MINIMIZING EXPANSIONS OF RECURSIONS

Jeffrey F. Naughton
Yehoshua Sagiv

CS-TR-150-88

April 1988

Minimizing Expansions of Recursions

Jeffrey F. Naughton* Princeton University
Yehoshua Sagiv† Hebrew University

April 18, 1988

Abstract

In recent years function-free horn clauses have received a lot of attention as database query languages. Recursive definitions in such a language are particularly problematic in that they are hard to implement efficiently. As most evaluation procedures at least implicitly evaluate the expansion of the recursion, it is natural to consider optimizing the recursion by minimizing its expansion. In this paper we show how attempts to minimize expansions of recursions lead naturally to the issues of recursively redundant predicates and bounded recursions. We review current results, prove several new results about inter-element redundancy in expansions, and show how both recursively redundant predicates and bounded recursions are closely related to the existence of various types of paths in a graph constructed from the rule.

1 Introduction

In recent years there has been a growing acknowledgement among database researchers that relational query languages are not sufficiently powerful. One proposed alternative to relational query languages is query languages built upon horn-clause logic. While such logic-based query languages are more expressive than relational languages, this extra expressive power does not come for free, as logic-based languages are more difficult to implement efficiently. Particularly problematic for efficient implementation are recursive definitions.

There is a large body of literature proposing efficient evaluation algorithms for recursive definitions (for a survey, see Bancilhon and Ramakrishnan [BR86].) There has been less work on rewriting recursive definitions to remove redundancy. This paper summarizes and unifies some earlier work in that direction, and proves some new results.

In this paper we will consider the language of function-free horn clauses, which is popularly known as "Datalog" because it is the subset of prolog that obeys the first-normal form assumptions of relational databases. Datalog has none of the extra-logical features present in Prolog, such as the "cut" operator or a built-in procedural interpretation. However, we

*This work was done at Stanford University, and was supported by NSF grant IST-84-12791 and a grant from the IBM Corporation.

†Work supported in part by grant 85-00082/1 from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel.

will use Prolog notation and write the horn clauses as rules, with the positive literal in the head or consequent, and the conjunction of the negative literals as the body or antecedent.

We will use the terms "predicate" and "relation" interchangably, and assume that the relations in the system can be separated into two types. The EDB or extensional database predicates appear in the head of no rule and are defined completely by their extent, that is, by the tuples stored in the relations corresponding to the predicate. The IDB or intensional database predicates are defined in terms of other predicates by the rules.

If an IDB predicate is defined either directly or transitively in terms of itself, we say that the definition of that predicate is recursive. The relation for a recursively defined IDB predicate is the least fixpoint that includes the given values for the EDB predicates in the definition. The *expansion* of a recursively defined predicate will be defined precisely in Section 2; for now it suffices to state that by an infinite sequence of rule applications to a recursive predicate one can produce an infinite set of conjunctions of EDB predicates that also defines the recursive predicate. This infinite set of conjunctions is the expansion of the recursive predicate.

We say that two programs P and P' are equivalent if they compute the same relations for the IDB predicates. In this paper except where noted otherwise we consider programs consisting of a single, linear recursive rule. A rule is linear recursive if the predicate in the head appears exactly once in the body. We further assume that the head of the rule contain no repeated variables or constants, and that all nonrecursive predicates in the rule be EDB predicates.

Such a definition must have some initialization; we assume it is provided by a nonrecursive rule

$$t :- t_0.$$

where t_0 does not appear in the recursive rule, has the same number of arguments as t, and contains the same variables in the same position as the predicate instance in the rule head.

We say that a the expansion of a recursively defined predicate contains *inter-element redundancy* if there exist elements of the expansion s_1 and s_2 such that for all possible values of the EDB relations, evaluating s_1 will return a superset of the tuples returned by s_2. We say that the expansion contains *intra-element redundancy* if there is some element s such that there is a conjunction of some proper subset of the predicate instances in s, say s', such that s' and s define the same relation for all possible values of the EDB relations.

Evaluation algorithms for recursive definitions must at least implicitly evaluate enough elements of the expansion to construct the fixpoint or some portion thereof. If the expansion contains inter-element redundancy, then in the course of evaluating the expansion the evaluation algorithm will evaluate redundant elements of the expansion; if the expansion contains intra-element redundancy, then in the course of evaluating some element of the expansion the evaluation algorithm will evaluate redundant predicate instances. The efficiency of the evaluation algorithm will be improved if these redundancies are first removed — so a natural goal is to try to develop algorithms to convert recursive definitions to equivalent definitions that produce redundancy free expansions.

Example 1.1 Suppose that we have the following EDB relations: $l(X,Y)$ of consumers X and products they like Y; $c(Y)$ of inexpensive products Y; and $k(X,W)$ of pairs of people

X, W such that X knows W. Also, suppose that a person will buy a product if it's cheap and they like it; or if it's cheap and they know someone who bought the product. Then we can define the relation *b(X, Y)* of consumers X and products Y that they buy with the following rules:

r_1: $b(X,Y)$:- $l(X,Y), c(Y)$.
r_2: $b(X,Y)$:- $k(X,W), b(W,Y), c(Y)$.

The first three elements of the expansion of *buys* are

$$l(X,Y)c(Y)$$
$$k(X,W_0)l(W_0,Y)c(Y)c(Y)$$
$$k(X,W_0)k(W_0,W_1)l(W_1,Y)c(Y)c(Y)c(Y)$$

One can show that there is no inter-element redundancy in this expansion. However, there is intra-element redundancy: every element after the first contains redundant instances of $c(Y)$. The recursion can be re-written equivalently as

r_1: $b(X,Y)$:- $l(X,Y), c(Y)$.
r_2': $b(X,Y)$:- $k(X,W), b(W,Y)$.

Here the first three elements of the expansion are

$$l(X,Y)c(Y)$$
$$k(X,W_0)l(W_0,Y)c(Y)$$
$$k(X,W_0)k(W_0,W_1)l(W_1,Y)c(Y)$$

and there is no intra-element redundancy.

To see that this is not completely trivial, consider the rules

r_3: $b(X,Y)$:- $r(X), l(X,Y)$.
r_4: $b(X,Y)$:- $r(X), k(X,W), b(W,Y)$.

where $k(X)$ is perhaps interpreted as "X is rich." The first three elements of the expansion of b as defined by these rules begins

$$r(X)l(X,Y)$$
$$r(X)k(X,W_0)r(W_0)l(W_0,Y)$$
$$r(X)k(X,W_0)r(W_0)k(W_0,W_1)r(W_1)l(W_1,Y)$$

which contains no inter-element or intra-element redundancy. ▮

The predicate c in the first definition in the previous example is what was called a *recursively redundant predicate* in Naughton [Nau86d]. (The word "recursively" is included because neither rule r_1 nor r_2 contain any predicates that are redundant when the rule bodies are viewed as relational expressions.) Briefly, a predicate instance p in a recursive definition D is recursively redundant if and only if there is an equivalent definition D' in which that instance of p appears in no recursive rule. Recursively redundant predicates will produce intra-element redundancy.

This definition of redundancy is similar but not identical to that used in Sagiv [Sag87], which states that a predicate instance p in a recursive rule is redundant if the definition defines the same relation when that predicate instance is removed. One difference is that according to Naughton's definition, a predicate instance is recursively redundant if it can be removed from a recursive rule by changing the nonrecursive rules or adding some new nonrecursive rules; Sagiv's definition, on the other hand, considers only removing a predicate instance without changing anything else in the definition. Because of this difference the redundancy removing algorithms in Sagiv [Sag87] are able to handle more general recursions than those given in Naughton [Nau86c].

The two definitions happen to coincide on the previous example, but this is not always the case. In the recursion consisting of the two rules

r_4: $b(X,Y) :- l(X,Y).$
r_5: $b(X,Y) :- k(X,W), b(W,Y), c(Y).$

the instance of $c(Y)$ in r_4 is recursively redundant by Naughton's definition (and, therefore, will cause intra-element redundancy in the expansion) but not by Sagiv's. In this case an optimized equivalent program is

r_6: $b(X,Y) :- l(X,Y).$
r_7: $b(X,Y) :- b'(X,Y).$
r_8: $b'(X,Y) :- k(X,W), l(W,Y), c(Y).$
r_9: $b'(X,Y) :- k(X,W), b'(W,Y).$

There are also redundant predicates that will be detected by Sagiv's optimizing algorithms but not by Naughton's. These include predicate instances that are redundant in the body of the rule when viewed as a relational expression. In this paper we concentrate on recursively redundant predicates as defined by Naughton, because (as we will show) there is a close relationship between intra-element and inter-element redundancy and predicates of this type.

The following is a recursion that produces inter-element redundancy.

Example 1.2 Suppose that we introduce a new relation $i(X)$, which stands for "person X is impressionable." Also, suppose now that a person will buy a product if the person likes it, or if the person is impressionable and someone else has bought it. This situation is captured by the following rules:

r_{10}: $b(X,Y) :- l(X,Y).$
r_{11}: $b(X,Y) :- i(X), b(W,Y).$

The first three elements of this expansion are

$$l(X,Y)$$
$$i(X)l(W_0,Y)$$
$$i(X)i(W_0)l(W_1,Y)$$

One can prove that for any valuation of i and l, the relation returned by any element other than the first will be identical to that returned by the second element, so the recursion produces an expansion with inter-element redundancy. The recursion can be re-written equivalently as

r_{10}: $b(X,Y)$:- $l(X,Y)$.
r'_{11}: $b(X,Y)$:- $i(X), l(W,Y)$.
which produces no inter-element redundancy. ∎

In the previous example something interesting has happened. We were able to replace a recursive definition by a nonrecursive definition. Recursions that are replaceable by nonrecursive definitions are called *bounded recursions.* Bounded recursions are a particularly interesting subset of recursions with inter-element redundancy. Early work on sufficient conditions for a recursion to be bounded appeared in Minker and Nicolas [MN82]. Sagiv [Sag85] gave necessary and sufficient conditions for strongly typed, single predicate recursions to be bounded. Cosmadakis and Kanellakis [CK86] showed that Sagiv's result can be extended to certain untyped recursions. Recently Gaifman et al. [GMSV87] have proven that the general problem is undecidable. Here with respect to boundedness we summarize the line of research beginning in Ioannidis [Ioa85] and in Naughton [Nau86a] and continued in Naughton and Sagiv [NS87].

As we will show, all but a finite number of elements in the expansion of a bounded recursive definition are redundant. Thus, boundedness of a recursive definition means inter-element redundancy, but the converse is not necessarily true. Another observation is that boundedness is a special case of intra-element redundancy, as can be seen when we ask "what if every predicate in every recursive rule is recursively redundant?" Then, by definition of recursively redundant, there is an equivalent definition in which no predicate appears in any recursive rule. But this means that there is an equivalent nonrecursive definition, so that the recursion is bounded. The price we pay for this observation is an easy reduction that proves that detecting redundant predicates is undecidable.

The ideal would be an algorithm that, for any recursive definition, outputs an equivalent definition that contains no inter-element or intra-element redundancy in its expansion. The previous undecidabilty results show that this is unachievable, but actually it is worse than that — in Section 4 we prove that there are recursions that are not equivalent to any inter-element redundancy free recursion. However, we can isolate subclasses of recursions for which redundancy eliminating algorithms do exist. Inasmuch as these subclasses of recursions cover most of the recursions we have encountered in actual horn-clause programs, we expect these algorithms will be useful as part of the optimization of horn-clause database query languages.

We now summarize the rest of the paper. Section 2 precisely defines the expansion of an IDB predicate, inter-element redundancy, and intra-element redundancy. Section 3 introduces the *A/V graph*, a graph constructed from a recursive rule that is useful in detecting both types of redundancy. This section also introduces *chains* and *branches*, properties of elements of expansions that determine whether or not redundancy will occur.

Section 4 concentrates on inter-element redundancy. This section relates boundedness to inter-element redundancy, defines two types of boundedness, and reviews current results about the decidability of boundedness. Section 5 concentrates on intra-element redundancy. In it we relate intra-element redundancy to syntactic properties of the elements of an expansion, and show how to detect certain intra-element redundancies from the A/V graph for a recursive rule. We conclude in Section 6 with some open problems and directions for future research.

```
1)  Give all variables in rules subscript 0;
2)  S := $\emptyset$;
3)  $CurString$ := $t$;
4)  while $true$ do
6)      S := S $\cup$ {$CurString$ with $r_e$ applied};
7)      $CurString$ := $CurString$ with $r_r$ applied;
8)      increment the subscripts of all variables in $r_r$ and $r_e$;
9)  endwhile;
```

Figure 1: Procedure ExpandRule

2 Expansions of IDB Predicates

The expansion of an IDB predicate t is the set of all conjunctions of EDB predicates that can be generated by some sequence of rule applications to t. More precisely, let s be a conjunction of predicate instances. To apply a rule to s, find some rule r such that the head of r unifies with some predicate instance p in s. Then replace that instance of p by the body of r, after applying the most general unifier (of the rule head and p) to both the body of r and the predicate instances in $s - p$. Then the expansion of an IDB predicate t is the set of all conjunctions of EDB predicates such that each conjunction can be produced by some sequence of rule applications beginning with an application of some rule to t. For recursive predicates, the expansion is infinite.

Recall that in this paper, we are dealing with linear recursive rules without constants or repeated variables in the rule head. Procedure ExpandRule (Figure 1), enumerates the expansion of such definitions consisting of a recursive rule, r_r, and a nonrecursive rule, r_e. The output of ExpandRule is the expansion of the recursively defined predicate, represented by the infinite set S.

Throughout the procedure, the string-valued variable CurString will have exactly one occurrence of the recursive predicate t. To "apply" a rule r to CurString, replace that occurrence of t by the right side of r, after the substitutions required to unify it with the head of the rule. In the initialization, we subscript the variables in the rules so that no variable appears in both CurString and one of the rules. On each iteration, we increment the subscripts for the same reason.

Example 2.1 If e is the edge relation of a digraph, then the following rules define the transitive closure t of the graph.

r_r: $t(X,Y)$:- $e(X,Z), t(Z,Y)$.
r_e: $t(X,Y)$:- $e(X,Y)$.

In order to distinguish between an application of the recursive rule and an application of the nonrecursive rule, we let e denote the occurrence of e in the recursive rule, and e' denote the occurrence in the nonrecursive rule. The first four strings in the set S are

$$e'(X,Y),$$

$$e(X,Z_0)e'(Z_0,Y),$$
$$e(X,Z_0)e(Z_0,Z_1)e'(Z_1,Y),$$
$$e(X,Z_0)e(Z_0,Z_1)e(Z_1,Z_2)e'(Z_2,Y).$$

∎

A string in an expansion may contain multiple occurrences of each predicate appearing in the recursive rule. We will use "predicate instance" to refer to occurrences of predicates in the strings of the expansion.

The strings in an expansion are conjunctive queries, a subset of relational expressions. If a variable V appears in the head of the rule, then V is a *distinguished* variable; otherwise, it is *nondistinguished.* The distinguished variables appearing in each element of the expansion are those of the original recursive rule (possibly not all of them.) On the other hand, new nondistinguished variables are created in each element of the expansion. The variable W_i denotes a nondistinguished variable that appears for the first time in the $i+2$ element of the expansion, and it corresponds to the nondistinguished variable W in the body of the recursive rule.

If $V_1, V_2, \ldots, V_i$ are the distinguished variables, and $W_1, W_2, \ldots, W_j$ the nondistinguished variables, then the relation specified by the string $p_1p_2\ldots p_n$ is

$$\{(V_1, V_2, \ldots, V_i) | (\exists W_1, W_2, \ldots, W_j)(p_1 \wedge p_2 \wedge \ldots \wedge p_n)\}$$

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

In the next section we will need to decide equivalences between conjunctive queries; to do this, we use techniques developed by Aho et al. [ASU79] and by Chandra and Merlin [CM77].

Definition 2.1 A mapping m from the variables of a string s_1 to the variables of a string s_2 is a *containment mapping* if distinguished variables map to themselves, and if $p(X_1, \ldots, X_n)$ appears in s_1, then $p(m(X_1), \ldots, m(X_n))$ appears in s_2.

The following lemma was proved by Chandra and Merlin [CM77]. (Aho et al. also attributed this to Chandra and Merlin although they proved it independently, because [CM77] was published a year earlier.)

Lemma 2.1 *If a string s_1 maps to a string s_2, then the relation specified by s_2 is contained in the relation specified by s_1.*

Aho et al. [ASU79] and Chandra and Merlin [CM77] defined a conjunctive query c to be minimal if there is no conjunctive query c' such that c' is equivalent to c yet contains fewer predicate instances. Any conjunctive query c has a unique minimal form (up to a renaming of variables), and that minimal form is always a subset of the predicate instances in c.

We are now ready to define inter-element and intra-element redundancy.

Definition 2.2 An expansion S of an IDB predicate t contains *inter-element redundancy* if there are two elements s_1 and s_2 of S such that s_1 maps to s_2.

If an expansion contains no inter-element dependency we say that the expansion is *containment-free.*

Definition 2.3 An expansion S of an IDB predicate t contains *intra-element redundancy* if there exists an element s in S such that s is not minimal as a conjunctive query.

3 The A/V Graph and Branches

Inter-element and intra-element redundancies depend on certain properties of the strings. In this section we discuss these properties, and develop tools to detect them. The lemmas and facts cited in this section are proven in Naughton [Nau86b,Nau86c].

3.1 A/V Graphs and Expansions

To relate the patterns of variables appearing in the strings of S to the structure of the rules, we define the *argument/variable (A/V) graph*:

- For each variable appearing in the rules add a variable node.
- For each argument position in each rule body add an argument node.
- Draw an undirected edge from each argument node to the node for the variable that appears in that position in the rule. This kind of edge is called an *identity edge.*
- Draw a directed edge from each argument node corresponding to a position p in the recursive predicate to the node for the distinguished variable that appears in p in the rule head. This kind of edge is called a *unification edge.*

The node for a variable X is labeled X, and the node for argument position i of a predicate p is labeled p^i. A node for a distinguished variable is a distinguished variable node; all other variables nodes are nondistinguished. Because of the one-to-one correspondence between positions in the bodies of rules and the argument nodes in the A/V graph, we use position names to refer to both an argument position and the argument node it is represented by. Similarly, we use variable names to refer to variable nodes.

Many of the subsequent results depend on the existence of certain kinds of paths through the A/V graph. Some nonstandard terminology arises because we allow the directed edges in an A/V graph to be traversed from head to tail as well as from tail to head; thus a path in an A/V graph can contain unification edges traversed in either direction.

Example 3.1 Figure 2 gives the A/V graph for the rules of Example 2.1. ∎

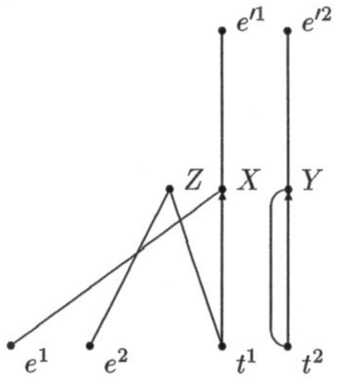

Figure 2: A/V graph for Example 2.1.

There is a close relationship between the A/V graph and procedure ExpandRule of Section 2. If a predicate instance first appears through applying a rule on iteration i, then we say that predicate instance was produced on iteration i. (The first iteration of the while loop is iteration 0.) There are two ways a predicate appearing in a string s of S can be produced on iteration i. It can be added to CurString through applying the recursive rule, or, if s was added to S on iteration i, it can be produced by applying the nonrecursive rule.

Consider iteration i. At line 8 on iteration $i-1$, the variables in the rules were given subscript i. Letting the argument nodes of the A/V graph represent the bodies of the rules, we represent iteration i by subscripting the labels of the variable nodes by i. (Figure 3(a)).

Because the heads of the rules contain no repeated variables or constants, the unification can be done by replacing the subscripted distinguished variables by the variables appearing in the instance of t in CurString. If we consider the argument nodes for t as representing that instance of t, the variable at the head of a unification edge is replaced by the variable appearing in the argument at the tail. On iteration 0, because of the initialization of CurString, the arguments contain the distinguished variables. On all other iterations, they hold the variables that were put there on the previous iteration — in this case, Z_{i-1} and Y. (Figure 3(b)).

After the substitution, argument a of a predicate instance produced on iteration i will contain the variable that is the label of the node at the end of its incident identity edge. In our example, the predicate instance added by the nonrecursive rule will be $e'(Z_{i-1}, Y)$ and the predicate instances added by the recursive rule will be $e(Z_{i-1}, Z_i)t(Z_i, Y)$.

The previous two paragraphs show how we can determine what variable appears in any position of any predicate instance in the expansion. The following two facts can be proven by induction:

Fact 3.1 A nondistinguished variable W_i appears in position p in a predicate instance produced on iteration $i+k$ if and only if there is a path from W to p containing k unification edges, all traversed in the forward direction.

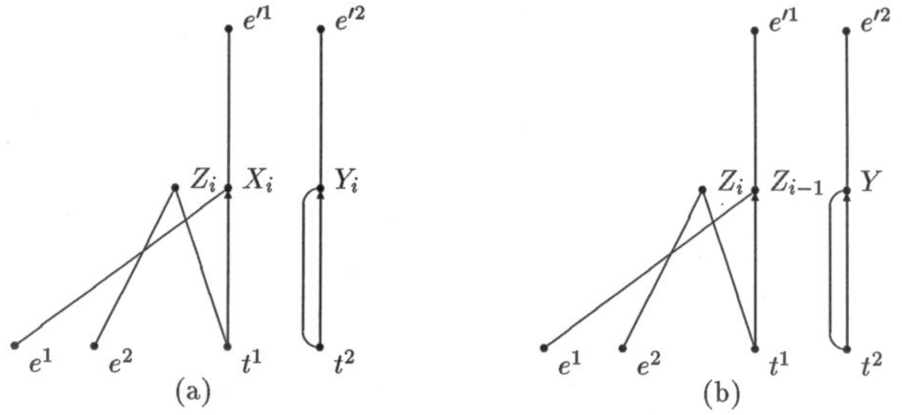

Figure 3: A/V graph for Example 2.1.

Fact 3.2 A distinguished variable V appears in position p on iteration i if and only if there is a path from V to p containing i unification edges, all traversed in the forward direction.

Any A/V graph can be divided into two kinds of connected components, those containing nondistinguished variables and those containing only distinguished variables. (Connected components in A/V graphs can require unification edges to be traversed in either direction.) Each type of component has a specific structure.

Lemma 3.1 *If a connected component in an A/V graph contains a nondistinguished variable W, it is a tree, and W is the only nondistinguished variable in the component.*

Lemma 3.2 *If a connected component contains no nondistinguished variable, that component must contain a cycle.*

Lemmas 3.1 and 3.2 combine with Facts 3.1 and 3.2 to prove that

1. Arguments in connected components that contain a cycle will always contain the distinguished variables appearing on the cycle.

2. Arguments in connected components that contain no cycles will eventually contain only subscripted instances of the nondistinguished variable in the component.

In view of point number 1 above, we have the following definition.

Definition 3.1 A *persistent variable* is a distinguished variable that appears in a cyclic component of the A/V graph.

Example 3.2 See Figure 2 for the A/V graph for Example 2.1. There are two connected components in this graph. The first, $\{t^2, Y, e'^2\}$, contains the cycle $t^2 \rightarrow Y \rightarrow t^2$. Then Fact 3.2 implies that Y always appears in e'^2. Y is a persistent variable. The remaining nodes form a tree, with Z at the root. By Fact 3.1, Z_i appears in e^2 and e'^1 on iteration i, and in e^1 on iteration $i + 1$. ▮

In the following subsection, it will be important to know how variables are shared between the predicate instances in the expansion. Things are complicated by the possibility of repeated variables in the rule body. Repeated variables give rise to branches in the paths from variable nodes to argument nodes, and the variable at the root of such paths appears on all branches. Thus to determine when argument positions share variables, we need to follow unification edges backward (toward the argument nodes) as well as forward.

To count the net number of forward unification edges in a path, we introduce weights on the edges of the A/V graph. The weight of an identity edge is 0; the weight of a unification edge traversed in the forward direction is 1, and the weight of a unification edge traversed in the reverse direction is -1. The weight of a path in the A/V graph is the sum of the weights of the edges in the path. With this definition, we have the following lemma:

Lemma 3.3 *A variable appears in position p^1 of a predicate instance produced on iteration i, and in position p^2 of a predicate instance produced on iteration $i+k$, if and only if there is a path from from p^1 to p^2 of weight k such that the weight of the minimal point on the path, relative to p^1, is $\geq -i$.*

3.2 Chains and Branches

In this subsection we discuss some properties of the strings of an expansion that will be useful in determining redundancy. These properties depend only on the recursive rule.

Definition 3.2 A *chain* is a sequence of predicate instances $p_1, p_2, \ldots, p_n$, such that for $1 \leq i < n$, p_i and p_{i+1} share a nondistinguished variable.

Definition 3.3 A *branch* is a sequence of predicate instances such that for all p and q in the sequence, there is a chain containing p and q.

Example 3.3 Assuming that X and Y are the only distinguished variables,

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

is a chain of length three. The sequence

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

contains three chains of length one. The sequence

$$p(X, W_0, Z_0)q(W_0)p(Z_0, W_1, Z_1)q(W_1)p(Z_1, W_2, Z_2)q(W_2)$$

is a branch. ▮

By examining the A/V graph for a recursive rule, we can tell what chains (and therefore, what branches) will appear in the strings of the expansion. Chains depend only on shared nondistinguished variables, so the first task is to eliminate from the A/V graph the nodes for arguments that contain only distinguished variables. By Lemmas 3.1 and 3.2, we can do this by removing all connected components that contain cycles.

For the second step, we augment the remaining subgraph by adding predicate edges between adjacent argument nodes of the same nonrecursive predicate. The result is the *augmented* A/V graph for the rule.

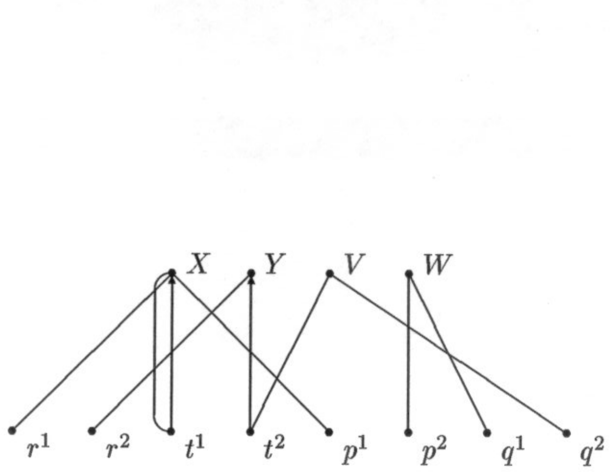

Figure 4: A/V graph for Example 3.4

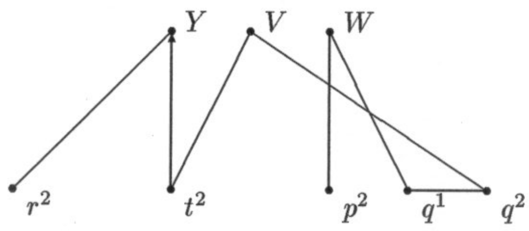

Figure 5: Augmented A/V graph for Example 3.4

Example 3.4 The following rule illustrates the concepts that will be developed in this section.

r_r: $t(X,Y) :- t(X,V), p(X,W), q(W,V), r(X,Y)$.

The A/V graph for this rule is given in Figure 4. X appears in a connected component with a cycle, so we remove the nodes for X, t^1, p^1, and r^1. The augmented A/V graph for the rule is given in Figure 5. ∎

The following lemma and its two corollaries show the relationship between the augmented A/V graph for a rule and the branches that appear in its expansion. Recall that in paths in an A/V graph, unification edges (arcs) can be traversed in either direction.

Lemma 3.4 *There is a chain containing an instance of predicate r produced on iteration i, and an instance of s produced on iteration $i+k$, if and only if there is a path of weight k from some argument of r to some argument of s.*

Corollary 3.1 *Instances of two predicates appear in the same branch if and only if the argument nodes for those predicates appear in the same connected component.*

Definition 3.4 The *rank* of a predicate is the weight of a maximal path from any variable node to an argument node of the predicate. The *span* of a branch is the rank of its maximal predicate.

If there are cycles the rank of some predicates may be infinite.

Corollary 3.2 *The span of a branch is the maximum weight of any path in the connected component for the branch.*

Two branches in a string are *instances of the same branch* if the predicates in each appear in the same connected component in the augmented A/V graph. A rule produces multiple branches if there are multiple connected components in the augmented A/V graph for the rule. An instance of each branch is begun on every iteration. If the span of a branch is k, the first complete instance of that branch will be produced by iteration $k + 1$. In addition to the complete instances of branches, there will also be incomplete instances of branches produced.

Example 3.5 In Figure 5, there is only one connected component, and it contains all the argument nodes. This implies that there is a branch (chain, in this case) containing instances of p, q, and r. There are paths of weight 0 from V and W to the nodes of p and q, thus p and q are of rank 0. There is a path from V to r^2 of weight 1, so r is of rank 1. The span of the branch is 1.

The third string in the expansion,

$$p_2(X, W_1)q_2(W_1, V_1)r_2(X, V_0)p_1(X, W_0)q_1(W_0, V_0)r_1(X, Y),$$

contains a partial instance of the branch produced on iteration 1 ($r_1(X, Y)$), a complete instance produced on iterations 1 and 2

$$p_1(X, W_0)q_1(W_0, V_0)r_2(X, V_0)$$

and another partial instance produced on iteration 2 ($p_2(X, W_1)q_2(W_1, V_1)$). This last partial instance will be completed on iteration 3. ∎

If there is a cycle of nonzero weight in a component of an augmented A/V, then there will be a branch with an infinite span. Such branches will never be completed; in this case we say the expansion contains *unbounded branches.*

Definition 3.5 A connected component of an augmented A/V graph is a *bounded component* if it contains no cycle of nonzero weight; otherwise it is an *unbounded component.*

Definition 3.6 A nonzero weight cycle in the augmented A/V graph is called a *chain-generating path.*

Definition 3.7 Any argument position on a chain-generating path is a *linking position*; the variables that appear in linking positions in the elements of the expansion are called *linking variables.*

Example 3.6 The augmented A/V graph for the recursive rule of the transitive closure example (Example 2.1), Figure 6, contains an unbounded component. The cycle in that component is a chain-generating path. Argument positions e^1 and e^2 are linking positions. The expansion contains unbounded branches — for any n, we can find a string in the expansion containing a branch of at least n instances of e. ∎

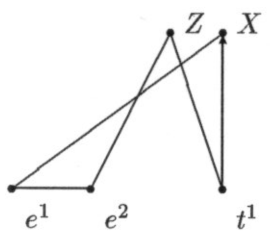

Figure 6: Augmented A/V graph for the recursive rule of Example 2.1.

4 Inter-Element Redundancy

In this section we concentrate on inter-element redundancy. We begin by giving a sufficient condition for an expansion to contain no inter-element redundancy.

Theorem 4.1 *Let D be a recursion with a single, linear recursive rule r. Suppose that the augmented A/V graph for D has some unbounded component and that the body of r contains no repeated predicates. Then the expansion of D contains no inter-element redundancy.*

Proof: Let the augmented A/V graph have a chain-generating path. We prove that if an element s_1 of the expansion maps to an element s_2 of the expansion, then the predicate instance appearing i predicate instances from the start of the chain in s_1 must map to the instance appearing i instances from the start in s_2. The proof is by induction on the predicate instances in the chains.

Let p be the first predicate in the chain. Then, by Fact 4.2 from Naughton [Nau86b], there is a linking position a in p that contains a distinguished variable. Because there are no repeated predicates in the rule, by Fact 4.1 from Naughton [Nau86b], this is the only instance of p such that a contains that distinguished variable. Thus if s_1 is to map to s_2, the first predicate instance of the chain in s_1 must map to the first predicate instance in the chain in s_2.

Now suppose that for $i < n$, if s_1 maps to s_2, then the *ith* predicate instance from the beginning of the chain in s_1 maps to the *ith* predicate instance from the beginning of the chain in s_2. The *nth* predicate in from the beginning of the chain in s_1 must, by definition of a chain, share some linking variable V with the instance $n-1$ from the beginning. By induction, any mapping m must map V to some variable, say V', that appears in the same position in the predicate $n-1$ from the beginning in s_2. But again by definition of chain, V' must appear in the same position in the predicate n from the beginning in s_2 as V does in s_1. But because there are no repeated predicates in the body of the recursive rule, this is the only instance in which V appears in that position, the instance in s_1 that is n instances from the beginning must map to the predicate instance n from the beginning in s_2.

But if $s_1 \neq s_2$, then the instance of the chain is longer in one string than in the other. If s_1 is longer than s_2, then we will some predicate instance in the chain in s_1 with no predicate instance in s_2 to map to, and hence s_1 cannot be mapped to s_2. If

s_2 is longer than s_1, then the exit predicate t_0 will have to be mapped to some other predicate in the chain. But this is impossible by definition of mapping, so again, s_1 cannot be mapped to s_2.

Because this holds for any distinct pair of strings s_1 and s_2, the expansion must contain no inter-element redundancy. ∎

We now turn to definitions of bounded recursions. Strictly speaking, because of the class of recursions we are considering, in this paper we are talking about *uniform boundedness.* For a discussion of uniform vs. simple boundedness see Naughton and Sagiv [NS87].

There are two natural ways to define boundedness.

Definition 4.1 A program P is bounded if there is a nonrecursive program P' such that P and P' are equivalent.

Definition 4.2 A program P is bounded if there is some k such that for any value of the relations, the iterative bottom-up construction of the fixpoint will converge in $\leq k$ iterations.

The following lemma is from Naughton and Sagiv [NS87].

Lemma 4.1 *Definitions 4.1 and 4.2 are equivalent.*

The following theorem is from Naughton [Nau86a]. It shows the relationship of boundedness to a particular kind of inter-element redundancy.

Theorem 4.2 *A set of rules defining a predicate t is bounded if and only if, in the expansion of t, there exists an n_0 such that for all $n > n_0$, s_n is mapped to by some previous string.*

Also in [Nau86a] it was shown that the absence of a chain generating path was a sufficient condition for bounded recursion. However, not all programs with chain generating paths are unbounded.

Example 4.1 The following program

$t(X,Y,Z)$:– $t(X,W,Z)$ & $e(W,Y)$ &
$\qquad\qquad\qquad e(W,Z)$ & $e(Z,Z)$ & $e(Z,Y)$.
$t(X,Y,Z)$:– $t_0(X,Y,Z)$.

has a chain generating path but is bounded. ∎

We now turn to a class C for which a chain-generating path is necessary and sufficient for the recursion to be unbounded. This class was defined in Naughton and Sagiv [NS87].

Definition 4.3 Let r be a linear recursive rule, and let s_i and s_j, where $i < j$, be any pair of strings in the expansion of r. Then r is in the class C if and only if there can be no mapping m from s_i to s_j such that m maps a linking variable in s_i to a persistent variable in s_j.

Theorem 4.3 *Let P be a recursive rule in class C. Then P is bounded if and only if there is no chain generating path in the A/V graph for P.*

The simplest way to prove that a recursive rule is in the class C is to prove that no persistent variable ever appears in a linking position. This technique allows us to prove that the following four classes are subclasses of C. The following lemmas are from Naughton and Sagiv [NS87].

Lemma 4.2 *If no sequence of argument positions in the recursive predicate in the body of the recursive rule in P contains a permutation of the variables that appear in the same positions in the instance of the recursive predicate in the head, then P is in C.*

Ioannidis first gave a necessary and sufficient condition for boundedness in this class in [Ioa85].

Lemma 4.3 *If the body of the recursive rule in a recursion P has no repeated predicates, then P is in the class C.*

A chain-generating path was first proven necessary and sufficient for recursions in this class in Naughton [Nau86a].

Lemma 4.4 *If no persistent variable appears in a nonrecursive predicate in the body of the recursive rule of P, then P is in C.*

In [Ioa86], Ioannidis has shown that his condition from [Ioa85] holds for this class as well. It includes recursions in which no distinguished variable appears both in a nonrecursive predicate and in the recursive predicate in the body.

Lemma 4.5 *If no persistent variable appears in a linking position in the body of the recursive rule of P, then P is in C.*

The subclasses of C described by Lemmas 4.2, 4.3, and 4.4 are incommensurate and there are linear time membership algorithms for each. The subclass described by Lemma 4.5 properly includes that described by Lemma 4.4 and has a polynomial time membership algorithm.

We finish this section with a negative result about eliminating all redundancy from an expansion:

Theorem 4.4 *There are recursively defined predicates that cannot be defined by any inter-element redundancy free definition.*

Proof: Consider the following definition:

$t(X,Y) \text{ :- } e(X,Y), e(Y,Y).$
$t(X,Y) \text{ :- } t(X,W), e(W,Y), e(W,W).$

The expansion S of this recursion begins

$$e(X,Y)e(Y,Y)$$
$$e(X,W_0)e(W_0,W_0)e(W_0,Y)e(Y,Y)$$
$$e(X,W_1)e(W_1,W_1)e(W_1,W_0)e(W_0,W_0)e(W_0,Y)e(Y,Y)$$

It is easy to verify that this expansion has the property that for all $i, j > 0$, if $j \geq i$ then s_j maps to s_i. Call this recursion D.

Now suppose that there is some other definition D' that is equivalent to D, and let S' be the expansion of D'. Furthermore, assume that S' has no inter-element redundancy. We first prove that S' must be infinite.

Assume that S' is finite. This means that S is definable by a finite union of conjunctive queries, and this finite union would then be a nonrecursive program equivalent to S. But by Theorem 4.3 and Lemma 4.5, S is not bounded. So S' must be infinite.

Take any element s'_k from S'. By a theorem due to Sagiv and Yannakakis [SY80], if S and S' are equivalent, then there is at least one element s_m in S such that s_m maps to s'_k. Similarly, s_m must be mapped to by some s'_n of S', and so s'_n maps to s'_k. Since S' does not have any inter-element redundancy, $k = n$, so each element s'_k of S' has an equivalent element s_m in S.

Now take any two distinct elements of S', say s'_k and s'_l, and let s_m and s_n be their equivalent elements in S. Because there is containment between every pair of elements in S, there is containment between s_m and s_n, hence there must also be containment between s'_k and s'_l, which is a contradiction. ▮

5 Intra-Element Redundancy

In this section we concentrate on intra-element redundancy. We show that if a predicate p appears in a bounded component of the augmented A/V graph, then p is recursively redundant, that is, the elements of the expansion will contain redundant p instances. For a useful subset of recursive rules, only predicates appearing in bounded components are redundant. These predicates can be found by a linear-time algorithm.

5.1 A Sufficient Condition

The following theorem from Naughton [Nau86d] relates redundancy to connected components in the augmented A/V graph for the recursive rule. (Recall that in a connected component of an A/V graph, unification edges can be traversed in either direction.)

Theorem 5.1 *Let a predicate p appear in a rule r, and suppose that no argument of p appears in an unbounded component of the augmented A/V graph for r. Then p is recursively redundant in r.*

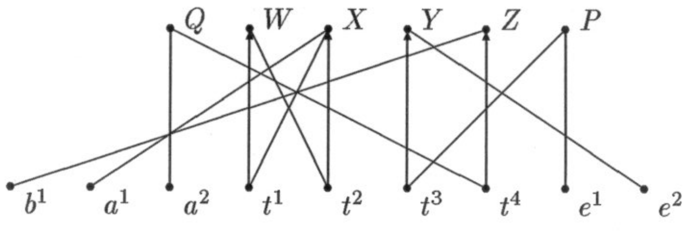

Figure 7: A/V graph for the recursive rule of Example 5.1

Note that if one argument of p appears in an unbounded component of the augmented A/V graph, then all arguments of p are in that component.

Example 5.1 Consider the following rules:

$$t(W,X,Y,Z) :- t_0(W,X,Y,Z).$$
$$t(W,X,Y,Z) :- t(X,W,P,Q), e(P,Y), a(X,Q), b(Z).$$

In the A/V graph for the recursive rule (Figure 7), there is a cycle involving X and W of weight 2. The nodes connected to this cycle form an unbounded component.

In the augmented A/V graph for the recursive rule (Figure 8) there are two components. The argument nodes of e appear in an unbounded component, while all other predicate arguments appear in a component with a maximal path of weight 1. The rank of a is 0, and the rank of b is 1. Consider the string produced on iteration five. We display it below, one branch to a line, one column for each iteration, starting with iteration 0 on the right.

1)	$t_0(X,W,P_4,Q_4)$	$a(X,Q_4)e(P_4,P_3)$	$e(P_3,P_2)$	$e(P_2,P_1)$	$e(P_1,P_0)$	$e(P_0,Y)$
2)		$b(Q_3)$	$a(W,Q_3)$			
3)			$b(Q_2)$	$a(X,Q_2)$		
4)				$b(Q_1)$	$a(W,Q_1)$	
5)					$b(Q_0)$	$a(X,Q_0)$
6)						$b(Z)$

The instances of a produced on iterations 2 and 3 ($a(X,Q_2)$ and $a(W,Q_3)$) and the instances of b produced on iterations 3 and 4 ($b(Q_2)$ and $b(Q_3)$) are redundant. ▮

More detail can be found in Naughton [Nau86c], which shows how to determine which instances of redundant predicates will be redundant, and gives an algorithm to rewrite recursive definitions to remove redundant predicates.

5.2 A Necessary and Sufficient Condition

In general, the converse of Theorem 5.1 fails to hold: some predicates appearing in unbounded components are redundant. If the nonrecursive predicates are IDB predicates, their definition may cause redundancy. For example, if we take the standard transitive closure rules,

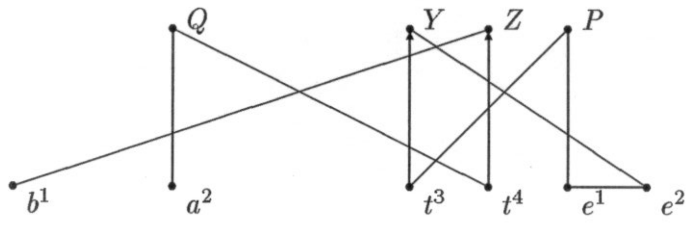

Figure 8: Augmented A/V graph for the recursive rule of Example 5.1

$t(X,Y)$:- $e(X,Y)$.
$t(X,Y)$:- $t(X,W), e(W,Y)$.

and add

$e(X,Y)$:- $a(X), b(Y)$.

then t is completely defined by the rule

$t(X,Y)$:- $a(X), b(Y)$.

and e is redundant in the recursive rule.

Even if the nonrecursive predicates are EDB predicates, interactions between the recursive and nonrecursive rule can make predicates redundant. The pair of rules

$t(X,Y)$:- $b(X), t(X,W), e(W,Y)$.
$t(X,Y)$:- $b(X), e(W,Y)$.

can be replaced by the nonrecursive rule alone.

However, we can state the following theorem, again from Naughton [Nau86d].

Theorem 5.2 *Let all nonrecursive predicates in r be EDB predicates, and suppose that there are no repeated nonrecursive predicates in r. Then a predicate p is recursively redundant in r if and only if p appears in a bounded component of the augmented A/V graph.*

5.3 Detecting Redundant Predicates

In this subsection we present a linear-time algorithm that detects redundant predicates in an augmented A/V graph. A variant of this algorithm was presented in Ioannidis [Ioa85]. There it was used on a different graph, the α-graph, to decide the bounded recursion problem for a restricted class of rules.

An A/V graph can be converted to an augmented A/V graph by a straightforward application of depth-first search. We assume that the augmented A/V graph is presented in adjacency list form. Each edge is represented by a pair of directed edges. Associated with each directed edge is a weight. Both directed edges for a predicate or identity edge have weight zero; the directed edge corresponding to a forward traversed unification edge has weight one, while the directed edge for a reverse unification edge has weight minus one.

A linear-time algorithm to detect redundant predicates is given in Figure 9.

Input. An augmented A/V graph $G = (V, E)$ represented by adjacency lists $L[v]$, for $v \in V$. The function $w(e(v,u))$ returns the weight for each edge $e(v,u)$.
Output. A list of the bounded components in the graph.

```
    cycle:    boolean;            /* true if current component contains a cycle */
    weight:   array of integer;   /* $weight[u]$ holds weight of $u$ if visited */

    Procedure SearchComp($v$, $cycle$);

    begin
    1.   mark $v$ old;
    2.   for each vertex $u$ on $L[v]$ do
    3.        if $u$ is marked "old" and $weight[u] \neq weight[v] + w(e(v,u))$ then
    4.             $cycle := true$;
    5.             return;
    6.        else
    7.             $weight[u] := weight[v] + w(e(v,u))$;
    8.             SearchComp($v$);
    9.        endif;
    10.  endfor;
    end;

begin
1.   mark all vertices "new";
2.   while there exists a vertex $v$ in $V$ marked "new" do
3.        $cycle$ := false;
4.        SearchComp($v$);
4.        if not $cycle$ then
6.             list bounded component containing $v$;
7.        endif;
8.   endwhile;
end.
```

Figure 9: An algorithm to detect redundant predicates.

6 Conclusion

Removing redundancy is only one part of efficient evaluation of queries on recursively defined relations. An evaluation algorithm should attempt to use selection constants to restrict the EDB tuples looked at during evaluation, and attempt to reuse the results from partial evaluation of one element of an expansion when evaluating another. These issues are ignored here because they are orthogonal to the issue of removing redundancy from a recursion.

This paper focussed on recursions containing one, linear recursive rule. We are currently extending the redundancy and boundedness detection procedures for more general recursions.

Another, more specific question is: is testing for containment-freedom (no inter-element redundancy) decidable? The general techniques given in Gaifman et al. [GMSV87] do not apply here, because although containment-freedom is strongly non-trivial, it is not semantic — there exist equivalent recursions D_1 and D_2 such that D_1 is containment free whereas D_2 is not.

Finally, the condition for redundancy given in Theorem 5.1 is similar to the condition for boundedness given by Theorem 4.3. Yet the class of recursions for which the condition of Theorem 5.1 is complete (no repeated predicates) is much more restrictive than the class C for which the condition of Theorem 4.3 is complete. Can the class of recursions for which Theorem 5.1 is complete be extended?

Acknowledgement: The importance of containment-free expansions in minimizing recursions first arose in discussions with Jeff Ullman and Allen Van Gelder.

References

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

[BR86] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1986.

[CK86] Stavros S. Cosmadakis and Paris C. Kanellakis. Parallel evaluation of recursive rule queries. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1986.

[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.

[GMSV87] Haim Gaifman, Harry Mairson, Yehoshua Sagiv, and Moshe Y. Vardi. Undecidable optimization problems for database logic programs. In *Proceedings of the Second IEEE Symposium on Logic in Computer Science*, pages 106–115, June 1987.

[Ioa85] Yannis E. Ioannidis. *Bounded recursion in deductive databases.* Technical Report UCB/ERL M85/6, UC Berkeley, February 1985.

[Ioa86] Yannis E. Ioannidis. A time bound on the materialization of some recursively defined views. 1986. To appear in *Algorithmica.*

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

[Nau86a] Jeffrey F. Naughton. Data independent recursion in deductive databases. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 267–279, 1986. Selected for special issue of JCSS.

[Nau86b] Jeffrey F. Naughton. *Data independent recursion in deductive databases.* Technical Report STAN-CS-86-1114, Stanford, May 1986.

[Nau86c] Jeffrey F. Naughton. *Optimizing Function-free Recursive Inference Rules.* Technical Report STAN-CS-86-1114, Stanford, May 1986.

[Nau86d] Jeffrey F. Naughton. Redundancy in function-free recursive inference rules. In *Proceedings of the IEEE Symposium on Logic Programming*, 1986.

[NS87] Jeffrey F. Naughton and Yehoshua Sagiv. A decidable class of bounded recursions. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 227–236, 1987. Selected for special issue of the Journal of Logic Programming.

[Sag85] Yehoshua Sagiv. On computing restricted projections of representative instances. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 171–180, 1985.

[Sag87] Yehoshua Sagiv. Optimizing datalog programs. In *Proceedings of the ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, March 1987.

[SY80] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *JACM*, 27(4):633–655, October 1980.