Bottom-Up Evaluation of Datalog: Preliminary Report

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. Bottom-up evaluation of Datalog has been studied for a long time, and is standard material in textbooks. However, if one actually wants to develop a deductive database system, it turns out that there are many implementation options. For instance, the sequence in which rule instances are applied is not given. In this paper, we study a method that immediately uses a derived tuple to derive more tuples. In this way, storage space for intermediate results can be reduced. The main contribution of our method is the way in which we minimize the copying of values at runtime, and do much work already at compile-time.

1 Introduction

The efficient evaluation of queries to logic programs remains an everlasting problem. Of course, big achievements have been made, but at the same time problem size and complexity grows. Any further progress can increase the practical applicability of logic-based, declarative programming.

Our long-term goal is to develop a new deductive database system. This has many aspects, for instance, language design. However, in the current paper, we exclude all special language features, including negation, and focus on efficient query evaluation for basic Datalog.

The magic set method is the standard textbook method for making bottom-up evaluation goal-directed. Many optimizations have been proposed, including our own SLDMagic method [1] and a method based on Earley deduction [3]. We assume in the current paper that such a rewriting of the program has been done, so we can concentrate on pure bottom-up evaluation.

As we understand it, bottom-up evaluation is an implementation of the T_P-operator that computes the minimal model of the program. However, an implementation is free in the order in which it applies the rule instances, while the T_P-operator first derives all facts that are derivable with a given set of known facts, before the derived facts are used (in the next iteration). Furthermore, facts do not have to be stored until the end of query evaluation, but can be deleted as soon as all possible derivations using them have been done, except for the facts that are relevant for the query. Therefore, the sequence of rule instance application becomes important. If one computes predicate by predicate as the

standard textbook method, one of course needs to store the entire extension of the predicates. However, if one uses derived tuples immediately, it might be possible to store only one tuple of the predicate during the evaluation. Of course, for duplicate elimination and termination, it might still be necessary to store extensions of a few selected predicates. It is also not given that tuples (facts) must be represented explicitly as records or objects in the program. It suffices if one knows where the values of single columns (predicate arguments) can be found. In this way, a lot of copying can be saved because tuples for the rule head are typically constructed from values bound in the rule body. Of course, one must ensure that the values are not changed before all usages are finished.

Our plan is to translate Datalog to C++, and to generate executable code from the resulting program. This permits to use existing compilers for low-level optimizations and gives an interface for defining built-in predicates. In [2], we already discussed implementation alternatives for bottom-up evaluation and did performance comparisons for a few example programs. Now we will improve the "push method" from that paper by changing the set of variables used to represent intermediate facts. This is essential for reducing the amount of copying. It also enables us to do more precomputation at "compile time".

The 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 [10, 12]. 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 [10] (and this is essential for the reduction of copying values). The paper [12] translates from a temporal Datalog extension to Prolog, which makes any efficiency comparison dependend on implementation details of the used Prolog compiler. We also believe that the rule application graph introduced in our paper is a useful concept. Further literature about the implementation of deductive database systems is, for instance, [8, 4, 9, 11, 7, 13]. A current commercial deductive DB system is Logicblox [5]. A benchmark collection is OpenRuleBench [6].

2 Basic Definitions

In this paper, we consider basic Datalog, i.e. pure Prolog without negation and without function symbols (i.e. terms can only be variables or constants). We also assume without loss of generality that all rules have at most two body literals. The output of our rewriting methods [1, 3] has this property. (But in any case, it is no restriction since one can introduce intermediate predicates.) Finally, we require range-restriction (allowedness), i.e. all variables in the head of the rule must also appear in a body literal. For technical purposes, we assume that each rule has a unique rule number.

As usual in deductive databases, we assume that EDB and IDB predicates are distinguished ("extensional" and "intensional database"). EDB predicates are defined by facts only, e.g. stored in a relational database or specially formatted files. Also program input is represented in this way. IDB predicates are defined

by rules. There is a special IDB-predicate answer that only appears in the head of one or more rules. The task is to compute the extension of this predicate in the minimal model of the program.

We assume that the logic program for the IDB predicates as well as the query (i.e. the answer-rules) are given at "compile time", whereas the database for the EDB predicates is only known at "runtime". Since the same program can be executed several times with different database states, any optimization or precomputation we can do at compile time will pay off in most cases. It might even be advantageous in a single execution because the database is large.

Since we want to generate C++ code, we assume that there is a data type known for every argument of an EDB predicate. The method does not need type information for IDB predicates (this is implicitly computed).

As mentioned above, our rewriting methods [1, 3] produce rules that have at most two body literals. Furthermore the case of two IDB-literals is rare—it is only used in special cases for translating complex recursions. Most rules have one body literal with IDB-predicate and one with EDB-predicate. Of course, there are also rules with only one body literal (EDB or IDB).

3 Accessing Database Relations

The approach we want to follow is to translate Datalog into C++, which can then be compiled to machine code. Of course, we need an interface to access relations for the EDB predicates. These relations can be stored in a standard relational database, but it is also possible to program this part oneself (at the moment, we do not consider concurrent updates and multi-user access).

We assume that it is possible to open a cursor (scan, iterator) over the relation, which permits to loop over all tuples. We assume that for every EDB predicate p there is a class `p_cursor` with the following methods:

- `void open()`: Open a scan over the relation, i.e. place the cursor before the first tuple.
- `bool fetch()`: Move the cursor to the next tuple. This function must also be called to access the first tuple. It returns `true` if there is a first/next tuple, or `false` if the cursor is at the end of the relation.
- T `col_`i`()`: Get the value of the i-th column (attribute) in the current tuple. Here T is the type of the i-th column.
- `close()`: Close the cursor.

For recursive rules, we will also need

- `push()`: Save the state of the cursor on a global stack.
- `pop()`: Restore the state of the cursor.

A relation may have special access structures (e.g. it might be stored in a B-tree or an array). Then not only a full scan (corresponding to binding pattern f...f) is possible, but also scans only over tuples with given values for certain arguments. We assume that in such cases there are additional cursor classes called

p_cursor_β, with a binding pattern β. These classes have the same methods as the other cursor classes, only the `open`-method has parameters for the bound arguments. E.g. if `p` is a predicate of arity 3 that permits particularly fast access to tuples with a given value of the first argument, and if this argument has type `int`, the class `p_cursor_bff` would have the method `open(int x)`.

4 Duplicate Elimination and Termination

The main contribution of this paper is the way in which copying and materialization of tuples is avoided. Our method basically pushes newly derived facts to body literals where they can be used to derive further facts.

However, in the presence of recursion, we must be able to notice whether a derived tuple is new or not. Therefore, in each recursive cycle, at least one predicate must be materialized ("tabled") to ensure termination. A simple solution is to create hash tables for the predicates in question.

This solution means that we materialize the extensions of some IDB predicates (hopefully, only a few) and copy all data values for the tuples of these predicates. In some cases, information about order or acyclicity might help to avoid this. Information about keys and data distribution could be used to make sensible optimization decisions. Furthermore, if tuples are produced in a sort order, the duplicate check can be done very efficiently and without storing the predicate extension. All this is subject of our future work.

It is also interesting that the data values in a derived tuple are stored at different times in program variables. For instance, we might know that when $\mathsf{p}(\mathsf{X},\mathsf{Y})$ is generated, X only seldom changes, and Y changes much more often. Then a nested relation might be best for tabling the predicate for the purpose of duplicate detection.

Of course, breaking each recursive cycle with a duplicate detection is only the minimum we have to do to ensure termination. Also non-recursive rules can generate duplicates, and in some cases it might be more efficient to detect these duplicates early in order to avoid duplicate computations (since the price for duplicate detection is quite high, in other cases it might be more efficient to simply do the duplicate work).

5 Code Generation: Overall Structure

The result of the translation looks basically as shown in Figure 1. So there are many small code pieces, each with a label that is suitable for a `goto`. Furthermore, when there are several things to do, e.g. a generated fact can be used in more than one rule, a backtrack point is set up for the second rule, and then a `goto` is done for the first. When an execution path reaches an end, the `switch` is left with `break`, and one of the delayed tasks is taken from the stack. Therefore, each code piece also has a unique number, which can be stored on the backtrack stack, and used in the `switch` to reach the code piece.

```
⟨Declaration Section⟩;
⟨Initialization Section⟩; // Initializes backtrack_stack
while(!backtrack_stack.is_empty()) {
     switch(backtrack_stack.pop()) {
          case L1:
          l1:
               ⟨Code Piece 1⟩;
               // break or goto at end of Code Piece
          case L2:
          l2:
               ⟨Code Piece 2⟩;
          ...
     }
}
```

Fig. 1. Overall structure of the generated code

Optimizations are possible, e.g. one can order the code pieces such that some jumps can be eliminated, because the target is immediately following. Some backtrack points can be avoided by finding a suitable code sequence.

5.1 Declaration Section

Data not known at compile time always originates from the database. In order to minimize copying, we (usually) introduce a C++ variable only for Datalog variables which

- occur in an EDB body literal,
- but do not occur in an IDB body literal of that rule (because then the value comes from another rule, where a variable has been created, if the value is not known at compile time),
- and occur in the head of that rule (because otherwise the value does not really have to be processed in the program).

For instance, consider the following rule:

$$\mathsf{p}(\mathsf{X},\mathsf{Y},\mathsf{a}) \leftarrow \mathsf{q}(\mathsf{Y}) \wedge \mathsf{r}(\mathsf{X},\mathsf{Y},\mathsf{Z},\mathsf{Z}).$$

If q is an IDB predicate and r an EDB predicate, we create a C++ variable only for X. A variable or constant for Y exists already when the rule is activated.

In seldom cases of recursive rule applications (see Section 5.4 below) we create C++ variables for all variables of the rule.

If the above condition shows the we must create a C++ variable for variable X in rule ρ, we generate the following code line in the declaration section:

$$T\ \mathtt{v}\rho\mathtt{\_}\mathsf{X};$$

We use the prefix with the rule number so that there can be no name conflicts between variables of different rules. T is the C++ data type for the database column in which X occurs.

5.2 Symbolic Facts

A symbolic fact consists of an IDB predicate p and a tuple $(\mathsf{t}_1, \ldots, \mathsf{t}_n)$ of C++ variables (i.e. their identifiers) and constants, where n is the arity of p. So a symbolic fact represents what is known at compile time about a fact that will be derived at runtime. For some arguments, we might know the exact value (a constant), for other arguments, we know the C++ variable which will contain the value.

An initial set of symbolic facts is derived by rules without IDB body literals. Then our task is to pass each derived symbolic fact to matching IDB body literals and to derive a symbolic fact for the rule head. For each such rule application, a code piece is generated which does the remaining computation at runtime.

"Matching" between a symbolic fact and a body literal means that they are unifiable. In general a full unification must be done (at compile time). Consider e.g. the body literal $\mathsf{p}(\mathsf{X}, \mathsf{X}, \mathsf{a})$ and the symbolic fact $\mathsf{p}(\mathsf{b}, \mathtt{v1\_Y}, \mathtt{v1\_Y})$. The rule cannot be applied to the symbolic fact, so no code is generated for this case.

5.3 Rule Application Graph

A "Symbolic Rule Application" is

- a rule from the logic program with one IDB body literal, together with a symbolic fact matching this body literal,
- a rule without IDB body literals,
- a rule with two IDB body literals, together with a symbolic fact matching one of them. (In the rare case of two IDB body literals, we use temporary tables for facts matching each body literal. The symbolic fact in this rule application describes the situation that we just computed a new fact for one of the IDB body literals. For the other body literal we use the table with previously computed facts.)

The result of a symbolic rule application is a symbolic fact. Let $\mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$ be the head of the rule, and ρ be its rule number. If the rule has an IDB body literal, let θ be a most general unifier with the input symbolic fact. We require that variable-to-variable bindings are done such that logic variables are replaced by C++ variables. Then the derived symbolic fact is $\mathsf{p}(\mathsf{u}_1, \ldots, \mathsf{u}_n)$, where u_i is

- t_i if t_i is a constant.
- $\mathsf{t}_i\theta$ if t_i is a variable which appears in the IDB body literal (if there is one).
- $\mathtt{v}\rho\mathtt{\_X}$ if t_i is a variable X which does not appear in the IDB body literal.

Now we can do a standard fixpoint computation to compute all symbolic facts which are derivable from the program. This process will come to an end, because the number of symbolic facts is bounded: There is only a finite number of C++ variables (at most the number of variables in the given logic program, where variables with the same name in different rules count as distinct). Furthermore, only a finite number of constants occurs in the given logic program (constants which appear only in the database are not known at "compile time" and not used for computing symbolic facts).

The structure of the computation can be shown in a "rule application graph". It has two types of nodes, namely symbolic facts ("fact nodes"), and symbolic rule applications ("rule nodes"). There is an edge from every symbolic fact to every symbolic rule application which uses the symbolic fact. Furthermore, there is an edge from every symbolic rule application to the symbolic fact it generates.

Of course, it is possible to show only the rule in nodes for symbolic rule applications (since the symbolic fact is identified by the incoming edge, except in the case of two IDB body literals). However, then there can be several nodes marked with the same rule: It is possible that a single rule is compiled several times for different symbolic facts matching its IDB body literal.

Note also that not every application of a recursive rule to a symbolic fact is actually recursive: Only if the same symbolic fact can be generated by applying this rule (maybe indirectly via other rules), we have to be prepared for recursive invocations of the code piece for the symbolic rule application. This can be seen from cycles in the graph.

Finally, nodes in the graph from which there is no path to an answer-node can be eliminated: They do not contribute to the computation of the answer. If the program is the result of a program transformation like magic sets, this path will not be followed at runtime, but it is better not to generate code for it. An example of such a program is

$$\begin{aligned} \mathsf{answer}(\mathsf{X}) &\leftarrow \mathsf{q}(\mathsf{X},\mathsf{a}).\\ \mathsf{q}(\mathsf{X},\mathsf{Y}) &\leftarrow \mathsf{p}(\mathsf{Y},\mathsf{X}).\\ \mathsf{p}(\mathsf{a},\mathsf{X}) &\leftarrow \mathsf{r}(\mathsf{X}).\\ \mathsf{p}(\mathsf{b},\mathsf{X}) &\leftarrow \mathsf{s}(\mathsf{X}). \end{aligned}$$

The rule application graph is shown in Figure 2. The right path is useless. In

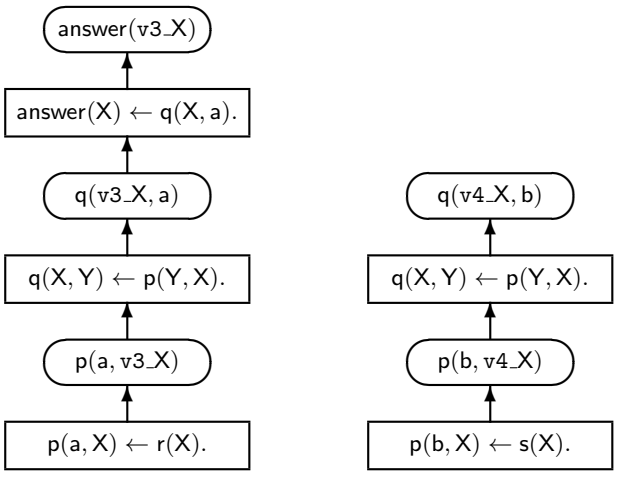

Fig. 2. Rule Application Graph with Useless Part (to be eliminated).

the code generation below, we assume that this has been removed, i.e. every fact node with a predicate different from "answer" has an outgoing edge.

5.4 Variable Conflicts

In rare cases of recursive rule applications, it is possible that a rule is applied to a symbolic fact which contains already a variable generated for that rule. An example is

$\mathsf{p}(\mathsf{X},\mathsf{Y}) \leftarrow \mathsf{r}(\mathsf{X},\mathsf{Y}).$
$\mathsf{p}(\mathsf{Y},\mathsf{Z}) \leftarrow \mathsf{p}(\mathsf{X},\mathsf{Y}) \wedge \mathsf{r}(\mathsf{Y},\mathsf{Z}).$

The first rule generates the symbolic fact $\mathsf{p}(\mathtt{v1\_X},\mathtt{v1\_Y})$. When we insert this into the second rule, we get $\mathsf{p}(\mathtt{v1\_Y},\mathtt{v2\_Z})$. Now we have to insert this again into the second rule: $\mathtt{v2\_Z}$ contains the input value for Y, but must also be set with a new data value from r. In this case, some copying seems unavoidable. While there are optimizations possible, the simplest solution is to create a C++ variable for each logical variable of the rule, and to copy first the values from the input fact to the right variable (which might need temporary variables, e.g. for swapping the values of two variables). For recursive rule applications, the previous variable values are also stored on a stack (see Section 5.6 below).

5.5 Labels for Code Pieces

We need a `goto` label and/or a `case` selector value (a unique number) for each code piece implementing a symbolic rule application. We write

$\mathtt{l\_start}(\mathsf{p}(\mathsf{t}_1,\ldots,\mathsf{t}_n),\ \rho,\ \mathsf{p}(\mathsf{u}_1,\ldots,\mathsf{u}_n))$

for the `goto`-label of the code piece for application of rule ρ with body literal $\mathsf{p}(\mathsf{u}_1,\ldots,\mathsf{u}_n)$ to the symbolic fact $\mathsf{p}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$. The implementation will replace this by $\mathtt{l\_start\_}n$ with some unique number n. The symbolic constant for the `case`-value is written as `L_START(...)` (and also made a legal C++ identifier by using the same unique number). Sometimes there are continuations or other code pieces, therefore the label is marked as "`start`".

5.6 Protection of Variable Values

Of course, when a code piece corresponding to a symbolic rule application is executed, the C++ variables in the symbolic fact $\mathsf{p}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$ must still have the same value as when this task was generated. It is possible that the ID/label of the code piece was pushed on the backtrack stack and it is executed only later.

However, for every C++ variable, a new value is assigned only in code pieces for the single rule for which the variable was introduced (to hold a data value for an EDB literal in that rule).

Furthermore, it is important that the backtrack points are kept on a stack. So we will return to that rule only after all backtrack points which use the value

(and are thus generated later) have been processed—unless the rule is recursive. In this case, the variable value must be saved (on another stack suitable for the data type), and we put the ID of a code piece on the backtrack stack which restores the variable value. This is done whenever we enter a recursive rule, and only for variables set in this rule (the derived fact might contain also variables passed from elsewhere and not changed in the rule).

If the backtrack stack shrinks below this point, all usages of the new variable value are done, and the old value is restored, so that older backtrack points find the value which was current when the backtrack point was created.

6 Code Pieces

In this section, we define a number of code pieces which are translations of different types of rules. Each code piece corresponds to a symbolic rule application. For simplicity, we do not consider variable conflicts (Section 5.4) here.

6.1 IDB-Facts

Suppose the program contains an IDB-fact $\mathsf{p}(c_1, \ldots, c_n)$. Whenever this matches a body literal $\mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n)$ of a rule ρ, the case selector value

```
L_START(p(c_1, ..., c_n), ρ, p(t_1, ..., t_n))
```

is pushed on the backtrack stack during initialization.

6.2 One EDB-Body Literal

Consider the rule $\mathsf{p}(\mathsf{t}_1, \ldots, \mathsf{t}_n) \leftarrow \mathsf{r}(\mathsf{u}_1, \ldots, \mathsf{u}_m)$ where r is an EDB predicate. Let ρ be the rule number. Let $\mathsf{p}(\bar{\mathsf{t}}_1, \ldots, \bar{\mathsf{t}}_n)$ be the symbolic fact generated by the rule ($\bar{\mathsf{t}}_i := \mathsf{t}_i$ if t_i is a constant, and $\bar{\mathsf{t}}_i := \mathsf{v}\rho\_\mathsf{X}$ if t_i is the variable X).

Among all possible cursors `cursor_r_`β for r choose one such that for all bound argument positions i (i.e. $\beta_i = \mathsf{b}$), u_i is a constant. This is always possible because every relation supports a full table scan, i.e. an access path with all argument positions "free". But obviously, if there are constants among the u_i, and there are available indexes, it is best to choose one with the smallest estimated result size. In the declaration section, generate

```
cursor_r_β cρ;
```

Define symbolic constants `L_INIT_`ρ and `L_CONT_`ρ as unique numbers for cases in the switch. Generate the following code in the initialization section:

```
backtrack_stack.push(L_INIT_ρ);
```

All following code is generated in the `switch`:

1. Generate
   ```
   case L_INIT_ρ:
   ```

2. Let $i_1, \ldots, i_k$ be the bound argument positions in β. Generate:
   `c`ρ`.open(`$\mathsf{u}_{i_1}, \ldots, \mathsf{u}_{i_k}$`);`
   (Note that although another `case` follows, execution simply continues.).
3. Generate:
   `case L_CONT_`ρ`:`
   The following loop (item 4) is left with `goto` when the first fact is generated. But before the jump, this case label is pushed on the backtrack stack, so that the loop is continued later.
4. Generate
   `while(c`ρ`.fetch()) {`
5. Let $\mathsf{u}_{i_1}, \ldots, \mathsf{u}_{i_k}$ be the constants among the $\mathsf{u}_1, \ldots, \mathsf{u}_m$ which correspond to free argument positions in β. If $k \geq 1$, generate
   `if(c`ρ`.col_`i_1`() != `u_{i_1}` || ··· || c`ρ`.col_`i_k`() != `u_{i_k}`)`
   `  continue;`
   I.e. if the current tuple of the EDB-predicate does not have the required values for the constant arguments, we immediately start the next iteration of the `while`-loop (i.e. fetch the next tuple).
6. For every variable Y, which appears more than once among the $\mathsf{u}_1, \ldots, \mathsf{u}_m$: Let $\mathsf{u}_{i_1}, \ldots, \mathsf{u}_{i_k}$ be all equal to Y (note that $k \geq 2$). Generate:
   `if(c`ρ`.col_`i_1`() != c`ρ`.col_`i_2` || ··· ||`
   `  c`ρ`.col_`i_{k-1}`() != c`ρ`.col_`i_k`())`
   `  continue;`
7. For every variable X_i in the head let u_j be any occurrence of this variable among the $\mathsf{u}_1, \ldots, \mathsf{u}_m$. Because of the range restriction (allowedness) condition on the rules, it must occur in the body. Generate for each X_i:
   `v`ρ`_`X_i` = c`ρ`.col_`j`();`
8. In case the predicate `p` was selected for a duplicate check, the following must be done here: The result tuple $\mathsf{p}(\bar{\mathsf{t}}_1, \ldots, \bar{\mathsf{t}}_n)$ with the current values of the C++ variables is entered into a hash table or other data structure. If the tuple was already present, one simply does "`continue;`" to skip it.
9. Generate:
   `backtrack_stack.push(L_CONT_`ρ`);`
   Since this code piece will change the values of the variables introduced in the rule, it must be on the stack below every task using the generated tuple.
10. Let $\rho_1, \ldots, \rho_k$ be all rules with an IDB body literal B_i, $i := 1, \ldots, k$, which matches the generated symbolic fact $\mathsf{p}(\bar{\mathsf{t}}_1, \ldots, \bar{\mathsf{t}}_n)$. For $i := 2, \ldots, k$, generate
    `backtrack_stack.push(L_START(`$\mathsf{p}(\bar{\mathsf{t}}_1, \ldots, \bar{\mathsf{t}}_n)$`, `$\rho_i$`, `$\mathsf{B}_i$`));`
    Finally, generate
    `goto l_start(`$\mathsf{p}(\bar{\mathsf{t}}_1, \ldots, \bar{\mathsf{t}}_n)$`, `$\rho_1$`, `$\mathsf{B}_1$`);`
11. Generate
    `} // End of while-loop`
    `break;`
    The `break;` is important if the `while`-loop ends because no (further) matching fact is found in the relation `r`. Otherwise, the loop is left with `goto` when the first/next matching fact is found.

6.3 Two EDB-Body Literals

In the output of SLDMagic, this case does not occur. However, it is easy to extend the above program code. One uses two cursors, one for each body literal, and two nested `while`-loops. For simplicity, we implement all joins as "nested loop join". Later, sort orders might be used, so that also a "merge join" can be generated (which is faster than the nested loop join).

6.4 One IDB-Body Literal

Consider the rule

$$\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n) \leftarrow \mathtt{q}(\mathtt{u}_1, \ldots, \mathtt{u}_m),$$

where $\mathtt{q}$ is an IDB-predicate. Let ρ be the number of this rule. There is one code piece per symbolic fact $\mathtt{q}(\bar{\mathtt{u}}_1, \ldots, \bar{\mathtt{u}}_m)$ which matches the body literal. Let θ be a most general unifier, where variable-to-variable bindings are done such that logic variables are replaced by C++ variables i.e. $\mathtt{u}_i$ is replaced by $\bar{\mathtt{u}}_i$, if both are variables. The generated symbolic fact is $\mathtt{p}(\mathtt{t}_1\theta, \ldots, \mathtt{t}_n\theta)$. Note that because of the range restriction requirement, every variable among the $\mathtt{t}_i$ also appears as an $\mathtt{u}_j$, and then it is unified with a constant or a C++ variable.

1. Generate
   `case L_START(`$\mathtt{q}(\bar{\mathtt{u}}_1, \ldots, \bar{\mathtt{u}}_m)$`, `$\rho$`, `$\mathtt{q}(\mathtt{u}_1, \ldots, \mathtt{u}_m)$`):`
   `l_start(`$\mathtt{q}(\bar{\mathtt{u}}_1, \ldots, \bar{\mathtt{u}}_m)$`, `$\rho$`, `$\mathtt{q}(\mathtt{u}_1, \ldots, \mathtt{u}_m)$`):`
2. Now the part of the unification which can only be done at runtime must be generated. Let $V_1, \ldots, V_k$ be all C++ variables which θ replaces by constants or a different variable (i.e. $V_i\theta \neq V_i$). If $k > 0$, generate:
   `if(`V_1 `!=` $V_1\theta$ `||` $\cdots$ `||` V_k `!=` $V_k\theta$`)`
   `  break;`
   So we simply stop executing this code piece if the current fact for $\mathtt{q}$ does not unify with the body literal.
3. In case predicate $\mathtt{p}$ was selected for a duplicate check, the code to enter the result tuple $\mathtt{p}(\mathtt{t}_1\theta, \ldots, \mathtt{t}_n\theta)$ with the current values of the C++ variables into a hash table would go here. If the tuple was already present (so we just computed a duplicate), one simply does "`break;`" to end the code piece. Then another task will be taken from the backtrack stack.
4. Let $\rho_1, \ldots, \rho_k$ be all rules with an IDB body literal $\mathtt{B}_i$, $i := 1, \ldots, k$, which matches the generated symbolic fact $\mathtt{p}(\mathtt{t}_1\theta, \ldots, \mathtt{t}_n\theta)$. For $i := 2, \ldots, k$, generate
   `backtrack_stack.push(L_START(`$\mathtt{p}(\mathtt{t}_1\theta, \ldots, \mathtt{t}_n\theta)$`, `$\rho_i$`, `$\mathtt{B}_i$`));`
   Finally, generate
   `goto l_start(`$\mathtt{p}(\mathtt{t}_1\theta, \ldots, \mathtt{t}_n\theta)$`, `$\rho_1$`, `$\mathtt{B}_1$`);`

6.5 One IDB- and one EDB-Body Literal

Consider the rule

$$\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n) \leftarrow \mathtt{q}(\mathtt{u}_1, \ldots, \mathtt{u}_m) \wedge \mathtt{r}(\mathtt{v}_1, \ldots, \mathtt{v}_l),$$

where $\mathtt{q}$ is an IDB-predicate and $\mathtt{r}$ is an EDB-predicate. Let ρ be the number of this rule. There is one code piece per symbolic fact $\mathtt{q}(\bar{\mathtt{u}}_1, \ldots, \bar{\mathtt{u}}_m)$ which matches the IDB body literal. Let θ be a most general unifier and $\mathtt{p}(\bar{\mathtt{t}}_1, \ldots, \bar{\mathtt{t}}_n)$ be the generated symbolic fact as defined in Section 5.3. As in Section 6.2, select a binding pattern β for accessing the EDB-relation $\mathtt{r}$. A value for $\mathtt{v}_i$ is known (i.e. β_i can be "bound") if $\mathtt{v}_i$ is a constant or a variable which also appears in $\mathtt{q}(\mathtt{u}_1, \ldots, \mathtt{u}_m)$ (when execution reaches this code piece, a concrete fact is given for the IDB body literal). In the declaration section, generate

```
cursor_r_β cρ;
```

All following code is generated in the `switch`:

1. Generate
   ```
   case L_START(q(ū1, ..., ūm), ρ, q(u1, ..., um)):
   l_start(q(ū1, ..., ūm), ρ, q(u1, ..., um)):
   ```
2. Now the part of the unification of the given fact with the IDB body literal, which can only be done at runtime, must be generated. Let $V_1, \ldots, V_k$ be all C++ variables with $V_i\theta \neq V_i$. If $k > 0$, generate:
   ```
   if(V1 != V1θ || ··· || Vk != Vkθ)
       break;
   ```
   This ends the execution of this code piece if the rule is not applicable.
3. Now we must use a cursor to access the tuples for the EDB body literal $\mathtt{r}(\mathtt{v}_1, \ldots, \mathtt{v}_l)$. In case this rule is recursive, it might be possible that the state of the cursor and the values of the variables $\mathtt{v}\rho\_\mathtt{X}$ set in this rule are still needed by backtrack points on the stack (unless we know that there are no such backtrack points, e.g. because the recursive rule application is the last use of the fact). Therefore, we generate
   ```
   cρ.push();
   ```
   And for each variable $\mathtt{v}\rho\_\mathtt{X}$ we generate
   ```
   value_stack.push(vρ_X);
   ```
   (There are probably several value stacks for different data types.) Finally we generate a backtrack point:
   ```
   backtrack_stack.push(L_RESTORE_ρ);
   ```
   The code for this `case` in the `switch` simply restores the variable values and the cursor state by popping them (in the inverse order). In this way, all earlier backtrack points (below the one just generated) find the old cursor state and variable values.
4. Let $i_1, \ldots, i_k$ be the bound argument positions in β. Generate:
   ```
   cρ.open(v̄i1, ..., v̄ik);
   ```
   where $\bar{\mathtt{v}}_{i_j}$ is
   - $\mathtt{v}_{i_j}$ if this is a constant,
   - $\mathtt{v}_{i_j}\theta$ if $\mathtt{v}_{i_j}$ is a variable which appears in the IDB body literal $\mathtt{q}(\ldots)$.

5. Generate:

   `case L_CONT_`ρ`:`

   When we are finished with using the computed fact, backtracking returns here to continue the following loop.
6. Generate

   `while(c`ρ`.fetch()) {`
7. Let $i_1, \ldots, i_k$ be the free argument positions in β such that $\mathtt{v}_{i_j}$ is a constant or a variable which appears the the IDB body literal $\mathtt{q}(\ldots)$. Let $\bar{\mathtt{v}}_{i_j}$ be defined as in 4 above. If $k \geq 1$, generate

   `if(c`ρ`.col_`i_1`() != `$\bar{\mathtt{v}}_{i_1}$` || ··· || c`ρ`.col_`i_k`() != `$\bar{\mathtt{v}}_{i_k}$`)`
   `    continue;`

   I.e. if the current tuple in the EDB relation does not have the required values, we continue with the next iteration of the `while`-loop under 6.
8. For every variable Y, which appears more than once among the $\mathtt{u}_1, \ldots, \mathtt{u}_m$, but not in the IDB literal $\mathtt{q}(\ldots)$: Let $\mathtt{u}_{i_1}, \ldots, \mathtt{u}_{i_k}$ be all equal to Y (note that $k \geq 2$). Generate:

   `if(c`ρ`.col_`i_1`() != c`ρ`.col_`i_2` || ··· ||`
   `   c`ρ`.col_`i_{k-1}`() != c`ρ`.col_`i_k`())`
   `        continue;`
9. For every variable X_i in the head, which does not appear in the IDB body literal $\mathtt{q}(\ldots)$, let $\mathtt{u}_j$ be any occurrence of this variable among the $\mathtt{u}_1, \ldots, \mathtt{u}_m$. Because of the range restriction (allowedness) condition on the rules, it must occur there. Generate for each X_i:

   `v`ρ`_`X_i` = c`ρ`.col_`j`();`
10. In case predicate `p` was selected for a duplicate check, we again enter the result tuple $\mathtt{p}(\bar{\mathtt{t}}_1, \ldots, \bar{\mathtt{t}}_n)$ with the current values of the C++ variables into a hash table. If the tuple was already present, one simply does "`continue;`" to compute the next tuple.
11. Generate:

    `backtrack_stack.push(L_CONT_`ρ`);`
12. Let $\rho_1, \ldots, \rho_k$ be all rules with an IDB body literal B_i, $i := 1, \ldots, k$, which matches the generated symbolic fact $\mathtt{p}(\bar{\mathtt{t}}_1, \ldots, \bar{\mathtt{t}}_n)$. For $i := 2, \ldots, k$, generate

    `backtrack_stack.push(L_START(`$\mathtt{p}(\bar{\mathtt{t}}_1, \ldots, \bar{\mathtt{t}}_n)$`, `$\rho_i$`, `$\mathsf{B}_i$`));`

    Then generate

    `goto l_start(`$\mathtt{p}(\bar{\mathtt{t}}_1, \ldots, \bar{\mathtt{t}}_n)$`, `$\rho_1$`, `$\mathsf{B}_1$`);`
13. Finally, we must close the open `while`-loop (6. above) and finish the code piece in case the loop does not find any (further) matching tuple. Generate:

    `}`
    `break;`

In addition, there is a code piece for `case L_RESTORE_`ρ as explained under item 3 above. It pops everything pushed there.

6.6 Two IDB-Body Literals

This is a complicated case and needs intermediate storage of tuples generated for the body literals. Fortunately, this occurs rarely in the output of the SLDMagic method (only when translating recursions that are not tail recursions).

A general solution, which does not need information about the order of generated tuples, is to have one set of tuples for each body literal. In the code piece for the case that a new tuple has been derived for the left body literal, this tuple is joined with all tuples in the current set for the right body literal. In the same way, when a new tuple is generated for the right body literal, it is joined with all existing tuples for the left body literal.

This means that we now need cursors also for the intermediate storage of generated IDB facts, and these cursors must keep information about the last fact when they were created (since new facts can be appended to the list while the cursor is active—these facts must not be returned by the cursor).

Recursion can be handled in the same way as before: When a new fact is generated for a body literal, we save the state of the cursor and all variables of that code piece, continue with derivations using the new fact, and later return to the old fact. However, since we anyway have intermediate storage now, it is also possible to create a queue of facts for each body literal, which must still be used in derivations.

If it is possible to generate all facts for the left body literal before the first fact for the right body literal, one obviously needs intermediate storage only for the left body literal. In this case it can later be treated like an EDB literal.

If one can generate facts for both literals in the sort order of the common variables (the join attributes), we would need intermediate storage only for a single tuple for each body literal (we would basically do a merge join).

6.7 Generated answer-Facts

For rules about the predicate answer, one can print the generated tuple or insert it into a result relation whenever the above code would jump to a body literal which uses the generated fact (there are no body literals with predicate answer). One can also offer the cursor interface of Section 3.

7 Conclusion

We have presented a detailed description of the push method, an efficient bottom-up evaluation algorithm for pure Datalog programs. In the push method, derived facts are immediately used to derive new facts without generally materializing immediate results. A specific feature is the representation of derived tuples which significantly reduces the amount of copying. The rule application graph introduced here is useful for planning the evaluation. First performance tests show some improvement over the previous version of the push method from [2]. We plan to develop a more complete implementation and to investigate further optimizations. The current state of the project will be reported at `http://www.informatik.uni-halle.de/~brass/botup/`

References

1. 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). pp. 1063–1077. No. 1861 in LNCS, Springer (2000)
2. 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
3. Brass, S., Stephan, H.: A variant of earley deduction with partial evaluation. In: Faber, W., Lembo, D. (eds.) Web Reasoning and Rule Systems - 7th International Conference, RR 2013. LNCS, vol. 7994, pp. 35–49. Springer-Verlag (2013), http://dbs.informatik.uni-halle.de/Earley/
4. Derr, M.A., Morishita, S., Phipps, G.: The Glue-Nail deductive database system: Design, implementation and evaluation. The VLDB Journal 3, 123–160 (1994)
5. Green, T.J., Aref, M., Karvounarakis, G.: Logicblox, platform and language: A tutorial. In: Barceló, P., Pichler, R. (eds.) Datalog in Academica and Industry, 2nd Int. Workshop, Datalog 2.0. LNCS, vol. 7494, pp. 1–8. Springer-Verlag (2012), https://developer.logicblox.com/2012/08/
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. Liu, M.: Design and implementation of the ROL system. Journal of Intelligent Information Systems 14, 1–21 (2000), http://www.scs.carleton.ca/~mengchi/papers/rol-JIIS00.ps
8. Ramamohanarao, K.: An implementation overview of the Aditi deductive database system. In: Ceri, S., Tanaka, K., Tsur, S. (eds.) Deductive and Object-Oriented Databases, Third Int. Conf., (DOOD'93). pp. 184–203. No. 760 in LNCS, Springer (1993)
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)
10. Schütz, H.: Tupelweise Bottom-up-Auswertung von Logikprogrammen (Tuple-wise bottom-up evaluation of logic programs). Ph.D. thesis, TU München (1993)
11. Seshadri, P., Flisakowski, S., Hersh, S.: Coral: The inside story. Shocking hacks revealed. Tech. rep., Department of Computer Sciences, The University of Wisconsin-Madison (1996), http://ftp.cs.wisc.edu/coral/doc/Inside.ps
12. Smith, D.A., Utting, M.: Pseudo-naive evaluation. In: Australasian Database Conference. pp. 211–223 (1999), http://www.cs.waikato.ac.nz/research/jstar/1999-ADC-pseudo-naive-eval.pdf
13. Yang, G., Kifer, M.: FLORA: Implementing an efficient DOOD system using a tabling logic engine. In: Lloyd, W., et al. (eds.) First International Conference on Computational Logic (CL'2000/DOOD'2000). pp. 1078–1093. No. 1861 in LNCS, Springer (2000)