13 Evaluation of Datalog

Alice: *I don't mean to sound naive, but isn't it awfully expensive to answer datalog queries?*

Riccardo: *Not if you use the right bag of tricks . . .*

Vittorio: *. . . and some magical wisdom.*

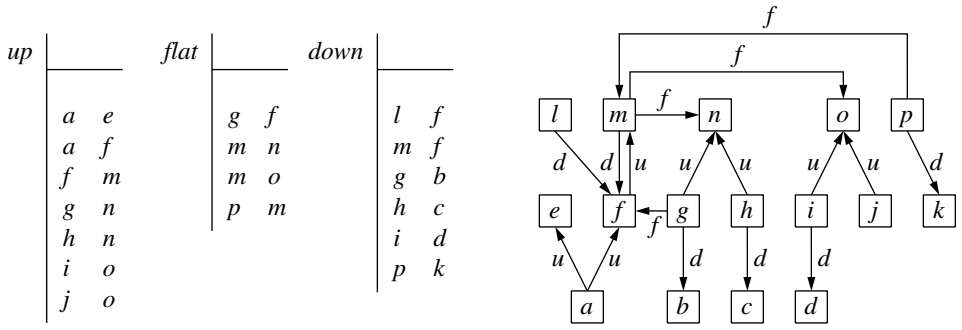
Sergio: *Well, there is no real need for magic. We will see that the evaluation is much easier if the algorithm knows where it is going and takes advantage of this knowledge.*

The introduction of datalog led to a flurry of research in optimization during the late 1980s and early 1990s. A variety of techniques emerged covering a range of different approaches. These techniques are usually separated into two classes depending on whether they focus on top-down or bottom-up evaluation. Another key dimension of the techniques concerns whether they are based on direct evaluation or propose some compilation of the query into a related query, which is subsequently evaluated using a direct technique.

This chapter provides a brief introduction to this broad family of heuristic techniques. A representative sampling of such techniques is presented. Some are centered around an approach known as "Query-Subquery"; these are top down and are based on direct evaluation. Others, centered around an approach called "magic set rewriting," are based on an initial preprocessing of the datalog program before using a fairly direct bottom-up evaluation strategy.

The advantage of top-down techniques is that selections that form part of the initial query can be propagated into the rules as they are expanded. There is no direct way to take advantage of this information in bottom-up evaluation, so it would seem that the bottom-up technique is at a disadvantage with respect to optimization. A rather elegant conclusion that has emerged from the research on datalog evaluation is that, surprisingly, there are bottom-up techniques that have essentially the same running time as top-down techniques. Exposition of this result is a main focus of this chapter.

Some of the evaluation techniques presented here are intricate, and our main emphasis is on conveying the essential ideas they use. The discussion is centered around the presentation of the techniques in connection with a concrete running example. In the cases of Query-Subquery and magic sets rewriting, we also informally describe how they can be applied in the general case. This is sufficient to give a precise understanding of the techniques without becoming overwhelmed by notation. Proofs of the correctness of these techniques are typically lengthy but straightforward and are left as exercises.



up	
a	*e*
a	*f*
f	*m*
g	*n*
h	*n*
i	*o*
j	*o*

flat	
g	*f*
m	*n*
m	*o*
p	*m*

down	
l	*f*
m	*f*
g	*b*
h	*c*
i	*d*
p	*k*

(a) The instance

(b) Represented as a graph

Figure 13.1: Instance $\mathbf{I}_0$ for RSG example

13.1 Seminaive Evaluation

The first stop on our tour of evaluation techniques is a strategy for improving the efficiency of the bottom-up technique described in Chapter 12. To illustrate this and the other techniques, we use as a running example the program "Reverse-Same-Generation" (RSG) given by

$$rsg(x, y) \leftarrow flat(x, y)$$
$$rsg(x, y) \leftarrow up(x, x1), rsg(y1, x1), down(y1, y)$$

and the sample instance $\mathbf{I}_0$ illustrated in Fig. 13.1. This is a fairly simple program, but it will allow us to present the main features of the various techniques presented throughout this chapter.

If the bottom-up algorithm of Chapter 12 is used to compute the value of *rsg* on input $\mathbf{I}_0$, the following values are obtained:

level 0: $\emptyset$
level 1: $\{\langle g, f\rangle, \langle m, n\rangle, \langle m, o\rangle, \langle p, m\rangle\}$
level 2: $\{\text{level 1}\} \cup \{\langle a, b\rangle, \langle h, f\rangle, \langle i, f\rangle, \langle j, f\rangle, \langle f, k\rangle\}$
level 3: $\{\text{level 2}\} \cup \{\langle a, c\rangle, \langle a, d\rangle\}$
level 4: $\{\text{level 3}\}$

at which point a fixpoint has been reached. It is clear that a considerable amount of redundant computation is done, because each layer recomputes all elements of the previous layer. This is a consequence of the monotonicity of the T_P operator for datalog programs P. This algorithm has been termed the *naive* algorithm for datalog evaluation. The central idea of the *seminaive* algorithm is to focus, to the extent possible, on the new facts generated at each level and thereby avoid recomputing the same facts.

Consider the facts inferred using the second rule of RSG in the consecutive stages of

the naive evaluation. At each stage, some new facts are inferred (until a fixpoint is reached). To infer a new fact at stage $i+1$, one must use at least one fact newly derived at stage i. This is the main idea of seminaive evaluation. It is captured by the following "version" of RSG, called RSG′:

$$\Delta^1_{rsg}(x, y) \leftarrow flat(x, y)$$
$$\Delta^{i+1}_{rsg}(x, y) \leftarrow up(x, x1), \Delta^i_{rsg}(y1, x1), down(y1, y)$$

where an instance of the second rule is included for each $i \geq 1$. Strictly speaking, this is not a datalog program because it has an infinite number of rules. On the other hand, it is not recursive.

Intuitively, Δ^i_{rsg} contains the facts in *rsg* newly inferred at the ith stage of the naive evaluation. To see this, we note a close relationship between the repeated applications of T_{RSG} and the values taken by the Δ^i_{rsg}. Let $\mathbf{I}$ be a fixed input instance. Then

- for $i \geq 0$, let $rsg^i = T^i_{RSG}(\mathbf{I})(rsg)$ (i.e., the value of *rsg* after i applications of T_{RSG} on $\mathbf{I}$); and
- for $i \geq 1$, let $\delta^i_{rsg} = RSG'(\mathbf{I})(\Delta^i_{rsg})$ (i.e., the value of Δ^i_{rsg} when $T_{RSG'}$ reaches a fixpoint on $\mathbf{I}$).

It is easily verified for each $i \geq 1$ that $T^{i-1}_{RSG'}(\mathbf{I})(\Delta^i_{rsg}) = \emptyset$ and $T^i_{RSG'}(\mathbf{I})(\Delta^i_{rsg}) = \delta^i_{rsg}$. Furthermore, for each $i \geq 0$ we have

$$rsg^{i+1} - rsg^i \subseteq \delta^{i+1}_{rsg} \subseteq rsg^{i+1}.$$

Therefore $\mathrm{RSG}(\mathbf{I})(rsg) = \cup_{1 \leq i}(\delta^i_{rsg})$. Furthermore, if j satisfies $\delta^j_{rsg} \subseteq \cup_{i<j}\delta^i_{rsg}$, then $\mathrm{RSG}(\mathbf{I})(rsg) = \cup_{i<j}\delta^i_{rsg}$, that is, only j levels of RSG′ need be computed to find $\mathrm{RSG}(\mathbf{I})(rsg)$. Importantly, bottom-up evaluation of RSG′ typically involves much less redundant computation than direct bottom-up evaluation of RSG.

Continuing with the informal development, we introduce now two refinements that further reduce the amount of redundant computation. The first is based on the observation that when executing RSG′, we do not always have $\delta^{i+1}_{rsg} = rsg^{i+1} - rsg^i$. Using $\mathbf{I}_0$, we have $\langle g, f \rangle \in \delta^2_{rsg}$ but not in $rsg^2 - rsg^1$. This suggests that the efficiency can be further improved by using $rsg^i - rsg^{i-1}$ in place of Δ^i_{rsg} in the body of the second "rule" of RSG′. Using a pidgin language that combines both datalog and imperative commands, the new version RSG″ is given by

$$\left\{ \begin{array}{lcl} \Delta^1_{rsg}(x, y) & \leftarrow & flat(x, y) \\ rsg^1 & := & \Delta^1_{rsg} \end{array} \right\}$$

$$\left\{ \begin{array}{lcl} temp^{i+1}_{rsg}(x, y) & \leftarrow & up(x, x1), \Delta^i_{rsg}(y1, x1), down(y1, y) \\ \Delta^{i+1}_{rsg} & := & temp^{i+1}_{rsg} - rsg^i \\ rsg^{i+1} & := & rsg^i \cup \Delta^{i+1}_{rsg} \end{array} \right\}$$

(where an instance of the second family of commands is included for each $i \geq 1$).

The second improvement to reduce redundant computation is useful when a given *idb* predicate occurs twice in the same rule. To illustrate, consider the nonlinear version of the ancestor program:

$$anc(x, y) \leftarrow par(x, y)$$
$$anc(x, y) \leftarrow anc(x, z), anc(z, y)$$

A seminaive "version" of this is

$$\left\{ \begin{array}{lcl} \Delta^1_{anc}(x, y) & \leftarrow & par(x, y) \\ anc^1 & := & \Delta^1_{anc} \end{array} \right\}$$

$$\left\{ \begin{array}{lcl} temp^{i+1}_{anc}(x, y) & \leftarrow & \Delta^i_{anc}(x, z), anc(z, y) \\ temp^{i+1}_{anc}(x, y) & \leftarrow & anc(x, z), \Delta^i_{anc}(z, y) \\ \Delta^{i+1}_{anc} & := & temp^{i+1}_{anc} - anc^i \\ anc^{i+1} & := & anc^i \cup \Delta^{i+1}_{anc} \end{array} \right\}$$

Note here that both Δ^i_{anc} and anc^i are needed to ensure that all new facts in the next level are obtained.

Consider now an input instance consisting of $par(1, 2), par(2, 3)$. Then we have

$$\Delta^1_{anc} = \{\langle 1, 2\rangle, \langle 2, 3\rangle\}$$
$$anc^1 = \{\langle 1, 2\rangle, \langle 2, 3\rangle\}$$
$$\Delta^2_{anc} = \{\langle 1, 3\rangle\}$$

Furthermore, both of the rules for $temp^2_{anc}$ will compute the join of tuples $\langle 1, 2\rangle$ and $\langle 2, 3\rangle$, and so we have a redundant computation of $\langle 1, 3\rangle$. Examples are easily constructed where this kind of redundancy occurs for at an arbitrary level $i > 0$ (see Exercise 13.2).

An approach for preventing this kind of redundancy is to replace the two rules for $temp^{i+1}$ by

$$temp^{i+1}(x, y) \leftarrow \Delta^i_{anc}(x, z), anc^{i-1}(z, y)$$
$$temp^{i+1}(x, y) \leftarrow anc^i(x, z), \Delta^i_{anc}(z, y)$$

This approach is adopted below.

We now present the seminaive algorithm for the general case. Let P be a datalog program over *edb* **R** and *idb* **T**. Consider a rule

$$S(u) \leftarrow R_1(v_1), \ldots, R_n(v_n), T_1(w_1), \ldots, T_m(w_m)$$

in P, where the R_k's are *edb* predicates and the T_j's are *idb* predicates. Construct for each $j \in [1, m]$ and $i \geq 1$ the rule

$$temp_S^{i+1}(u) \leftarrow R_1(v_1), \ldots, R_n(v_n),$$
$$T_1^i(w_1), \ldots, T_{j-1}^i(w_{j-1}), \Delta_{T_j}^i(w_j), T_{j+1}^{i-1}(w_{j+1}), \ldots, T_m^{i-1}(w_m).$$

Let P_S^i represent the set of all i-level rules of this form constructed for the *idb* predicate S (i.e., the rules for $temp_S^{i+1}$, j in $[1, m]$).

Suppose now that $T_1, \ldots, T_l$ is a listing of the *idb* predicates of P that occur in the body of a rule defining S. We write

$$P_S^i(\mathbf{I}, T_1^{i-1}, \ldots, T_l^{i-1}, T_1^i, \ldots, T_l^i, \Delta_{T_1}^i, \ldots, \Delta_{T_l}^i)$$

to denote the set of tuples that result from applying the rules in P_S^i to given values for input instance $\mathbf{I}$ and for the T_j^{i-1}, T_j^i, and $\Delta_{T_j}^i$.

We now have the following:

ALGORITHM 13.1.1 (Basic Seminaive Algorithm)

Input: Datalog program P and input instance $\mathbf{I}$

Output: $P(\mathbf{I})$

1. Set P' to be the rules in P with no *idb* predicate in the body;
2. $S^0 := \emptyset$, for each *idb* predicate S;
3. $\Delta_S^1 := P'(\mathbf{I})(S)$, for each *idb* predicate S;
4. $i := 1$;
5. do begin
   for each *idb* predicate S, where $T_1, \ldots, T_l$
   are the *idb* predicates involved in rules defining S,
   begin
   $S^i := S^{i-1} \cup \Delta_S^i$;
   $\Delta_S^{i+1} := P_S^i(\mathbf{I}, T_1^{i-1}, \ldots, T_l^{i-1}, T_1^i, \ldots, T_l^i, \Delta_{T_1}^i, \ldots, \Delta_{T_l}^i) - S^i$;
   end;
   $i := i + 1$
   end
   until $\Delta_S^i = \emptyset$ for each *idb* predicate S.
6. $s := s^i$, for each *idb* predicate S. ■

The correctness of this algorithm is demonstrated in Exercise 13.3. However, it is still doing a lot of unnecessary work on some programs. We now analyze the structure of datalog programs to develop an improved version of the seminaive algorithm. It turns out that this analysis, with simple control of the computation, allows us to know in advance which predicates are likely to grow at each iteration and which are not, either because they are already saturated or because they are not yet affected by the computation.

Let P be a datalog program. Form the *precedence graph* G_P for P as follows: Use the *idb* predicates in P as the nodes and include edge (R, R') if there is a rule with head predicate R' in which R occurs in the body. P is *recursive* if G_P has a directed cycle. Two predicates R and R' are *mutually recursive* if $R = R'$ or R and R' participate in the same

cycle of G_P. Mutual recursion is an equivalence relation on the *idb* predicates of P, where each equivalence class corresponds to a strongly connected component of G_P. A rule of P is *recursive* if the body involves a predicate that is mutually recursive with the head.

We now have the following:

ALGORITHM 13.1.2 (Improved Seminaive Algorithm)

Input: Datalog program P and edb instance $\mathbf{I}$

Output: $P(\mathbf{I})$

1. Determine the equivalence classes of *idb(P)* under mutual recursion.
2. Construct a listing $[R_1], \ldots, [R_n]$ of the equivalence classes, according to a topological sort of G_P (i.e., so that for each pair $i < j$ there is no path in G_P from R_j to R_i).
3. For $i = 1$ to n do
   Apply Basic Seminaive Algorithm to compute the values of predicates in $[R_i]$, treating all predicates in $[R_j]$, $j < i$, as *edb* predicates. ■

The correctness of this algorithm is left as Exercise 13.4.

Linear Datalog

We conclude this discussion of the seminaive approach by introducing a special class of programs.

Let P be a program. A rule in P with head relation R is *linear* if there is at most one atom in the body of the rule whose predicate is mutually recursive with R. P is *linear* if each rule in P is linear. We now show how the Improved Seminaive Algorithm can be simplified for such programs.

Suppose that P is a linear program, and

$$\rho : R(u) \leftarrow T_1(v_1), \ldots, T_n(v_n)$$

is a rule in P, where T_j is mutually recursive with R. Associate with this the "rule"

$$\Delta_R^{i+1}(u) \leftarrow T_1(v_1), \ldots, \Delta_{T_j}^i(v_j), \ldots, T_n(v_n).$$

Note that this is the only rule that will be associated by the Improved Seminaive Algorithm with ρ. Thus, given an equivalence class $[T_k]$ of mutually recursive predicates of P, the rules for predicates S in $[T_k]$ use only the Δ_S^i, but not the S^i. In contrast, as seen earlier, both the Δ_S^i and S^i must be used in nonlinear programs.

13.2 Top-Down Techniques

Consider the RSG program from the previous section, augmented with a selection-based query:

$$rsg(x, y) \leftarrow flat(x, y)$$
$$rsg(x, y) \leftarrow up(x, x1), rsg(y1, x1), down(y1, y)$$
$$query(y) \leftarrow rsg(a, y)$$

where a is a constant. This program will be called the *RSG query*. Suppose that seminaive evaluation is used. Then each pair of *rsg* will be produced, including those that are not used to derive any element of *query*. For example, using $\mathbf{I}_0$ of Fig. 13.1 as input, fact $rsg(f, k)$ will be produced but not used. A primary motivation for the top-down approaches to datalog query evaluation is to avoid, to the extent possible, the production of tuples that are not needed to derive any answer tuples.

For this discussion, we define a *datalog query* to be a pair (P, q), where P is a datalog program and q is a datalog rule using relations of P in its body and the new relation *query* in its head. We generally assume that there is only one rule defining the predicate *query*, and it has the form

$$query(u) \leftarrow R(v)$$

for some *idb* predicate R.

A fact is *relevant* to query (P, q) on input $\mathbf{I}$ if there is a proof tree for *query* in which the fact occurs. A straightforward criterion for improving the efficiency of any datalog evaluation scheme is to infer only relevant facts. The evaluation procedures developed in the remainder of this chapter attempt to satisfy this criterion; but, as will be seen, they do not do so perfectly.

The top-down approaches use natural heuristics to focus attention on relevant facts. In particular, they use the framework provided by SLD resolution. The starting point for these algorithms (namely, the query to be answered) often includes constants; these have the effect of restricting the search for derivation trees and thus the set of facts produced. In the context of databases without function symbols, the top-down datalog evaluation algorithms can generally be forced to terminate on all inputs, even when the corresponding SLD-resolution algorithm does not. In this section, we focus primarily on the query-subquery (QSQ) framework.

There are four basic elements of this framework:

1. Use the general framework of SLD resolution, but do it set-at-a-time. This permits the use of optimized versions of relational algebra operations.
2. Beginning with the constants in the original query, "push" constants from goals to subgoals, in a manner analogous to pushing selections into joins.
3. Use the technique of "sideways information passing" (see Chapter 6) to pass constant binding information from one atom to the next in subgoals.
4. Use an efficient global flow-of-control strategy.

Adornments and Subqueries

Recall the RSG query given earlier. Consider an SLD tree for it. The child of the root would be $rsg(a, y)$. Speaking intuitively, not all values for *rsg* are requested, but rather only those

with first coordinate a. More generally, we are interested in finding derivations for *rsg* where the first coordinate is *bound* and the second coordinate is *free*. This is denoted by the expression rsg^{bf}, where the superscript 'bf' is called an *adornment*.

The next layer of the SLD tree will have a node holding $flat(a, y)$ and a node holding $up(a, x1)$, $rsg(y1, x1)$, $down(y1, y)$. Answers generated for the first of these nodes are given by $\pi_2(\sigma_{1 = \text{'a'}}(flat))$. Answers for the other node can be generated by a left-to-right evaluation. First the set of possible values for $x1$ is $J = \pi_2(\sigma_{1 = \text{'a'}}(up))$. Next the possible values for $y1$ are given by $\{y1 \mid \langle y1, x1\rangle \in rsg \text{ and } \langle x1\rangle \in J\}$ (i.e., the first coordinate values of *rsg* stemming from second coordinate values in J). More generally, then, this calls for an evaluation of rsg^{fb}, where the second coordinate values are bound by J. Finally, given $y1$ values, these can be used with *down* to obtain y values (i.e., answers to the query).

As suggested by this discussion, a top-down evaluation of a query in which constants occur can be broken into a family of "subqueries" having the form (R^γ, J), where γ is an adornment for *idb* predicate R, and J is a set of tuples that give values for the columns bound by γ. Expressions of the form (R^γ, J) are called *subqueries*. If the RSG query were applied to the instance of Fig. 13.1, the first subquery generated would be $(rsg^{fb}, \{\langle e\rangle, \langle f\rangle\})$. As we shall see, the QSQ framework is based on a systematic evaluation of subqueries.

Let P be a datalog program and $\mathbf{I}$ an input instance. Suppose that R is an *idb* predicate and γ is an adornment for R (i.e., a string of b's and f's having length the arity of R). Then $bound(R, \gamma)$ denotes the coordinates of R bound in γ. Let t be a tuple over $bound(R, \gamma)$. Then a *completion* for t in R^γ is a tuple s such that $s[bound(R, \gamma)] = t$ and $s \in P(\mathbf{I})(R)$. The *answer* to a subquery (R^γ, J) over $\mathbf{I}$ is the set of all completions of all tuples in J.

The use of adornments within a rule body is a generalization of the technique of sideways information passing discussed in Chapter 6. Consider the rule

$$(*) \qquad R(x, y, z) \leftarrow R_1(x, u, v), R_2(u, w, w, z), R_3(v, w, y, a).$$

Suppose that a subquery involving R^{bfb} is invoked. Assuming a left-to-right evaluation, this will lead to subqueries involving R_1^{bff}, R_2^{bffb}, and R_3^{bbfb}. We sometimes rewrite the rule as

$$R^{bfb}(x, y, z) \leftarrow R_1^{bff}(x, u, v), R_2^{bffb}(u, w, w, z), R_3^{bbfb}(v, w, y, a)$$

to emphasize the adornments. This is an example of an *adorned rule*. As we shall see, the adornments of *idb* predicates in rule bodies shall be used to guide evaluations of queries and subqueries. It is common to omit the adornments of *edb* predicates.

The general algorithm for adorning a rule, given an adornment for the head and an ordering of the rule body, is as follows: (1) All occurrences of each bound variable in the rule head are bound, (2) all occurrences of constants are bound, and (3) if a variable x occurs in the rule body, then all occurrences of x in subsequent literals are bound. A different ordering of the rule body would yield different adornments. In general, we permit different orderings of rule bodies for different adornments of a given rule head. (A generalization of this technique is considered in Exercise 13.19.)

The definition of adorned rule also applies to situations in which there are repeated

variables or constants in the rule head (see Exercise 13.9). However, adornments do not capture all of the relevant information that can arise as the result of repeated variables or constants that occur in *idb* predicates in rule bodies. Mechanisms for doing this are discussed in Section 13.4.

Supplementary Relations and QSQ Templates

A key component of the QSQ framework is the use of *QSQ templates* which store appropriate information during intermediate stages of an evaluation. Consider again the preceding rule (*), and imagine attempting to evaluate the subquery (R^{bfb}, J). This will result in calls to the generalized queries $(R_1^{bff}, \pi_1(J))$, (R_2^{bffb}, K), and (R_3^{bbfb}, L) for some relations K and L that depend on the evaluation of the preceding queries. Importantly, note that relation K relies on values passed from both J and R_1, and L relies on values passed from R_1 and R_2. A QSQ template provides data structures that will remember all of the values needed during a left-to-right evaluation of a subquery.

To do this, QSQ templates rely on *supplementary relations*. A total of $n + 1$ supplementary relations are associated to a rule body with n atoms. For example, the supplementary relations $sup_0, \ldots, sup_3$ for the rule (*) with head adorned by R^{bfb} are

$$\begin{array}{l} R^{bfb}(x, y, z) \leftarrow R_1^{bff}(x, u, v), \;\; R_2^{bffb}(u, w, w, z), \;\; R_3^{bbfb}(v, w, y, a) \\ \qquad\qquad\quad \uparrow \qquad\qquad\quad \uparrow \qquad\qquad\quad \uparrow \qquad\qquad\quad \uparrow \\ \qquad\qquad sup_0[x, z] \quad sup_1[x, z, u, v] \quad sup_2[x, z, v, w] \quad sup_3[x, y, z] \end{array}$$

Note that variables serve as attribute names in the supplementary relations. Speaking intuitively, the body of a rule may be viewed as a process that takes as input tuples over the bound attributes of the head and produces as output tuples over the variables (bound and free) of the head. This determines the attributes of the first and last supplementary relations. In addition, a variable (i.e., an attribute name) is in some supplementary relation if it is has been bound by some previous literal and if it is needed in the future by some subsequent literal or in the result.

More formally, for a rule body with atoms $A_1, \ldots, A_n$, the set of variables used as attribute names for the i^{th} supplementary relation is determined as follows:

- For the 0^{th} (i.e., zeroth) supplementary relation, the attribute set is the set X_0 of bound variables of the rule head; and for the last supplementary relation, the attribute set is the set X_n of variables in the rule head.
- For $i \in [1, n-1]$, the attribute set of the i^{th} supplementary relation is the set X_i of variables that occur both "before" X_i (i.e., occur in $X_0, A_1, \ldots, A_i$) and "after" X_i (i.e., occur in $A_{i+1}, \ldots, A_n, X_n$).

The *QSQ template* for an adorned rule is the sequence $(sup_0, \ldots, sup_n)$ of relation schemas for the supplementary relations of the rule. During the process of QSQ query evaluation, relation instances are assigned to these schemas; typically these instances repeatedly acquire new tuples as the algorithm runs. Figure 13.2 shows the use of QSQ templates in connection with the RSG query.

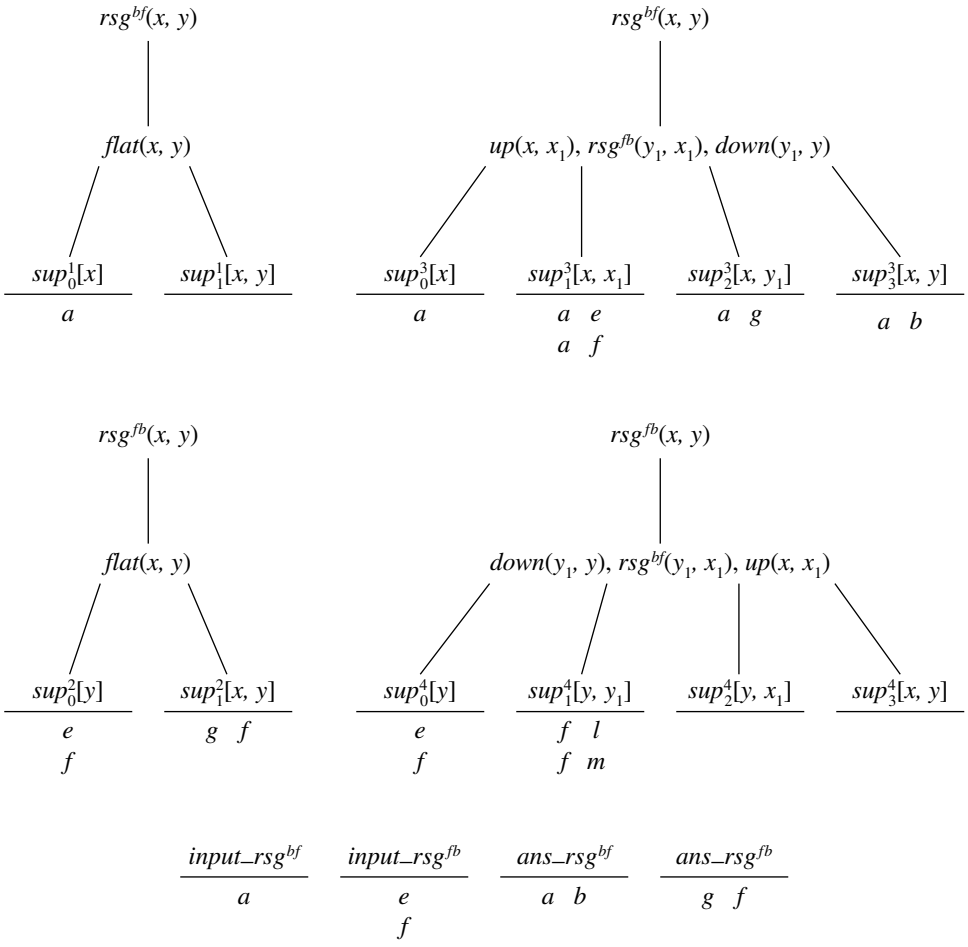


Figure 13.2: Illustration of QSQ framework

The Kernel of QSQ Evaluation

The key components of QSQ evaluation are as follows. Let (P, q) be a datalog query and let $\mathbf{I}$ be an *edb* instance. Speaking conceptually, QSQ evaluation begins by constructing an adorned rule for each adornment of each *idb* predicate in P and for the query q. In practice, the construction of these adorned rules can be lazy (i.e., they can be constructed only if needed during execution of the algorithm). Let (P^{ad}, q^{ad}) denote the result of this transformation.

The relevant adorned rules for the RSG query are as follows:

1. $rsg^{bf}(x, y) \leftarrow flat(x, y)$
2. $rsg^{fb}(x, y) \leftarrow flat(x, y)$

3. $rsg^{bf}(x, y) \leftarrow up(x, x1),\ rsg^{fb}(y1, x1),\ down(y1, y)$
4. $rsg^{fb}(x, y) \leftarrow down(y1, y),\ rsg^{bf}(y1, x1),\ up(x, x1).$

Note that in the fourth rule, the literals of the body are ordered so that the binding of y in *down* can be "passed" via $y1$ to *rsg* and via $x1$ to *up*.

A QSQ template is constructed for each relevant adorned rule. We denote the j^{th} (counting from 0) supplementary relation of the i^{th} adorned rule as sup_j^i. In addition, the following relations are needed and will serve as variables in the QSQ evaluation algorithm:

(a) for each *idb* predicate R and relevant adornment γ the variable ans_R^γ, with same arity as R;
(b) for each *idb* predicate R and relevant adornment γ, the variable $input\_R^\gamma$ with same arity as $bound(R, \gamma)$ (i.e., the number of b's occurring in γ); and
(c) for each supplementary relation sup_j^i, the variable sup_j^i.

Intuitively, $input\_R^\gamma$ will be used to form subqueries $(R^\gamma, input\_R^\gamma)$. The completion of tuples in $input\_R^\gamma$ will go to ans_R^γ. Thus ans_R^γ will hold tuples that are in $P(\mathbf{I})(R)$ and were generated from subqueries based on R^γ.

A QSQ algorithm begins with the empty set for each of the aforementioned relations. The query is then used to initialize the process. For example, the rule

$$query(y) \leftarrow rsg(a, y)$$

gives the initial value of $\{\langle a \rangle\}$ to $input\_rsg^{bf}$. In general, this gives rise to the subquery $(R^\gamma, \{t\})$, where t is constructed using the set of constants in the initial query.

There are essentially four kinds of steps in the execution. Different possible orderings for these steps will be considered. The first of these is used to initialize rules.

(A) Begin evaluation of a rule: This step can be taken whenever there is a rule with head predicate R^γ and there are "new" tuples in a variable $input\_R^\gamma$ that have not yet been processed for this rule. The step is to add the "new" tuples to the 0^{th} supplementary relation for this rule. However, only "new" tuples that unify with the head of the rule are added to the supplementary relation. A "new" tuple in $input\_R^\gamma$ might fail to unify with the head of a rule defining R if there are repeated variables or constants in the rule head (see Exercise 13.9).

New tuples are generated in supplementary relations sup_j^i in two ways: Either some new tuples have been obtained for sup_{j-1}^i (case B); or some new tuples have been obtained for the *idb* predicate occurring between sup_{j-1}^i and sup_j^i (case C).

(B) Pass new tuples from one supplementary relation to the next: This step can be taken whenever there is a set T of "new" tuples in a supplementary variable sup_{j-1}^i that have not yet been processed, and sup_{j-1}^i is not the last supplementary relation of the corresponding rule. Suppose that A_j is the atom in the rule immediately following sup_{j-1}^i.

Two cases arise:

(i) A_j is $R^\gamma(u)$ for some *edb* predicate R. Then a combination of joins and projections on R and T is used to determine the appropriate tuples to be added to sup_j^i.

(ii) A_j is $R^\gamma(u)$ for some *idb* predicate R. Note that each of the bound variables in γ occurs in sup_{j-1}^i. Two actions are now taken.
   (a) A combination of joins and projections on ans_R^γ (the current value for R) and T is used to determine the set T' of tuples to be added to sup_j^i.
   (b) The tuples in $T[bound(R, \gamma)] - input\_R^\gamma$ are added to $input\_R^\gamma$.

(C) Use new idb *tuples to generate new supplementary relation tuples:* This step is similar to the previous one but is applied when "new" tuples are added to one of the *idb* relation variables ans_R^γ. In particular, suppose that some atom A_j with predicate R^γ occurs in some rule, with surrounding supplementary variables sup_{j-1}^i and sup_j^i. In this case, use join and projection on all tuples in sup_{j-1}^i and the "new" tuples of ans_R^γ to create new tuples to be added to sup_j^i.

(D) Process tuples in the final supplementary relation of a rule: This step is used to generate tuples corresponding to the output of rules. It can be applied when there are "new" tuples in the final supplementary variable sup_n^i of a rule. Suppose that the rule predicate is R^γ. Add the new tuples in sup_n^i to ans_R^γ.

Example 13.2.1 Figure 13.2 illustrates the data structures and "scratch paper" relations used in the QSQ algorithm, in connection with the RSG query, as applied to the instance of Fig. 13.1. Recall the adorned version of the RSG query presented on page 321. The QSQ templates for these are shown in Fig. 13.2. Finally, the scratch paper relations for the *input*- and *ans*-variables are shown.

Figure 13.2 shows the contents of the relation variables after several steps of the QSQ approach have been applied. The procedure begins with the insertion of $\langle a \rangle$ into $input\_rsg^{bf}$; this corresponds to the rule

$$query(y) \leftarrow rsg(a, y)$$

Applications of step (A) place $\langle a \rangle$ into the supplementary variables sup_0^1 and sup_0^3. Step (B.i) then yields $\langle a, e \rangle$ and $\langle a, f \rangle$ in sup_1^3. Because ans_rsg^{fb} is empty at this point, step (B.ii.a) does not yield any tuples for sup_2^3. However, step (B.ii.b) is used to insert $\langle e \rangle$ and $\langle f \rangle$ into $input\_rsg^{fb}$. Application of steps (B) and (D) on the template of the second rule yield $\langle g, f \rangle$ in ans_rsg^{fb}. Application of steps (C), (B), and (D) on the template of the third rule now yield the first entry in ans_rsg^{bf}. The reader is invited to extend the evaluation to its conclusion (see Exercise 13.10). The answer is obtained by applying $\pi_2\sigma_{1 = \text{'a'}}$ to the final contents of ans_rsg^{bf}.

Global Control Strategies

We have now described all of the basic building blocks of the QSQ approach: the use of QSQ templates to perform information passing both into rules and sideways through rule bodies, and the three classes of relations used. A variety of global control strategies can be used for the QSQ approach. The most basic strategy is stated simply: Apply steps (A) through (D) until a fixpoint is reached. The following can be shown (see Exercise 13.12):

THEOREM 13.2.2 Let (P, q) be a datalog query. For each input $\mathbf{I}$, any evaluation of QSQ on (P^{ad}, q^{ad}) yields the answer of (P, q) on $\mathbf{I}$.

We now present a more specific algorithm based on the QSQ framework. This algorithm, called *QSQ Recursive* (QSQR) is based on a recursive strategy. To understand the central intuition behind QSQR, suppose that step (B) described earlier is to be performed, passing from supplementary relation sup^i_{j-1} across an *idb* predicate R^γ to supplementary relation sup^i_j. This may lead to the introduction of new tuples into sup^i_j by step (B.ii.a) and to the introduction of new tuples into $input\_R^\gamma$ by step (B.ii.b). The essence of QSQR is that it now performs a recursive call to determine the R^γ values corresponding to the new tuples added to $input\_R^\gamma$, before applying step (B) or (D) to the new tuples placed into sup^i_j.

We present QSQR in two steps: first a subroutine and then the recursive algorithm itself. During processing in QSQR, the global state includes values for ans_R^γ and $input\_R^\gamma$ for each *idb* predicate R and relevant adornment γ. However, the supplementary relations are not global—local copies of the supplementary relations are maintained by each call of the subroutine.

Subroutine Process subquery on one rule

Input: A rule for adorned predicate R^γ, input instance $\mathbf{I}$, a QSQR "state" (i.e., set of values for the *input*- and *ans*-variables), and a set $T \subseteq input\_R^\gamma$. (Intuitively, the tuples in T have not been considered with this rule yet).

Action:

1. Remove from T all tuples that do not unify with (the appropriate coordinates of) the head of the rule.
2. Set $sup_0 := T$. [This is step (A) for the tuples in T.]
3. Proceed sideways across the body $A_1, \ldots, A_n$ of the rule to the final supplementary relation sup_n as follows:
   For each atom A_j
   (a) If A_j has *edb* predicate R', then apply step (B.i) to populate sup_j.
   (b) If A_j has *idb* predicate R'^δ, then apply step (B.ii) as follows:
   (i) Set $S := sup_{j-1}[bound(R', \delta)] - input\_R'^\delta$.
   (ii) Set $input\_R'^\delta := input\_R'^\delta \cup S$. [This is step (B.ii.b).]
   (iii) (Recursively) call algorithm QSQR on the query (R'^δ, S).

[This has the effect of invoking step (A) and its consequences for the tuples in S.]

(iv) Use sup_{j-1} and the current value of global variable $ans\_R'^{\delta}$ to populate sup_j. [This includes steps (B.ii.a) and (C).]

4. Add the tuples produced for sup_n into the global variable ans_R^{γ}. [This is step (D).]

The main algorithm is given by the following:

ALGORITHM 13.2.3 (QSQR)

Input: A query of the form (R^{γ}, T), input instance $\mathbf{I}$, and a QSQR "state" (i.e., set of values for the *input*- and *ans*-variables).

Procedure:

1. Repeat until no new tuples are added to any global variable:
   Call the subroutine to process subquery (R^{γ}, T) on each rule defining R. ■

Suppose that we are given the query

$$query(u) \leftarrow R(v)$$

Let γ be the adornment of R corresponding to v, and let T be the singleton relation corresponding to the constants in v. To find the answer to the query, the QSQR algorithm is invoked with input (R^{γ}, T) and the global state where $input\_R^{\gamma} = T$ and all other *input*- and *ans*-variables are empty. For example, in the case of the *rsg* program, the algorithm is first called with argument $(rsg^{bf}, \{\langle a \rangle\})$, and in the global state $input\_rsg^{bf} = \{\langle a \rangle\}$. The answer to the query is obtained by performing a selection and projection on the final value of ans_R^{γ}.

It is straightforward to show that QSQR is correct (Exercise 13.12).

13.3 Magic

An exciting development in the field of datalog evaluation is the emergence of techniques for bottom-up evaluation whose performance rivals the efficiency of the top-down techniques. This family of techniques, which has come to be known as "magic set" techniques, simulates the pushing of selections that occurs in top-down approaches. There are close connections between the magic set techniques and the QSQ algorithm. The magic set technique presented in this section simulates the QSQ algorithm, using a datalog program that is evaluated bottom up. As we shall see, the magic sets are basically those sets of tuples stored in the relations $input\_R^{\gamma}$ and sup_j^i of the QSQ algorithm. Given a datalog query (P, q), the magic set approach transforms it into a new query (P^m, q^m) that has two important properties: (1) It computes the same answer as (P, q), and (2) when evaluated using a bottom-up technique, it produces only the set of facts produced by top-down approaches

(s1.1)	$rsg^{bf}(x, y)$	$\leftarrow input\_rsg^{bf}(x), flat(x, y)$
(s2.1)	$rsg^{fb}(x, y)$	$\leftarrow input\_rsg^{fb}(y), flat(x, y)$
(s3.1)	$sup_1^3(x, x1)$	$\leftarrow input\_rsg^{bf}(x), up(x, x1)$
(s3.2)	$sup_2^3(x, y1)$	$\leftarrow sup_1^3(x, x1), rsg^{fb}(y1, x1)$
(s3.3)	$rsg^{bf}(x, y)$	$\leftarrow sup_2^3(x, y1), down(y1, y)$
(s4.1)	$sup_1^4(y, y1)$	$\leftarrow input\_rsg^{fb}(y), down(y1, y)$
(s4.2)	$sup_2^4(y, x1)$	$\leftarrow sup_1^4(y, y1), rsg^{bf}(y1, x1)$
(s4.3)	$rsg^{fb}(x, y)$	$\leftarrow sup_2^4(y, x1), up(x, x1)$
(i3.2)	$input\_rsg^{bf}(x1)$	$\leftarrow sup_1^3(x, x1)$
(i4.2)	$input\_rsg^{fb}(y1)$	$\leftarrow sup_1^4(y, y1)$
(seed)	$input\_rsg^{bf}(a)$	$\leftarrow$
(query)	$query(y)$	$\leftarrow rsg^{bf}(a, y)$

Figure 13.3: Transformation of RSG query using magic sets

such as QSQ. In particular, then, (P^m, q^m) incorporates the effect of "pushing" selections from the query into bottom-up computations, as if by magic.

We focus on a technique originally called "generalized supplementary magic"; it is perhaps the most general magic set technique for datalog in the literature. (An earlier form of magic is considered in Exercise 13.18.) The discussion begins by explaining how the technique works in connection with the RSG query of the previous section and then presents the general algorithm.

As with QSQ, the starting point for magic set algorithms is an adorned datalog query (P^{ad}, q^{ad}). Four classes of rules are generated (see Fig. 13.3). The first consists of a family of rules for each rule of the adorned program P^{ad}. For example, recall rule (3) (see p. 321) of the adorned program for the RSG query presented in the previous section:

$$rsg^{bf}(x, y) \leftarrow up(x, x1), rsg^{fb}(y1, x1), down(y1, y).$$

We first present a primitive family of rules corrresponding to that rule, and then apply some optimizations.

(s3.0') $sup_0^3(x) \leftarrow input\_rsg^{bf}(x)$

(s3.1') $sup_1^3(x, x1) \leftarrow sup_0^3(x), up(x, x1)$

(s3.2) $sup_2^3(x, y1) \leftarrow sup_1^3(x, x1), rsg^{fb}(y1, x1)$

(s3.3') $sup_3^3(x, y) \leftarrow sup_2^3(x, y1), down(y1, y)$

(S3.4') $rsg^{bf}(x, y) \leftarrow sup_3^3(x, y)$

Rule (s3.0') corresponds to step (A) of the QSQ algorithm; rules (s3.1') and (s3.3') correspond to step (B.i); rule (s3.2) corresponds to steps (B.ii.a) and (C); and rule (s3.4') corresponds to step (D). In the literature, the predicate $input\_rsg^{fb}$ has usually been denoted as $magic\_rsg^{fb}$ and sup_j^i as $supmagic_j^i$. We use the current notation to stress the connection with the QSQ framework. Note that the predicate rsg^{bf} here plays the role of ans_rsg^{bf} there.

As can be seen by the preceding example, the predicates sup_0^3 and sup_3^3 are essentially redundant. In general, if the i^{th} rule defines R^γ, then the predicate sup_0^i is eliminated, with $input\_R^\gamma$ used in its place to eliminate rule (3.0') and to form

(s3.1) $sup_1^3(x, x1) \leftarrow input\_rsg^{bf}(x), up(x, x1).$

Similarly, the predicate of the last supplementary relation can be eliminated to delete rule (s3.4') and to form

(s3.3) $rsg^{bf}(x, y) \leftarrow sup_2^3(x, y1), down(y1, y).$

Therefore the set of rules (s3.0') through (s3.4') may be replaced by (s3.1), (s3.2), and (s3.3). Rules (s4.1), (s4.2), and (s4.3) of Fig. 13.3 are generated from rule (4) of the adorned program for the RSG query (see p. 321). (Recall how the order of the body literals in that rule are reversed to pass bounding information.) Finally, rules (s1.1) and (s2.1) stem from rules (1) and (2) of the adorned program.

The second class of rules is used to provide values for the *input* predicates [i.e., simulating step (B.ii.b) of the QSQ algorithm]. In the RSG query, one rule for each of $input\_rsg^{bf}$ and $input\_rsg^{fb}$ is needed:

(i3.2) $input\_rsg^{bf}(x1) \leftarrow sup_1^3(x, x1)$

(i4.2) $input\_rsg^{fb}(y1) \leftarrow sup_1^4(y, y1).$

Intuitively, the first rule comes from rule (s3.2). In other words, it follows from the second atom of the body of rule (3) of the original adorned program (see p. 321). In general, an adorned rule with k *idb* atoms in the body will generate k *input* rules of this form.

The third and fourth classes of rules include one rule each; these initialize and conclude the simulation of QSQ, respectively. The first of these acts as a "seed" and is derived from the initial query. In the running example, the seed is

$$input\_rsg^{bf}(a) \leftarrow .$$

The second constructs the answer to the query; in the example it is

$$query(y) \leftarrow rsg^{bf}(a, y).$$

From this example, it should be straightforward to specify the magic set rewriting of an adorned query (P^{ad}, q^{ad}) (see Exercise 13.16a).

The example showed how the "first" and "last" supplementary predicates sup_0^3 and sup_4^3 were redundant with $input\_rsg^{bf}$ and rsg^{bf}, respectively, and could be eliminated. Another improvement is to merge consecutive sequences of *edb* atoms in rule bodies as follows. For example, consider the rule

$$\text{(i)} \qquad R^{\gamma}(u) \leftarrow R_1^{\gamma_1}(u_1), \ldots, R_n^{\gamma_n}(u_n)$$

and suppose that predicate R_k is the last *idb* relation in the body. Then rules $(si.k), \ldots, (si.n)$ can be replaced with

$$(si.k'') \qquad R^{\gamma}(u) \leftarrow sup_{k-1}^{i}(v_{k-1}), R_k^{\gamma_k}(u_k), R_{k+1}^{\gamma_{k+1}}(u_{k+1}), \ldots, R_n^{\gamma_n}(u_n).$$

For example, rules (s3.2) and (s3.3) of Fig. 13.3 can be replaced by

$$(s3.2'') \qquad rsg^{bf}(x, y) \leftarrow sup_1^3(x, x1), rsg^{fb}(y1, x1), down(y1, y).$$

This simplification can also be used within rules. Suppose that R_k and R_l are *idb* relations with only *edb* relations occurring in between. Then rules $(i.k), \ldots, (i.l-1)$ can be replaced with

$$(si.k'') \qquad sup_{l-1}^{i}(v_{l-1}) \leftarrow sup_{k-1}^{i}(v_{k-1}), R_k^{\gamma_k}(u_k), R_{k+1}^{\gamma_{k+1}}(u_{k+1}), \ldots, R_{l-1}^{\gamma_{l-1}}(u_{l-1}).$$

An analogous simplification can be applied if there are multiple *edb* predicates at the beginning of the rule body.

To summarize the development, we state the following (see Exercise 13.16):

THEOREM 13.3.1 Let (P, q) be a query, and let (P^m, q^m) be the query resulting from the magic rewriting of (P, q). Then

(a) The answer computed by (P^m, q^m) on any input instance **I** is identical to the answer computed by (P, q) on **I**.

(b) The set of facts produced by the Improved Seminaive Algorithm of (P^m, q^m) on input **I** is identical to the set of facts produced by an evaluation of QSQ on **I**.

13.4 Two Improvements

This section briefly presents two improvements of the techniques discussed earlier. The first focuses on another kind of information passing resulting from repeated variables and constants occurring in *idb* predicates in rule bodies. The second, called counting, is applicable to sets of data and rules having certain acyclicity properties.

Repeated Variables and Constants in Rule Bodies (by Example)

Consider the program P_r:

(1) $T(x, y, z) \leftarrow R(x, y, z)$

(2) $T(x, y, z) \leftarrow S(x, y, w), T(w, z, z)$

$query(y, z) \leftarrow T(1, y, z)$

Consider as input the instance $\mathbf{I}_1$ shown in Fig. 13.4(a). The data structures for a QSQ evaluation of this program are shown in Fig. 13.4(b). (The annotations '$2 = $3', '$2 = $3 = 4', etc., will be explained later.)

A magic set rewriting of the program and query yields

$$
\begin{aligned}
T^{bff}(x, y, z) &\leftarrow input\_T^{bff}(x), R(x, y, z)\\
sup_1^2(x, y, w) &\leftarrow input\_T^{bff}(x), S(x, y, w)\\
T^{bff}(x, y, z) &\leftarrow sup_1^2(x, y, w), T^{bff}(w, z, z)\\
input\_T^{bff}(w) &\leftarrow sup_1^2(x, y, w)\\
input\_T^{bff}(1) &\leftarrow\\
query(y, z) &\leftarrow T^{bff}(1, y, z).
\end{aligned}
$$

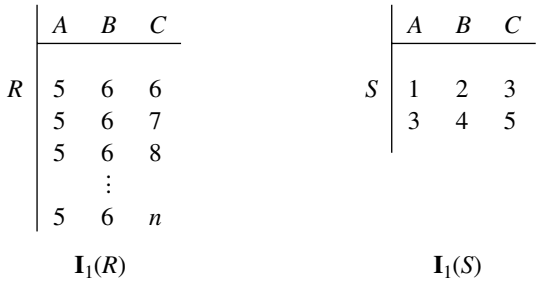
On input $\mathbf{I}_1$, the query returns the empty instance. Furthermore, the SLD tree for this query on $\mathbf{I}_1$ shown in Fig. 13.5, has only 9 goals and a total of 13 atoms, regardless of the value of n. However, both the QSQ and magic set approach generate a set of facts with size proportional to n (i.e., to the size of $\mathbf{I}_1$).

Why do both QSQ and magic sets perform so poorly on this program and query? The answer is that as presented, neither QSQ nor magic sets take advantage of restrictions on derivations resulting from the repeated z variable in the body of rule (2). Analogous examples can be developed for cases where constants appear in *idb* atoms in rule bodies.

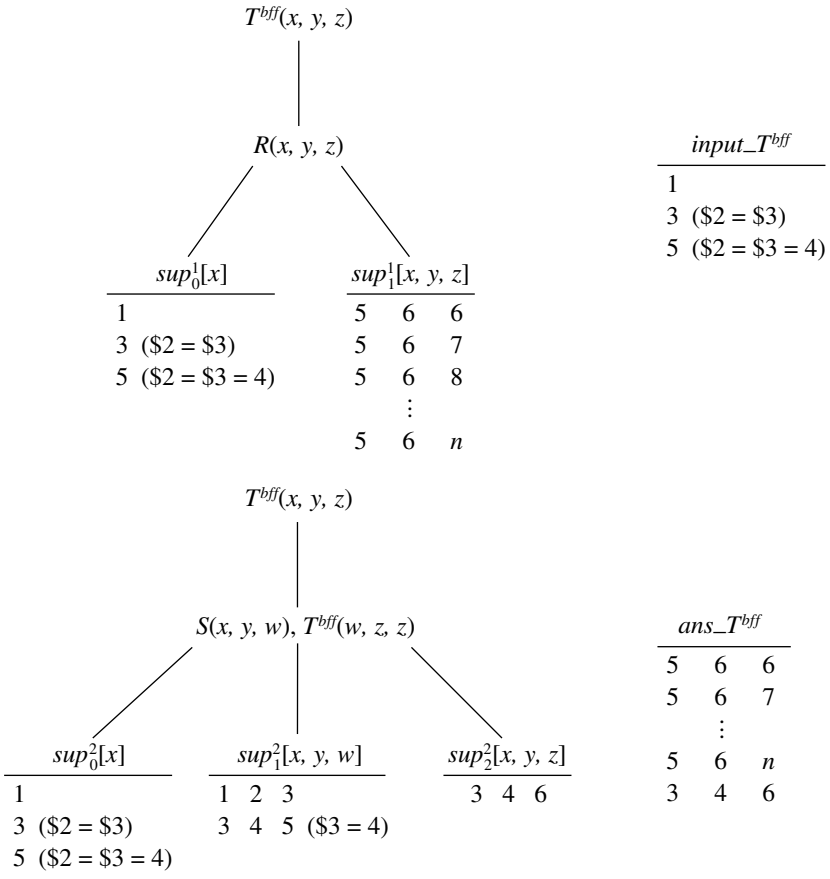
Both QSQ and magic sets can be enhanced to use such information. In the case of QSQ, the tuples added to supplementary relations can be annotated to carry information about restrictions imposed by the atom that "caused" the tuple to be placed into the leftmost supplementary relation. This is illustrated by the annotations in Fig. 13.4(b). First consider the annotation '$2 = $3' on the tuple $\langle 3\rangle$ in $input\_T^{bff}$. This tuple is included into $input\_T^{bff}$ because $\langle 1, 2, 3\rangle$ is in sup_1^2, and the next atom considered is $T^{bff}(w, z, z)$. In particular, then, any valid tuple (x, y, z) resulting from $\langle 3\rangle$ must have second and third coordinates equal. The annotation '$2 = $3' is passed with $\langle 3\rangle$ into sup_0^1 and sup_0^2.

Because variable y is bound to 4 in the tuple $\langle 3, 4, 5\rangle$ in sup_1^2, the annotation '$2 = $3' on $\langle 3\rangle$ in sup_0^2 "transforms" into '$3 = 4' on this new tuple. This, in turn, implies the annotation '$2 = $3 = 4' when $\langle 5\rangle$ is added to $input\_T^{bff}$ and to both sup_0^1 and sup_0^2.

Now consider the tuple $\langle 5\rangle$ in sup_0^1, with annotation ($2 = $3 = 4). This can generate a tuple in sup_1^1 only if $\langle 5, 4, 4\rangle$ is in R. For input $\mathbf{I}_1$ this tuple is not in R, and so the annotated



(a) Sample input instance $\mathbf{I}_1$



(b) QSQ evaluation

Figure 13.4: Behavior of QSQ on program with repeated variables

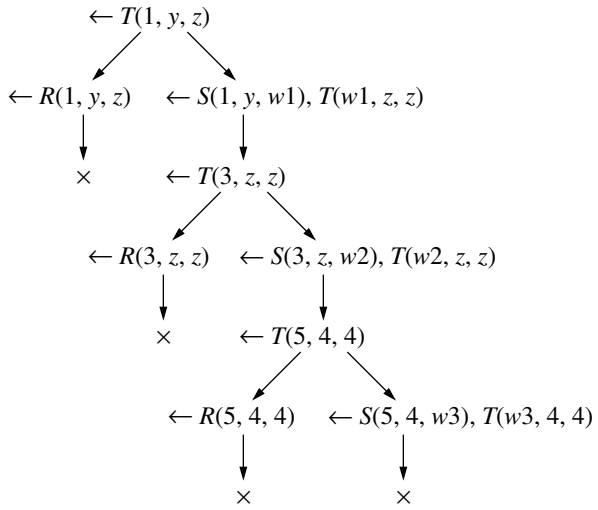


Figure 13.5: Behavior of SLD on program with repeated variables

tuple $\langle 5 \rangle$ in sup_0^1 generates nothing (even though in the original QSQ framework many tuples are generated). Analogously, because there is no tuple $\langle 5, 4, w \rangle$ in S, the annotated tuple $\langle 5 \rangle$ of sup_0^2 does not generate anything in sup_1^2. This illustrates how annotations can be used to restrict the facts generated during execution of QSQ.

More generally, annotations on tuples are conjunctions of equality terms of the form '$\$i = \$j$' and '$\$i = a$' (where a is a constant). During step (B.ii.b) of QSQ, annotations are associated with new tuples placed into relations $input\_R^\gamma$. We permit the same tuple to occur in $input\_R^\gamma$ with different annotations. This enhanced version of QSQ is called *annotated QSQ*. The enhancement correctly produces all answers to the initial query, and the set of facts generated now closely parallels the set of facts and assignments generated by the SLD tree corresponding to the QSQ templates used.

The magic set technique can also be enhanced to incorporate the information captured by the annotations just described. This is accomplished by an initial preprocessing of the program (and query) called "subgoal rectification." Speaking loosely, a subgoal corresponding to an *idb* predicate is *rectified* if it has no constants and no repeated variables. Rectified subgoals may be formed from nonrectified ones by creating new *idb* predicates that correspond to versions of *idb* predicates with repeated variables and constants. For example, the following is the result of rectifying the subgoals of the program P_r:

$$
\begin{aligned}
T(x, y, z) &\leftarrow R(x, y, z) \\
T(x, y, z) &\leftarrow S(x, y, w), T_{\$2=\$3}(w, z)
\end{aligned}
$$

$$
\begin{aligned}
T_{\$2=\$3}(x, z) &\leftarrow R(x, z, z) \\
T_{\$2=\$3}(x, z) &\leftarrow S(x, z, w), T_{\$2=\$3}(w, z)
\end{aligned}
$$

$$query(y, z) \leftarrow T(1, y, z)$$
$$query(z, z) \leftarrow T_{\$2=\$3}(1, z).$$

It is straightforward to develop an iterative algorithm that replaces an arbitrary datalog program and query with an equivalent one, all of whose *idb* subgoals are rectified (see Exercise 13.20). Note that there may be more than one rule defining the query after rectification.

The magic set transformation is applied to the rectified program to obtain the final result. In the preceding example, there are two relevant adornments for the predicate $T_{\$2=\$3}$ (namely, *bf* and *bb*).

The following can be verified (see Exercise 13.21):

THEOREM 13.4.1 (Informal) The framework of annotated QSQ and the magic set transformation augmented with subgoal rectification are both correct. Furthermore, the set of *idb* predicate facts generated by evaluating a datalog query with either of these techniques is identical to the set of facts occurring in the corresponding SLD tree.

A tight correspondence between the assignments in SLD derivation trees and the supplementary relations generated both by annotated QSQ and rectified magic sets can be shown. The intuitive conclusion drawn from this development is that top-down and bottom-up techniques for datalog evaluation have essentially the same efficiency.

Counting (by Example)

We now present a brief sketch of another improvement of the magic set technique. It is different from the previous one in that it works only when the underlying data set is known to have certain acyclicity properties.

Consider evaluating the following SG query based on the Same-Generation program:

(1) $sg(x, y) \leftarrow flat(x, y)$

(2) $sg(x, y) \leftarrow up(x, x1), sg(x1, y1), down(y1, y)$

$query(y) \leftarrow sg(a, y)$

on the input $\mathbf{J}_n$ given by

$$\mathbf{J}_n(up) = \{\langle a, b_i\rangle \mid i \in [1, n]\} \cup \{\langle b_i, c_j\rangle \mid i, j \in [1, n]\}$$
$$\mathbf{J}_n(flat) = \{\langle c_i, d_j\rangle \mid i, j \in [1, n]\}$$
$$\mathbf{J}_n(down) = \{\langle d_i, e_j\rangle \mid i, j \in [1, n]\} \cup \{\langle e_i, f\rangle \mid i \in [1, n]\}.$$

Instance $\mathbf{J}_2$ is shown in Fig. 13.6.

The completed QSQ template on input $\mathbf{J}_2$ for the second rule of the SG query is shown in Fig. 13.7(a). (The tuples are listed in the order in which QSQR would discover them.) Note that on input $\mathbf{J}_n$ both sup_1^2 and sup_2^2 would contain $n(n + 1)$ tuples.

Consider now the proof tree of SG having root $sg(a, f)$ shown in Fig. 13.8 (see Chapter 12). There is a natural correspondence of the children at depth 1 in this tree with the supplementary relation atoms $sup_0^2(a)$, $sup_1^2(a, b_1)$, $sup_2^2(a, e_1)$, and $sup_3^2(a, f)$ generated

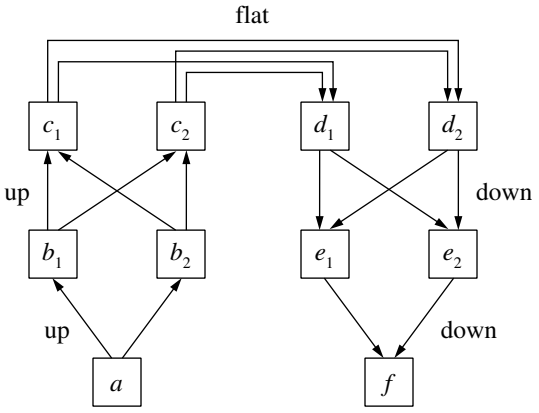


Figure 13.6: Instance $\mathbf{J}_2$ for counting

by QSQ; and between the children at depth 2 with $sup_0^2(b_1)$, $sup_1^2(b_1, c_1)$, $sup_2^2(b_1, d_1)$, and $sup_3^2(b_1, e_1)$.

A key idea in the counting technique is to record information about the depths at which supplementary relation atoms occur. In some cases, this permits us to ignore some of the specific constants present in the supplementary atoms. You will find that this is illustrated in Fig. 13.7(b). For example, we show atoms $count\_sup_0^2(1, a)$, $count\_sup_1^2(1, b_1)$, $count\_sup_2^2(1, e_1)$, and $count\_sup_3^2(1, f)$ that correspond to the supplementary atoms $sup_0^2(a)$, $sup_1^2(a, b_1)$, $sup_2^2(a, e_1)$, and $sup_3^2(a, f)$. Note that, for example, $count\_sup_1^2(2, c_1)$ corresponds to both $sup_1^2(b_1, c_1)$ and $sup_1^2(b_2, c_1)$.

More generally, the modified supplementary relation atoms hold an "index" that indicates a level in a proof tree corresponding to how the atom came to be created. Because of the structure of SG, and assuming that the *up* relation is acyclic, these modified supplementary relations can be used to find query answers. Note that on input $\mathbf{J}_n$, the relations $countsup_1^{2'}$ and $count\_sup_2^{2'}$ hold $2n$ tuples each rather than $n(n+1)$, as in the original QSQ approach.

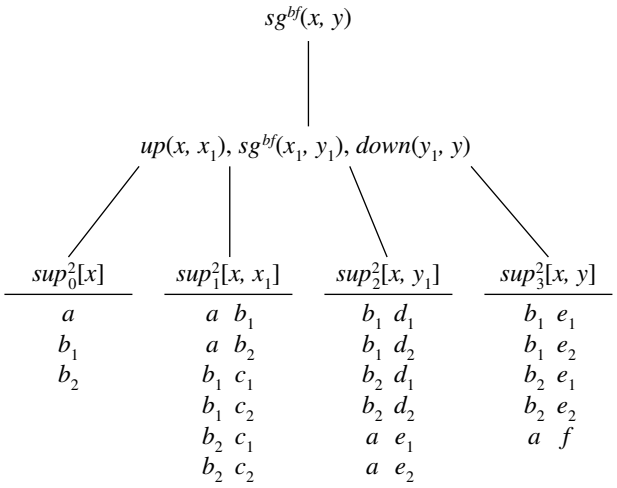
We now describe how the magic set program associated with the SG query can be transformed into an equivalent program (on acyclic input) that uses the indexes suggested by Fig. 13.7(b). The magic set rewriting of the SG query is given by

(s1.1) $sg^{bf}(x, y) \leftarrow input\_sg^{bf}(x), flat(x, y)$

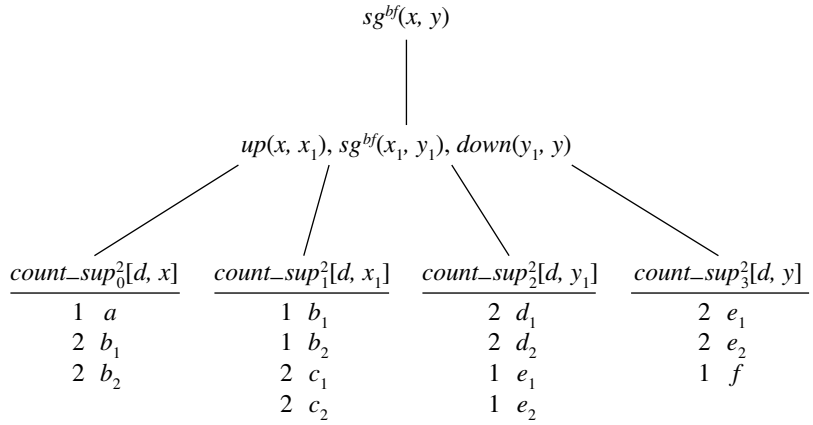
(s2.1) $sup_1^2(x, x1) \leftarrow input\_sg^{bf}(x), up(x, x1)$

(s2.2) $sup_2^2(x, y1) \leftarrow sup_1^2(x, x1), sg^{bf}(x1, y1)$

(s2.3) $sg^{bf}(x, y) \leftarrow sup_2^2(x, y1), down(y1, y)$



(a) Completed QSQ template for sg^{bf} on input $\mathbf{J}_2$



(b) Alternative QSQ "template," using indices

Figure 13.7: Illustration of intuition behind counting

(i2.2) $\quad input\_sg^{bf}(x1) \leftarrow sup_1^2(x, x1)$

(seed) $\quad input\_sg^{bf}(a) \leftarrow$

(query) $\quad query(y) \leftarrow sg^{bf}(a, y).$

The counting version of this is now given. (In other literature on counting, the seed is initialized with 0 rather than 1.)

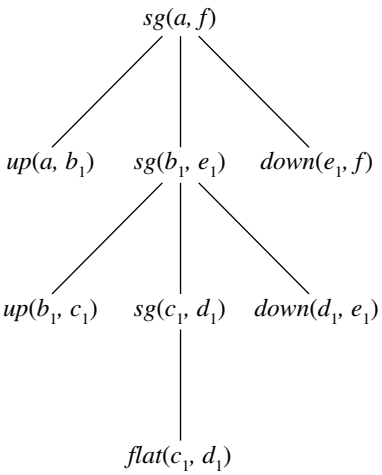


Figure 13.8: A proof tree for $sg(a, f)$

(c-s1.1) $count\_sg^{bf}(I, y) \leftarrow count\_input\_sg^{bf}(I, x), flat(x, y)$

(c-s2.1) $count\_sup_1^2(I, x1) \leftarrow count\_input\_sg^{bf}(I, x), up(x, x1)$

(c-s2.2) $count\_sup_2^2(I, y1) \leftarrow count\_sup_1^2(I, x1), count\_sg^{bf}(I + 1, y1)$

(c-s2.3) $count\_sg^{bf}(I, y) \leftarrow count\_sup_2^2(I, y1), down(y1, y)$

(c-i2.2) $count\_input\_sg^{bf}(I + 1, x1) \leftarrow count\_sup_1^2(I, x1)$

(c-seed) $count\_input\_sg^{bf}(1, a) \leftarrow$

(c-query) $query(y) \leftarrow count\_sg^{bf}(1, y)$

In the preceding, expressions such as $I + 1$ are viewed as a short hand for using a variable J in place of $I + 1$ and including $J = I + 1$ in the rule body.

In the counting version, the first coordinate of each supplementary relation keeps track of a level in a proof tree rather than a specific value. Intuitively, when "constructing" a sequence of supplementary atoms corresponding to a given level of a proof tree, each *idb* atom used must have been generated from the next deeper level. This is why $count\_sg^{bf}(I + 1, y1)$ is used in rule (c-s2.2). Furthermore, rule (c-i2.2) initiates the "construction" corresponding to a new layer of the proof tree.

The counting program of the preceding example is not safe, in the sense that on some inputs the program may produce an infinite set of tuples in some predicates (e.g., $count\_sup_1^2$). For example, this will happen if there is a cycle in the *up* relation reachable from a. Analogous situations occur with most applications of counting. As a result, the counting technique can only be used where the underlying data set is known to satisfy certain restrictions.

This preceding example is a simple application of the general technique of counting. A more general version of counting uses three kinds of indexes. The first, illustrated in the example, records information about levels of proof trees. The second is used to record information about what rule is being expanded, and the third is used to record which atom of the rule body is being considered (see Exercise 13.23). A description of the kinds of programs for which the counting technique can be used is beyond the scope of this book. Although limited in applicability, the counting technique has been shown to yield significant savings in some contexts.

Bibliographic Notes

This chapter has presented a brief introduction to the research on heuristics for datalog evaluation. An excellent survey of this work is [BR88a], which presents a taxonomy of different techniques and surveys a broad number of them. Several books provide substantial coverage of this area, including [Bid91a, CGT90, Ull89b]. Experimental results comparing several of the techniques in the context of datalog are described in [BR88b]. An excellent survey on deductive database systems, which includes an overview of several prototype systems that support datalog, is presented in [RU94].

The naive and seminaive strategies for datalog evaluation underlie several early investigations and implementations [Cha81b, MS81]; the seminaive strategy for evaluation is described in [Ban85, Ban86], which also propose various refinements. The use of T^{i-1} and T^i in Algorithm 13.1.1 is from [BR87b]. Reference [CGT90] highlights the close relationship of these approaches to the classical Jacobi and Gauss-Seidel algorithms of numerical analysis.

An essential ingredient of the top-down approaches to datalog evaluation is that of "pushing" selections into recursions. An early form of this was developed in [AU79], where selections and projections are pushed into restricted forms of fixpoint queries (see Chapter 14 for the definition of fixpoint queries).

The Query-Subquery (QSQ) approach was initially presented in [Vie86]; the independently developed method of "extension tables" [DW87] is essentially equivalent to this. The QSQ approach is extended in [Vie88, Vie89] to incorporate certain global optimizations. An extension of the technique to general logic programming, called SLD-AL, is developed in [Vie87a, Vie89]. Related approaches include APEX [Loz85], Earley Deduction [PW80, Por86], and those of [Nej87, Roe87]. The connection between context-free parsing and datalog evaluation is highlighted in [Lan88].

The algorithms of the QSQ family are sometimes called "memo-ing" approaches, because they use various data structures to remember salient inferred facts to filter the work of traditional SLD resolution.

Perhaps the most general of the top-down approaches uses "rule/goal" graphs [Ull85]; these potentially infinite trees intuitively correspond to a breadth-first, set-at-a-time execution of SLD resolution. Rule/goal graphs are applied in [Van86] to evaluate datalog queries in distributed systems. Similar graph structures have also been used in connection with general logic programs (e.g., [Kow75, Sic76]). A survey of several graph-based approaches is [DW85].

Turning to bottom-up approaches, the essentially equivalent approaches of [HN84] and

[GdM86] develop iterative algebraic programs for linear datalog programs. [GS87] extends these. A more general approach based on rewriting iterative algebra programs is presented in [CT87, Tan88].

The magic set and counting techniques originally appeared for linear datalog in [BMSU86]. Our presentation of magic sets is based on an extended version called "generalized supplementary magic sets" [BR87a, BR91]. That work develops a general notion of sideways information passing based on graphs (see Exercise 13.19), and develops both magic sets and counting in connection with general logic programming. The Alexander method [RLK86, Ker88], developed independently, is essentially the same as generalized supplementary magic sets for datalog. This was generalized to logic programming in [Sek89]. Magic set rewriting has also been applied to optimize SQL queries [MFPR90].

The counting method is generalized and combined with magic sets in [SZ86, SZ88]. Supplementary magic is incorporated in [BR91]. Analytic comparisons of magic and counting for selected programs are presented in [MSPS87].

Another bottom-up technique is Static Filtering [KL86a, KL86b]. This technique forms a graph corresponding to the flow of tuples through a bottom-up evaluation and then modifies the graph in a manner that captures information passing resulting from constants in the initial query.

Several of the investigations just mentioned, including [BR87a, KL86a, KL86b, Ull85, Vie86], emphasize the idea that sideways information passing and control are largely independent. Both [SZ88] and [BR91] describe fairly general mechanisms for specifying and using alternative sideways information passing and related message passing. A more general form of sideways information passing, which passes bounding inequalities between subgoals, is studied in [APP+86]. A formal framework for studying the success of pushing selections into datalog programs is developed in [BKBR87].

Several papers have studied the connection between top-down and bottom-up evaluation techniques. One body of the research in this direction focuses on the sets of facts generated by the top-down and bottom-up techniques. One of the first results relating top-down and bottom-up is from [BR87a, BR91], where it is shown that if a top-down technique and the generalized supplementary magic set technique use a given family of sideways information passing techniques, then the sets of intermediate facts produced by both techniques correspond. That research is conducted in the context of general logic programs that are range restricted. These results are generalized to possibly non-range-restricted logic programs in the independent research [Ram91] and [Sek89]. In that research, bottom-up evaluations may use terms and tuples that include variables, and bottom-up evaluation of rewritten programs uses unification rather than simple relational join. A close correspondence between top-down and bottom-up evaluation for datalog was established in [Ull89a], where subgoal rectification is used. The treatment of Program P_r and Theorem 13.4.1 are inspired by that development. This close correspondence is extended to arbitrary logic programs in [Ull89b]. Using a more detailed cost model, [SR93] shows that bottom-up evaluation asymptotically dominates top-down evaluation for logic programs, even if they produce nonground terms in their output.

A second direction of research on the connection between top-down and bottom-up approaches provides an elegant unifying framework [Bry89]. Recall in the discussion of Theorem 13.2.2 that the answer to a query can be obtained by performing the steps of

the QSQ until a fixpoint is reached. Note that the fixpoint operator used in this chapter is different from the conventional bottom-up application of T_P used by the naive algorithm for datalog evaluation. The framework presented in [Bry89] is based on meta-interpreters (i.e., interpreters that operate on datalog rules in addition to data); these can be used to specify QSQ and related algorithms as bottom-up, fixpoint evaluations. (Such meta-programming is common in functional and logic programming but yields novel results in the context of datalog.) Reference [Bry89] goes on to describe several top-down and bottom-up datalog evaluation techniques within the framework, proving their correctness and providing a basis for comparison.

A recent investigation [NRSU89] improves the performance of the magic sets in some cases. If the program and query satisfy certain conditions, then a technique called factoring can be used to replace some predicates by new predicates of lower arity. Other improvements are considered in [Sag90], where it is shown in particular that the advantage of one method over another may depend on the actual data, therefore stressing the need for techniques to estimate the size of *idb*'s (e.g., [LN90]).

Extensions of the datalog evaluation techniques to stratified datalog$^\neg$ programs (see Chapter 15) include [BPR87, Ros91, SI88, KT88].

Another important direction of research has been the parallel evaluation of datalog programs. Heuristics are described in [CW89b, GST90, Hul89, SL91, WS88, WO90].

A novel approach to answering datalog queries efficiently is developed in [DT92, DS93]. The focus is on cases in which the same query is asked repeatedly as the underlying *edb* is changing. The answer of the query (and additional scratch paper relations) is materialized against a given *edb* state, and first-order queries are used incrementally to maintain the materialized data as the underlying *edb* state is changed.

A number of prototype systems based on variants of datalog have been developed, incorporating some of the techniques mentioned in this chapter. They include DedGin [Vie87b, LV89], NAIL! [Ull85, MUV86, MNS$^+$87], LDL [NT89], ALGRES [CRG$^+$88], NU-Prolog [RSB$^+$87], GLUE-NAIL [DMP93], and CORAL [RSS92, RSSS93]. Descriptions of projects in this area can also be found in [Zan87], [RU94].

Exercises

Exercise 13.1 Recall the program RSG′ from Section 13.1. Exhibit an instance **I** such that on this input, $\delta^i_{rsg} \neq \emptyset$ for each $i > 0$.

Exercise 13.2 Recall the informal discussion of the two seminaive "versions" of the nonlinear ancestor program discussed in Section 13.1. Let P_1 denote the first of these, and P_2 the second. Show the following.

(a) For some input, P_2 can produce the same tuple more than once at some level beyond the first level.

(b) If P_2 produces the same tuple more than once, then each occurrence corresponds to a distinct proof tree (see Section 12.5) from the program and the input.

(c) P_1 can produce a given tuple twice, where the proof trees corresponding to the two occurrences are identical.

Exercise 13.3 Consider the basic seminaive algorithm (13.1.1).

(a) Verify that this algorithm terminates on all inputs.

(b) Show that for each $i \geq 0$ and each *idb* predicate S, after the i^{th} execution of the loop the value of variable S^i is equal to $T_P^i(\mathbf{I})(S)$ and the value of Δ_S^{i+1} is equal to $T_P^{i+1}(\mathbf{I})(S) - T_P^i(\mathbf{I})(S)$.

(c) Verify that this algorithm produces correct output on all inputs.

(d) Give an example input for which the same tuple is generated during different loops of the algorithm.

Exercise 13.4 Consider the improved seminaive algorithm (13.1.2).

(a) Verify that this algorithm terminates and produces correct output on all inputs.

(b) Give an example of a program P for which the improved seminaive algorithm produces fewer redundant tuples than the basic seminaive algorithm.

Exercise 13.5 Let P be a linear datalog program, and let P' be the set of rules associated with P by the improved seminaive algorithm. Suppose that the naive algorithm is performed using P' on some input $\mathbf{I}$. Does this yield $P(\mathbf{I})$? Why or why not? What if the basic seminaive algorithm is used?

Exercise 13.6 A set X of relevant facts for datalog query (P, q) and input $\mathbf{I}$ is *minimal* if (1) for each answer β of q there is a proof tree for β constructed from facts in X, and (2) X is minimal having this property. Informally describe an algorithm that produces a minimal set of relevant facts for a query (P, q) and input $\mathbf{I}$ and is polynomial time in the size of $\mathbf{I}$.

Exercise 13.7 [BR91] Suppose that program P includes the rule

$$\rho : S(x, y) \leftarrow S_1(x, z), S_2(z, y), S_3(u, v), S_4(v, w),$$

where S_3, S_4 are *edb* relations. Observe that the atoms $S_3(u, v)$ and $S_4(v, w)$ are not connected to the other atoms of the rule body or to the rule head. Furthermore, in an evaluation of P on input $\mathbf{I}$, this rule may contribute some tuple to S only if there is an assignment α for u, v, w such that $\{S_3(u, v), S_4(v, w)\}[\alpha] \subseteq \mathbf{I}$. Explain why it is typically more efficient to replace ρ with

$$\rho' : S(x, y) \leftarrow S_1(x, z), S_2(z, y)$$

if there is such an assignment and to delete ρ from P otherwise. Extend this to the case when S_3, S_4 are *idb* relations. State a general version of this heuristic improvement.

Exercise 13.8 Consider the adorned rule

$$R^{bf}(x, w) \leftarrow S_1^{bf}(x, y), S_2^{bf}(y, z), T_1^{ff}(u, v), T_2^{bf}(v, w).$$

Explain why it makes sense to view the second occurrence of v as bound.

Exercise 13.9 Consider the rule

$$R(x, y, y) \leftarrow S(y, z), T(z, x).$$

(a) Construct adorned versions of this rule for R^{ffb} and R^{fbb}.

(b) Suppose that in the QSQ algorithm a tuple $\langle b, c\rangle$ is placed into $input\_R^{fbb}$. Explain why this tuple should not be placed into the 0^{th} supplementary relation for the second adorned rule constructed in part (a).

(c) Exhibit an example analogous to part (b) based on the presence of a constant in the head of a rule rather than on repeated variables.

Exercise 13.10

(a) Complete the evaluation in Example 13.2.1.

(b) Use Algorithm 13.2.3 (QSQR) to evaluate that example.

★ **Exercise 13.11** In the QSQR algorithm, the procedure for processing subqueries of the form (R^γ, S) is called until no global variable is changed. Exhibit an example datalog query and input where the second cycle of calls to the subqueries (R^γ, S) generates new answer tuples.

♠ **Exercise 13.12** (a) Prove Theorem 13.2.2. (b) Prove that the QSQR algorithm is correct.

★ **Exercise 13.13** The *Iterative QSQ* (*QSQI*) algorithm uses the QSQ framework, but without recursion. Instead in each iteration it processes each rule body from left to right, using the values currently in the relations ans_R^γ when computing values for the supplementary relations. As with QSQR, the variables $input\_R^\gamma$ and ans_R^γ are global, and the variables for the supplementary relations are local. Iteration continues until there is no change to the global variables.

(a) Specify the QSQI algorithm more completely.

(b) Give an example where QSQI performs redundant work that QSQR does not.

Exercise 13.14 [BR91] Consider the following query based on a nonlinear variant of the same-generation program, called here the SGV query:

(a) $sgv(x, y) \leftarrow flat(x, y)$

(b) $sgv(x, y) \leftarrow up(x, z1), sgv(z1, z2), flat(z2, z3), sgv(z3, z4), down(z4, y)$
$query(y) \leftarrow sgv(a, y)$

Give the magic set transformation of this program and query.

Exercise 13.15 Give examples of how a query (P^m, q^m) resulting from magic set rewriting can produce nonrelevant and redundant facts.

♠ **Exercise 13.16**

(a) Give the general definition of the magic set rewriting technique.

(b) Prove Theorem 13.3.1.

Exercise 13.17 Compare the difficulties, in practical terms, of using the QSQ and magic set frameworks for evaluating datalog queries.

★ **Exercise 13.18** Let (P, q) denote the SGV query of Exercise 13.14. Let (P^m, q^m) denote the result of rewriting this program, using the (generalized supplementary) magic set transformation presented in this chapter. Under an earlier version, called here "original" magic, the rewritten form of (P, q) is (P^{om}, q^{om}):

(o-m1) $sgv^{bf}(x, y) \leftarrow input\_sgv^{bf}(x), flat(x, y)$

(o-m2) $sgv^{bf}(x, y) \leftarrow input\_sgv^{bf}(x), up(x, z1), sgv^{bf}(z1, z2), flat(z2, z3), sgv^{bf}(z3, z4), down(z4, y)$

(o-i2.2) $input\_sgv^{bf}(z1) \leftarrow input\_sgv^{bf}(x), up(x, z1)$

(o-i2.4) $input\_sgv^{bf}(z3) \leftarrow input\_sgv^{bf}(x), up(x, z1), sgv^{bf}(z1, z2), flat(z2, z3)$

(o-seed) $input\_sgv(a) \leftarrow$

(o-query) $query(y) \leftarrow sgv^{bf}(a, y)$

Intuitively, the original magic set transformation uses the relations $input\_R^{\gamma}$, but not supplementary relations.

(a) Verify that (P^{om}, q^{om}) is equivalent to (P, q).

(b) Compare the family of facts computed during the executions of (P^m, q^m) and (P^{om}, q^{om}).

(c) Give a specification for the original magic set transformation, applicable to any datalog query.

★ **Exercise 13.19** Consider the adorned rule

$$R^{bbf}(x, y, z) \leftarrow T_1^{bf}(x, s), T_2^{bf}(s, t), T_3^{bf}(y, u), T_4^{bf}(u, v), T_5^{bbf}(t, v, z).$$

A *sip graph* for this rule has as nodes all atoms of the rule and a special node *exit*, and edges (R, T_1), (T_1, T_2), (R, T_3), (T_3, T_4), (T_2, T_5), (T_4, T_5), $(T_5, exit)$. Describe a family of supplementary relations, based on this sip graph, that can be used in conjunction with the QSQ and magic set approaches. [Use one supplementary relation for each edge (corresponding to the output of the tail of the edge) and one supplementary relation for each node except for R (corresponding to the input to this node—in general, this will equal the join of the relations for the edges entering the node).] Explain how this may increase efficiency over the left-to-right approach used in this chapter. Generalize the construction. (The notion of sip graph and its use is a variation of [BR91].)

♠ **Exercise 13.20** [Ull89a] Specify an algorithm that replaces a program and query by an equivalent one, all of whose *idb* subgoals are rectified. What is the complexity of this algorithm?

♠ **Exercise 13.21**

(a) Provide a more detailed specification of the QSQ framework with annotations, and prove its correctness.

(b) [Ull89b, Ull89a] State formally the definitions needed for Theorem 13.4.1, and prove it.

Exercise 13.22 Write a program using counting that can be used to answer the RSG query presented at the beginning of Section 13.2.

(c-s1.1) $count\_sgv^{bf}(I, K, L, y) \leftarrow count\_input\_sgv^{bf}(I, K, L, x), flat(x, y)$

(c-s2.1) $count\_sup_1^2(I, K, L, z1) \leftarrow count\_input\_sgv^{bf}(I, K, L, x), up(x, z1)$

(c-s2.2) $count\_sup_2^2(I, K, L, z2) \leftarrow count\_sup_1^2(I, K, L, z1),$
$count\_sgv^{bf}(I + 1, 2K + 2, 5L + 2, z2)$

(c-s2.3) $count\_sup_3^2(I, K, L, z3) \leftarrow count\_sup_2^2(I, K, L, z2), flat(z2, z3)$

(c-s2.4) $count\_sup_4^2(I, K, L, z4) \leftarrow count\_sup_3^2(I, K, L, z3),$
$count\_sgv^{bf}(I + 1, 2K + 2, 5L + 4, z4),$

(c-s2.5) $count\_sgv^{bf}(I, K, L, y) \leftarrow count\_sup_4^2(I, K, L, z4), down(z4, y)$

(c-i2.2) $count\_input\_sgv^{bf}(I + 1, 2K + 2, 5L + 2, z1)$
$\leftarrow count\_sup_1^2(I, K, L, z1)$

(c-i2.4) $count\_input\_sgv^{bf}(I + 1, 2K + 2, 5L + 4, z3)$
$\leftarrow count\_sup_3^2(I, K, L, z3)$

(c-seed) $count\_input\_sgv^{bf}(1, 0, 0, a) \leftarrow$

(c-query) $query(y) \leftarrow count\_sgv^{bf}(1, 0, 0, y)$

Figure 13.9: Generalized counting transformation on SGV query

★ **Exercise 13.23** [BR91] This exercise illustrates a version of counting that is more general than that of Exercise 13.22. Indexed versions of predicates shall have three index coordinates (occurring leftmost) that hold:

(i) The level in the proof tree of the subgoal that a given rule is expanding.

(ii) An encoding of the rules used along the path from the root of the proof tree to the current subgoal. Suppose that there are k rules, numbered $(1), \ldots, (k)$. The index for the root node is 0 and, given index K, if rule number i is used next, then the next index is given by $kK + i$.

(iii) An encoding of the atom occurrence positions along the path from root to the current node. Assuming that l is the maximum number of *idb* atoms in any rule body, this index is encoded in a manner similar to item (ii).

A counting version of the SGV query of Exercise 13.14 is shown in Fig. 13.9. Verify that this is equivalent to the SGV query in the case where there are no cycles in *up* or *down*.