An Abstract Machine for Push Bottom-Up Evaluation with Declarative Output

Stefan Brass

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

Abstract. The Push Method for Bottom-Up Evaluation in deductive databases was previously defined as a translation from Datalog to C++. Performance tests on some benchmarks from the OpenRuleBench collection gave very encouraging results. However, most of the systems used for comparison compile the query into code of an abstract machine and then use an emulator for this code. Therefore, runtimes cannot be directly compared. In this paper, we propose an abstract machine for bottom-up evaluation of Datalog based on the Push Method. This also helps to clarify some optimizations we previously expected from the C++ compiler. Since the interpreted code of the abstract machine must do something useful "standalone", we also consider declarative output with templates.

1 Introduction

The database language SQL is a very successful declarative language, but usually only parts of applications are developed in SQL, the rest is written in a standard language like Java or PHP. The purpose of deductive databases is to increase the declaratively specified part of an application (ideally to 100% for many applications). Declarative languages have important advantages:

- Programs are compact (shorter than equivalent programs in classical languages), thus program development is faster.
- The language is relatively simple, therefore it can be used also by non-experts (e.g., not everybody using SQL is a professional programmer).
- The language has a mathematical precise semantics (usually based on logic), which makes programs easier to verify.
- The language is not tied to a specific execution model, thus it is easier to execute on new computing platforms, such as multicore processors or massively parallel clouds.

One reason for the current revival of Datalog is that it is used also for applications which are not traditional database applications, such as static analysis of program code [14], cloud computing [10,16] and semantic web applications [7,13]. The commercial deductive database system LogicBlox [1] is successful probably because it offers many functions in an integrated system with only one language.

Heike Stephan and the author have developed the "Push" method for bottom-up evaluation of Datalog [3,5,6]. 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 this way, the derived facts often do not have to be materialized, and temporary storage can be saved. It can be seen as an extreme form of seminaive evaluation that dates back to the PhD thesis of Heribert Schütz [15], see also [17]. However, we use partial evaluation and treat database predicates specially, and did performance evaluations with very promising results. In the last time, "pushing" tuples through relational algebra expressions has also become an attractive technique for standard databases [12]. It seems well suited for modern hardware because it keeps the actively used set of data small.

The Push Method was defined as a translation from Datalog to C++. Performance tests on some benchmarks from the OpenRuleBench collection [11] gave encouraging results. However, most of the systems used for comparison (XSB, YAP, DLV, HSQLDB) compile the query into code of an abstract machine (an extension of the WAM) and then use an emulator for this virtual machine. Since we compiled to native code, runtimes were not directly comparable. Experiences with compiling Prolog to machine code [8] suggest that this gives approximately a factor of 3 (the range in that paper was between 1.3 and 5.6). In a first test of an early prototype, our abstract machine was only 1.5 times worse than the compiler-based approach. This strengthens our previous performance claims.

Our performance comparison also did not contain the runtime of the C++ compiler (because the compilation result can be executed many times on different database states). For large benchmark programs such as the wine ontology, the compilation runtime is quite substantial. In general, during program development, when the program changes often and is executed only on small test data, it is preferable to reduce the compilation overhead.

Another advantage of the program execution with an abstract machine is that the user does not need to install a C++ compiler. In this way, also the interfacing with prossibly different compilers is avoided.

Furthermore, in the last version of the Push Method with procedures [6], many optimizations were delegated to the C++ compiler. While this makes the method easy to understand and produces quite readable code, it reduces the understanding of the lower-level execution which contributes to the good performance results. Going down to the level of an abstract machine can help here.

The abstract machine could also be a step towards a direct translation to machine code based on the LLVM compiler infrastructure (in [12], performance improvements were noticed by compilation to LLVM code instead of C++).

So far we were concerned mainly with the computation of derived data (query results). We expected that the generated C++ program can be linked with a manually created main program that uses the computed data. If one uses an emulator for an abstract machine, however, it must be sufficiently self-contained to do something useful with the computed data. In this paper, we consider declarative output based on templates that we introduced in [4].

2 Input Language: An Example

The input language is basic Datalog, i.e. pure Prolog (Horn clauses) with only constants and variables, but no function symbols. Obviously, the language should be extended later (e.g., by adding negation, aggregation and arithmetic computations). But for the first performance tests to check the approach, this very basic language is sufficient. As an example, consider the well-known transitive closure program, which is one benchmark from the OpenRuleBench collection [11]:

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

The query is `tc(X,Y)`, i.e. all derivable facts should be computed. Other benchmarks check goal-directed computation, and we have also good results for the query `tc(1,X)` by applying our SLDMagic transformation [2], but the focus in this paper is on pure bottom-up evaluation.

The predicate `par` is a database predicate. The benchmark contains files with e.g. 50 000 facts for this "parent" relation, such as `par(1,2)`. In our approach such predicates must be declared with argument types:

```
db par(int, int) facts 'par.dl'.
```

For our compiler-based approach, we have implemented a data loader for Datalog facts. We plan to develop additional loaders for different file formats (e.g., CSV, JSON, XML). The input to the program is made available by reading the data files into main memory relations for the database predicates. Also command line arguments or the data of an HTTP request could be represented in this form.

We use main-memory relations matching the required access patterns in the rules. Therefore, in the example, the `edge` facts are stored two times:

- In a list `par_ff` that permits to iterate over all facts for the first rule (`ff` is the corresponding binding pattern: "free, free").
- In a multi-map (e.g. a hash table) `par_fb` for the literal `par(X, Y)` in the second rule. This data structure permits to iterate over all `X`-values given a value for `Y` (binding pattern "free, bound"). The Push Method will activate the second rule when a new fact for the body literal `tc(Y, Z)` is found, therefore we know a value for `Y` when we access `par` here.

This corresponds to a relation in a classical database with an index over the second column (however, the index usually contains ROWIDs, which are pointers into the main heap file for the relation, in our case it directly contains the data).

Output is defined by a series of "templates". The idea is that one first computes the necessary data by means of Datalog rules and then instantiates templates with parameters to actually generate the output. One can also understand the templates as special procedures that contain only printing commands, calls to other templates, and accesses to data from the computed "answer predicates".

A template definition starts with the name of the template and an optional parameter list. This "template head" looks like a Prolog/Datalog literal with only variables as arguments. It is followed by the "template body", which is written in "`[...]`". This is a list consisting of

- Text pieces written in `'...'` (string constants),
- parameters of the template,
- calls to other templates possibly with values for the parameters in `(...)`,
- calls to other templates with an iterator query in the following form

  ```
  template(Args)<Sort>+Sep :- Lit.
  ```

  This means that `template(Args)` is called once for each answer to the Datalog literal `Lit`. The literal can contain the template parameters, new variables, and constants. The variables can be used as arguments to the template call (in `Args`), and in the specification of the sort order of the template calls (`Sort`). This is a comma-separated list of sort criteria (similar to the `ORDER BY` clause of SQL). One can also optionally specify a separator string `Sep` that is inserted between each two instantiations of the template.

In the example, output is specified as follows (generation starts from "`main`"):

```
main: [
    html_begin('All Derivable tc-Facts')
    ul_begin
    result_item(X, Y)<X, Y> :- tc(X, Y).
    ul_end
    html_end
].
result_item(X, Y): [
    li_begin
        X ', ' Y
    li_end
].
```

It contains calls to six templates of a small HTML library (see below). The third call contains the loop over the answers and produces a sorted list.

There is also a "verbatim mode", that starts and ends with "`|`". In this mode, all characters except "`|`" and "`[`" (and a newline immediately after the opening "`|`") are copied literally to output. With "`[`" one can nest the interpreted "code mode" within the verbatim mode (e.g., for inserting a parameter value).

```
html_begin(Title): [|
<html>
    <head>
        <title>[Title]</title>
    </head>
    <body>
        <h1>[Title]</h1>
|].

html_end: [|
    </body>
</html>
|].
```

```
ul_begin: [|
        <ul>
|].

ul_end: [|
        </ul>
|].

li_begin: [|
        <li>|].

li_end: [|</li>
|].
```

3 The Push Method: Explanation with Procedures

The idea of the Push Method is that the producer of derived facts has the control and actively "pushes" these facts to the consumer (rules in which the produced facts match body literals). In [3], we contrasted it with the "Pull" method where the consumer fetches the next fact when needed. This open-next-close interface of the query plan operators is also known as "Vulcano-style" [9] and is very common in relational database systems. However, in the last time, push methods have also been used successfully in standard databases [12].

In [3,5], we defined the Push Method for bottom-up evaluation of Datalog relatively low-level, managed our own stack and used C++ basically as a portable assembler. Then it turned out that a high-level version with procedures worked more or less equally fast for several tested benchmarks [6]. Current compilers are able to do many of the optimizations that were explicit in the low-level version. The version with procedures is actually very similar to an approach proposed by Heribert Schütz in his PhD thesis [15]. However, at that time, the results of first performance tests were not very encouraging. The approaches differ also in the details, for instance, we treat the database predicates specially.

The idea of the method is quite simple: One creates a procedure for each derived predicate p that is called whenever a fact $p(c_1, \ldots, c_n)$ is derived. The task of the procedure call is to make sure that all rule instances with $p(c_1, \ldots, c_n)$ in the body are eventually applied. This is simple for rules that contain only a single body literal with a derived predicate ("linear rules"). Since the complete relations for the other (database) body literals are known, the necessary joins, selections and projections can be immediately done in order to perform the procedure calls corresponding to the head of the rule. For instance, the procedure for the "transitive closure" program from Section 2, looks as follows:

```
void tc(int c1, int c2) {
    // Is this fact a duplicate?
    if(!tc_bb.insert(c1, c2)) // Set data structure
        return;

    // This is the query predicate, store answer:
    tc_ff.insert(c1, c2); // List data structure

    // Rule tc(X, Z) :- par(X, Y), tc(Y, Z):
    int Y = c1;
    int Z = c2;
    cur_1_1_c lit_1(&par_fb); // iterator over multimap
    lit_1.open(Y);
    while(lit_1.fetch()) { // Loop over all X with par(X,Y)
        int X = lit_1.out_1(); // First&only output column
        tc(X, Z);
    }
}
```

`cur_1_1_c` is a class for cursors/iterators over the result of a lookup in multimaps from rows with one column to rows with one column, generated from a template class for general multimap cursors.

The procedure for a predicate p contains one code block for each rule that contains the predicate p in the body (in this case, it is only one rule).

For "complex rules" that have more than one body literal with a derived predicate, temporary storage of derived facts seems unavoidable. While for special cases, optimizations are possible, in general one creates a temporary table for each body literal with a derived fact. When a rule is activated for a new fact for a specific body literal, one applies all rule instances with this fact and the previously derived facts stored in the temporary tables for the other body literals. As explained in [15], this can be seen as an extreme form of seminaive evaluation, where the "delta" consists of a single fact.

Finally, a procedure "`start`" applies all rules without derived predicates in the body. Note that the database predicates already have been loaded:

```
void start() {
    // Rule tc(X, Y) :- par(X, Y):
    cur_2_c lit_1(&par_ff); // iterator over par-list
    lit_1.open();
    while(lit_1.fetch()) { // Loop over (X,Y) with par(X,Y)
        int X = lit_1.col_1(); // First column of current row
        int Y = lit_1.col_2(); // Second column
        tc(X, Y);
    }
    lit_1.close();
}
```

4 Memory Areas of the Bottom-Up Abstract Machine

Now we present an abstract machine that has instructions for the building blocks of the previous generated C++ code. Let us start with memory.

4.1 String Memory

Names of relations, output texts, and string constants in the given rules must be stored. All arguments of machine instructions are integers, therefore string constants are mapped to unique integers. When facts for the database predicates are loaded, these mappings are used and extended. One can define several domains, and assign each column of a database predicate to one domain. This leads to small, sequential numbers for each domain.

There is a different map data structure for each string domain. For large domains, we use an efficient implementation of a radix tree, for small domains, a simple fixed-size hash table. Of course, the inverse mapping (from integers back to strings) must also be supported, so that query results can be printed.

There is also a system string table for texts to be printed, where uniqueness is not important. This supports only the mapping from numbers to strings.

4.2 Relations

As explained above, relations are used for storing extensions of database predicates. Currently, all columns have type `int`, strings are mapped to integers as explained above. Later, we will have to add `float` or `double` values. Also 8-bit and 16-bit integers could lead to more compact data. But in this first version, all data are standard `int` values.

Each relation data structure supports a specific binding pattern (bound/free, i.e. input/output arguments). For instance, lists correspond to "all arguments free" binding pattern and support only a "full table scan". Sets correspond to the "all arguments bound" binding pattern and support only an element test. Multimaps have bound and free arguments: One can loop over values for the free arguments given values for the bound arguments (we do not have keys yet, otherwise we might know that there is only a single tuple for given values of the input arguments).

We assume that the user specifies for which derived predicates duplicates should be detected (similar to requesting tabling for a predicate, e.g. in XSB). This must be done at least once in each recursive cycle in order to guarantee termination. There is a set data structure for each selected predicate.

Finally, the query predicate or the predicates used in output templates must also be stored in relations. While for a classical query predicate, a list data structure suffices, output templates can use a specific binding pattern, and might also require sorted output of the selected rows. Our plan is to use tree data structures for these output predicates, so that the tuples are already stored in the order in which they are required. This might also be useful for doing merge joins. Currently we only have index joins.

Note that the same predicate with the same extension can be stored in different relation data structures, if there are body literals that require different access structures.

Relations are identified in the program by relation IDs (small numbers).

4.3 Load Specifications

We have implemented a loader for data files formatted as Datalog facts. The loader needs the file names and the set of predicates to be expected in each file. For predicates, the list of argument types is needed, and the domain (string table) to be used if the argument is of type string.

Furthermore, each predicate is linked to a set of relation data structures, in which the loaded tuples are inserted. For each such relation, an extended binding pattern is specified, which defines the mapping of the loaded data tuple to an entry in the relation. There are five options for each argument i, most of which have an integer parameter value j:

- Store argument i of the loaded fact in input column j of the relation data structure (this corresponds to the "bound" case).
- Store argument i of the loaded fact in output column j of the relation data structure (this corresponds to the "free" case").

- Check that argument i of the loaded fact has value j. Otherwise, the fact is not stored. This corresponds to the case that the body literal to be supported by the relation contains constant j (or a string mapped to j). Thus, it is possible to do the selection already when the data is loaded. For instance, in the DBLP benchmark [11], this is a very useful feature.
- Check that argument i of the loaded fact has the same value as argument j. Otherwise, the fact is not stored. This corresponds to multiple occurrences of the same variable in a body literal.
- Do nothing with argument i. This corresponds to an anonymous variable.

In the transitive closure example, loaded `par`-facts are stored

- once in a list `par_ff` (lists have only output columns: argument 1 is stored in output column 1, argument 2 stored in output column 2), and
- once in a multimap `par_fb` (with argument 1 stored in output column 1, and argument 2 stored in input column 1).

4.4 Variables/Registers

Data values during the computation are stored in variables (registers of the abstract machine). Each variable can store a single `int` value. Also arguments of procedures are passed in these variables. Variables are identified by a single (quite small) integer value. In the transitive closure example, only two variables are used (for the two arguments of the `tc` procedure). A program must define how many variables it is going to use.

4.5 Cursors

Cursors are data structures used for iterating over the result of a relation access. If the relation is a list, the cursor supports a loop over all elements of the list. If the relation is a multimap, the cursor supports to loop over the result values for a given tuple of input values.

4.6 Stack

Values of type `int` can be pushed on the stack of the abstract machine. The stack is used for saving the following data:

- Return addresses for procedure calls. In this case, the `int` value is an address in the code area (the next instruction to be executed after the procedure call).
- Saved values of variables: For recursive procedure calls, it might be necessary to save the value of a variable, and restore the old value later from the stack.
- Saved positions of cursors: For recursive procedure calls, it might be necessary to save the current state of a cursor on the stack, and restore the cursor after the procedure call to that state. This is not always a single integer (the position in the list), e.g. it might also include the number of elements in the list: If the cursor is used for looping over a set of derived facts for a complex rule, it is required that insertions do not change the set of tuples over which the cursor runs.

4.7 Code (Machine Instructions)

Of course, the abstract machine also contains a storage area for machine instructions. In the current prototype, this is an array of type `unsigned char`. We try to keep the code compact, therefore the instruction length is not a multiple of 32-bit integers. Each instruction has an 8-bit opcode, and then arguments as needed for the instruction. In the current (very first, experimental) prototype, the longest instruction contains an 8-bit cursor ID, a 16-bit address, and an 8-bit variable number. It is clear that this will not be sufficient for larger programs. However, in the interest of compact code (which is more cache-friendly), there probably will be short and long forms of instructions.

4.8 Instruction Pointer

As any CPU, the abstract machine has an instruction pointer (program counter) that contains the address of the next instruction to execute.

4.9 Error Indicator

There is a boolean variable (a flag) that is set when an error is detected, e.g. a stack overflow or a failed insertion into a relation due to insufficient memory. In this case, program execution terminates at the next instruction.

5 Instructions of the Bottom-Up Abstract Machine

5.1 NULL

There is a `NULL` (no op) instruction that does nothing. It could be used to fill space in the code area when alignment of arguments, e.g. on 32-bit boundaries, becomes interesting.

5.2 HALT

The `HALT` instruction finishes program execution. It is written at the end of the main program (procedure `start`).

5.3 Procedure Calls: CALL and RETURN

The instruction `CALL` has an argument for the address of the called procedure. It pushes the instruction pointer on the stack (which has already been incremented to point to the next instruction after the `CALL`), and jumps to the given address (by setting the instruction pointer).

`RETURN` sets the instruction pointer to the address on top of the stack (and pops that address).

5.4 Duplicate Check

The `DUPCHECK` instruction has two arguments: A set ID s and a variable number v. The set data structure has a method for getting the number of columns n. Then the tuple stored in variables v, $v+1$, ..., $v+n-1$ is inserted into the set. If this insertion fails, because the tuple is already contained in the set, the machine implicitly performs a `RETURN`, i.e. jumps to the topmost address on the stack. We try to keep the code compact, so a single instruction corresponds to the following code from the `tc`-procedure shown above:

```
// Is this fact a duplicate?
if(!tc_bb.insert(c1, c2)) // Set data structure
    return;
```

The larger the granularity of the single instructions, the smaller is the overhead for interpretation.

We also added special instructions for small numbers of columns (currently 1 and 2). This permits to have the constants compiled into the code and use a series of instructions instead of a loop ("loop unrolling").

Requiring that the argument values are stored in consecutive variables means that it might be necessary to copy values from variables at non-consecutive numbers to a fresh set of variables. The compiler can of course try to pass arguments to predicate procedures in consecutive variables, but this is not always possible without copying values (which we want to avoid). Therefore, we will add a `DUPCHECK` instruction with a list of variable numbers (corresponding to the number of columns in the set). While the copying to an internal array must still be done (to construct the tuple for the `insert` operation), the code is more compact than with a series of copy operations. Furthermore, the intention of the instruction in the assembler program is clearer.

If we later do inlining of procedures, a duplicate check instruction will be needed that does a jump instead of the procedure return.

5.5 Saving and Restoring Values of Variables

Arguments are passed to procedures in variables (which are global storage locations), not on the stack. With the methods explained in [5], one can save a lot of copying in this way. However, sometimes one has to choose between copying or generating several procedures for the same predicate with different variables for the arguments. We did a partial evaluation that also handles the case that sometimes constants are known for arguments. Compilers that do inlining of procedure calls and copy propagation also might generate code for the same procedure with arguments in different storage locations.

If a procedure uses a variable that might contain a value that is still needed by the caller (or possibly an indirect caller), the variable must be saved and later restored. Variables can be assigned in such a way that this happens only in recursive procedures. Because not all arguments necessarily change from one recursive call to the next, it is better to save only what is actually needed than

to allocate everything on the stack. E.g., in the transitive closure example, the second argument of `tc` is not changed from the body to the head. It might be helpful to write the recursive rule as

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

Therefore, no copying is done for the variable corresponding to the second argument (in this example, the tail recursion optimization of standard Prolog implementations would basically to the same, although the rule is applied in the opposite direction).

The instruction `SAVE_VAR` pushes the value of a variable on the stack, and the instruction `RESTORE_VAR` pops it again into the variable. Both have a variable number as argument.

5.6 Saving and Restoring Cursor States

In the same way, the current state of a cursor must be saved before it is opened if it might already be open in a procedure invocation somewhere up in the call tree. It seemed better to use a global cursor and push only its position on the stack, because cursors can be quite big objects that contain parts that do not depend on the current position (the link to the relation) and parts that are redundant for speeding up access (derived from the current position).

We use different instructions for different cursor types. E.g. `SAVE_MMAP_1_1` calls the `push`-method of a cursor class for multimaps from one column to one column (remember that we treat common cases specially). We try to avoid virtual methods because of their overhead. However, there will also be a `SAVE_CURSOR` instruction that looks up the type of the cursor at runtime and does the corresponding type cast. In any case, the current position of the cursor is pushed on the stack (plus other state-dependent data, like the current length of the list). The corresponding instruction `RESTORE_*` restores the cursor to the previously saved state. All these instructions have a single parameter for the cursor ID.

5.7 Instructions for Loops Over Cursors

A very common operation is to loop over tuples of a relation with a cursor. E.g., for the first rule `tc(X,Y) :- par(X,Y)` in the example, we must loop over all tuples in the `par_ff` relation. Such loops have a `LOOP_*` instruction at the beginning, and an `END_LOOP_*` instruction at the end. We use again specific instructions for common cases, so in the example the instructions are `LOOP_LIST_2` and `END_LOOP_LIST_2` (the relation is a list of tuples with two columns). These instructions have two parameters: An ID of the cursor (which is already linked to the relation) and a code address.

The instruction at the start of the loop opens the cursor and fetches the first tuple. If there is none, it closes the cursor, and jumps to the address, which should be the instruction just after the loop.

The instruction at the end of the loop fetches the next tuple. If this is successful, it jumps to the given address, which should be the first instruction of

the loop body. If there is no further tuple, the instruction closes the cursor, and control passes to the next instruction (after the loop). We try to do the entire loop control in one instruction to reduce the interpretation overhead.

The loop opening instructions for multimaps have an additional parameter for the input value (see the discussion about input tuples in Subsection 5.4).

As a possible alternative, we think about a stack of pointers to cursors. The loop instructions would then use implicitly always the top cursor.

5.8 Instructions for Accessing Values from Cursors

In the body of a loop controlled by a cursor, one obviously needs to get data values of the current tuple. This is done by means of `GET_*` instructions. There are again specialized instructions for common cases, e.g. `GET_LIST_2_COL_1` to access the value of the first column of a cursor over a list with two columns. This instruction has two parameters: The ID of the cursor and the number of the variable into which the value should be stored.

There is also a general `GET` instruction that can be applied to any type of cursor and has an additional parameter for the column number.

5.9 Conditions, Jumps

The body literal that is matched with the derived literal of the procedure invocation might contain constants or the same variable in different arguments. In these cases, we must check whether the rule can be applied to the derived literal. For this purpose, `IF`-instructions (conditional jumps) are needed.

The instruction `IF_VAR_IS` has three arguments: the number v of a variable, a code address a, and a data value n. If variable v contains value n, execution continues normally with the next instruction. Otherwise, control jumps to the instruction at address a.

The instruction `IF_VAR_EQ` is similar, but compares the values of two variables. The jump is done if they are distinct (i.e. the following code block is executed if they are equal).

There is also a `GOTO` instruction for an unconditional jump.

5.10 Copying Variables and Assigning Values to Variables

The head literal of a rule might contain constants. If the corresponding procedure has a parameter for the argument, we need to assign this constant value to the corresponding variable. This is done by the `ASSIGN` instruction, which has a parameter for the variable and a parameter for an integer constant. Note that it is possible to create a specialized procedure for a predicate that handles only the case with this specific constant for the selected predicate argument. We formalized this kind of partial evaluation with the notion of "fact types" in [5]: A fact type consists of a predicate and for each argument either a storage location (i.e. variable/register number in our case), or a constant.

The instruction `COPY` copies the value of one variable to another variable.

5.11 Inserting Tuples Into Relations

In order to store tuples in result relations or temporary tables for complex rules, there is an `INSERT` instruction. Again, there are different variants for special cases, e.g. `INSERT_LIST_2` inserts a tuple into a list of tuples with two columns. The relation (in this case, a list) is specified with a relation ID as first parameter. The second parameter is the number of the variable with the value for the first column, the second column value must be in the following variable. See Subsection 5.4 for a discussion of alternatives for the specification of tuples.

5.12 Output

Templates are translated to special procedures that contain output instructions, calls to other templates, and loops over result relations. A minimal set of output instructions is:

- `PRINT_TEXT(`n`)`: This prints string n from the first (system) string table.
- `PRINT_STR(`s`,`v`)`: This prints a string from domain (string table) with ID `s`, where the string number is stored in variable v.
- `PRINT_INT(`v`)`: This prints the value of variable v as an integer.

6 Example

In this section, we show the instructions of the abstract machine for the transitive closure example from Section 2:

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

The shown instructions run in our first (still experimental and not quite finished) prototype. This prototype does not have output templates yet, therefore we show a variant that simply inserts the derived tuples into a list (as we already did in Section 3). This also makes the runtime more comparable with the OpenRuleBench TCFF benchmark, which does not contain output. The example uses the following relations:

ID	Relation	Comment
0	`par_ff`	Use of `par` in first rule
1	`par_fb`	Use of `par` in second rule
2	`tc_bb`	For duplicate check
3	`tc_ff`	Result

Two cursors are used (in general, there might be several cursors over the same relation, but in this example, there is only one cursor for each of the `par`-relations):

ID	Relation	Comment
0	`par_ff`	`par(X,Y)` in first rule
1	`par_fb`	`par(X,Y)` in second rule

The program for the `tc` example consists of 17 instructions stored in 48 bytes:

```
// Procedure start: tc(X, Y) :- par(X, Y).
 0: LOOP_LIST_2(0, 17)       // Loop over par_ff, if empty goto 17
 4: GET_LIST_2_COL_1(0, 0)   // Var[0] = X from par cursor (ID 0)
 7: GET_LIST_2_COL_2(0, 1)   // Var[1] = Y value from par cursor
10: CALL(18)                 // Call tc(Var[0],Var[1])
13: END_LOOP_LIST_2(0, 4)    // If next par-tuple exists goto 4
17: HALT                     // End of "main" procedure start
// Procedure tc(Var[0],Var[1]): tc(X, Z) :- par(X, Y), tc(Y, Z).
18: DUPCHECK_2(2, 0)         // If (Var[0],Var[1]) in tc_bb: return
21: INSERT_LIST_2(3, 0)      // Store result tuple in tc_ff (ID 3)
24: SAVE_VAR(0)              // This invocation will change Var[0]
26: SAVE_MMAP_1_1_CUR(1)     // Cursor 1 will be used here
28: LOOP_MMAP_1_1(1, 43, 0)  // Loop over par(X,Y) given Y=Var[0]
33: GET_MMAP_1_1_OUT_1(1, 0) // Store X with par(X,Y) in Var[0]
36: CALL(18)                 // Recursive call: tc(Var[0],Var[1])
39: END_LOOP_MMAP_1_1(1, 33) // If next tuple exists: goto 33
43: RESTORE_MMAP_1_1_CUR(1)  // Restore used cursor
45: RESTORE_VAR(0)           // Restore changed variable
47: RETURN                   // End of procedure tc
```

7 Performance

In our previous performance comparisons of the Push Method with benchmarks from the from the OpenRuleBench suite [11], we have assumed that a factor of 3 must be attributed to the compilation to machine code. The results were still encouraging. Now the important question was of course whether an interpreted version of abstract machine code is not worse. Fortunately, the first benchmark we were able to execute with our still incomplete prototype, namely the transitive closure example, is only 1.5 times slower than the machine code version:

System	Load	Execution	Total time	Factor	Memory
Push (Switch)	0.004s	1.145s	1.147s	1.0	23.535 MB
Push (Proc.)	0.004s	1.176s	1.177s	1.0	31.392 MB
Push (Abstr.M.)	0.004s	1.714s	1.713s	1.5	31.397 MB
Seminaïve	0.004s	2.225s	2.227s	1.9	31.360 MB
XSB	0.239s	4.668s	5.103s	4.4	135.693 MB
YAP	0.240s	10.432s	10.840s	9.5	147.544 MB
DLV	(0.373s)	—	51.660s	45.0	513.748 MB
Soufflé (SQlite)	(0.113s)	—	11.240s	9.8	43.083 MB
(compiled)	(0.030s)	—	0.797s	0.7	3.867 MB

Transitive Closure Benchmark `tc(_,_)`, 50 000 `par`-facts (cyclic) [11]

Our current prototype can also execute the Join1 benchmark of [11], with the same factor 1.5 compared to the native code version (which even beats single core compiled Soufflé, probably due to a bitmap duplicate check).

8 Conclusion

Both, the translation of Datalog to C++ (and possibly other languages in future), and the translation to code of an abstract machine, have advantages of their own.

For instance, a Datalog system with an abstract machine can work standalone, and does not need a C++ compiler. Lower level optimizations can be studied better with the abstract machine than with a generation of readable C++ code and relying on optimizations of the compiler for that language. Distribution of applications as abstract machine code is simpler. If a user trusts the abstract machine, he/she does not need to trust the code for a specific application, whereas binary code is inherently more dangerous.

Program execution with an abstract machine is slower than execution of native machine code. In order to keep the overhead small, we did the following:

- Arguments to procedures are passed in variables/registers, and not on the stack. Several versions of a procedure can be generated where the arguments are contained in different variables, or are known constants (in particular, there is no fixed register for the n-th argument). In this way, copying of values is reduced, which is an important characteristic of our Push Method.
- We tried to make the granularity of the instructions large, i.e. let a single instruction of the abstract machine do a lot of things to reduce the overhead of interpretation.
- We also introduced specialized versions of instructions for common cases. (We still have to check how much runtime is really gained in this way.)
- We tried to make the code of the abstract machine compact in order to better utilize the cache.

The current state of the project is reported at the following web address:

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

References

1. Aref, M., ten Cate, B., Green, T.J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T.L., Washburn, G.: Design and implementation of the LogicBlox system. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 1371–1382. ACM (2015), `https://developer.logicblox.com/wp-content/uploads/2016/01/logicblox-sigmod15.pdf`
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). pp. 1063–1077. No. 1861 in LNCS, 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.: Language constructs for a Datalog compiler. In: Benslimane, D., Damiani, E., Grosky, W.I., Hameurlain, A., Sheth, A., Wagner, R.R. (eds.) 28th International Conference on Database and Expert Systems Applications (DEXA 2017). No. 10438 and 10439 in LNCS, Springer-Verlag (2017), http://www.informatik.uni-halle.de/~brass/push/publ/dexa17.pdf
5. Brass, S., Stephan, H.: Bottom-up evaluation of Datalog: Preliminary report. In: Schwarz, S., Voigtländer, J. (eds.) Proceedings 29th and 30th Workshops on (Constraint) Logic Programming and 24th International Workshop on Functional and (Constraint) Logic Programming (WLP'15/'16/WFLP'16). pp. 13–26. No. 234 in Electronic Proceedings in Theoretical Computer Science, Open Publishing Association (2017), https://arxiv.org/abs/1701.00623
6. Brass, S., Stephan, H.: Pipelined bottom-up evaluation of Datalog: The Push method. In: Petrenko, A.K., Voronkov, A. (eds.) 11th A.P. Ershov Informatics Conference (PSI'17) (2017), http://www.informatik.uni-halle.de/~brass/push/publ/psi17.pdf
7. Calì, A., Gottlob, G., Lukasiewicz, T.: A general Datalog-based framework for tractable query answering over ontologies. In: Proc. of the 28th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS'09). pp. 77–86. ACM (2009)
8. 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)
9. Graefe, G.: Vulcano: An extensible and parallel query evaluation system. Tech. Rep. 192, Oregon Graduate Center / OHSU Digital Commons (1989), http://digitalcommons.ohsu.edu/csetech/192
10. Hellerstein, J.M.: The declarative imperative. SIGMOD Record 39(1), 5–19 (2010), http://db.cs.berkeley.edu/papers/sigrec10-declimperative.pdf
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. 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
13. Polleres, A.: How (well) do Datalog, SPARQL and RIF interplay? In: Barceló, P., Pichler, R. (eds.) Datalog in Academica and Industry, 2nd Int. Workshop, Datalog 2.0. pp. 27–30. No. 7494 in LNCS, Springer (2012)
14. 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)
15. Schütz, H.: Tupelweise Bottom-up-Auswertung von Logikprogrammen (Tuple-wise bottom-up evaluation of logic programs). Ph.D. thesis, TU München (1993)
16. Shkapsky, A., Yang, M., Interlandi, M., Chiu, H., Condie, T., Zaniolo, C.: Big data analytics with Datalog queries on Spark. In: Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16). pp. 1135–1149. ACM (2016), http://yellowstone.cs.ucla.edu/~yang/paper/sigmod2016-p958.pdf
17. 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