Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 44–53
http://www.floc-conference.org/ICLP-home.html

IMPLEMENTATION ALTERNATIVES FOR BOTTOM-UP EVALUATION

STEFAN BRASS [1]

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

Abstract. Bottom-up evaluation is a central part of query evaluation / program execution in deductive databases. It is used after a source code optimization like magic sets or SLDmagic that ensures that only facts relevant for the query can be derived. Then bottom-up evaluation simply performs the iteration of the standard T_P-operator to compute the minimal model. However, there are different ways to implement bottom-up evaluation efficiently. Since this is most critical for the performance of a deductive database system, and since performance is critical for the acceptance of deductive database technology, this question deserves a thorough analysis. In this paper we start this work by discussing several different implementation alternatives. Especially, we propose a new implementation of bottom-up evaluation called "Push-Method".

1. Introduction

Deductive databases [Min88, Ull90, Fre91, Ram94, Vag94, Fri95, Ram95] have not yet been very successful in practice (at least in terms of market share). However, their basic idea is practically very important: Deductive databases aim at an integrated system of database and programming language that is based on the declarative paradigm which was so successful in database languages. Currently, database programming is typically done in languages like PHP or Java. The programs construct SQL statements, send them to the database server, fetch the results, and process them. The interface is not very smooth, and although the situation can be improved with specific database languages like PL/SQL and server-side procedures / user-defined functions within the DBMS, the language paradigms remain different. Object-oriented databases were one approach to develop an integrated system based on a single paradigm, but there the declarativity of the database query part was sacrificed, and they did not get a significant market share, either. Nevertheless, there is an obvious demand for integrated database/programming systems, and this demand has even grown because of object-relational features that need programming inside the database server, and because of web and XML applications.

One of the reasons why deductive databases were not yet successful is the non-satisfying performance of many prototypes. This is also related to the impression that most deductive database prototypes have not really tried to be useful also as a programming platform — they were concentrated only on recursive query evaluation.

1998 ACM Subject Classification: I.2.3 Logic Programming, H.2.4 Query processing.
Key words and phrases: deductive databases, bottom-up evaluation, implementation.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.44

Of course, recursive query evaluation is an important task, because many applications use tree-structured or graph-structured data. There has been a lot of progress over the years in this area [Ban86b]. A large part of this work was about source-level optimizations, like the well-known magic-set method [Ban86a, Bee91] and its many optimizations, including the SLDMagic-method of the author [Bra00].

However, this all depends on an efficient implementation of bottom-up evaluation. If one wants to build a new deductive database system which a real chance for acceptance in practice, one needs to clarify first how bottom-up evaluation should be done. This is not obvious, and several alternatives will be discussed in this paper.

Note that programming in deductive databases is not the same as programming in Pure Prolog. In deductive databases one thinks in the direction of the arrow, because they are based on bottom-up evaluation. For instance left recursion is very natural in this way, whereas in Prolog it must be avoided.

Top-down systems with tabling (like the XSB system [Sag94]) have a middle position between Prolog and deductive databases. Currently they have better performance than systems based on bottom-up evaluation. Our belief is that the bottom-up approach has still room for improvement in order to deliver competitive performance. In [Bra00] we proposed a source-level transformation that is for tail-recursions asymptotically faster than the standard magic set method (and also than the tabling method underlying the XSB system). It is also interesting because it unifies many improvements which were proposed for the magic set method over time.

After a source-level transformation like SLDmagic, which solves the problem of goal-direction, one needs an efficient implementation of bottom-up evaluation. The research reported in this paper is a step in this direction.

The approach we want to follow is to translate Datalog into C++, which can then be compiled to machine code. We did first performance tests with the methods described in this paper, but because of space restrictions, we must refer to

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

for the results.

2. Basic Framework

There are three types of predicates:

- EDB predicates ("extensional database"), the given database relations,
- IDB predicates ("intensional database"), which are defined by means of rules,
- built-in predicates like $<$, which usually have an infinite extension, and are defined by means of program code inside the system.

The purpose of bottom-up evaluation is to compute the extensions of the IDB relations. Actually, only one of them is the "answer predicate", the extension of which must be printed, or otherwise made available to the user.

Bottom-up evaluation works by applying the rules from right to left, so basically it computes the minimal model by iterating the standard T_P-operator. Of course, an important goal is to apply every applicable rule instance only once via rule-ordering and managing deltas for recursive rules ("seminaive evaluation"). However, slight exceptions are possible, because there is a tradeoff with the work needed for storing and accessing again intermediate facts.

Because of the infinite extension, built-in predicates can only be called when certain arguments are bound (i.e. input arguments, known values). In contrast, a free argument position permits a variable (output argument). The restrictions for the predicates are described by binding patterns (modes, adornments), e.g. $<$ can be called for the binding pattern `bb` only (every letter in a binding pattern corresponds to an argument position, `b` means bound, `f` means free).

A basic interface for relations is 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 normal predicate p, there is a class p`_cursor` with the following methods:

- `void open()`: Open a scan over the relation (cursor is placed 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 (with data type T).
- `close()`: Close the cursor.

For the push method, we will also need

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

(A merge join sometimes needs the possibility to return to a saved position in a scan, too.) 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 `ff`...`f`) is possible, but also scans only over tuples with a given value 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 which permits especially 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)`.

Actually, some access structures can efficiently evaluate small conjunctions with parameters, e.g. a B-tree over the first argument of `p` would also support a query of the form `p(X,Y,Z)` $\wedge$ `X` $\geq c_1 \wedge$ `X` $< c_2$, where c_1 and c_2 are integer constants or bound variables. This is not in the focus of the current paper, but a realistic system must be able to make use of such possibilities.

In addition, we need a possibility to create new tuples for predicates defined by rules (IDB predicates). We assume that for each predicate p there is a class p with a class method `insert` that creates a new tuple in p. Of course, it is possible that the objects of class p correspond to individual tuples, but since we only use the cursor interface, this is only one possible implementation.

For seminaive evaluation of recursive programs, additional cursor types are needed (e.g. `p_cursor_diff` runs only over tuples generated in the previous step of the fixpoint iteration), and a method to switch to the next iteration step (class method `next_iter` of `p`).

3. Materializing Derived Predicates

The first, most basic method for implementing bottom-up evaluation is to explicitly create a stored relation for every IDB-prediate. As an example, let us consider

$$\texttt{p(X, Z, 2)} \leftarrow \texttt{q(X, Y)} \wedge \texttt{r(Y, 5, Z)}.$$

```
q_cursor q1;
q1.open();
while(q1.fetch()) {
    int X = q1.col_1();
    int Y = q1.col_2();
    r_cursor_bff r1; // if there is an index on the first argument
                     // (and none for the binding pattern bbf)
    r1.open(Y);
    while(r1.fetch()) {
        if(r1.col_2() == 5) {
            int Z = r1.col_3();
            p::insert(X, Z, 2);
        }
    }
}
```

Figure 1: Materializing an IDB-Predicate: `p(X, Z, 2)` $\leftarrow$ `q(X, Y)` $\wedge$ `r(Y, 5, Z)`.

We assume that all columns in the examples have type `int`.

Of course, one option is to use a standard relational database, create a table for every IDB predicate, and send SQL statements to the database to execute the rules. But using a separate system for the management of facts causes performance penalties. Furthermore, for the seminaive evaluation of recursive rules, there is no good and efficient way to manage the deltas with SQL (the set of tuples newly derived in an iteration). Another interesting problem is that a good sideways information passing (SIP) strategy in the magic set method or selection function in the SLDMagic method needs already knowledge about existing indexes and relation sizes. Therefore it is not a good idea to do query optimization in two completely separate systems: The chosen SIP strategy/selection function more or less prescribes the evaluation of the resulting rules. For instance, if one wants to use a merge join, less "sideways information passing" is possible than with a nested loop/index join.

Therefore, one would do basic rule evaluation in the deductive database system itself, although it might be possible to use parts of a standard relational system (e.g. the storage manager). E.g. with a nested loop/index join, the implementation of the above rule would look as shown in Figure 1.

Of course, the materialization method causes a lot of copying. E.g., disjunctions must be expressed in standard Datalog with a derived predicate:

$$\begin{aligned} \texttt{p(X, Y)} &\leftarrow \texttt{q(X, Y).} \\ \texttt{p(X, Y)} &\leftarrow \texttt{r(X, Y).} \end{aligned}$$

The materialization method would copy all `q`- and `r`-facts. This is especially expensive if the data values `X` and `Y` are large (e.g. longer strings). The problem can be reduced by working only with pointers to the real data values (but that might not make optimal use of the memory cache in the CPU, because values are more scattered around in memory).

4. Pull-Method

Of course, explicitly materializing every intermediate predicate needs a lot of memory. Therefore, it is a standard technique in databases to compute tuples only on demand, or

```
class p_cursor {
public:
    void open()
    {
        q1.open();
        q1_more = q1.fetch();
        if(q1_more)
            r1.open(q1.col_2()); // Assuming again index on first argument
    }
    bool fetch()
    {
        while(q1_more) {
            while(r1.fetch()) {
                if(r1.col_2() == 5)
                    return true;
            }
            r1.close();
            q1_more = q1.fetch();
            if(q1_more)
                r1.open(q1.col_2());
        }
        return false;
    }
    int col_1() { return q1.col_1(); }
    int col_2() { return r1.col_3(); }
    int col_3() { return 2; }
private:
    q_cursor q1;
    r_cursor_bff r1;
    bool q1_more;
};
```

Figure 2: Pull-Method (Lazy Evaluation): `p(X, Z, 2)` $\leftarrow$ `q(X, Y)` $\wedge$ `r(Y, 5, Z)`.

actually not even compute the entire tuple, but permit access to its columns (in this way, possibly large data values do not have to be copied). In order to get such "lazy" evaluation, one only needs to support the cursor interface for each predicate. Let us consider again

`p(X, Z, 2)` $\leftarrow$ `q(X, Y)` $\wedge$ `r(Y, 5, Z)`.

If `p` is nonrecursive, and this is the only rule, and no duplicate elimination is needed, the code would look as shown in Figure 2.

If duplicate elimination is needed, storing all tuples is necessary (unless the tuples for the body literals are generated in a fitting sort order). One can then apply the materialization method, or extend the pull-method by building a hash table of all previously returned tuples, and adding a check that the derived tuple is new.

An important disadvantage of the pull-method is that it causes recomputation if multiple scans over a predicate are performed. This does not only happen when there are several body literals with the same predicate p, but also when a single p-literal appears in the inner loop of a nested loop join. However, recomputation is not necessarily something evil that must be avoided at any price. If the recomputation is not expensive, as in the example with the predicate describing a disjunction, it is a possible alternative.

Recursion with the pure pull-method is not possible: If one tries to implement recursion with the recursive opening of cursors, this leads to an infinite recursion even for acyclic relations since no bindings are passed to the recursive call: Each call does the same work again. Of course, it is possible to integrate standard seminaive evaluation, but this simply means to use the materialization method at least once in each recursive cycle.

5. Push-Method

It is also possible to apply the rules strictly from right to left, and move generated facts immediately to the place where they are needed. In contrast to the materialization method, a rule is not applied to produce all consequences in the current state, but only a single fact is derived each time. This reduces the need for intermediate storage and copying, which was also the main motivation for the pull-method. But here the producer of facts is in control, not the consumer as in the pull-method.

Of course, usually several facts can be derived with a rule. Therefore, once a fact is derived, one must store the current state of rule application for later backtracking. Then control jumps to a rule where this fact matches a body literal. There can be several rules that might use the produced fact, in which case again a backtrack point is generated.

This method basically works only with rules that have at most one body literal with IDB-predicate, because then matching facts for the other (EDB) body literals are available when a fact for the IDB body literal arrives. The SLDMagic method [Bra00] produces such rules as output of the program transformation, therefore this case is practically interesting. Furthermore, when there are several body literals with IDB-predicates, it is often possible to use the materialization or pull method for the predicates of all but one body literal.

The push method is applicable to linear recursion, and that is in fact one of its strengths (it can be very efficient in this case).

Let us explain how it works. First one creates a variable for every column of an IDB predicate. Consider again the example rule:

```
p(X, Z, 2) ← q(X, Y) ∧ r(Y, 5, Z).
```

If q is an IDB predicate, r is an EDB-predicate, and all arguments have type int, we get:

```
int q_1, q_2;
int p_1, p_2, p_3;
```

In addition, there is a code piece for every body literal with IDB-predicate. Control jumps to this code piece when a new fact for this predicate was derived. The argument values of the fact are stored in the above variables. The purpose of the code is to check whether new facts can be derived with this rule with the given instantiation of the IDB body literal, and if yes, to store the arguments of the derived fact in the corresponding variables and to jump to every place where the newly derived fact is used. For the example rule, the code looks as shown in Figure 3 (there are in fact many optimization possibilities, which we cannot discuss here for space reasons). Rules with only EDB-predicates in the body act as starting points. For instance, consider

```
q(X, Y) ← s(X, Y) ∧ Y ≥ 0.
```

```
q:  r_cursor_bff r1;
    r1.open(q_2); // Assuming index on first arg
    while(r1.fetch()) {
        if(r1.col_2() == 5) {
            p_1 = q_1;
            p_2 = r1.col_3();
            p_3 = 2;
            if(r1.fetch()) {
                push_int(q_1);
                push_int(q_2);
                r1.push();
                push_cont(CONT_q);
            }
            goto p;
        }
    }
    goto backtrack; // if this is the last place where q is used
cont_q:
    r1.pop();
    q_2 = pop_int();
    q_1 = pop_int();
    do {
        if(r1.col_2() == 5) {
            p_1 = q_1;
            p_2 = r1.col_3();
            p_3 = 2;
            if(r1.fetch()) {
                push_int(q_1);
                push_int(q_2);
                r1.push();
                push_cont(CONT_q);
            }
            goto p;
        }
    } while(r1.fetch());
    goto backtrack; // if this is the last place where q is used
backtrack:
    if(stack_empty()) return false;
    switch(pop_task()) {
    case CONT_q; goto cont_q;
    ...
    }
```

Figure 3: Push-Method: `p(X, Z, 2)` $\leftarrow$ `q(X, Y)` $\wedge$ `r(Y, 5, Z).`

```
init:
    s_cursor s1;
    s1.open();
    while(s1.fetch()) {
        if(s1.col_2() >= 0) {
            q_1 = s1.col_1();
            q_2 = s1.col_2();
            if(s1.fetch()) {
                s1.push();
                push_cont(CONT_init);
            }
            goto q;
        }
    }
    return false; // if this is the last/only initialization rule
cont_init:
    s1.pop();
    do {
        if(s1.col_2() >= 0) {
            q_1 = s1.col_1();
            q_2 = s1.col_2();
            if(s1.fetch()) {
                s1.push();
                push_cont(CONT_init);
            }
            goto q;
        }
    } while(s1.fetch());
    return false; // if this is the last/only initialization rule
```

Figure 4: Push-Method: Initialization with $\texttt{q(X, Y)} \leftarrow \texttt{s(X, Y)} \wedge \texttt{Y} \geq \texttt{0}$.

Then for every `s`-fact, we would fill the variables `q_1` and `q_2` and jump to the place where `q`-facts are used (label `q:`). Again, this loop is implemented with backtracking.
Note that the push method can be made to fit into the cursor interface: If for instance `p` needs to be queried with a cursor, the above code is inside the `fetch` method. The code jumps to label `p:` when a new `p`-fact is derived, the `fetch`-method returns `true` to the caller. Each call to the `fetch`-method starts at the label `backtrack`. In the `open`-method the backtrack-stack is initialized with a value that causes a jump to the initialization (`init`).

6. Pull-Method with Passing of Bindings

In deductive databases, one normally uses first a program transformation like magic sets, which is responsible for passing bindings from the caller to the callee, so that only relevant facts are computed when the transformed rules are evaluated strictly bottom-up (i.e. the entire minimal model of the transformed program is computed). The “magic predicates” contain values for the input/bound arguments of a predicate. Calls to these predicates are added as conditions to the rule body, so that the rule can only "fire" when the result is needed.

With the magic set transformation, all calls to a predicate are put together in one set. This is good, if there are several calls to a complex predicate with the same input values: Then the answer is computed only once. But it is also bad, because the results for different calls to a predicate are all in one set, from which one has to select the result matching the current input arguments.

The Pull Method works already not completely bottom-up, but is controlled from the caller (top-down) who requests the next tuple. Therefore it is very natural that the caller passes all information he has about the required tuples. This would replace the magic set transformation, but it is not exactly the same, because now there is not one big magic set for a predicate (and a binding pattern), but for each call there are given values for certain arguments. This has positive and negative effects: When the call returns, one gets a tuple with the required values in the given positions, so no further check/selection is necessary. This is especially important since the pull method repeats computations, so non-matching tuples would simply be wasted. On the negative side, if the same call appears more than once, the result is computed repeatedly.

Passing bindings to called predicates fits nicely into the cursor interface, because for EDB-predicates with special access structures, it is already possible. In this way, this is also possible for IDB predicates.

For the materialization method and the push method, magic sets (or one of its variants) works well. However, if one wants to combine the different methods in one program (which is advisable, since each has its strengths and weaknesses), it would be possible to treat the magic set specially and to initialize it each time with only a single tuple.

7. Related Work

There are still more variants of bottom-up evaluation proposed in the literature, which we intend to include in our comparison in a future version of this article:

- In [Liu03], an extreme form of materialization is proposed: Not only facts about the derived predicates are explicitly stored, but also intermediate results during rule evaluation. This is combined with a clever selection of data structures.
- In [Wun95], a method similar to the push method is used, but with a materialization of the derived predicates (our push method avoids this). Similar methods are also used to propagate changes from base relations to materialized views.
- In [Cod99], bottom-up evaluation is implemented with a meta-interpreter running in Prolog. An important idea is also how to handle rules where the body of one rule is prefix of a body of another rule (as generated by the magic set method).

8. Conclusions

Our long-term goal is to develop a deductive database system that supports stepwise migration from classical SQL. Of course, the system will use our SLDmagic method [Bra00] for goal-direction, but it also needs an efficient bottom-up engine to run the transformed program. In this paper, we investigated several implementation variants based on a translation to C++. In summary, the methods differ in what they materialize (store in memory for a longer time), what they recompute, and the order in which applicable rule instances are considered (and also the duration: the pull method has a rule instance "open" for a longer time). It turned out that the optimal method depends on the input program and also on the compiler and the hardware, as well as keys and access structures for the relations. But the push method performed constantly quite well. It has the restriction that it can work directly only with rules having only a single IDB-literal in the body, but it can be combined with other methods, or applied in several steps with different components of the program. Also, the SLDmagic method produces rules with only one IDB-predicate in the body.

The source code and performance results for the tests are available at

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

Results of future tests will also be posted on this page.

References

[Ban86a] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. 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.

[Ban86b] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing. In Carlo Zaniolo (ed.), *Proceedings of the 1986 ACM SIGMOD international conference on Management of Data (SIGMOD'86)*, pp. 16–52. 1986.

[Bee91] Catril Beeri and Raghu Ramakrishnan. On the power of magic. *The Journal of Logic Programming*, 10:255–299, 1991.

[Bra00] Stefan Brass. SLDMagic — the real magic (with applications to web queries). In W. Lloyd et al. (eds.), *First International Conference on Computational Logic (CL'2000/DOOD'2000)*, no. 1861 in LNCS, pp. 1063–1077. Springer, Heidelberg, Berlin, 2000.

[Cod99] Michael Codish. Efficient goal directed bottom-up evaluation of logic programs. *Journal of Logic Programming*, 38(3):355–370, 1999.

[Fre91] Burkhard Freitag, Heribert Schütz, and Günter Specht. LOLA — A logic language for deductive databases and its implementation. In *Proc. of 2nd Int. Symp. on Database Systems for Advanced Applications (DASFAA'91)*, pp. 216–225. World Scientific, 1991.

[Fri95] Oris Friesen, Gilles Gauthier-Villars, Alexandre Lefebvre, and Laurent Vieille. Applications of deductive object-oriented databases using DEL. In Raghu Ramakrishnan (ed.), *Applications of Logic Databases*, pp. 1–22. Kluwer, 1995.

[Liu03] Yanhong A. Liu and Scott D. Stoller. From datalog rules to efficient programs with time and space guarantees. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pp. 172–183. ACM, 2003.

[Min88] Jack Minker. Perspectives in deductive databases. *The Journal of Logic Programming*, 5:33–60, 1988.

[Ram94] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. The CORAL deductive system. *The VLDB Journal*, 3:161–210, 1994.

[Ram95] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of deductive database systems. *The Journal of Logic Programming*, 23:125–149, 1995.

[Sag94] Konstantinos Sagonas, Terrance Swift, and David S. Warren. XSB as an efficient deductive database engine. In Richard T. Snodgrass and Marianne Winslett (eds.), *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'94)*, pp. 442–453. 1994.

[Ull90] Jeffery D. Ullman and Carlo Zaniolo. Deductive databases: Achievements and future directions. *ACM SIGMOD Record*, 19:75–82, 1990.

[Vag94] Jayen Vaghani, Kotagiri Ramamohanarao, David Kemp, Zoltan Somogyi, Peter J. Stuckey, Tim S. Leask, and James Harland. The Aditi deductive database system. *The VLDB Journal*, 3:245–288, 1994.

[Wun95] Jens E. Wunderwald. Memoing evaluation by source-to-source transformation. In Maurizio Proietti (ed.), *Logic Programming Synthesis and Transformation, 5th International Workshop (LOPSTR'95)*, no. 1048 in LNCS, pp. 17–32. Springer-Verlag, 1995.