COMPILING SEPARABLE RECURSIONS

Jeffrey F. Naughton

CS-TR-140-88

March 1988

Compiling Separable Recursions

Jeffrey F. Naughton
Princeton University

March 21, 1988

Abstract

In this paper we consider evaluating queries on relations defined by a combination of recursive rules. We first define separable recursions. We then give a specialized algorithm for evaluating selections on separable recursions. Like the Magic Sets and Counting algorithms, this algorithm uses selection constants to avoid examining irrelevant portions of the database; however, on some simple recursions this algorithm is $O(n)$, whereas the Magic Sets algorithm is $\Omega(n^2)$ and the Generalized Counting Method is $\Omega(2^n)$.

1 Introduction

Evaluation algorithms for queries on recursively defined relations can be divided into two classes: those intended for a specific class of recursion, and those intended for general recursions. This paper continues research into algorithms for special classes of recursions. Developing specialized algorithms in addition to general algorithms will be a "win" if the classes for which we have specialized evaluation algorithms are frequently used, and the specialized evaluation algorithms are better than the general algorithms on these recursions.

In this paper we identify a class of recursions called "separable recursions." While it is impossible to be certain how frequently separable recursions will appear in as-yet unavailable systems, initial studies of logic programs suggest that they will be common. Furthermore, for selections on separable recursions, our evaluation algorithm can outperform popular general evaluation algorithms by a factor proportional to the size of the database.

Separable recursions are a subset of linear recursions, and include non-chain rules and non-binary predicates. Section 2 gives a precise definition of separable recursions; here we present two examples.

Example 1.1 Suppose that we have a relation $perfectFor(X, Y)$ of people X and products Y such that Y is perfect for X, and also two relations $friend(X, Y)$ and $idol(X, Y)$ of people X and their friends and idols Y. Furthermore, suppose that a person will buy a product if it is perfect for them, or if their friend or idol has bought it. Then the following recursion defines a relation $buys(X, Y)$ of people X and products Y that they buy.

r_1: $buys(X,Y)$:- $friend(X,W)$ & $buys(W,Y)$.
r_2: $buys(X,Y)$:- $idol(X,W)$ & $buys(W,Y)$.
r_3: $buys(X,Y)$:- $perfectFor(X,Y)$.
This is a separable recursion. ∎

Example 1.2 Suppose now that someone will no longer buy a product if their idol buys it, but that they will buy a product if it is cheaper than another product they will buy. Assuming that we have a relation $cheaper(X,Y)$ of products X and Y such that X is cheaper than Y, we can now define the relation *buys* as follows:

r_1: $buys(X,Y)$:- $friend(X,W)$ & $buys(W,Y)$.
r_2: $buys(X,Y)$:- $buys(X,W)$ & $cheaper(Y,W)$.
r_3: $buys(X,Y)$:- $perfectFor(X,Y)$.
This is also a separable recursion. ∎

Two currently popular evaluation algorithms are Magic Sets [BMSU86,BR87] and Counting [BMSU86,BR87,SZ86]. Section 4 discusses the relative worst case performance of Magic Sets, Counting, and Separable on some general classes of separable recursions. That section also shows that if n is the number of distinct constants in the base relations mentioned in the recursion, on the query *buys(tom,Y)?* on the relation *buys* defined in Example 1.2, the Generalized Magic Sets algorithm generates relations of size is $\Omega(n^2)$. For the same query on the relation *buys* defined in Example 1.1, the Generalized Counting Method generates relations of size $\Omega(2^n)$. The evaluation algorithm presented here is $O(n)$ for both queries.

In related work, Beeri et al. [BKBR87] suggest that an evaluation algorithm uses a selection effectively if it reduces the arity of the recursion, and prove that a selection on a binary chain rule program can be replaced by a monadic recursion if and only if the underlying language for the program is regular. For binary chain programs that are separable, this work provides an algorithm to do so. However, there are separable recursions that are not binary chain programs; for such programs the connection between the two papers is less obvious.

Chang [Cha81] presents an algorithm for evaluating queries on relations defined by a system of regular chain rules. As separable recursions do not include mutually recursive predicates, but do include non-chain rules, the class of programs to which Chang's algorithm applies and the class to which our algorithm applies are incommensurate. Chang's algorithm requires that the base relations be acyclic, and is $\Omega(n^2)$ on the queries in Examples 1.1 and 1.2. Minker and Nicolas [MN82] note that Chang's algorithm can be extended to some nonlinear chain rule recursions.

Henschen and Naqvi [HN84] note that for some special cases of single rule linear recursions (no "induced part" or no "determined part") their general algorithm can be simplified. Han and Henschen [HH87] mention a similar algorithm for computing selections on transitive closure queries. The separable recursion evaluation algorithm reduces to this simplification for these special cases, but is more general in that it applies to multiple rule recursions. The general Henschen and Naqvi algorithm [HN84] fails for cyclic data and, like generalized counting, is $\Omega(2^n)$ on the query in Example 1.1.

The one-sided recursion evaluation algorithm in Naughton [Nau87] and the separable algorithm are identical for separable single rule recursions. However, as not all one-sided recursions are separable and not all separable recursions are one-sided, the one-sided evaluation algorithm and the separable evaluation algorithm apply to different classes of programs.

Aho and Ullman [AU79] present a technique of pushing selections into fixpoints that, when combined with semi-naive evaluation, produces an instance of our algorithm if the selection is on a "stable" variable and the recursion is separable. As their algorithm applies to recursions that are not separable but not to all selections on separable recursions, their algorithm also applies to a set of programs and queries incommensurate with that to which our separable recursion evaluation algorithm applies.

As the algorithm presented here applies to a special class of recursions, it must supplement more general algorithms such as Generalized Magic Sets rather than replace them. However, because of its superior performance on queries on separable recursions, and because it is computationally simple to detect separable recursions, we expect that this evaluation algorithm will be a useful component of a recursive query processor.

2 Definitions

We consider queries on relations defined by function-free pure horn clause programs. We use Prolog syntax, and require that the heads of the rules contain no repeated variables and no constants. (The effect of both repeated variables and constants in rule heads can be handled by adding equalities to the rule bodies.) The predicates are divided into two types: IDB predicates, which appear in the head of some rule, and EDB predicates, which appear in the head of no rule and are defined by their extent.

If the predicate appearing in the rule head appears exactly once in the rule body, then the rule is *linear recursive*. The *definition* of an IDB predicate t is the set of all rules in which t appears in the head. In this paper we consider queries on relations defined by one or more linear recursive rules r_1 through r_n. We assume without loss of generality that each definition also contains a single nonrecursive rule r_e. Furthermore, we assume that the definitions of the predicates other than t do not depend on t — that is, they are not mutually recursive with t. We will call any predicate other than t a *base* predicate.

A *query* is some predicate instance, possibly containing variables and constants. The evaluation algorithm must return the set of all tuples of values for the variables that make the query true. In this paper we consider queries in which at least one argument of the query predicate is a constant.

The *expansion* of a predicate t is the set of all conjunctions of EDB predicates that can be generated by some sequence of rule applications beginning with applying some rule to t. To "apply" a rule to a conjunction of predicate instances, choose some predicate instance in the conjunction and some rule with a head that unifies with that predicate instance. Then replace the chosen predicate instance with the body of the chosen rule after the most general unifier has been applied. For recursive predicates, the expansion is infinite.

For the recursions considered in this paper, the expansion can be generated by procedure Expand in Figure 1. The input to that procedure is some recursion, and an instance of the recursive predicate t; the output is the infinite set S, the expansion of that recursion.

```
1)   Give all variables in rules subscript 0;
2)   S := ∅;
3)   Fringe := {t};
4)   while true do
5)        NewFringe := ∅;
6)        for each element f of Fringe do
7)             S := S ∪ {f with r_e applied};
8)             for i := 1 to n do
9)                  NewFringe := NewFringe ∪ {f with r_i applied};
10)            end;
11)       end;
12)       increment subscripts in all rules;
13)       Fringe := NewFringe;
     endwhile;
```

Figure 1: Procedure Expand

The elements of an expansion are conjunctive queries, and will be called *strings*. If a variable V appears in the initial instance of t, then V is a *distinguished* variable; otherwise, it is *nondistinguished*. By our assumption that the heads of rules contain no repeated variables, the unifications in line 7 and 10 of Procedure Expand can always be done by replacing the variables in the heads of r_e or r_i by the corresponding variable in the instance of the recursive predicate in f. As $Fringe$ is initialized to the instance of t containing the distinguished variables, this implies that the distinguished variables will always appear in the elements of the expansion without subscripts, while the nondistinguished variables will always appear with subscripts.

If $V_1, \ldots, V_i$ are the distinguished variables, and $W_1, \ldots, W_j$ the nondistinguished variables, then the relation specified by the string $p_1 \ldots p_n$ is

$$\{(V_1, \ldots, V_i) | (\exists W_1, \ldots, W_j)(p_1 \wedge \ldots \wedge p_n)\}$$

The recursively defined relation is the union of the relations for the strings in the expansion.

Example 2.1 The expansion of the definition in Example 1.1 begins

$$p(X,Y),$$
$$f(X,W_0)p(W_0,Y),$$
$$i(X,W_0)p(W_0,Y),$$
$$f(X,W_0)f(W_0,W_1)p(W_1,Y),$$
$$f(X,W_0)i(W_0,W_1)p(W_1,Y),$$
$$i(X,W_0)f(W_0,W_1)p(W_1,Y),$$
$$i(X,W_0)i(W_0,W_1)p(W_1,Y),$$

∎

Definition 2.1 A predicate instance p_1 is *connected* to a predicate instance p_2 if p_1 shares a variable with p_2, or shares a variable with a predicate instance connected to p_2.

Definition 2.2 A subset of predicate instances C is a *maximal connected set* if

1. For every pair of predicate instances p_1 and p_2 in C, p_1 and p_2 are connected, and
2. No predicate instance in C shares a variable with any predicate instance not in C.

Example 2.2 The predicate instances in

$$a(X, Z_0)a(Z_0, Z_1)b(Z_1, Y)$$

form a maximal connected set of size 3. The instances in

$$a(X, Y)b(Y, Z)c(W)$$

form two maximal connected sets — one of size 2 containing instances of a and b, the other of size one containing the instance of c. ∎

Definition 2.3 Let r be a linear recursive rule and let t be the recursive predicate in r. Then r contains *shifting variables* if there is some variable X such that X appears in position p_1 in the instance of t in the head of r and in position p_2 in the instance of t in the body of r, where $p_1 \neq p_2$.

Definition 2.4 (Separable Recursions) Let t be defined by n recursive rules r_1 through r_n. Furthermore, let t_i^h be the argument positions of t such that in the instance of t at the head of rule r_i, each argument position in t_i^h shares a variable with a nonrecursive predicate in the body of r_i. Similarly, let t_i^b be the argument positions of t such that in the instance of t in the body of rule r_i, each argument position in t_i^b shares a variable with a nonrecursive predicate in the body of r_i. Then the definition of t is a *separable recursion* if

1. For $1 \leq i \leq n$, r_i has no shifting variables, and
2. For $1 \leq i \leq n$, $t_i^h = t_i^b$, and
3. For $1 \leq i \leq n$ and $i < j \leq n$, either $t_i^h = t_j^h$ or t_i^h and t_j^h are disjoint, and
4. For $1 \leq i \leq n$, removing the instance of t from the body of r_i leaves a maximal connected set.

Section 5 discusses what happens when each of these restrictions is removed. Note that Condition 3 of the above definition partitions the recursive rules into equivalence classes, where rules r_i and r_j are in the same equivalence class if $t_i^h = t_j^h$. These equivalence classes can be evaluated essentially independently; this is the motivation for the term "separable recursion." If a separable recursion has n such equivalence classes, we will denote these

equivalence classes e_i, for $1 \leq i \leq n$. The columns of t that share variables with nonrecursive predicate instances in equivalence class e_i will be denoted by $t|_{e_i}$.

In general, there may be columns of t that share variables with no equivalence class. These columns are denoted $t|_{pers}$, because the variables in these positions are *persistent*, that is, they always appear in the same position in the instances of t in *Fringe* throughout the expansion.

Example 2.3 In Example 1.1, $n = 2$. The recursive rules contain no shifting variables, so the recursion satisfies Condition 1 of Definition 2.4. Also, $buys_1^h = \{buys^1\}$, $buys_1^b = \{buys^1\}$, while $buys_2^h = \{buys^1\}$, and $buys_2^b = \{buys^1\}$, so that recursion satisfies Condition 2 of Definition 2.4. Condition 3 is satisfied since $\{buys_1^h\} = \{buys_2^h\}$. Finally, removing the instance of *buys* from r_1 and r_2 leaves a maximal connected set of size 1 in each case, so that recursion satisfies Condition 4 of Definition 2.4. This recursion has one equivalence class, e_1, containing rules r_1 and r_2. Here $t|_{e_1}$ is the first column of t, and $t|_{pers}$ is the second.

In Example 1.2, there are again no shifting variables, and we have $buys_1^h = \{buys^1\}$, $buys_1^b = \{buys^1\}$, while $buys_2^h = \{buys^2\}$, and $buys_2^b = \{buys^2\}$, so that recursion satisfies Conditions 1 and 2 of Definition 2.4. Also, $\{buys_1^h\}$ and $\{buys_2^h\}$ are disjoint, so that recursion satisfies Condition 3. Finally, removing the instance of *buys* from r_1 and r_2 again leaves a maximal connected set of size 1 in each case, so that recursion is also separable. This recursion has two equivalence classes, e_1 containing r_1, and e_2 containing r_2. Here $t|_{e_1}$ is the first column of t, while $t|_{e_2}$ is the second and $t|_{pers}$ is empty. ∎

In procedure Expand, the elements in *Fringe* grow from iteration to iteration as predicate instances are added by rule applications. For any predicate instance p in string of the expansion, there is an iteration i and a rule r such that p first appeared in an element of the fringe on iteration i, and appeared due to the application of the rule r. We say that p was *produced by r on iteration i.*

Definition 2.5 Let s be an element of the expansion of a separable recursion R, and let e_i be an equivalence class of R. Then $D(s)$, the *derivation of s*, is the sequence of rule applications that produced s. Similarly, $D_i(s)$, the projection of $D(s)$ onto e_i, is the subsequence of $D(s)$ formed by deleting from $D(s)$ all rule applications from equivalence classes other than e_i. Also, $D_i(s)_k$ is the first k rules of $D_i(s)$.

Definition 2.6 Let s be an element of the expansion of a separable recursion R, and let e_i be an equivalence class of R. Then $P_i(s)$, the projection of s onto $D_i(s)$, is the predicate instances of s that were produced by the application of rules from e_i. Also, let $P_i(s)_k$ be the predicates produced by the rules in $D_i(s)_k$.

The following theorem demonstrates an important property of separable recursions.

Theorem 2.1 *Let s and s' be two elements from the expansion of a separable recursion R, and let the equivalence classes of R be e_i, for $1 \leq i \leq n$. Furthermore, let $D_i(s) = D_i(s')$ for $1 \leq i \leq n$. Then s and s' define the same relation.*

Proof: The proof proceeds by demonstrating that there are containment mappings in both directions between s and s'. A *containment mapping* from s to s' is a mapping m from the variables of s to the variables of s' such that

- If V is a distinguished variable, then $m(V) = V$.
- If $p(V_1, \ldots, V_n)$ appears in s, then $p(m(V_1), \ldots, m(V_n))$ appears in s'.

Containment mappings were defined in [CM77,ASU79]; those papers also proved that two conjunctive queries q_1 and q_2 define the same relation if and only if there are containment mappings in both directions between the two queries.

We first show that for all i, $1 \leq i \leq n$, the predicate instances of s that were produced by the application of rules from e_i can be mapped to the predicate instances of s' that were produced by the application of rules from e_i. The proof is by induction on prefixes of $D_i(s)$.

For the base case, note that $D_i(s)_0$ is empty, so, vacuously, the predicate instances in $D_i(s)_0$ can be mapped to the predicate instances in $D_i(s')_0$. Next, suppose that the predicate instances in $D_i(s)_{k-1}$ can be mapped to the predicate instances in $D_i(s')_{k-1}$. Consider the case $D_i(s)_k$, where $k > 0$.

Let m_{ik-1} be the containment mapping from the predicate instances produced by the first $k-1$ rule applications in $D_i(s)$ to the predicate instances produced by the first $k-1$ rule applications in $D_i(s')$. Extend m_{ik-1} to m_{ik} as follows:

- For any variable V that appears in $P_i(s)_{k-1}$ and also in the predicate instances produced by rule k of $D_i(s)$, let $m_{ik}(V) = m_{ik-1}(V)$.
- For any variable W_p that appears for the first time in instances produced by rule application k of $D_i(s)$, let $m_{ik}(W_p) = W_{p'}$, where $W_{p'}$ is the corresponding variable in the predicate instance produced by rule application k of $D_i(s')$.

We now show that m_{ik} defined in this manner is a containment mapping from $P_i(s)_k$ to $P_i(s')_k$.

Consider the predicate instances produced by rule k of $D_i(s)$ and $D_i(s')$. (By the assumption that $D_i(s) = D_i(s')$, rule k is the same in each.) Let f_k be the element of *Fringe* to which rule k of $W_i(s)$ was applied, and let f'_k be the element of *Fringe* to which rule k of $D_i(s')$ was applied. Also, let rule k of D_i be applied on on iteration p of Procedure Expand, and let rule k of $D(s')$ be applied on iteration p' of Procedure Expand.

Any variable V that appears in $P_i(s)_{k-1}$ and also in the predicate instances produced by rule k must appear in $t|_{e_i}$ in the instance of t in f_k before rule k is applied. Note that, by definition of Separable recursion, only the application of rules from e_i can change variables in positions of $t|_{e_i}$; rules from other equivalence classes simply use the identity substitution on variables in $t|_{e_i}$. Thus V

appears in the same position in f_k as it did immediately after the application of rule r_{k-1} to f_{k-1}.

Now consider the variable $m_{ik-1}(V)$. The assumption that m_{ik-1} is a containment mapping from $P_i(s)_{k-1}$ to $P_i(s')_{k-1}$ implies that $m_{ik-1}(V)$ must appear in the positions of $t|_{e_i}$ in f'_{k-1} in which V appears in $t|_{e_i}$ in f_{k-1}. By the arguments of the preceding paragraph, V and $m_{ik-1}(V)$ must also appear in the same positions in f_k and f'_k.

On iteration p, the variables in rule k have subscripts p. In the unification necessary to apply rule k, the variables that appear in the head of rule k will be replaced by the corresponding variables in $t|_{e_i}$ in f_k. Hence the predicate instances added to $P_i(s)_k$ by the application of rule k will be the predicate instances from the body of rule k, with variables that appear in the head of the rule replaced by the variables in $t|_{e_i}$ of f_k, and variables that do not appear in the head subscripted by p. Similarly, the predicate instances added to $P_i(s')_k$ by the application of rule k will be the predicate instances from the body of rule k, with the variables that appear in the head of the rule replaced by the variables in $t|_{e_i}$ in f'_k, and variables that do not appear in the head of the rule subscripted by p'.

Hence all variables W_p that do not appear in the head of rule k can be consistently mapped by setting $m_{ik}(W_p) = W_{p'}$.

Now consider the variables that do appear in the head. There are two cases to consider: if $k = 1$, then no rule from e_i has been applied, and the positions $t|_{e_i}$ in both s and s' contain the distinguished variables that appear in those positions in the original instance of t. These variables haven't yet appeared in any previously produced predicate instances of $P_i(s)$ (there are none), hence for each such variable we set $m_{i0}(W_p) = W_{p'}$, which reduces to $m_{i0}(V) = V$. If $k > 1$, then we know that V and $m_{ik-1}(V)$ appear in the same positions in f_k and f'_k, so they will appear in the same positions in the predicate instances produced by applying rule k. Hence the mapping is consistent, that is, it maps predicate instances in $P_i(s)$ to predicate instances in $P_i(s')$.

We still must show that if V is a distinguished variable appearing in $P_i(s)$, then $m_i(V) = V$. First, note that by definition of distinguished variable, if a distinguished variable appears in predicate instances added by rule k of $D_i(s)$, it must do so through being substituted for some variable in the head of the rule. Hence $m_{i1}(V) = V$, and $m_{ik}(V) = m_{ik-1}(V)$ if $k - 1 > 0$. By induction, this shows that if V is a distinguished variable, then $m_{ik}(V) = V$, as required.

Up to this point, we have shown that, for each i, there is a containment mapping m_i from $P_i(s)$ to $P_i(s')$. Now consider the mapping m_{total} defined as

- If V appears in predicate instances produced by a rule of equivalence class e_i, then $m_{total}(V) = m_i(V)$.
- If V appears only in the instance of t_0 (the only remaining possibility), then $m_{total}(V) = V$.

We claim that m_{total} is a containment mapping from s to s'.

By definition of m_i, for each i the mapping m_{total} is a containment mapping from $P_i(s)$ to $P_i(s')$. By the definition of separable recursion, the variables appearing in $P_i(s)$ and $P_j(s)$ are disjoint when $i \neq j$. Hence m_{total} is a containment mapping from the predicates in $\cup_{i=1}^{n} P_i(s)$ to the predicates in $\cup_{i=1}^{n} P_i(s')$. Now consider the instance of t_0.

By an easy induction, if some variable V appears in position p of t_0 in s, and V also appears in $P_i(s)$ for some i, then $m_i(V)$ appears in position p in s'. Similarly, if V appears in position p in t_0 but not in $P_i(s)$ for any i, then V appears in position p in s'. Hence m_{total} is a containment mapping from the instance of t_0 in s to the instance in s'.

This shows that there is a containment mapping from s to s'. By interchanging s and s' in the proof above, we get that there is a containment mapping from s' to s as well. Hence s and s' must define the same relation. ∎

We conclude this section with a definition of *full selections*. Intuitively, a full selection binds every column of some equivalence class. As Lemma 2.1 shows, we need only consider full selections in the Separable evaluation algorithm.

Definition 2.7 A selection on a separable recursion is a *full selection* if either the query predicate contains a constant in $V(t|_{pers})$, or there is at least one equivalence class e_i such that in the query predicate, all variables in $V_h(t|_{e_i})$ are replaced by constants.

Lemma 2.1 *Any selection on a separable recursion can be evaluated by taking the union of full selections.*

Proof: Clearly any full selection can be evaluated by a union of full selections. Consider an arbitrary selection Q that is not a full selection on an arbitrary separable recursion R defining a recursive predicate t. We can write R as

r_{11}: $t :- t, a_{11}.$
r_{12}: $t :- t, a_{12}.$
.
.
r_{1m_1}: $t :- t, a_{1m_1}.$
r_{21}: $t :- t, a_{21}.$
r_{22}: $t :- t, a_{22}.$
.
.
r_{2m_2}: $t :- t, a_{2m_2}.$
.
.
r_{n1}: $t :- t, a_{n1}.$
r_{n2}: $t :- t, a_{n2}.$

.

.

r_{nm_m}: $\quad t :- t, a_{nm_n}.$

where in general the a_{ij} are conjunctions of base predicate instances, and the equivalence classes of this recursion are $e_1 = \{r_{11}, r_{12}, \ldots, r_{1m_1}\}$, $\ldots$, $e_n = \{r_{n1}, r_{n2}, r_{nm_n}\}$.

By definition of full selection, since Q is not full, there must be some equivalence class e_i such that a proper subset of the variables in $V_h(t|_{e_i})$ are replaced by constants. Assume without loss of generality that this equivalence class is e_1. Construct two new recursive predicates t^{full} and t^{part}, where these predicates appear nowhere else in R. Replace R by the two recursions

r_{11}^{full}: $\quad t^{full} :- t^{full}, a_{11}.$
r_{12}^{full}: $\quad t^{full} :- t^{full}, a_{12}.$

.

.

$r_{1m_1}^{full}$: $\quad t^{full} :- t^{full}, a_{1m_1}.$
r_{21}^{full}: $\quad t^{full} :- t^{full}, a_{21}.$
r_{22}^{full}: $\quad t^{full} :- t^{full}, a_{22}.$

.

.

$r_{2m_2}^{full}$: $\quad t^{full} :- t^{full}, a_{2m_2}.$

.

.

r_{n1}^{full}: $\quad t^{full} :- t^{full}, a_{n1}.$
r_{n2}^{full}: $\quad t^{full} :- t^{full}, a_{n2}.$

.

.

$r_{nm_n}^{full}$: $\quad t^{full} :- t^{full}, a_{nm_n}.$
r_e^{full}: $\quad t^{full} :- t_0.$

and

r_{21}^{part}: $\quad t^{part} :- t^{part}, a_{21}.$
r_{22}^{part}: $\quad t^{part} :- t^{part}, a_{22}.$

.

.

$r_{2m_2}^{part}$: $\quad t^{part} :- t^{part}, a_{2m_2}.$

.

.

r_{n1}^{part}: $\quad t^{part} :- t^{part}, a_{n1}.$
r_{n2}^{part}: $\quad t^{part} :- t^{part}, a_{n2}.$

.
.

$r^{part}_{nm_n}$: $\quad t^{part} :- t^{part}, a_{nm_n}.$

r^{part}_{e}: $\quad t^{part} :- t_0.$

where the variables in each rule in the new recursions are the same as the variables in the corresponding rules in the original recursion. Finally, add the rules

$t :- t^{part}.$
$t :- t^{full}, a_{11}.$
$t :- t^{full}, a_{12}.$
.
.
$t :- t^{full}, a_{1m_1}.$

where the variables in t^{part} in the first rule are the same as the variables in t in the heads of the original recursion, and the variables in t^{full} and the a_{1j} in each of the remaining rules are the same as the variables in the corresponding rules of the original recursion.

We now prove that the new recursion R' computes the same relation for t as the original recursion R. By Theorem 2.1, all that is necessary is to show that for each element s of the expansion of R, there is an element s' in R' such that, for $1 \leq i \leq n$, $D_i(s) = D_i(s')$, and vice-versa.

Let s be an arbitrary string in the expansion of R. Suppose that $D_1(s)$ is empty. By definition of $D_1(s)$, this implies that s was produced without any applications of rules from e_1. Now consider s' in the expansion of R' produced by first applying the rule $t :- t^{part}$, and then applying the sequence of rules $D(s')$ where $D(s')$ is defined by replacing r_{ij} in $D(s)$ by r^{full}_{ij}. As r_{ij} and r^{full}_{ij} are identical, s and s' will be identical, hence s and s' define the same relation.

Now suppose that $D_1(s)$ is not empty. Let r_{1j} be the first rule application in $D_1(s)$. Consider the string s' produced by first applying the rule $t :- t^{part}, a_{1j}$, then applying the sequence of rules $D(s')$ where $D(s')$ is defined by deleting the first rule application from $D_1(s)$, then replacing r_{ij} in $D(s)$ by r^{part}_{ij}. As r_{ij} and r^{part}_{ij} are identical, and the rule $t :- t^{part}, a_{1j}$ is the same as the first rule application in $D_1(s)$, $D_i(s) = D_i(s')$. Then by Theorem 2.1, s and s' define the same relation.

Hence we have shown that for every string in the expansion of R, there is a string in the expansion of R' that defines the same relation. Now consider an arbitrary string s' in the expansion of R'. Suppose that s' was produced by first applying the rule $t :- t^{part}$, then applying the sequence of rules $D(s')$. Then the string s in the expansion of R, where $D(s) = D(s')$ with r^{part}_{ij} replaced by r_{ij}, defines the same relation.

Now suppose that s' was produced by the sequence of rules $D(s')$, where the first rule in $D(s')$ is the rule $t :- t^{full}, a_{1j}$. Consider the string s in the expansion of R, where s was produced by the sequence of rule applications $r_{1j}D(s)$, where $D(s)$ is $D(s')$ with r_{ij}^{full} replaced by r_{ij}. For $1 \leq i \leq n$ we have $D_i(s) = D_i(s')$, so again s and s' define the same relations.

Thus the new recursion and the original recursion define the same relations. To complete the proof, note that a the partial selection on t gets transformed (via sideways information passing) to full selections on t^{full} and t^{part}. In more detail, through sideways information passing the original selection binds all columns of $t|_{e_1}$ in t^{full}, while the original selection is on a variable in $t^{part}|_{pers}$, so both are full selections. ∎

Example 2.4 If $t|_{e_i}$, for all equivalence classes e_i of a recursion, consists of a single column, any selection on t will be a full selection. This is the case for the recursions in Examples 1.1 and 1.2. Let R be the recursion

$t(X,Y,Z) :- a(X,Y,U,V), t(U,V,Z).$
$t(X,Y,Z) :- t(X,Y,W), b(W,Z).$
$t(X,Y,Z) :- t_0(X,Y,Z).$

and the query $t(c,Y,Z)?$. Here there are two equivalence classes. The class e_1 consists of the first rule, and $t|_{e_1}$ consists of the first two columns of t. The class e_2 consists of the second rule, and $t|_{e_2}$ consists of the third column of t. The selection is not a full selection, as it only binds one of the two columns of $t|_{e_1}$. The recursion can be rewritten as

$t^{part}(X,Y,Z) :- t^{part}(X,Y,W), b(W,Z).$
$t^{part}(X,Y,Z) :- t_0(X,Y,Z).$

$t^{full}(X,Y,Z) :- a(X,Y,U,V), t^{full}(U,V,Z).$
$t^{full}(X,Y,Z) :- t^{full}(X,Y,W), b(W,Z).$
$t^{full}(X,Y,Z) :- t_0(X,Y,Z).$

$t(X,Y,Z) :- t^{part}(X,Y,Z).$
$t(X,Y,Z) :- a(X,Y,U,V), t^{full}(U,V,Z).$

Note that the original selection, $t(c,Y,Z)?$, is passed unchanged to the query $t^{part}(c,Y,Z)?$, which is a full selection, as the first column of t^{part} belongs to $t^{part}|_{pers}$. The original selection is passed through the instance of $a(X,Y,U,V)$ to give bindings for the first two columns of t^{full}, which completely binds $t^{full}|e_1$, so that is also a full selection. Hence the original selection is transformed to a union of full selections. ∎

3 Algorithms

3.1 Detection Algorithms

If an algorithm for a subset of recursive definitions is to be practical, we must have an efficient way of determining whether or not a given recursion falls into the class. To decide

if a recursion is a separable recursion, we have to verify each of the conditions of Definition 2.4. Below we give upper bounds on the time required to do so using straightforward algorithms.

Let r be the number of rules in the recursion, k be the maximum over all predicates occuring in the recursion of the number of arguments of a predicate, and let l be the maximum over all rules of the number of predicates in the rule body.

Condition 1 can be verified in time $O(k^2r)$ by, for each rule, checking each variable in the head to see where it appears (if at all) in the instance of the recursive predicate in the body. Similarly, condition 2 can be verified in time $O(k^2lr)$ by, for each i, first identifying t_i^h ($O(k^2l)$), then identifying t_i^b (also $O(k^2l)$), and finally checking that they are equal ($O(k^2)$.)

Once t_i^h and t_i^b have been identified, for $1 \leq i \leq n$, then condition 3 can be verified in $O(k^2r^2)$ by checking, for each i and for all $j \neq i$, that either $t_i^h = t_j^h$ or t_i^h and t_j^h are disjoint. Finally, condition 4 can be verified in time $O(rk^2l^2)$ by, for each i and each predicate instance p in r_i, checking each argument of p to see if it shares a variable with some other predicate instance p' in r_i.

Clearly, the above can be improved upon. The important thing to notice, however, is that the parameters r, k, and l refer to the rules and predicates in the recursion, not to the database. If we let n be the size of the database, in general we expect n will be much large than r, l, or k. Hence the time to evaluate the query, which is proportional to n, will dominate the time to verify that the recursion is separable. Put another way, spending time that is a small polynomial in the size of the query to deduce that an algorithm that is $O(n)$ can be used instead of one that is $O(n^2)$ or even $O(2^n)$ will be a "win" in almost all cases.

3.2 Motivation for Evaluation Algorithm

In this subsection we present an algorithm to evaluate queries of the form "column = constant" on relations defined by separable recursions. We begin with a motivating abstract example. Consider the recursion

$t(X,Y) :- a_1(X,W)\ \&\ t(W,Y).$
$t(X,Y) :- a_2(X,W)\ \&\ t(W,Y).$
$t(X,Y) :- t(X,W)\ \&\ b_1(W,Y).$
$t(X,Y) :- t(X,W)\ \&\ b_2(W,Y).$
$t(X,Y) :- t_0(X,Y).$

This is a separable recursion. Ignoring variables, the elements of the expansion can be described by the regular expression $(a_1 + a_2)^* t_0 (b_1 + b_2)^*$.

Suppose that we wish to evaluate the query $t(x_0, Y)$?. Then, after substituting x_0 for X in each string, the strings of the expansion can be evaluated from left to right, using the selection constant to restrict the first lookup and shared variables to restricted subsequent lookups. The answer to the query is the union of the values for Y, which will appear in column 2 of the last predicate of the string.

We will describe our algorithms using Datalog notation rather than relational algebra. For example, instead of writing $p := \pi_{1,3}(p \bowtie_{2=1} q)$ we will write $p(X,Y) :=$

$p(X,W)$ & $q(W,Y)$. This is not a Datalog rule; in particular, it is not a fixpoint equation. The right side of the statement is evaluated, and assigned to the left side, once. For convenience we extend the notation to allow unions and differences on the right side. For example, $p(X,Y) := q(X,Y) \cup r(X,Y)$ assigns the union of the relations q and r to p.

Every value that could be passed to t_0 in the evaluation of some string of the expansion can be found by the following algorithm:

$carry_1(x_0)$;
$seen_1(X) := carry_1(X)$;
while $carry_1$ not empty do
    $carry_1(W) := carry_1(X)$ & $a_1(X,W) \cup carry_1(X)$ & $a_2(X,W)$;
    $carry_1(X) := carry_1(X) - seen_1(X)$;
    $seen_1(X) := seen_1(X) \cup carry_1(X)$;
endwhile;

In the above, $carry_1$ and $seen_1$ are unary, relation-valued variables. At the end of the algorithm, $seen_1$ contains all values that will be passed to t_0 in any string of the expansion.

By taking $carry_2(W)$:=$seen_1(X)$ & $t_0(X,W)$, we get all values that can be passed from t_0 to an instance of b_1 or b_2 in any string of the expansion. Thus the following algorithm finds all values that appear at the right end of some string (the Y values):

$carry_2(W) := seen_1(X)$ & $t_0(X,W)$;
$seen_2(W) := carry_2(W)$;
while $carry_2$ not empty do
    $carry_2(Y) := carry_2(W)$ & $b_1(W,Y) \cup carry_2(W)$ & $b_2(W,Y)$;
    $carry_2(Y) := carry_2(Y) - seen_2(Y)$;
    $seen_2(Y) := seen_2(Y) \cup carry_2(Y)$;
endwhile;

Here $carry_2$ and $seen_2$ are unary, relation-valued variables. At the end of this algorithm, $seen_2$ will be the answer to the query.

There is no need to record how a tuple got into $seen_1$ or $seen_2$; this independence of the evaluation of the a and b parts of the recursion is crucial to the efficiency of the separable recursion evaluation algorithm. Note also that the above evaluation procedure only looks at tuples along a path from x_0, and it examines each tuple at most once.

3.3 The Separable Evaluation Algorithm

Figure 2 gives a general schema for evaluating a full selection on a separable recursion. The $carry_i$, the $seen_i$, and ans are relation-valued variables. The f_i, g_i, and h are relational operators. In addition to the arguments listed, the f_i, g_i, and h may involve relations and constants appearing in the rules and in the query.

We assume that the rules are "rectified," that is, they have been re-written so that heads of the rules are identical and contain no repeated variables. (The term "rectified"

```
1)  init carry_1;
2)  seen_1 := carry_1;
3)  while carry_1 not empty do
4)        carry_1 := f_1(carry_1);
5)        carry_1 := carry_1 − seen_1;
6)        seen_1 := seen_1 ∪ carry_1;
7)  endwhile;
8)  carry_2 := g_2(seen_1);
9)  seen_2 := carry_2;
10) while carry_2 not empty do
11)       carry_2 := f_2(carry_2);
12)       carry_2 := carry_2 − seen_2;
13)       seen_2 := seen_2 ∪ carry_2;
14) endwhile;
15) ans := seen_2;
```

Figure 2: A schema for evaluating single selections on separable recursions.

is from Ullman [Ull88].) Furthermore, we assume that the rules are re-written so that if $t_b^i = t_b^j$, then the variables in corresponding positions of t_b^i and t_b^j are identical.

Definition 2.4 partitions the rules of a separable recursion into equivalence classes, where rule r_i and r_j are in the same equivalence class if $t_i^h = t_j^h$. Let there be n equivalence classes $e_1, \ldots, e_n$. To describe how the schema in Figure 2 is instantiated, we will use the following notation:

- Let the rules in equivalence class e_k be r_{k1}, r_{k2}, ..., r_{km_k}. We represent the nonrecursive predicate instances in the body of the rule r_{ki} by a_{ki}.
- Let the predicate instance in the body of the nonrecursive rule be t_0.
- Recall that by definition of a separable recursion, for each rule r_{ki} in e_k, the argument positions $t_{ki}^b = t_{ki}^h$. Furthermore, by definition of e_k, for each pair of rules r_{ki} and r_{kj} in e_k, the argument positions $t_{ki}^h = t_{kj}^h$. We represent these positions by $t|_{e_k}$.
- For $1 \leq k \leq n$, let $V_h(t|_{e_k})$ be the variables that appear in $t|_{e_k}$ in the head of the rules in e_k, in the order in which they appear in $t|_{e_k}$ in the heads of the rules. (Because we assumed the recursion is rectified, this is well defined.)
- For $1 \leq k \leq n$, let $V_b(t|_{e_k})$ be the variables that appear in $t|_{e_k}$ in the body of the rules in e_k, in the order in which they appear in $t|_{e_k}$ in the bodies of the rules. (By our additional assumptions on variable renaming, this is also well defined.)
- Let $V(t|_{pers})$ be the variables that appear in $t|_{pers}$ in the heads of the rules.

- Let x_0 be the vector of selection constants, and let X be the vector of variables that appear in the selected-on columns in the heads of the rules.

We now describe how to instantiate the separable recursion evaluation schema for a full selection on a separable recursion.

If the selection constants appear in $t|_{pers}$, then replace lines 1-7 of Figure 2 with the single statement $seen_1(x_0)$ and create a dummy equivalence class e_1, setting $V(t|_{e_1}) = X$, the where X is the vector of variables replaced by the selection constants in the query. Also, delete the positions in which the variables X appeared from $t|_{pers}$, and renumber the other equivalence classes so there is only one e_1.

If the selection constants do not appear in $t|_{pers}$, then without loss of generality assume that the selection constants appear in $t|_{e_1}$. We create a pair of relation-valued variables $carry_1$ and $seen_1$. These relation variables have one column for each variable in $V_h(t|_{e_1})$. The carry initialization statement (line 1 in Figure 2) is the fact

$carry_1(V_h(t|_{e_1}))$.

with the variables in $V_h(t|_{e_1}))$ replaced by the corresponding selection constants. The carry extension operator f_1 (line 4 of Figure 2) is given by

$carry_1(V_b(t|_{e_1})) := a_{11}$ & $carry_1(V_h(t|_{e_1})) \cup \ldots \cup a_{1m_1}$&$carry_1(V_h(t|_{e_1}))$;

Next create another pair of relation-valued variables $carry_2$ and $seen_2$. These variables have one column for each variable in the concatenation of $V_h(t|_{e_2}), \ldots, V_h(t|_{e_n}), V(t|_{pers})$. The initialization of $carry_2$ is

$carry_2(V_h(t|_{e_2}), \ldots, V(t|_{e_n}), V(t|_{pers}) := t_0$ & $seen_1(V_h(t|_{e_1}))$;

The carry extension operator f_2 (line 11 of Figure 2) is given by

$carry_2(V_h(t|_{e_2}), \ldots, V_h(t|_{e_n}), V(t|_{pers})) :=$
$\bigcup_{i=2}^{n} \bigcup_{j=1}^{m_i} a_{ij}$&$carry_2(V_b(t|_{e_2}), \ldots, V_b(t|_{e_n}), V(t|_{pers}))$;

Finally, the answer operator h is given by

$$ans(V_h(t|_{e_2}), \ldots, V_h(t|_{e_n}), V(t|_{pers})) := seen_2(V_h(t|_{e_2}), \ldots, V_h(t|_{e_n}), V(t|_{pers}))$$

Example 3.1 On the query $buys(tom, ?)$ on the recursion of Example 1.1, the evaluation algorithm produced by instantiating the schema is shown in Figure 3.

For the same query on the recursion of Example 1.2, instantiating the schema produces the algorithm of Figure 4. ∎

3.4 Correctness of Separable

In this section we prove the following theorem:

Theorem 3.1 *The separable recursion evaluation algorithm produced by instantiating the schema in Figure 2 terminates and correctly evaluates any full selection on a separable recursion.*

$carry_1(tom)$;
$seen_1(W) := carry_1(W)$;
while $carry_1$ not empty do
    $carry_1(W) := carry_1(X)\ \&\ f(X,W)) \cup carry_1(X)\ \&\ i(X,W)$;
    $carry_1(W) := carry_1(W) - seen_1(W)$;
    $seen_1(W) := seen_1(W) \cup carry_1(W)$;
endwhile;
$seen_2(Y) := seen_1(X)\ \&\ p(X,Y)$;
$ans(Y) := seen_2(Y)$;

Figure 3: The instantiated separable recursion algorithm for Example 1.1.

$carry_1(tom)$;
$seen_1(W) := carry_1(W)$;
while $carry_1$ not empty do
    $carry_1(W) := carry_1(X)\ \&\ f(X,W)$;
    $carry_1(W) := carry_1(W) - seen_1(W)$;
    $seen_1(W) := seen_1(W) \cup carry_1(W)$;
endwhile;
$carry_2(Z) := seen_1(X)\ \&\ p(X,Y)$;
$seen_2(Z) := carry_2(Z)$;
while $carry_2$ not empty do
    $carry_2(Y) := carry_2(Z)\ \&\ c(Y,Z)$;
    $carry_2(Y) := carry_2(Y) - seen_2(Y)$;
    $seen_2(Y) := seen_2(Y) \cup carry_2(Y)$;
endwhile;
$ans(Y) := seen_2(Y)$;

Figure 4: The instantiated separable recursion algorithm for Example 1.2.

To prove this theorem we need some auxiliary definitions and lemmas. The first definition is that of the *justification* of an answer produced by Separable. Intuitively, a justification of an answer is a record of how the tuple came to appear in the *ans* relation through the course of the algorithm. More formally, the *justification J of a tuple a in ans* produced by an instantiation of the separable schema of Figure 2 is a string defined recursively as follows: The tuple a must have appeared in $seen_2$. There are two places at which a could have been added to $seen_2$ — either at line 11, or at line 8.

If a was added to $seen_2$ at line 11, then it must have been added by some term $a_{ij}\&carry_2$ of the operator f_2. This in turn implies that there was some tuple a' in $carry_2$ that joined with a_{ij} to produce a. In this case, define $J(a) = J(a')r_{ij}$.

If a was added to $seen_2$ at line 8, then there must have been some tuple a' in t_0 such that a' joined with a tuple b of $seen_1$ to produce a. Here we define $J(a) = J(b)$.

Finally, let b be some tuple in $seen_1$. The tuple b must have appeared in $carry_1$; either b was added to $carry_1$ through the initialization at line 1, or it was added by some term $a_{1j}\&carry_1$ of f_1 at line 4. In the first case, $J(b)$ is the empty string. In the second case, let b' be the tuple that joined with a_{1j} to produce b. Then $J(b) = J(b')r_{1j}$.

The elements of an expansion are unordered, but the predicate instances in an element can be ordered by applying the following rule in Procedure Expand: Whenever a rule from equivalence class e_1 is applied, add the instance of the recursive predicate to the right of the instances of the nonrecursive predicates in the rule. Whenever a rule from equivalence class e_i, where $i > 1$, is applied, add the instance of the recursive predicate to the left of the instances of the nonrecursive predicates in the rule. Thus in an element s of the expansion, the predicate instances due to rules from equivalence class e_1 appear to the left of the instance of t_0, and ordered from left to right in the the order in which they were added. The predicate instances due to rules from equivalence class e_i, where $i > 2$, appear to the right of t_0, in the reverse of the order in which they were added.

Once this order has been established, it makes sense to talk about evaluating the elements of the expansion "from left to right," using the shared variables from one predicate evaluation to restrict the lookup of the next. The values taken on by these shared variables are critical to the correctness of the algorithm.

For any string s, let $L_{1k}(s)$ be the relation of values found for variables shared between predicate instances produced by the application of rule k in $D_1(s)$ and the predicate instances produce by rule $k+1$ in $D_1(s)$ in a left-to-right evaluation of s.

Similarly, let $L_{rk}(s)$, for $i > 1$, be the relation of values found for the persistent variables and the variables shared between predicate instances produced by the application of rule k in $D(s) - D_1(s)$ and the predicate instances produced by the application of rule $k-1$ in $D(s) - D_1(s)$ in a left-to-right evaluation of s. ($D(s) - D_1(s)$ is simply $D(s)$ with all rules from e_1 deleted.)

Lemma 3.1 *Let a be an answer produced by applying the Separable algorithm to a separable recursion R and a full selection query Q. Then a is an answer to Q.*

> **Proof:** Consider an answer a with justification $J(a)$. Let s be the string in the expansion of R such that $D(s) = J(a)$. We show that a is the in the relation defined by s, after substituting the query constants from Q for the corresponding

variables in s. By definition of the expansion of a recursion, this implies that a is an answer to Q.

First we prove, by induction on the rules in $D_1(s)$, that any tuple appearing in $carry_1$ at line 3 in Figure 2 on iteration k of the while loop of lines 3–7 appears as a value in $L_{1k}(s)$ in the left-to-right evaluation of s.

The basis, $k = 0$, is trivial. Any tuple b appearing at line 3 on iteration 0 must have been added by the initialization statement $carry_1(c)$, where c is the vector of selection constants from Q. The variables in $L_{10}(s)$ are the variables that were replaced by constants in the query Q, so after the corresponding substitution in s, we have that c appears in $L_{10}(s)$.

If a tuple b appears at line 3 on iteration $k > 0$, it must have been added at line 4 on iteration $k - 1$. Suppose that b was added by the term $carry_1$ & a_{1j}. Then there must have been some tuple b' in $carry_1$ on iteration $k - 1$ such that b was produced by the join of b' and tuples of a_{1j}; by induction, b' appears in $L_{1k-1}(s)$.

By the way we chose s, the predicate instances produced by e_1 on iteration k were also instances of a_{1j}. Hence if b' appears in $L_{1k-1}(s)$, b will appear in $L_{1k}(s)$.

Let $|D_1(s)|$, the number of rules in $D_1(s)$, be k_1. Also, let $|D(s) - D_1(s)|$, the number of rules in $D(s)$ but not in $D_1(s)$, be k_2. Consider the initialization of $carry_2$. Let a' be the tuple, mentioned in the definition of the justification of an answer tuple, that was added to $carry_2$ at line 8 of Figure 2. We claim that a' is in the relation $L_{rk_2}(s)$.

The tuple a' must have been added to $carry_2$ at line 8 by the statement $carry_2$:=$seen_1$ & t_0. There must have been some tuple b in $seen_1$ that joined with a tuple in t_0 to produce a'. By the preceding induction, that tuple b appeared in $L_{1k_1}(s)$, so a' is also produced by evaluating the join of the predicate instances produced by rule k_1 of $D_1(s)$ and t_0, which again implies that a' is in $L_{rk_2}(s)$.

Finally, we show by induction that if a tuple a appears in $carry_2$ at line 10 of Figure 2 on iteration k, a also appears in $L_{r(k_2-k)}(s)$. The basis, $k = 0$, is given by the preceding paragraph.

If a tuple a appears at line 3 on iteration $k > 0$, it must have been added at line 11 on iteration $k - 1$. Suppose that a was added by the term $carry_2$ & a_{ij}. Then there must have been some tuple a' in $carry_2$ on iteration $k - 1$ such that a was produced by the join of a' and tuples of a_{ij}; by induction, a' appears in $L_{r(k_2-(k-1))}(s)$.

By the way we chose s, the predicate instances produced by rule k of $D(s) - D_1(s)$ were also instances of a_{ij}. Hence if a' appears in $L_{r(k_2-(k-1))}(s)$, a will appear in $L_{r(k_2-k)}(s)$.

To complete the proof we note that because Q is a full selection query, the relation for string s after the substitution of the selection constants of Q is the relation for $L_{r0}(s)$. Hence if a is an answer produced by separable, a is produced by the left-to-right evaluation of s. ∎

Definition 3.1 Let s be an element of the expansion of a separable recursion R, and consider a query Q on R over database EDB. Furthermore, let R have n equivalence classes e_1 through e_n. Then s is a *minimal element with respect to EDB* if, when evaluating s over EDB by left-to-right evaluation, for all k such that $1 \leq k \leq n$, whenever $i \neq j$, we have that $L_{ki}(s) \neq L_{kj}(s)$.

Lemma 3.2 *Let a be an answer to a query Q on R over EDB. Then there exists an s in the expansion of R such that s is minimal with respect to Q and EDB, and a is in the relation for s.*

Proof: By a splicing argument. By definition of answer and expansion, if an answer a appears in Q, it must be produced by some string s in the expansion of R. If s is not minimal, then by definition of minimal there must be i, j, where $i \neq j$, such that for some k, the relation for $L_{ki}(s)$ equals the relation for $L_{kj}(s)$. Consider the string s' defined such that $D(s')$ is just $D(s)$ with the rule applications from i up to (but not including) j from $D_k(s)$ removed. Clearly, s' is also in the expansion of R. We claim that a is also in the relation for s'.

If a is in the relation for s, then there must be an answer mapping h from the variables of s to constants in EDB such that

- If V is a distinguished variable, then $h(V) = c$, where c is the constant appearing in a in the position in which V appears in the original query.
- If $p(V_1, \ldots, V_n)$ appears in s, then the tuple $(h(V_1), \ldots, h(V_n))$ appears in the relation for p.

Consider the mapping h' from the variables of s' to the constants of the EDB defined as follows:

- If V is a distinguished variable, then $h'(V) = h(V)$.
- If V is a nondistinguished variable not appearing in $P_i(s)$, then $h'(V) = h(V)$.
- If V is a nondistinguished variable appearing in predicates produced by the first i rule applications in $D_k(s)$, then $h'(V) = h(V)$.
- If V_p is a nondistinguished variable appearing in predicates produced by a rule application $p > i$ of $D_k(s)$, then $h'(V_p) = h(V_{p+(j-i)})$.

As $h'(V) = h(V)$ for all distinguished variables V, then if h' is indeed a proper answer mapping it will demonstrate that a is an answer to s'.

The only place h' differs from h is on nondistinguished variables in $P_k(s)$ appearing due to rule applications after rule application i. The only place s'

differs from s is in $P_k(s)$ and $P_k(s')$ — that is, the predicate instance produced by rules of equivalence class e_k. Hence we must show that h' is a proper answer mapping on $P_k(s')$.

Distinguished variables either appear only in predicate instances produced by rule application 1, or are persistent and appear in the same positions in every predicate instance produced by a given rule in $D_k(s)$. (This follows from the "no shifting variables" assumption in the definition of Separable recursions.) Hence h' consistently maps distinguished variables.

For the nondistinguished variables, note that on variables appearing in predicate instances produced after rule application i of $D_k(s')$, h' is h shifted by $j-i$. But the predicate instances of s' appearing after those produced by rule application i are just those of s produced after rule application j, with the exception that the subscripts on nondistinguished variables have been decremented by $j-i$. Hence h' consistently maps predicate instances produced after rule application i in $P_k(s')$.

Thus h is an answer mapping that demonstrates that a is an answer of s', hence we've shown that if for some $i \neq j$, the relation for $L_{ki}(s)$ and $L_{kj}(s)$ are identical, then there is a shorter string s' that also returns a. This "splicing" can be repeated until we have a string that returns a but has no repeats. ∎

Lemma 3.3 *Let R be a separable recursion, let Q be a full selection query on R, and let s be an element in the expansion of R, with the constants of Q replacing the corresponding variables in s. Then if a is a tuple in the relation for s, a is in the answer returned by Separable.*

Proof: By Lemma 3.2, we need only consider minimal s. Let s have the derivation $D(s)$, and let a be a tuple in the relation for s. We must prove that a is in ans.

Suppose that for $1 \leq i \leq n$, the projection of $D(s)$ onto e_i is $D_i(s)$. Consider the string s' such that $D(s') = D_1(s)D_2(s)\ldots D_n(s)$. By Theorem 2.1, the a is in the relation for s if and only if a is in the relation for s'. The lemma is proven by demonstrating that the tuple a is in ans, with justification $J(a) = D(s')$.

After evaluating (from left-to-right) the prefix of s' given by predicate instances produced by the first k rule applications in $D_1(s')$, any tuple b in $L_{1k}(s')$ is a tuple in $carry_1$ with justification $J(b) = D_1(s')_k$. This is proven by an induction that mirrors that in the proof of Lemma 3.1. The basis is trivial — if $k = 0$, the relation $L_{10}(s)$ contains just the vector of selection constants c. This tuple will be added to $carry_1$ at line 1.

Now consider a tuple b in $L_{1k}(s')$. The tuple b must have been added to $L_{1i}(s')$ through a join of some a_{1j} and $L_{1(k-1)}$. Let b' be the tuple in $L_{1(k-1)}$ that participated in this join to produce b. By induction, b' appeared in $carry_1$ on iteration $k-1$ with justification $J(b') = D_1(s')_{k-1}$. Since s' is minimal, b' did not appear in $carry_1$ on any previous iteration, so b' is not removed by

the difference at line 5. Hence on iteration k, the join $carry_1$ & a_{1j} at line 4 produces b with justification $J(b')r_{1j} = D_1(s')_k$. This completes the induction.

Let $|D_1(s')| = k_1$, and let $|D(s') - D_1(s)| = k_2$. Then by the previous induction, any tuple b that appears in $L_{1k_1}(s')$ also appears in $carry_1$ on iteration k_1, hence also appears in $seen_1$. Therefore the join $seen_1$ & t_0 produces exactly the values for $seen_2$ that were added to L_{rk_2} in the left-to-right evaluation of s'.

We now show that if a tuple a appears in $L_{r(k_2-k)}(s')$, then a appears in $carry_2$ on iteration k of the second while loop of Figure 2. The basis, $k = 0$, is given be the previous paragraph.

Now consider a tuple a in $L_{r(k_2-k)}(s')$. The tuple a must have been added to $L_{r(k_2-k)}(s')$ through a join of some a_{ij} and $L_{r(k_2-(k-1))}$. Let a' be the tuple in $L_{r(k_2-(k-1))}$ that participated in this join to produce a. By induction, a' appeared in $carry_2$ on iteration $k-1$. Since s' is minimal, a' did not appear in $carry_2$ on any previous iteration, so a' is not removed by the difference at line 12. Hence on iteration k, the join $carry_2$ & a_{ij} at line 11 produces a. This completes the induction.

Because Q is a full selection, the relation for s' is just $L_{rk_2}(s')$, which by the above induction is $carry_2$ on iteration k_2. Hence if a is in the relation for s', a is in $carry_2$, hence a must appear in ans. ∎

Lemma 3.4 *The separable algorithm terminates for any query on any Separable recursion.*

Proof: Consider the instantiation of Separable for a query on an arbitrary recursion R. Let $carry_1$ have k_1 columns, and let $carry_2$ have k_2 columns. Then if n is the number of distinct constants appearing in the base relations of R, there are n^{k_1} possible tuples for $carry_1$, and n^{k_2} possible tuples for $carry_2$. In the while loops of the algorithm, no tuple ever appears in $carry_1$ or $carry_2$ at the top of the loop twice. (The first time a tuple appears in $carry_1$, it is added to $seen_1$; if it appears again, it is removed at line 5 by the difference operation. Similarly, repeated tuples are eliminated from $carry_2$ at line 12.) Hence the total number of iterations of the while loops is bounded by n^{k_1} and n^{k_2}, so the algorithm terminates. ∎

Now we can complete the proof of Theorem 3.1.

Proof: (Of Theorem 3.1) By Lemma 3.1, Separable only returns tuples that are indeed answers to the query. By Lemma 3.3, it returns all answers. Finally, by Lemma 3.4, it terminates. ∎

4 Comparison

There are a number of measures one can use to compare evaluation algorithms for recursive queries. An important measure is the "focus" of an algorithm, that is, how well it uses

selection constants to limit the portion of the database searched. Here we ignore this measure, as the three algorithms are equivalent in that respect. Instead, we focus on the size of the relations generated by each algorithm in the course of answering a query. In [Nau88] we give empirical average case performance figures for the evaluation algorithms on some representative recursions; here we state lemmas about the worst-case figures for classes of separable recursions.

Of course, we must be more precise about what we mean by "relations generated by an algorithm." The difficulty is that one can put an arbitrary project-join query into the body of a recursive rule. Clearly, we cannot put a bound on the size of relations constructed in solving an arbitrary project-join query. Because of this, we focus on the relations that the algorithms would produce even if the bodies of the recursive rule consisted only of a single, nonrecursive predicate, and the recursive predicate. For example, Separable generates $carry_1$, $carry_2$, $seen_1$, $seen_2$, and *ans*. Generalized Magic Sets constructs *magic* and t, the original recursive predicate. Generalized Counting produces *count* and a modified version of t.

Definition 4.1 Let S_p^k be the class of all separable recursions with p recursive rules and a recursive predicate of arity k.

Definition 4.2 Let R be a recursion, let Q be a query on the recursively defined relation of R, let n be the number of distinct constants in the base relations of R, and let M be a query evaluation method. Then we say that M is $O(f(n))$ on Q if, in evaluating Q, M constructs only relations of size $O(f(n))$. Similarly, we say that M is $\Omega(f(n))$ on Q if, in evaluating Q, M constructs at least one relation of size $\Omega(f(n))$.

Definition 4.3 Let R be a separable recursion, let Q be a selection query on the recursive predicate of R, and let e_1 be the equivalence class such that the constant(s) that appear in Q appear in the columns of $V(t|_{e_1})$ in R. The *width of* e_1, written $w(e_1)$, is the number of columns in $V(t|_{e_1})$.

Lemma 4.1 *For all $k > 0$ and $p > 0$, for any recursion R in S_p^k, on any full selection query Q on the recursive relation of R, Separable is $O(n^{\max(w(e_1),k-w(e_1))})$.*

> **Proof:** Let R be any recursion in S_p^k, and let Q be an arbitrary query on the recursively defined relation in R. By definition of S_p^k, R must be separable. Then by Theorem 3.1, Separable applies to Q. The only relations generated by Separable in answering Q are *ans*, $seen_1$, $carry_1$, $seen_2$, and $carry_2$. Each of these relations has at most $\max(w(e_1), k - w(e_1))$ columns; thus if there are n distinct constants in the base relations mentioned in R, each of these relations is of size at most $n^{\max(w(e_1),k-w(e_1))}$. ∎

Lemma 4.2 *For all $k > 0$ and $p > 0$, there is a recursion R in S_p^k, and a full selection query Q on R, such that Generalized Magic Sets is $\Omega(n^k)$ on Q.*

> **Proof:** For any $k > 0$ and $p > 0$, let R be the recursion

$t(X_1, X_2, \ldots, X_k) :- a_1(X_1, W)\ \&\ t(W, X_2, \ldots, X_k).$
$t(X_1, X_2, \ldots, X_k) :- a_2(X_1, W)\ \&\ t(W, X_2, \ldots, X_k).$
.
.
.
$t(X_1, X_2, \ldots, X_k) :- a_p(X_1, W)\ \&\ t(W, X_2, \ldots, X_k).$
$t(X_1, X_2, \ldots, X_k) :- t_0(X_1, X_2, \ldots, X_k).$

Let the n distinct constants appearing in the a_i and t_0 be $c_1, c_2, \ldots, c_n$. Furthermore, let a_i, for $i > 1$, be empty, and let a_1 consist of the tuples $\{(c_1, c_2), (c_2, c_3), \ldots, (c_{n-1}, c_n)\}$, let t_0 consist of the n^k tuples $\{(c_{i_1}, c_{i_2}, \ldots, c_{i_k})\}$, for $1 \leq i_1, i_2, \ldots, i_k \leq n$. Finally, let Q be the query $t(c_1, Y)?$. In response to this query, Generalized Magic Sets will generate (among others) the rules

$magic(c_1).$
$magic(W) :- magic(X)\ \&\ a_1(X, W).$

$t(X_1, X_2, \ldots, X_k) :- magic(X_1)\ \&\ t_0(X_1, X_2, \ldots, X_k).$
$t(X_1, X_2, \ldots, X_k) :- magic(X_1)\ \&\ a_1(X_1, W)\ \&\ t(W, X_2, \ldots, X_k).$

The relation *magic* contains the n tuples $\{(c_1), (c_2), \ldots, (c_n)\}$. Hence the base rule of the re-written definition (the one with t_0 in the right hand side) will add to t the n^k tuples $\{(c_{i_1}, c_{i_2}, \ldots, c_{i_k})\}$, for $1 \leq i_1, i_2, \ldots, i_m \leq n$. Hence Generalized Magic Sets is $O(n^k)$ on Q. ∎

Note that as $w(e_1) \leq k$, $\max(w(e_1), k - w(e_1)) \leq k$, so by this measure, Separable is never worse than Generalized Magic Sets. The only time $\max(w(e_1), k - w(e_1)) = k$ is when $w(e_1) = k$, that is, every column of the recursively defined relation belongs to the same equivalence class.

Lemma 4.3 *For all $k > 0$ and $p > 0$, there is a recursion R in S_p^k, and a full selection query Q on R, such that Generalized Counting is $\Omega(p^n)$ on Q.*

Proof: For any $k > 0$ and $p > 0$, let R again be the recursion

$t(X_1, X_2, \ldots, X_k) :- a_1(X_1, W)\ \&\ t(W, X_2, \ldots, X_k).$
$t(X_1, X_2, \ldots, X_k) :- a_2(X_1, W)\ \&\ t(W, X_2, \ldots, X_k).$
.
.
.
$t(X_1, X_2, \ldots, X_k) :- a_p(X_1, W)\ \&\ t(W, X_2, \ldots, X_k).$
$t(X_1, X_2, \ldots, X_k) :- t_0(X_1, X_2, \ldots, X_k).$

Let the n distinct constants appearing in the a_i and t_0 be $c_1, c_2, \ldots, c_n$. In this case, let the a_i, for $1 \leq i \leq n$, be identical and consist of the tuples

$\{(c_1, c_2), (c_2, c_3), \ldots, (c_{n-1}, c_n)\}$. (Here t_0 is arbitrary.) Finally, let Q be the query $t(c_1, Y)$?. In response to this query, Generalized Counting will generate (among others) the rules

$count(0, 0, 0, c_1)$.
$count(I+1, J, (p+1)*K+1, W)$:- $count(I, J, K, X)$ & $a_1(X, W)$.
$count(I+1, J, (p+1)*K+2, W)$:- $count(I, J, K, X)$ & $a_2(X, W)$.
.
.
$count(I+1, J, (p+1)*K+p, W)$:- $count(I, J, K, X)$ & $a_p(X, W)$.

The relation *count* contains the tuples $(I, 0, J, c_I)$, for $1 \leq I < n$ and $(p+1)^{I-1} \leq J < (p+1)^I$. Hence Generalized Counting is $\Omega(p^n)$ on Q. ∎

These lemmas are illustrated by the following examples from Section 1.

On the recursion of Example 1.2, and the query $buys(tom, Y)$? the Magic Sets algorithm [BMSU86,BR87] will generate the rules

$magic(tom)$.
$magic(W)$:- $magic(X)$ & $friend(X, W)$.

$buys(X, Y)$:- $magic(X)$ & $perfectFor(X, Y)$.
$buys(X, Y)$:- $magic(X)$ & $friend(X, W)$ & $buys(W, Y)$.
$buys(X, Y)$:- $magic(X)$ & $buys(X, Z)$ & $cheaper(Z, Y)$.

Let *friend* contain the tuples $(a_1 = tom, a_2)$,(a_2, a_3),...,(a_{n-1}, a_n). Also, let *cheaper* consist of the tuples (b_n, b_{n-1}), ..., (b_2, b_1) and *perfectFor* consist of the tuple (a_n, b_n). At the end of the evaluation of these rules, *buys* will contain the n^2 tuples (a_i, b_j), for $1 \leq i, j \leq n$. Thus Magic Sets generates relations that are $\Omega(n^2)$.

On the recursion of Example 1.1 and the query $t(tom, Y)$?, the Generalized Counting Method [BR87,SZ86] constructs (among others) the rules

$count(1, 1, 1, tom)$.
$count(i+1, 2j, 2k, W)$:- $count(i, j, k, X)$ & $friend(X, W)$.
$count(i+1, 2j+1, 2k, W)$:- $count(i, j, k, X)$ & $idol(X, W)$.

Consider the sample database in which both *friend* and *idol* contain the tuples (a_1, a_2), (a_2, a_3), ..., (a_{n-1}, a_n). At the end of evaluating these rules the *count* relation contains the tuples $(i, j, 2^{i-1}, a_i)$ for $1 \leq i < n$, $2^{i-1} \leq j < 2^i$. Thus Generalized Counting is $\Omega(2^n)$. This implies that a 30 tuple database can generate a several gigabyte relation.

On both of the preceding queries, the separable recursion algorithm generates only monadic relations. If the *friend*, *idol*, and *cheaper* relations contain n distinct constants, then no more than $O(n)$ tuples can appear in these monadic relations. Hence the separable algorithm is $O(n)$ on both of the queries above.

5 Conclusion

We have demonstrated that for some separable recursions, the separable recursion evaluation algorithm is significantly more efficient than Magic Sets and Generalized Counting. However, as the separable algorithm is limited in applicability, it must be used to supplement these more general algorithms rather than to replace them.

It is interesting to attempt to apply the separable evaluation algorithm to more general recursions. Relaxing the conditions in Definition 2.4 sheds some light on the difficulties encountered in doing so.

If we remove condition 1, the independence of the evaluation of each equivalence class in Section 3 no longer holds. Intuitively, variables in one equivalence class can shift into another. The separable evaluation algorithm may be extendible to handle some shifting variable definitions if the algorithm maintains some state about rule applications.

If we remove conditions 2 and 3, in general we can no longer guarantee that the equivalence classes will be well defined. Again, a modification of the separable algorithm may work for some recursions in this class.

Finally, if we remove condition 4, the separable evaluation algorithm will still produce the correct answer. However, it loses the "focussing" effect of the selection constant. For example, if we apply the separable recursion algorithm to the recursion

$t(X,Y) :- a(X,W)\ \&\ t(W,Z)\ \&\ b(Z,Y).$
$t(X,Y) :- t_0(X,Y).$

and the query $t(x_0,Y)?$, we will examine the entire b relation. (The initialization of $carry_1$ will be $carry_1(x_0,Y) := a(x_0,W)\ \&\ b(Z,Y).$)

Perhaps the most promising direction is to attempt to generalize the separable evaluation algorithm to non-linear recursions and to recursions that contain mutually recursive predicates. We are currently investigating this possibility.

References

[ASU79] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalence of relational expressions. *SIAM Journal of Computing*, 8(2):218–246, 1979.

[AU79] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*, pages 110–120, 1979.

[BKBR87] Catriel Beeri, Paris Kanellakis, Francois Bancilhon, and Raghu Ramakrishnan. Bounds on the propagation of selection into logic programs. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 214–226, 1987.

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 1–15, 1986.

[BR87] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 269–283, 1987.

[Cha81] C. Chang. On the evaluation of queries containing derived relations in a relational data base. In H. Gallaire, J. Minker, and J. Nicolas, editors, *Advances in Data Base Theory, Volume 1*, Plenum Press, 1981.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing*, pages 77–90, 1977.

[HH87] Jiawei Han and Lawrence J. Henschen. Handling redundancy in the processing of recursive database queries. In *Proceedings of the ACM-SIGMOD Conference on the Management of Data*, pages 73–81, 1987.

[HN84] Lawrence J. Henschen and Shamim A. Naqvi. On compiling queries in recursive first order databases. *JACM*, 31(1):47–85, 1984.

[MN82] Jack Minker and Jean M. Nicolas. On recursive axioms in relational databases. *Information Systems*, 8(1):1–13, 1982.

[Nau87] Jeffrey F. Naughton. One sided recursions. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 340–348, 1987.

[Nau88] Jeffrey F. Naughton. *Benchmarking Multi-Rule Recursion Evaluation Strategies.* Technical Report CS-TR-141-88, Princeton University, 1988.

[SZ86] Domenico Sacca and Carlo Zaniolo. The generalized counting methods for recursive logic queries. In *Proceedings of the First International Conference on Database Theory*, 1986.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems.* Volume 1, Computer Science Press, 1988.