam`, given database relations `mother` and `father`. We assume that there is a "main" predicate `answer`, for which we want to compute the derivable instances. All other predicates are views or names for subexpressions of the query.

We require range-restriction (allowedness), i.e. all variables in the head of the rule must also appear in a body literal. This ensures that when rules are applied bottom-up (from right to left), only variable-free atomic formulas (facts) are derived. 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 specially formatted files). In the above example, `mother` and `father` are EDB predicates.
- IDB-predicates ("intensional database"), which are defined by rules. In the above example, `parent`, `grandparent` and `answer` are IDB predicates.

We also say "EDB body literal" for a body literal with EDB-predicate, and "IDB body literal" for one with IDB-predicate.

Since we use a compiler-based approach, one would probably not want to compile the constant "`sam`" into the program, but to execute the compilation result many times for computing the grandparents of different people. This can easily be achieved by replacing the last rule by

```
answer(Z) :- input(X), grandparent(X, Z).
```

Here, `input` is a new EDB-predicate which is set before each execution, and contains e.g. the single fact `input(sam)`. The result of the compilation depends only on the rules (and possibly facts) for the IDB-predicates. The database state, i.e. the extensions of the EDB-predicates, can be arbitrarily updated.

3 Goal-Directed Query Evaluation with SLDMagic

Pure bottom-up evaluation is not goal-directed, i.e. it computes all derivable facts without looking at the given query. It is standard to do the magic sets

program transformation [1] first, which reduces the applicability of rules such that only facts relevant to the given query can be derived. The output of this transformation is again a Datalog program, which is then used as input for bottom-up evaluation.

In our case, we use an improved "Magic Set" method, SLDMagic [2,5]. This is interesting because it simulates SLD-resolution (the algorithm underlying Prolog) more precisely than Magic Sets (furthermore, the output is mostly linear Datalog, which is good for the Push method, see below). Basically, the rules resulting from the SLDMagic transformation compute the nodes of the SLD-tree.

Example 1. The result of the SLDMagic prototype[1] for the above program is (with slight renamings of variables):

```
p0(X) :- input(X).
p4(Y) :- p0(X), mother(X,Y).
p4(Y) :- p0(X), father(X,Y).
p7(Z) :- p4(Y), mother(Y,Z).
p7(Z) :- p4(Y), father(Y,Z).
answer(Z) :- p7(Z).
```

A fact of the form `p0(X)` corresponds to the goal `grandparent(X,Z)` in the SLD-tree, where `Z` is the result variable (i.e. the variable appearing in the original query). Since SLDMagic must produce range-restricted rules, only variables that are bound to a constant appear as arguments in the generated predicates like `p0`. A fact of the form `p4(Y)` corresponds to the goal `parent(Y,Z)` in the SLD-tree, still with `Z` as result variable. I.e. if one finds a `Z` with `parent(Y,Z)`, this `Z` is an answer to the original query.

SLD-Resolution does several steps in between, which have been removed by the "copy rule elimination" phase of the SLDMagic prototype (this explains why there are holes in the predicate numbers). Finally, `p7(Z)` corresponds to the empty goal in the SLD-tree, i.e. the query is proven, and `Z` is a result value.

4 The Push Method with Procedure Calls

Whereas Magic Sets and SLDMagic are translations from Datalog to Datalog, the Push method is implemented here by a translation from Datalog to C++. The resulting program is then compiled to machine code by a standard optimizing compiler. In our tests, we have used `g++`. Because the C++ compiler performs powerful optimizations, such as inlining and copy propagation, the code we generate from a Datalog program can be simple.

4.1 Basic Code Structure, Requirements

Because we want to "push" derived facts through rules (from a body literal to the head), rules with a single IDB body literal (where derived facts can match) are especially simple:

[1] Available at `http://users.informatik.uni-halle.de/~brass/sldmagic/`

Definition 1 (Rule Classification). *Rules in the given Datalog program are classified into*

- *Start rules: Rules with only EDB-body literals,*
- *Simple (or "linear") rules: Rules with exactly one IDB body literal,*
- *Complex rules: Rules with more than one IDB body literal.*

A Datalog program without complex rules is a linear Datalog program. Our SLDMagic transformation produces programs that are mostly linear (only for rules that are recursive, but not tail recursive, complex rules are generated).

The C++ program that is generated from a given set of rules basically contains for each IDB-predicate p of arity n a corresponding procedure `p` with n arguments that is called (at least) once for each derivable fact $p(c_1, \ldots, c_n)$. If one looks at the procedure calls ordered by the time when they occur, one gets the following notion of a computation sequence:

Definition 2 (Computation Sequence). *Let a Datalog program P with rules defining IDB-predicates and a database D with facts for EDB-predicates be given. Let $\mathcal{M}$ be the minimal Herbrand model of $P \cup D$ (i.e. the set of all derivable facts). A computation-sequence is a sequence $\mathcal{S}$ of facts for IDB-predicates.*

- *The computation is correct iff all facts in $\mathcal{S}$ appear in $\mathcal{M}$.*
- *It is complete iff all facts for IDB-predicates in $\mathcal{M}$ appear in $\mathcal{S}$.*

Any algorithm for bottom-up evaluation of a Datalog program computes the facts of the minimal model in some such sequence. The specific order of the facts can be very different, though, which also influences how long intermediate facts must be stored. However, any reasonable computation sequence must have the following property: All facts in the sequence must be derivable from previously derived facts or given database facts. We call such computation sequences causal.

Definition 3 (Causal Computation Sequence). *A computation sequence $\mathcal{S} = F_1, F_2, \ldots$ is causal iff for every fact F_i, $i = 1, 2, \ldots$ there is a rule $A \leftarrow B_1 \wedge \cdots \wedge B_n$ in P and a ground substitution θ for this rule such that*

- *$A\theta = F_i$ and*
- *$\{B_1\theta, \ldots, B_n\theta\} \subseteq D \cup \{F_1, \ldots, F_{i-1}\}$.*

Lemma 1. *Every causal computation sequence is correct.*

The translation result of the "Push" method contains a procedure "`start()`" which is basically the main program. It ensures that the start rules are applied which then lead to further procedure calls. Start rules are easy to execute because they contain only EDB body literals. The extensions of these predicates (database relations) are given, so the query corresponding to the rule body can be executed and the calls for the head literal can be done.

Definition 4 (Requirements for the `start()` procedure). *Let a program P and database D be given. The procedure* `start()` *ensures that for every start rule $A \leftarrow B_1 \wedge \cdots \wedge B_n$ in P and every ground substitution θ for this rule such that $B_i\theta \in D$ for $i = 1, \ldots, n$ the procedure call corresponding to $A\theta$ is executed, i.e. the computation sequence contains this fact (under the assumption that the sequence is finite).*

Besides the `start()` procedure, the translation result contains one procedure per predicate. It must ensure that when a fact $p(c_1, \ldots, c_n)$ is derived, all applicable rule instances are fired that contain $p(c_1, \ldots, c_n)$ in the body. For simple (linear) rules this is easy, because they contain only one IDB body literal, so the true instances of the other body literals can be looked up in the database. For instances of complex rules (containing more than one IDB body literal), we must look at the global computation sequence: We can only require that the rule instance is applied if the needed instances of the other IDB body literals have already been derived earlier in the sequence. So in the case of complex rules, we need temporary storage for previously derived instances of IDB body literals.

Definition 5 (Requirements for procedures implementing predicates). *Let a program P and database D be given. For every IDB-predicate p of arity n there is a procedure* `p` *with n arguments that ensures the following condition for the computation sequence $\mathcal{S} = F_1, F_2, \ldots$:*

For each $i = 1, 2 \ldots$, if F_i is the first occurrence of the fact (i.e. there is no F_j with $j < i$ and $F_j = F_i$), the following holds:

- *For each rule $A \leftarrow B_1 \wedge \cdots \wedge B_m$ and each ground substitution θ for that rule such that there is a $k \in \{1, \ldots, m\}$ with $B_k\theta = F_i$ and*

$$\{B_1\theta, \ldots, B_m\theta\} \subseteq \{F_1, \ldots, F_i\} \cup D,$$

  $A\theta$ appears somewhere in the sequence, i.e. this fact also occurs in $\mathcal{S}$ (under the assumption that the sequence is finite).

It is not required that the position of the fact is after i (e.g., if it was already derivable with another rule instance).

This simply means that all rule instances that are applicable when F_i is derived for the first time, are eventually applied. If we can ensure this condition, the completeness easily follows:

Theorem 1 (Completeness). *Let a program P and database D be given. If a computation sequence $\mathcal{S}$ satisfies the conditions of Definitions 4 and 5, and if it is finite, it is complete, i.e. contains all IDB-facts in the minimal model of $P \cup D$.*

After clarifying the general approach and the requirements for each procedure, let us look a bit closer at the code that is generated. The overall code structure can be visualized by means of a rule application graph:

Definition 6 (Rule Application Graph). *Let a Datalog program P be given and let $\mathsf{IDB}(P)$ be the set of IDB-predicates of this program, i.e. the predicates appearing in rule heads. The rule application graph for P is a bipartite directed graph $(\mathcal{V}, \mathcal{E})$ with*

- $\mathcal{V} := \mathcal{V}_1 \cup \mathcal{V}_2$, *where*
  - $\mathcal{V}_1 := \mathsf{IDB}(P)$ *are "predicate nodes", and*
  - $\mathcal{V}_2 := \big\{(A \leftarrow B_1 \wedge \cdots \wedge B_m,\ i) \;\big|\; A \leftarrow B_1 \wedge \cdots \wedge B_m \in P,$ *and B_i is a literal with IDB-predicate*$\big\}$ *are "rule activation nodes".*
  (The "active literal" B_i will be shown underlined.)
- $\mathcal{E} := \mathcal{E}_1 \cup \mathcal{E}_2$ *with*
  - $\mathcal{E}_1 := \big\{\big(p,\ (A \leftarrow B_1 \wedge \cdots \wedge B_m,\ i)\big) \;\big|\; p$ *is the predicate of* $B_i\big\}$,
  - $\mathcal{E}_2 := \big\{\big((A \leftarrow B_1 \wedge \cdots \wedge B_m,\ i),\ p\big) \;\big|\; p$ *is the predicate of* $A\big\}$.

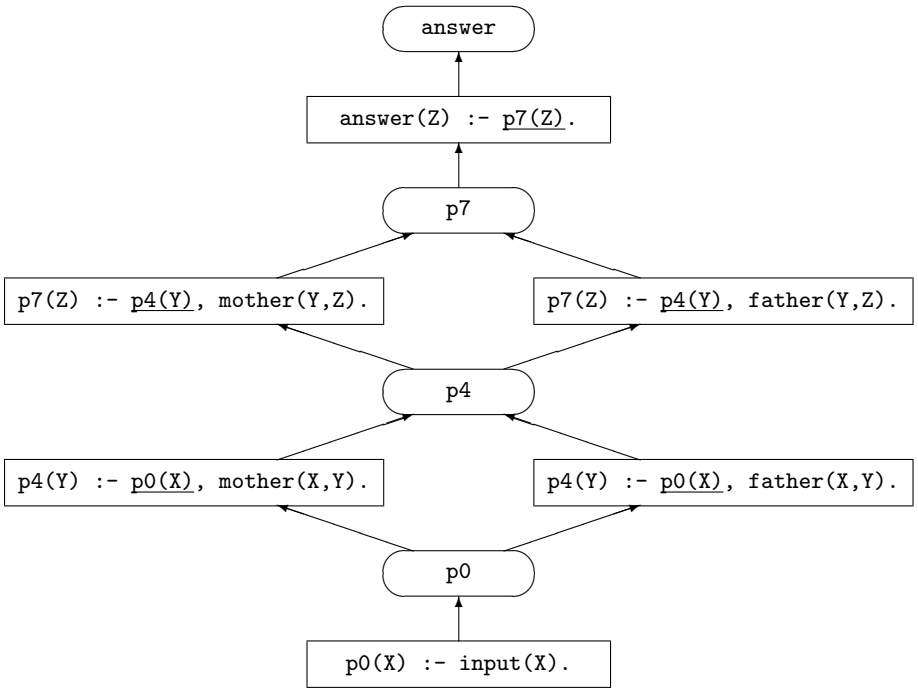

Fig. 1. Rule Application Graph for Example 1

The rule application graph for Example 1 is shown in Figure 1. Each predicate node p corresponds to a procedure that contains a code block for each rule activation node that has an incoming edge from the predicate node. This code block ensures that the given fact for p is inserted for the active literal in that

rule activation and the corresponding calls for the predicate in the head (to which the outgoing edge leads) are done. Rule activation nodes without incoming edges correspond to start rules. They are executed in the procedure `start()` and initialize the stream of facts which is pushed along the edges through the nodes. In each rule activation node, an incoming fact can lead to multiple outgoing facts, but it can also be stopped if the actual data values do not match the active literal or if it does not find a join partner for the other body literals.

4.2 Example, Data Structures for Query Evaluation

The generated code for Example 1 is shown in Figure 2. Loops over tuples returned from relation data structures are shown in pseudocode, one would use a cursor/iterator here.

```
void start()
{
   // p0(X) :- input(X).
   foreach X in input_f
      p0(X);
}

inline void p0(int c1)
{
   // p4(Y) :- p0(X), mother(X,Y).
   // X = c1
   foreach Y in mother_bf(c1)
      p4(Y);

   // p4(Y) :- p0(X), father(X,Y).
   // X = c1
   foreach Y in father_bf(c1)
      p4(Y);
}
```

```
inline void p4(int c1)
{
   // p7(Z) :- p4(Y), mother(Y,Z).
   // Y = c1;
   foreach Z in mother_bf(c1)
      p7(Z);

   // p7(Z) :- p4(Y), father(Y,Z).
   // Y = c1;
   foreach Z in father_bf(c1)
      p7(Z);
}

inline void p7(int c1)
{
   // answer(Z) :- p7(Z).
   // Z = c1;
   insert c1 into answer;
}
```

Fig. 2. Result of the Push Transformation for the Program from Example 1

The data structures for the relations (EDB predicates) are as follows:

- `input_f` is a list of input values (probably only one). The suffix `f` is a binding pattern, it indicates that the relation is accessed with a free variable for the only column, i.e. the argument is an output argument. Our current implementation first loads all data to main memory, so a vector/list data structure is used in such cases, when only a full table scan is needed.
- `mother_bf` and `father_bf` support an access with a given ("bound") first argument, where the second ("free") argument is required. A map/multimap data structure is used in such cases.

When the data is loaded, we construct for each EDB body literal a relation data structure which just fits its use. I.e. if the literal contains constants or the same variable more than once, these selections are already evaluated while the data is loaded, and only matching tuples are stored. Variables which are bound when the literal is accessed form the search key of the index, i.e. we do an index join. Values for the other ("free") variables are found with the index lookup. It might seem that we need a lot of memory for this, but benchmarks so far have shown that memory consumption is low compared to other systems and loading time is quick. Of course, one can construct examples where using such an optimal index for each body literal leads to a blowup. One should introduce a limit. It is always possible to simply load all data for an EDB predicate into a list, and then do the selection at runtime (which corresponds to a nested loop join).

For string data, we use a radix tree similar to the one in [9] to map each string to a unique, sequential integer. In this way, later comparisons can be done in one machine instruction, and index structures for relations are simpler and more efficient. It is usual also in Prolog systems to have a hash table for atoms (which are one type of strings). Therefore, the arguments and variables are shown with type `int` in the code. However, this is not essential for the Push method and will probably be relaxed in future versions of our implementation. Basically, one has to define the types for all columns of the database relations (EDB-predicates) and can then derive types of the IDB-predicate arguments.

4.3 Duplicate Elimination, Termination

We need to make sure that query evaluation terminates. Since there are only finitely many constants in the program and the database, the only reason for non-termination can be duplicates. In every recursive cycle, we must check for duplicate facts at least once. Even for non-recursive predicates, if many duplicates are generated for the predicate, it might greatly improve performance if these are detected early and further computations with them are avoided.

Of course, there is also a price to pay in form of a hash table or similar "set" data structure in which tuples must be inserted. Since the push method intends to avoid materialization of tuples, this is not nice. Note, however, that a data structure only for the purpose of duplicate elimination is still smaller than storing a general relation with arbitrary access. Of course, if we should know that tuples arrive in a sort order, duplicate elimination would be much cheaper. Since the variables which make up a tuple are assigned to at different frequencies (e.g., one in an outer loop, and one in an inner loop), we also intend to experiment with nested hash tables.

The user of our system can select for which predicates duplicate elimination is done (of course, recursive cycles must be broken). This is similar to switching tabling on for a predicate in a logic programming system that uses tabling.

If duplicate elimination is selected for a predicate p, the first thing the procedure for p does is to insert the argument tuple into a "set" data structure (e.g., a hash table). If this fails (because the tuple is already contained in the set), the procedure immediately returns.

4.4 Temporary Relations for Complex Rules

The Push method tries to use each fact immediately when it is derived. This works nicely with simple rules which have only a single IDB body literal, and therefore all other facts needed to apply the rule are available in the database.

For complex rules (with more than one IDB body literal) it is unavoidable to keep derived facts matching a body literal until we have the matching facts (join partners) for the other IDB body literals. A rule instance is applied as soon as the last IDB fact used in this instance is derived (which is again the earliest possible time).

Thus, for complex rules, we create temporary relations for each IDB body literal. These relations contain all facts matching the respective body literal which have previously been derived. The code ensures the following condition: All ground instances of the rule where the instance of each body literal is contained in its temporary relation have already been applied, or are currently being applied (somewhere above in the recursion). Now, if for one of the body literals a new instance is derived, only this new instance is joined with all facts in the temporary relations for the other body literals. This new instance is also inserted into the temporary relation for its own body literal.

This also works if the same fact is used for two body literals: It is tried first for one of them, and then for the other. When it is tried for the first, the join partner is still missing, but it is inserted into its temporary relation. When it is matched with the second body literal, it is already stored in the relation for the first one, so it finds its join partner, and the rule instance is applied. Basically, this is seminaïve evaluation, where the "delta" consists of a single fact.

Note again that complex rules are an exceptional case in the output of the SLDMagic transformation. If the program was not generated by SLDMagic, one might eliminate some complex rules by unfolding. Furthermore, the above description works already in the most general case, where all IDB body literals are potentially recursive. In other cases, when we know that facts are first produced for one body literal, and then for the other one, we might eliminate some temporary relations. See subsection 4.6.

Example 2. As an example of a complex and recursive rule, consider this version of the transitive closure:

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

The corresponding code is shown in Figure 3. Note that when a `foreach`-loop starts, the set of values over which it iterates must be fixed (insertions into the temporary relation have no effect on already active iterators).

4.5 Code Block for a Rule Activation

Now we are ready to look at the code block for a rule activation

$$A \leftarrow B_1 \wedge \cdots \wedge \underline{B_i} \wedge \cdots \wedge B_m$$

```
void start()
{
   // tc(X, Y) :- edge(X, Y).
   foreach (X,Y) in edge_ff
      tc(X, Y);
}
```

```
void tc(int c1, int c2)
{
   // Duplicate check:
   if(!tc_bb.insert(c1, c2))
      return;

   // tc(X,Z) :- tc(X,Y), tc(Y,Z).
   // X=c1, Y=c2 (first body literal)
   tc_1_fb.insert(c1, c2);
   foreach Z in tc_2_bf(c2)
      tc(c1, Z);

   // tc(X,Z) :- tc(X,Y), tc(Y,Z).
   // Y=c1, Z=c2 (second body literal)
   tc_2_bf.insert(c1, c2);
   foreach X in tc_1_fb(c1)
      tc(X, c2);
}
```

Fig. 3. Result of the Push Transformation for the Program from Example 2

in a bit more detail. It is structured as follows:

- It first has to check whether the given fact $p(c_1, \ldots, c_n)$ really matches the IDB body literal $B_i = p(t_1, \ldots, t_n)$, i.e. for $j = 1, \ldots, n$
  - if t_j is a constant, then $t_j = c_j$ must hold,
  - if t_j is a variable that appeared first in t_k, $k < j$, then $c_j = c_k$ must be satisfied (otherwise, the following code block is skipped).

  Note that if the procedure call contains constants as arguments, or the same variable in different arguments, the compiler might be able to evaluate the condition, and possibly remove the code block.
- If the rule is a complex rule, $(c_1, \ldots, c_n)$ is inserted into the temporary relation for B_i. (One can improve this by storing only values of variables that are later needed.)
- Then one executes the query $B_1 \wedge \cdots \wedge B_{i-1} \wedge B_{i+1} \wedge \cdots \wedge B_m$ with the given values for the variables among the $t_1, \ldots, t_n$. For simple rules, all these literals are EDB literals, so there is a given database relation for them. For complex rules, we created temporary relations for each IDB body literal in them, which are used here. So in both cases, relations are given for all B_j, $j \neq i$, and it is a standard task to evaluate this query.
- The query result contains bindings for the remaining variables of the query. So we now have a ground substitution θ for the rule such that $(B_1 \wedge \cdots \wedge B_m)\theta$ is true. With these variable values, the corresponding instance of the head literal A can be determined. Then the corresponding procedure call is done.

4.6 An Optimization for Complex Rules

Consider a rule with two IDB body literals, e.g. $p(X)$ and $q(X)$. When a start rule is executed, only a subset of the nodes in the rule application graph are

reachable from this start rule, i.e. only for some predicates facts are derived. If the first start rule yields only p-facts, but no q-facts, it is clear that the temporary relation for the second body literal is still empty. So the derived p-facts will be stored in the temporary relation for the first body literal, but cannot be pushed further. The corresponding code can be removed.

If for a later start rule p-facts are derived when q-facts have been derived in the meantime, this code is needed. Thus two different versions of the procedure are needed, which are used in the derivation for different start rules (also predicate-procedures between the start rule and this rule must be duplicated).

Another interesting special case is when all start rules that yield p-facts are executed before the first start rule with yields q-facts. In this case, the temporary relation for the body literal $q(X)$ is not needed, and its facts can be simply pushed through the rule as if p were an EDB-predicate.

So in general, we have a sequence of components of the program, one for each start rule. For this optimization of the complex rules evaluation, it is be useful to distinguish three possible status values of a predicate p in component i:

- The predicate extension is empty, i.e. the predicate node for p is not reachable from any start rule $\leq i$.
- The predicate extension is partial, i.e. p is reachable in a component $\leq i$ and p is also reachable in a component $\geq i$.
- The predicate extension is complete, i.e. p is not reachable in components $\geq i$.

4.7 Remarks about Inlining

Of course, unfolding procedure calls through inlining often leads to an increased code size, which might have negative effects on cache utilization and thus runtime. Currently, we leave the decision which procedure calls are inlined to the compiler (it does a cost-benefit estimation). With an analysis of the rules, we might be able to do better in future and use `__attribute__((always_inline))` or `__forceinline` on selected procedures.

5 Benchmark Results

We implemented our transformation and a supporting C++ library and checked several benchmarks from the OpenRuleBench benchmark suite [11]. The current version of our method can handle only basic Datalog without negation and structured terms. So we could try only such benchmark problems. However, for these we get encouraging results, often significantly faster than systems which are well established in the community. We tried systems that were considered also in the original OpenRuleBench test, namely XSB [15], YAP [8], and DLV [10]. Note that XSB is called a deductive database in [15], and clearly beats older, well-known deductive database prototypes like CORAL [14]. The DLV system is strong in answer set programming, but it only uses system modules necessary for a given problem [10]. So it is not unfair to use it in a comparison for simpler problems. Recently, the Soufflé system has been developed [16]. It is not

advertised as a deductive database, but as a system for doing static program analysis using Datalog. Nevertheless, it can be used to execute the benchmarks, and does a compilation to C++ as we do, although the code structure is different (it is based on relations and relational algebra). The performance results on the benchmarks we tried are comparable to our system (we restricted Soufflé to one core, because our prototype cannot do parallel evaluation yet). The significantly lower memory consumption of Soufflé in the compiler version is probably due to the use of a Trie data structure for relations (something we should try, too).

Of course, the above systems (except Soufflé) have been developed for more than a decade, and offer many features which are needed for the practical development of large projects. In contrast, our implementation is just a first prototype in order to check the potential of our evaluation method.

For space reasons, we show here only the results of two benchmarks [11]: The standard tail-recursive transitive closure program (with a cyclic graph with 1000 nodes and 50 000 edges), and a join of five relations with 10 000 rows each.

System	Load	Execution	Total time	Factor	Memory
Push (Proc.)	0.005s	1.039s	1.044s	1.0	31.246 MB
Push (Switch)	0.005s	1.030s	1.033s	1.0	23.303 MB
Seminaïve	0.005s	1.670s	1.672s	1.6	31.210 MB
XSB	0.247s	4.740s	5.090s	4.9	135.925 MB
YAP	0.240s	10.549s	10.833s	10.4	147.601 MB
DLV	(0.373s)	—	52.300s	50.1	513.753 MB
Soufflé (SQlite)	(0.113s)	—	11.237s	10.8	43.083 MB
(compiled)	(0.030s)	—	0.790s	0.8	3.810 MB

Transitive Closure Benchmark with query tc(_,_) [11]

System	Load	Execution	Total Time	Factor	Memory
Push (Proc.)	0.004s	1.043s	1.043s	1.0	16.863 MB
Push (Switch)	0.004s	1.031s	1.032s	1.0	9.010 MB
XSB	0.128s	6.056s	6.460s	6.2	127.259 MB
YAP	0.207s	3.572s	3.840s	3.7	135.921 MB
DLV	(0.253s)	—	80.237s	76.9	603.141 MB
Soufflé (SQlite)	(0.100s)	—	12.680s	12.2	38.548 MB
(compiled)	(0.040s)	—	1.450s	1.4	4.264 MB

Join1 Benchmark with query a(X,Y) [11]

We compile to machine code, whereas XSB, YAP and DLV emulate an abstract machine. This explains a factor of about 3, see, e.g. [7]. In case of Soufflé, the compiled version uses an own implementation, whereas the interpreted version uses SQLite. Thus, the factor of 10 is not only caused by compilation.

Our transformation and the compilation together take about 0.5s, which are not included in the above numbers (because they are done only once).

The additional memory in the procedure version of the Push method is required because the query result needs to be stored, whereas the original version could offer an iterator interface without having to store answer tuples.

6 Related Work

The general idea of immediately using derived facts to derive more facts is not new. For instance, variants of semi-naive evaluation have been studied which work in this way [17,18]. Heribert Schütz proposed in his 1993 PhD thesis already a version of bottom-up evaluation that used procedures for each predicate called when a new fact was derived. In contrast to our approach, the rules were first normalized, and EDB predicates were not treated specially. There are differences also in the data structures and the structure of the generated code (our goal is doing partial evaluation). Furthermore, there was only a prototypical implementation in Lisp (as part of the LOLA system), and benchmark comparisons with established systems were not done or did not give the desired results.

The method is also related to the propagation of updates to materialized views. In [12], compiling Datalog rules to program code was studied based on incremental computation of the derived facts.

"Pushing" tuples through relational algebra expressions has also become an attractive technique for standard databases [13].

7 Conclusion

In this paper, we have explained a new version of the Push method for bottom-up evaluation in deductive databases. The main improvement is that the method is much simpler to understand, and the program code resulting from the Datalog-to-C++ transformation is much better structured. Current C++ compilers are able to do many optimizations automatically that were explicitly built into the old transformation. Our performance measurements on some benchmarks of the OpenRuleBench collection are encouraging. It seems that we are able to be faster than some well-established systems in the area. The current state of the project is reported on the following web page:

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

References

1. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic sets and other strange ways to implement logic programs. In: Proc. of the 5th ACM Symp. on Principles of Database Systems (PODS'86). pp. 1–15. ACM Press (1986)
2. Brass, S.: SLDMagic — the real magic (with applications to web queries). In: Lloyd, W., et al. (eds.) First International Conference on Computational Logic (CL'2000/DOOD'2000). LNCS, vol. 1861, pp. 1063–1077. Springer (2000), `http://users.informatik.uni-halle.de/~brass/sldmagic/`
3. Brass, S.: Implementation alternatives for bottom-up evaluation. In: Hermenegildo, M., Schaub, T. (eds.) Technical Communications of the 26th International Conference on Logic Programming (ICLP'10). Leibniz International Proceedings in Informatics (LIPIcs), vol. 7, pp. 44–53. Schloss Dagstuhl (2010), `http://drops.dagstuhl.de/opus/volltexte/2010/2582`

4. Brass, S.: Order in Datalog with applications to declarative output. In: Barceló, P., Pichler, R. (eds.) Datalog in Academica and Industry, 2nd Int. Workshop, Datalog 2.0. LNCS, vol. 7494, pp. 56–67. Springer-Verlag (2012), http://users.informatik.uni-halle.de/~brass/order/
5. Brass, S.: A framework for goal-directed query evaluation with negation. In: Calimeri, F., Ianni, G., Truszczynski, M. (eds.) Logic Programming and Nonmonotonic Reasoning, 13th Int. Conf. (LPNMR 2015). pp. 151–157. No. 9345 in LNAI, Springer (2015)
6. Brass, S., Stephan, H.: Bottom-up evaluation of Datalog: Preliminary report. In: Schwarz, S., Hölldobler, S. (eds.) 29th Workshop on (Constraint) Logic Programming (WLP 2015). pp. 21–35. HTWK Leipzig (2015), http://www.imn.htwk-leipzig.de/WLP2015/
7. Costa, V.S.: Optimizing bytecode emulation for Prolog. In: Nadathur, G. (ed.) Principles and Practice of Declarative Programming, International Conference PPDP'99. LNCS, vol. 1702, pp. 261–277. Springer (1999)
8. 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
9. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: ARTful indexing for main-memory databases. In: Proc. of the 2013 IEEE International Conference on Data Engineering (ICDE'2013). pp. 38–49. IEEE Computer Society (1997), http://www3.informatik.tu-muenchen.de/~leis/papers/ART.pdf
10. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Logic 7(3), 499–562 (2006), https://arxiv.org/pdf/cs/0211004
11. 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/
12. Liu, Y.A., Stoller, S.D.: From Datalog rules to efficient programs with time and space guarantees. In: Proc. of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03). pp. 172–183. ACM (2003), http://www3.cs.stonybrook.edu/~liu/papers/Rules-PPDP03.pdf
13. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. Proceedings of the VLDB Endowment 4(9), 539–550 (2011), http://www.vldb.org/pvldb/vol4/p539-neumann.pdf
14. Ramakrishnan, R., Srivastava, D., Sudarshan, S., Seshadri, P.: The CORAL deductive system. The VLDB Journal 3, 161–210 (1994)
15. 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
16. Scholz, B., Jordan, H., Subotić, P., Westmann, T.: On fast large-scale program analysis in Datalog. In: Proceedings of the 25th International Conference on Compiler Construction (CC'2016). pp. 196–206. ACM (2016)
17. Schütz, H.: Tupelweise Bottom-up-Auswertung von Logikprogrammen (Tuple-wise bottom-up evaluation of logic programs). Ph.D. thesis, TU München (1993)
18. Smith, D.A., Utting, M.: Pseudo-naive evaluation. In: Australasian Database Conference. pp. 211–223 (1999), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.177.5047