Constant Propagation Versus Join Reordering in Datalog

Mariano P. Consens[1], Alberto O. Mendelzon[2], Dimitra Vista[2], and Peter T. Wood[3*]

[1] Dep. of Computer Science, Univ. of Waterloo, Waterloo, Canada N2L 3G1
[2] Computer Systems Research Institute, Univ. of Toronto, Toronto, Canada M5S 1A1
[3] Dep. of Computer Science, Univ. of Cape Town, Rondebosch 7700, South Africa

Abstract. Constant propagation and join reordering are two standard optimization techniques used for Datalog programs. These techniques have typically been studied independently of one another. However, in order to achieve constant propagation it is sometimes necessary to impose a certain join ordering on a given program. In the worst case this ordering may result in efficiencies which overcome the benefit of constant propagation. Thus the goal of constant propagation should not necessarily be pursued to the exclusion of all else. We study this problem in the context of GraphLog, a visual language whose queries are graphical notations which are translated into Datalog. We study two translation schemes from GraphLog to Datalog, one which always achieves constant propagation and another which does not. We show that each translation can significantly outperform the other. We demonstrate this by both measuring execution times using actual application data, and by providing analytical formulae to explain the trade-off between constant propagation and join reordering.

1 Introduction

Two topics of research in the area of query evaluation for recursive Datalog rules have been constant propagation and join ordering. In the former, given a recursive Datalog program with a constant in the query, the goal is to propagate the constant from the query down to the base (EDB) relations in order to reduce the number of tuples computed by the program. In the latter, the goal is to order the joins in the bodies of rules in such a way as to minimize the size of intermediate results.

Up to now, the topics of constant propagation and join ordering have been pursued independently. In addition, ordering of joins has been restricted to consideration of one rule at a time. Of course, if a program is nonrecursive, it can be translated to relational algebra (say), after which conventional approaches to join ordering can be applied. In the case of recursive programs, however, we know

* Most of this work was done while the author was on sabbatical at the University of Toronto.

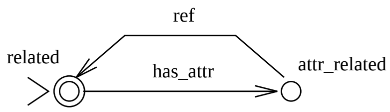

Fig. 1. The automaton M corresponding to `(has_attr · ref)`*

of no work on global rewriting of a program in order to improve the ordering of joins.

In this paper, we investigate the interaction of constant propagation and join ordering in recursive Datalog programs. We restrict our attention to a class of queries which correspond to path computations on graph databases as specified by regular expressions. These are the queries which can be computed by the Hy$^+$ database visualization system developed at the University of Toronto [4]. We show that, if we insist on propagating constants to EDB relations, then certain join orderings are precluded. One of these orderings may be more efficient than that imposed by constant propagation. We provide cost formulas and heuristics to guide a choice among the alternative strategies.

Example 1. Consider a graph G representing schema information for an object database. The nodes of G represent class names and attribute names. There are two types of edges: `has_attr` relates a class name with an attribute name which is a property of the class, and `ref` relates an attribute name to a class name when the attribute is a reference (relationship) attribute. These edges can be represented by two binary relations `has_attr(class, attr)` and `ref(attr, class)`. We say that classes c_1 and c_2 are *related* if c_1 has an attribute which references c_2 either directly or indirectly. We can represent this relationship using the regular expression $R =$ `(has_attr · ref)`* (where $\cdot$ represents concatenation and * represents reflexive transitive closure).

If we construct a Datalog program based on the parse tree of R, we obtain the program P_1:

```
related(ClassX, ClassX).
related(ClassX, ClassY) ← related(ClassX, ClassZ),
                          directly_related(ClassZ, ClassY).
directly_related(ClassX, ClassY) ← has_attr(ClassX, Attr),
                                   ref(Attr, ClassY).
?- related(ClassX, ClassY).
```

Here the concatenation in R is represented by the `directly_related` predicate. The two rules for `related` compute the closure.

On the other hand, we can first construct a nondeterministic finite automaton (NFA) M for recognizing strings in the language denoted by R. The NFA M has two states: `related`, which is both the initial and final state, and `attr_related`. Figure 1 shows M. Each state of M corresponds to a derived predicate: `related` is the relation of interest, and `attr_related` is an intermediate relation. The intuition behind the automaton representation is the following: the initial state, **related**, at start, knows that any class is related to itself. Then, alternatively examining a **has_attr** and **ref** fact, more **related** facts are produced, until no more facts can be thus derived. We use the NFA M to produce P_2.

```
related(ClassX, ClassX).
attr_related(ClassX, Attr) ← related(ClassX, ClassY),
                             has_attr(ClassY, Attr).
related(ClassX, ClassY) ← attr_related(ClassX, Attr),
                          ref(Attr, ClassY).
?- related(ClassX, ClassY).
```

The bottom-up evaluation of the rules of P_2 mimics the breadth-first traversal of graph G starting from all class nodes, and alternatively following **has_attr** and **ref** edges. The rules are obtained from M as follows. The first rule comes from the fact that **related** is the initial state of M. The next two rules are derived from transitions from **related** to **attr_related** (labelled **has_attr**) and from **attr_related** to **related** (labelled **ref**), respectively. The query comes from the fact that **related** is the final state of M.

The above two translations from regular expressions to Datalog are implemented in the GraphLog query language [5, 7] provided by the Hy$^+$ system. We call the former the *closure* translation, because it computes closures of EDB or IDB relations, and the latter the *nfa* translation. Datalog programs are passed by Hy$^+$ to the CORAL deductive database system for evaluation [12]. Which translation should be adopted is one of the issues explored in this paper.

Consider the bottom-up evaluation of P_1. If we abbreviate **has_attr** by h and **ref** by r, then the sequence of joins performed during the evaluation is

$$
\begin{gathered}
(h \bowtie r) \\
(h \bowtie r) \bowtie (h \bowtie r) \\
((h \bowtie r) \bowtie (h \bowtie r)) \bowtie (h \bowtie r) \\
\vdots
\end{gathered}
$$

Now consider the bottom-up evaluation of P_2. The sequence of joins is:

$$
\begin{gathered}
(h \bowtie r) \\
((h \bowtie r) \bowtie h) \\
(((h \bowtie r) \bowtie h) \bowtie r) \\
\vdots
\end{gathered}
$$

Note that both translations produce left-linear programs. Right-linear versions of each would produce other join orderings.

Now assume that we are interested only in those classes related to the particular class called **object**. One crucial difference between the two translations is that the *nfa* translation admits the propagation of constants to base relations using the technique of factoring [11], while the *closure* translation does not. For program P_1 from the *closure* translation, we simply substitute `object` for `ClassX` in the query clause and apply the magic sets transformation [2].

The *nfa* translation modifies P_2 as follows:

```
class_related_to_object(object).
attr_related_to_object(Attr) ← class_related_to_object(Class),
                                has_attr(Class, Attr).
class_related_to_object(Class) ← attr_related_to_object(Attr),
                                  ref(Attr, Class).
related(object, Class) ← class_related_to_object(Class).
?- related(object, Class).
```

The above program corresponds to traversing graph G starting at the node representing the class `object`. This ability to perform constant propagation might lead one to conclude that the *nfa* translation would always be preferable to the *closure* translation when constants appear in a GraphLog query. This is not the case, however. The reason is that the join ordering imposed by the *nfa* translation may be particularly expensive. The different ordering adopted by the *closure* translation may overcome the initial loss in performance due to a constant not being propagated, outperforming the *nfa* translation after one or two iterations of the fixed point computation.

The remainder of the paper is organized as follows. In the next section, we use regular expressions, which are a fundamental component of GraphLog, to introduce the two translation schemes: the *closure* translation in Section 2.1, and the *nfa* translation in Section 2.2. An example from the application of performance tuning during software development is presented in Section 3, which also reports on some performance results which suggest that the *nfa* scheme is always better. This conclusion is refuted in Section 4 where we show through further empirical results as well as analytical formulae that each translation can outperform the other. Concluding remarks are found in Section 5.

2 Translating Regular Queries to Datalog

In this section, we describe the regular queries which form a fundamental part of the GraphLog query language as well as their translation to Datalog. Two translations are described: the first is the *closure* translation, which defines the semantics of GraphLog; the second is the *nfa* translation. Section 3 and 4 presents both experimental and analytical comparisons between the two methods.

Since for the purposes of this paper we do not require a number of features of GraphLog, we simplify the following description accordingly. For a complete description of the language, the reader is referred to [3].

In GraphLog, a *term* is either a constant or a variable (once again, more general terms are in fact allowed). An *edge label* is an expression generated by the following grammar, where p is a predicate:

$$E \rightarrow E|E;\ E \cdot E;\ -E;\ (E);\ E^+;\ E^*;\ p;\ \neg p$$

This is essentially a grammar for regular expressions, with "|" representing alternation, "·" concatenation, "−" inversion (a traversal from head to tail, rather than tail to head), "¬" negation (the absence of a path), "+" transitive closure, and "∗" reflexive transitive closure.

Database instances are directed graphs whose nodes are labelled with ground terms and whose edges are labelled with predicates. Database instances of the object-oriented or relational model can easily be visualized as graphs. For example, an edge labelled p from a node labelled T_1 to a node labelled T_2, corresponds to tuple (T_1, T_2) of relation p in the relational model.

Queries are directed graphs whose nodes are labelled by terms and each edge is labelled by an edge label. The query graph represents a pattern; the query evaluator searches the database graph for all occurrences of that pattern.

A query must have one *distinguished* edge, labelled by a positive literal. The meaning of a query is to define the predicate in this distinguished literal in terms of the rest of the pattern. The semantics of queries is given by a translation to stratified Datalog. Each query graph G translates to a rule with the label of the distinguished edge in the head, and as many literals in the body as there are non-distinguished edges in G. If an edge is labelled with a regular expression, additional rules to define its predicate are necessary.

2.1 The Closure Translation

Suppose between nodes labelled X and Y, there is an edge labelled p, where X, Y are terms and p is a predicate. We will denote this by $p(X, Y)$. In fact, the meaning of such an edge is given by the Datalog fact $p(X, Y)$.

Given a GraphLog query Q with non-distinguished edges $p_1, \ldots, p_n$, where p_i is from node labelled X_{i-1} to node labelled X_i, and with distinguished edge p from node labelled X_{j_1} to node labelled X_{j_2}, the translation for Q contains the following rule that defines the predicate p:

$$p(X_{j_1}, X_{j_2}) \leftarrow l_1(X_0, X_1), l_2(X_1, X_2), \ldots, l_n(X_{n-1}, X_n).$$

where $l_i(X_{i-1}, X_i)$ is $p_i(X_{i-1}, X_i)$, if the edge is positive, $\neg p_i(X_{i-1}, X_i)$, if the edge is negated, or it is defined recursively on the structure of the label according to the following algorithm (assuming that the edge is from a node labelled X to one labelled Y):

- *Inversion*: Label $-e$ corresponds to the literal $e(Y, X)$.
- *Concatenation*: Label $e_1 \cdot e_2$ corresponds to the rule

$$conc(X, Y) \leftarrow e_1(X, Z), e_2(Z, Y).$$

- *Alternation*: Label $e_1 | e_2$ corresponds to the rules

$$alter(X, Y) \leftarrow e_1(X, Y).$$
$$alter(X, Y) \leftarrow e_2(X, Y).$$

– *Closure*: Label e^+ corresponds to the rules

$$tc\_e(X,Y) \leftarrow e(X,Y).$$
$$tc\_e(X,Y) \leftarrow tc\_e(X,Z), e(Z,Y).$$

– *Kleene closure*: Label e^* corresponds to the rules

$$kleene\_e(X,X).$$
$$kleene\_e(X,Y) \leftarrow kleene\_e(X,Z), e(Z,Y).$$

Note that the rules for the two forms of closure above are left-linear. The choice between left- and right-linear rules is another which could be made by the query language optimizer; for simplicity and consistency, we will use left-linear rules throughout. The reader is referred back to program P_1 of Example 1 for an example of the output of the *closure* translation.

GraphLog users are allowed to label nodes with (single) constants. The *closure* translation of such a query to Datalog will result in a program containing a single constant in the query goal, whereupon the rewriting techniques such as magic sets [2], factoring [11] and context rewriting [10] become applicable. We assume that readers are familiar with each of these.

CORAL provides an annotation for controlling evaluation called **`@factoring`**. This applies the context rewriting transformation of [10]. We will use the term *factoring* in the same way as in [11] to mean the reduction in the number of arguments in recursive predicates. Context rewriting does not do this, so to avoid confusion we will refer to the CORAL annotation as *context rewriting*.

There are at least two limitations of context rewriting. Firstly, it is not applicable to all programs generated from GraphLog queries. Secondly, context rewriting may be an order of magnitude less efficient than a true factoring transformation. The original factoring transformation applies to a very restricted class of programs [11]. However, it was shown in [13, 14] that the factoring transformation can be applied to any program generated by an alternative translation from (single-edge) GraphLog queries to Datalog. A simplified version of this translation is presented below; for a complete version, the reader is referred to [13].

2.2 The NFA Translation

Rather than a translation based on the structure of the regular expression R in the GraphLog query, in this subsection we describe a translation based on a nondeterministic finite automaton (NFA) constructed from R. We assume the reader is familiar with the definition of a nondeterministic finite automaton [9]. Given NFA M, the language $L(M)$ *accepted* by M is the set of all strings accepted by M.

We assume that we are given a single edge from a GraphLog query. The edge is labelled with an arbitrary regular expression R. The first step in the translation is to construct an NFA M from R. The transitions in M are labelled with predicates from R. The second step is to generate a Datalog program P

from M. This uses the standard technique of generating a regular grammar from an NFA, and yields a regular chain program.

The details of the translation from regular expression R to program P are as follows:

1. Construct an NFA M which accepts the language $L(R)$, as described, for example, in [9].
2. For each transition t in M from p to q labelled with e, generate a Datalog rule as follows:
$$q(X,Y) \leftarrow p(X,Z), e(Z,Y).$$
If, instead, t is labelled with $-e$, generate:
$$q(X,Y) \leftarrow p(X,Z), e(Y,Z).$$
3. For the initial state s_0 of M, generate the fact $s_0(X,X)$., and for each final state f of M, generate a rule
$$s(X,Y) \leftarrow f(X,Y).$$
where s is the distinguished (query) predicate.

Note, once again, that left-linear rules are produced by the above translation.

When one of the nodes v in the GraphLog query is labelled with a constant, we modify the translation as follows. If v is the head node, we first reverse the automaton M, also performing the inversion of each term labelling a transition in M. Now we modify steps (2) and (3) above by deleting all occurrences of X from the rules generated in (2), and replacing $s_0(X,X)$. by $s_0(c)$. in (3), where c is the node constant in the GraphLog query. Also in (3) for each final state we generate $s(c,X) : -f(Y)$. We have thus successfully propagated the constant to the EDB predicates and reduced the number of arguments in all the IDB predicates.

We refer back to the program P_2 of Example 1. When comparing P_2 to P_1 (the output of the *closure* translation), there are two differences to note. Firstly, the recursion in the *nfa* translation is in terms of mutually recursive rules, while that in the *closure* translation is in terms of the closure of an IDB predicate. This results in different join orderings being chosen by the two translations. Secondly, when a constant is present, the *nfa* translation factors the recursive predicates and propagates `object` into the program.

These two differences, join ordering and constant propagation, provide the principal explanations for the performance results presented in the following two sections.

3 Application-Based Performance Comparisons

In this section we describe the differences in performance stemming from the two different translations in an application of performance tuning during software development (described in detail in [3, 6]). In this setting, we regard the code as a set of functions that call each other. The code is partitioned into *sections*, which are load modules. Each section consists of a set of functions and a set of overlay *areas* that are arranged in *series* in memory. An area consists of a set of sections that are arranged in *parallel* in memory.

The data of interest are generated by a linker and are represented by these sets of facts: `section_function(S,F)` denotes that section `S` contains function `F`; `section_area(S,A)` that section `S` contains area `A`; `area_section(A,S)` that area `A` contains section `S`; and, `calls(F1,F2)` that function `F1` calls function `F2`. The database comprises about 180K bytes, with the number of tuples as follows: `calls` (4593), `section_function` (995), `section_area` (31), and `area_section` (107).

When the program is executed, the root section is loaded and remains resident in memory until termination. All other sections are loaded on demand, and the loader guarantees that all ancestors of a section in the section-area tree are in memory whenever the section is. In order to retain acceptable performance, the loader must keep track of when it is actually necessary to reload a section.

We look at two relations in this database: `overwrites` and `smashable_caller`.

`overwrites`

When a section is loaded it may *overwrite* other sections. For example, parallel sections in the same overlay area overwrite each other. We say that section `S1` overwrites section `S2` if there exist two distinct sections, `SP1` and `SP2`, at the top level of the same area `A`, such that `SP1` (resp. `SP2`) is an ancestor of `S1` (resp. `S2`) or is `S1` (resp. `S2`) itself.

The definition of `overwrites` has a *common sub-expression*: the relation that associates a given section `A` with a section `S1` that `A` contains. We define the auxiliary relation `aux(A,SP1,S1)` whenever area `A` directly contains `SP1`, and `SP1` is either `S1` or an ancestor of `S1`. In other words, `aux` corresponds to the regular expression `area_section` $\cdot$ `(section_area` $\cdot$ `area_section)`*. The predicate `overwrites` is then defined as:

```
overwrites(S1,S2) ←
        aux(A,SP1,S1),
        aux(A,SP2,S2),
        not SP1 = SP2.
```

We can use either the *closure* or the *nfa* translation for the definition of `aux`. Moreover, in the presence of constants, we can project out the `A` argument of the `aux` predicate, and push the constant as far as possible (in both schemes). For example, if we are interested in overwrites within the `deskArea` area, the *closure* translation of the `aux` predicate is (`A` argument dropped):

```
aux(SP1,S1) ← area_section(deskArea,SP1), kl_compo(SP1,S1).
kl_compo(SP1,SP1).
kl_compo(SP1,S1) ← kl_compo(SP1,X), compo(X,S1).
compo(X,Y) ← section_area(X,Z), area_section(Z,Y).
```

while the *nfa* translation is:

```
p(SP1) ← area_section(deskArea,SP1).
q(SP1,X) ← p(SP1), section_area(SP1,X).
r(SP1,S1) ← q(SP1,X), area_section(X,S1).
q(SP1,X) ← r(SP1,S1), section_area(S1,X).

aux(SP1,SP1) ← p(SP1).
aux(SP1,S1) ← r(SP1,S1).
```

Note that the *nfa* translation is not able to factor all IDB predicates since both variables of the closure are required as join attributes in the rule for `overwrites` above. Nevertheless, as shown in Figure 2, the *nfa* translation performs slightly better than the *closure* translation. An explanation of the improvement in the absence of constants is postponed to Section 4.

`smashable_caller`

When a function call returns, the loader checks that the section being returned to is still in memory, and if not, reloads it. Recall that while any function is executing, its section and all of its ancestors in the section area tree will be in memory. We say that a section `S2` is a *smashable caller* of function `F1` if `S2` contains a function `F2` that calls `F1` and `S2` is not an ancestor in the section area tree of the section `S1` that contains `F1`. Section `S2` is potentially smashable because it is not guaranteed to be in memory while `F1` is executing.

The Datalog definition of `smashable_caller` is

```
smashable_caller(S2, F1) ←
        section_function(S2, F2),
        calls(F2, F1),
        section_function(S1, F1),
        not kl_compo(S1, S2).
```

where `kl_compo` is the (Kleene) closure of the relation among sections in the section area tree. The *closure* definition for it is as follows:

```
kl_compo(X, X).
kl_compo(X, Y) ← kl_compo(X, Z), compo(Z, Y).
compo(X, Y) ← section_area(X, Z), area_section(Z, Y).
```

Assume now that `S1` is bound to the constant `initSect`. In the *closure* translation, all occurrences of `S1` in the rule for `smashable_caller` are replaced with `initSect`; the rules for `kl_compo` are unchanged. In contrast, the *nfa* definition yields the following:

```
p(initSect).
q(Temp) ← p(S1), section_area(S1, Temp).
p(S2) ← q(Temp), area_section(Temp, S2).
kl_compo1(S2) ← p(S2).
```

Query	Constant	Which Constant	Output	Closure	Nfa
overwrites	–	–	6 684	6.08	**4.84**
overwrites	$\surd$	deskArea	352	0.87	**0.49**
smashable_caller	–	–	2001	21.73	**2.91**
smashable_caller	$\surd$	initSect	95	0.58	**0.19**

Fig. 2. Performance results on computing `overwrites, and smashable_caller`

Predicates p and q correspond to the two states of an NFA for the regular expression `(section_area` $\cdot$ `area_section)`*. State p is both the initial and final state.

Note that, in the presence of the constant, the *nfa* translation is able to factor `kl_compo`. This explains why it outperforms the *closure* translation on this example. It should be noted that `area_section` and `section_area` are small relations, so the performance difference is not that great. By increasing the size of these relations, the performance differential can be made arbitrarily large.

One reason for the relatively large difference in performance in the absence of constants is that CORAL uses ordered search for the evaluation of negated subgoals [12]. This also precludes the use of context rewriting when constants are present.

Figure 2 is a summary of our performance results (all times indicate the user cpu time in seconds and are averaged over a number of runs on a lightly loaded Sun Sparc10). As indicated, each query was run with and without a constant labelling one of the nodes in order to compare the effects of constant propagation. The column labelled "output" indicates the number of tuples in the answer of the query. The programs produced by the *closure* translation were run with the default rewriting scheme of CORAL, namely supplementary magic sets. Those produced by the *nfa* translation were run with no rewriting.

The results in Figure 2 suggest that the *nfa* translation always outperforms the *closure* translation. However, it is easy to find other examples where the *closure* translation runs faster than the *nfa* translation for queries with or without constants. We explore this in the next section.

4 Performance Analysis

Previous results on comparing performance costs induced by different evaluation strategies or optimization methods seem to suggest that an overriding factor is the presence or absence of constants in the query [1, 8]. In this section, we demonstrate both analytically and experimentally that the choice of which translation is preferable is *not* guided by the presence or absence of constants. It is guided only by the shape of the graph, that is, the join selectivity of the relations involved.

Query	Constant	Output	Closure	Nfa
(section_function.calls.(-section_function))$^+$	–	10 713	**14.41**	217.77
(function_call.section_call)$^{+}$[4]	–	14 224	120.12	**7.19**
(section_call.function_call)$^+$	$\surd$	4	**0.16**	0.24
(section_function.calls.(-calls).(-section_function))$^+$	$\surd$	104	46.52	**2.87**

Fig. 3. More experimental results on the overlay database

We focus on expressions involving the closure of a concatenation. The reason for this is that the two translations give rise to different join orderings for this class of expressions, while producing identical programs for other classes. The concatenation of a with b, $a \cdot b$, is the join of a and b, $a \bowtie b$, with the join attributes projected out. We will denote the concatenation as simply ab.

4.1 In the Absence of Constants

Suppose that the relation to be computed is the transitive closure of ab, that is, $(ab)^+$. The program of the *closure* translation computes ab and then joins every intermediate result $(ab)^i$ with ab in order to compute the next intermediate result $(ab)^{i+1}$. The program of the *nfa* translation, on the other hand, first joins $(ab)^i$ with a, and then the result, $(ab)^i a$, with b to get $(ab)^{i+1}$. Note that the *nfa* program requires twice as many semi-naive iterations, and therefore twice as many actual joins, as the *closure* program. The point is that the actual joins that are needed for the *nfa* program may be *cheaper*.

The algebraic expression that expresses the cost of evaluating the *closure* program is

$$\mathrm{cost}(a \bowtie b) + \sum_{i=1}^{l-1} \mathrm{cost}((ab)^i \bowtie ab)$$

while that for the *nfa* program is

$$\mathrm{cost}(a \bowtie b) + \sum_{i=1}^{l-1} \big(\mathrm{cost}((ab)^i \bowtie a) + \mathrm{cost}((ab)^i a \bowtie b)\big)$$

where l is the maximum length of computed paths.

Case 1: Where Closure Beats Nfa If the cost of joining $(ab)^i$ with ab is smaller than the cost of first joining $(ab)^i$ with a and then joining the result with b for each i, then the *closure* translation is better than the *nfa* translation. If the condition holds in the first iteration, and we assume some uniformity of the property across iterations, then it likely holds in subsequent iterations. So,

[4] Only a subset of the database is considered.

checking the condition for $i = 1$ provides a heuristic for choosing between the two translations in the absence of constants.

For illustrative purposes, we use the simplest measure for the cost of computing a join below, namely, the product of the sizes of its operand relations. Our motivation for this is that using a more sophisticated and accurate measure does not alter which of the translations is the more efficient in each of the cases we consider.

Figure 3 shows the results on further experiments on the database of Section 3. The following table computes the costs for $i = 1$, first for the *closure* translation, then for the *nfa* translation corresponding to the first row of Figure 3.

relation	no. of tuples	cost
`section_function`	995	
`calls`	4592	
`r = section_function` $\bowtie$ `calls` $\bowtie$ `(-section_function)`	511	
`r` $\bowtie$ `r`	1993	2.6×10^5
total for *closure* in 2nd iteration ($i = 1$)		2.6×10^5
`s = r` $\bowtie$ `section_function`	37220	5.1×10^5
`t = s` $\bowtie$ `calls`	28432	1.7×10^8
`t` $\bowtie$ `(-section_function)`	1993	2.8×10^7
total for *nfa* in 2nd iteration ($i = 1$)		1.9×10^8

Clearly, it is more beneficial to produce the join once and then use it repeatedly. This explains why the *closure* translation for this program requires only 14.41 sec while the *nfa* translation requires 217.77 sec. Note that the numbers in the `cost` column above are derived by multiplying the sizes of the operand relations in the corresponding join. Hence, the first number is $511 * 511 = 2,6 * 10^5$.

Case 2: Where Nfa Beats Closure If the cost of joining $(ab)^i$ with ab is greater than the cost of joining $(ab)^i$ first with a and then with b, then it is likely that the *nfa* translation is better than the *closure* translation.

As an example consider the program that corresponds to the second row of Figure 3.

relation	no. of tuples	cost
`function_call`	115	
`section_call`	139	
`r = function_call` $\bowtie$ `section_call`	13376	
`r` $\bowtie$ `r`	14098	1.8×10^8
total for *closure* in 2nd iteration ($i = 1$)		1.8×10^8
`s = r` $\bowtie$ `function_call`	212	1.5×10^6
`s` $\bowtie$ `section_call`	14098	2.9×10^4
total for *nfa* in 2nd iteration ($i = 1$)		1.5×10^6

Two new relations are used in this example: `function_call(F, S)` which is true if `F` calls a function in section `S`; and, `section_call(S, F)` which is true if a function in section `S` calls `F`. The comparative costs for $i = 1$ are given above. Clearly, in this case the *nfa* translation is more efficient (cf. Figure 3), 7.19 sec vs 120.12 sec. The same reasoning explains why the *nfa* translation outperforms the *closure* translation on the examples in Section 3.

4.2 In the Presence of Constants

Suppose now that we request the transitive closure of ab starting from a given constant, say c. Let us denote the computed relation by $\pi_c((ab)^+)$. The program of the *closure* translation first computes the whole of ab and then projects it on c. Subsequently, it joins each $\pi_c(ab)^i$ with ab in order to compute $\pi_c(ab)^{i+1}$. The program of the *nfa* translation, on the other hand, first projects a on c (to get $\pi_c(a)$) and then joins it with b. Then, it joins each $\pi_c(ab)^i$ with a and the result, $\pi_c(ab)^i a$, with b to get $\pi_c(ab)^{i+1}$.

Note that the *nfa* program requires twice as many semi-naive iterations, and therefore twice as many actual joins, as the *closure* program. Also, note the difference with the case of no constants. Given the initial constant to start the computation from, the *nfa* program never in fact computes the whole join ab. The initial step in the computation is $a \bowtie b$, for the *closure* program, but, $\pi_c(a) \bowtie b$, for the *nfa* program. The algebraic expression that expresses the cost of the *closure* program is

$$\mathrm{cost}(a \bowtie b) + \sum_{i=1}^{l-1} \mathrm{cost}(\pi_c(ab)^i \bowtie ab)$$

while that for the *nfa* program is

$$\mathrm{cost}(\pi_c(a) \bowtie b) + \sum_{i=1}^{l-1} \left(\mathrm{cost}(\pi_c(ab)^i \bowtie a) + \mathrm{cost}(\pi_c(ab)^i a \bowtie b)\right)$$

where l is the maximum length of computed paths.

If we hope to devise a heuristic for choosing between the two methods, as in the previous case, we have to consider the following two factors. The first is the initial cost of computing $a \bowtie b$, for the *closure* program; the *nfa* program computes initially just $\pi_c(a) \bowtie b$. The second factor is, as before, to compare the cost of joining $\pi_c(ab)^i$ with ab to the cost of joining first with a and then with b.

Case 1: Where Closure Beats Nfa Consider the program corresponding to the third row of the Figure 3. We consider the cost for the case $i = 1$ below.

It is clear that joining with `section_call` expands the size of the intermediate relation by so much that after a few iterations (in fact in no more than two iterations) the total cost paid by the *nfa* program exceeds the initial cost of the *closure* program, and the *closure* program performs better overall.

relation	no. of tuples	cost
section_call	568	
function_call	928	
r = section_call $\bowtie$ function_call	16	5.2×10^5
s = $\pi_{\texttt{specialSect}}$(r)	4	16
s $\bowtie$ r	4	64
total for *closure* after 2nd iteration		5.2×10^5
v = $\pi_{\texttt{specialSect}}$(section_call)	324	
s = v $\bowtie$ function_call	4	3.0×10^5
t = s $\bowtie$ section_call	423	2.2×10^3
t $\bowtie$ function_call	4	3.9×10^5
total for *nfa* after 2nd iteration		5.7×10^5

Case 2: Where Nfa Beats Closure Finally, consider the two translations for the program that gives better performance to the *nfa* program, the last row of Figure 3.

relation	no. of tuples	cost
section_function	995	
calls	4592	
r = section_function $\bowtie$ calls $\bowtie$ (-calls) $\bowtie$ (-section_function)	5878	2.1×10^{13}
s = $\pi_{\texttt{scanselSect}}$(r)	92	5.8×10^3
s $\bowtie$ r	102	5.4×10^5
total for *closure* after 2nd iteration		2.1×10^{13}
t1 = $\pi_{\texttt{scanselSect}}$(section_function)	13	
v1 = t1 $\bowtie$ calls	44	5.9×10^4
u1 = v1 $\bowtie$ (-calls)	417	2.0×10^5
w1 = u1 $\bowtie$ (-section_function)	92	4.1×10^5
t2 = w1 $\bowtie$ section_function	934	9.1×10^4
v2 = t2 $\bowtie$ calls	917	4.2×10^6
u2 = v2 $\bowtie$ (-calls)	743	4.2×10^6
u2 $\bowtie$ (-section_function)	102	7.4×10^5
total for *nfa* after 2nd iteration		9.9×10^6

It is clear that the initial cost of computing the complete join in the *closure* program results in its poor performance. Hence constant propagation is to be favored in this case.

5 Conclusions

In this paper we have compared the relative merits of constant propagation and join reordering as optimization methods for Datalog programs. This has been investigated using two different translations from the visual language GraphLog to Datalog. Our motivation is that visual presentation and querying of data will not be adopted by practitioners unless usefulness is matched by efficiency.

The *nfa* translation permits the propagation of constants in a query to the base relations, while the *closure* translation does not. Common wisdom suggests that this should favour the *nfa* translation in the presence of constants. We showed, however, that this is not the case. Given a query representing the closure of a concatenation of relations, the two translations produce programs which result in different join orderings being chosen during evaluation. This difference can override the benefit of constant propagation, thereby making it undesirable in some circumstances.

Recently it has been shown that the original factoring translation of [11] can be correctly applied to closure programs [15]. In addition, after further optimizations, a closure program generated from regular expression R is transformed into precisely the program generated by the *nfa* translation for R [15]. The results of the present paper indicate, however, that this transformation may not always be desirable.

References

1. F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *SIGMOD*, pages 16–52, 1986.
2. C. Beeri and R. Ramakrishnan. On the Power of Magic. In *PODS*, pages 269–283, 1987.
3. M.P. Consens. *Creating and Filtering Structural Data Visualizations using Hygraph Patterns*. PhD thesis, Department of Computer Science, University of Toronto, February 1994. (Available as Technical Report CSRI-302).
4. M.P. Consens, F.Ch. Eigler, M.Z. Hasan, A.O. Mendelzon, E.G. Noik, A.G. Ryman, and D. Vista. Architecture and Applications of the Hy$^+$ Visualization System. *IBM Systems Journal*, 33(3):458–476, 1994.
5. M.P. Consens and A.O. Mendelzon. GraphLog: A Visual Formalism for Real Life Recursion. In *PODS*, pages 404–416, 1990.
6. M.P. Consens, A.O. Mendelzon, and A. Ryman. Visualizing and Querying Software Structures. In *ICSE '92*, pages 138–156, 1992.
7. M.P. Consens, A.O. Mendelzon, and D. Vista. Deductive Database Support for Data Visualization. In *EDBT*, pages 45–58, 1994.
8. J. Han and H. Lu. Some Performance Results on Recursive Query Processing in Relational Database Systems. In *ICDE*, pages 533–539, 1986.
9. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
10. D. Kemp, K. Ramamohanarao, and Z. Somogyi. Right-, Left-, and Multi-Linear Rule Transformations that Maintain Context Information. In *VLDB*, pages 380–391, 1990.
11. J.F. Naughton, R. Ramakrishnan, Y. Sagiv, and J.D. Ullman. Argument Reduction by Factoring. In *VLDB*, pages 173–182, 1989.
12. R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. Implementation of the CORAL Deductive Database System. In *SIGMOD*, pages 167–176, 1993.
13. D. Vista and P.T. Wood. Efficient Evaluation of Visual Queries Using Deductive Databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 143–161. Kluwer Academic Publishers, Boston, 1995.
14. P.T. Wood. Factoring Augmented Regular Chain Programs. In *VLDB*, pages 255–263, 1990.
15. P.T. Wood. Magic Factoring of Closure Programs. In *PODS*, pages 174–183, 1995.