DISTRIBUTED PROCESSING OF LOGIC PROGRAMS

Ouri Wolfson[1] and *Avi Silberschatz*[2,3]

ABSTRACT

This paper is concerned with the issue of parallel evaluation of logic programs. To address this issue we define a new concept of *predicate decomposability*. If a predicate is decomposable, it means that the load of evaluating it can be divided among a number of processors, without a need for communication among them. This in turn results in a very significant speed-up of the evaluation process.

We completely characterize three classes of single rule programs (sirups) with respect to decomposability: nonrecursive, linear, and simple chain programs. All three classes were studied previously in various contexts. We establish that nonrecursive programs are decomposable, whereas for the other two classes we determine which ones are, and which ones are not decomposable. We also establish two sufficient conditions for sirup decomposability.

1. Introduction

We propose a new method of evaluating logic programs in parallel. The method is suitable for sharing the computation load among an arbitrary number of processors, which have common memory or communicate by message passing. This makes it applicable to a large class of hardware architectures. Let us demonstrate the method using the classical example of the program computing the transitive closure of a graph. The arcs of the graph are given by the tuples of a database relation A. The program is written in DATALOG (see [MW]), i.e., pure PROLOG.

$$T(x,y) :- T(x,z), A(z,y).$$

$$T(x,y) :- A(x,y).$$

If the relation A is replicated at two different processors, p1 and p2, we can partition the work of computing (the relation for) the predicate T as follows. Processor p1 executes the program

$$T(x,y) :- T(x,z), A(z,y).$$

$$T(x,y) :- A(x,y), even(x).$$

and processor p2 executes the program

$$T(x,y) :- T(x,z), A(z,y).$$

$$T(x,y) :- A(x,y), odd(x).$$

1. Computer Science Department, The Technion - Israel Institute of Technology, Haifa 32000, Israel
2. Computer Science Department, University of Texas at Austin, Austin, TX 78712
3. Research sponsored in part by the NSF grant DCR-8507224 and ONR Contract N00014-86-K-0161

In other words, p1 computes the tuples (x,y) of the transitive closure, in which x is even, and p2 computes those tuples in which x is odd. A moment of reflection will reveal that this partitioning of the work has several nice properties. First, no processor computes a tuple which is also computed by the other processor. Second, if the relation computed by each processor is output to the same device, or stored in the same file, the result is always the complete transitive closure, regardless of the input graph. Third, no communication between the two processors is required during the computation. Fourth, the work-partitioning does not require complicated program transformations, only adding evaluable predicates to the body of some rules of the original program.

Assume that the whole relation for T has to be evaluated, and p1 and p2 start at the same time and execute their programs in parallel. Assume further that at the same time a single processor, using the original program, starts the evaluation of T. It is quite intuitive that, for an "average" (large enough) graph, the partitioned evaluation of T will complete much sooner than the single-processor evaluation. Furthermore, note that the evaluation can be divided among k processors, for any $k \geq 2$. The only difference from the above example is that processor pi executes a copy of the program with the predicate $i \bmod k(x)$ added to the nonrecursive rule. The exact time-speedup achieved by the work-partitioning scheme depends on many parameters outside the scope of this paper, however, here we are interested in a qualitative issue.

We postulate that in general, a work-partitioning scheme with the properties enumerated above, is very desirable. If it can be applied to the evaluation of a predicate in a program, then we say that the predicate is decomposable. Not every predicate is decomposable. Even for the same problem of computing the transitive closure, we will prove that the predicate T' in the program

$$T'(x,y) :- T'(x,z), T'(z,y).$$

$$T'(x,y) :- A(x,y).$$

is not decomposable. The proof of this fact will be given in section 5. Therefore, we feel that it is practically and theoretically important to first formally define decomposability, and then characterize the decomposable predicates.

In this paper we completely characterize three subclasses of single rule programs (sirups) with respect to decomposability: nonrecursive, linear programs, and simple chain programs. Sirups were first studied as a syntactically restricted class of programs by Cosmodakis and Kanellakis ([CK]). They have only one output predicate, therefore we interchageably use the term decomposability of a predicate or of a program. We also provide two sufficient conditions for any sirup to be decomposable. Linear programs and simple chain programs are important subclasses of sirups from the practical point of view. Simple chain programs were completely characterized with respect to membership in the complexity class NC by Afrati and Papadimitriou ([AP]). Linear sirups were studied as a distinct subclass in the context of bounded recursion ([I], [N1], [NS]) and one sided recursion ([N2]).

This work is related to the general subject of parallel evaluation of logic programs. The subject has recently emerged as a very important and active area of research ([K], [U]). However, as far as we know, existing research is concerned with membership in the complexity class NC. This class is a mathematical tool for analyzing parallel algorithms in general. Here we show that for analyzing parallel evaluation of logic programs, a different tool can be used. Loosely speaking, if a logic program is in NC it does not guarantee that it has all the nice properties of a decomposable predicate. In particular, the processors executing an NC type algorithm usually have to communicate extensively, and therefore communication is assumed to take place through common memory. Also, a speedup for such an algorithm is not guaranteed unless the number of processors is polynomial in the size of the input. The technique of program modification that we discuss here is also related to the magic sets technique ([BMSU]). Magic sets and decomposability, both aim at increasing the efficiency of query evaluation. However, the means of magic sets is selection propagation, whereas the means of decomposability is parallel evaluation.

In the next section we introduce the necessary definitions and notations, and prove that any nonrecursive sirup is decomposable. In section 3 we provide two sufficient conditions for a general sirup to be decomposable, and in section 4 we show that one of these conditions, called pivoting, is also necessary for decomposability of a linear sirup. In section 5 it is proven that a simple chain program is decomposable if and only if it is regular. In section 6 we discuss future work.

2. Preliminaries

An *atom* is a predicate symbol with a constant or a variable in each argument position. We assume that the constants are the natural numbers. An R-atom is an atom having R as the predicate symbol. A *rule* consists of an atom, Q, designated as the *head*, and a conjunction of one or more atoms, denoted $Q^1, \ldots, Q^k$, designated as the *body*. Such a rule is denoted $Q :- Q^1, \ldots, Q^k$, which should be read "Q if Q^1 and Q^2, and, ...,and Q^k." A rule or an atom is an *entity*. If an entity has a constant in each argument position, then it is a *ground* entity. For a predicate symbol, R, a finite set of R-ground-atoms is a *relation* for R.

A DATALOG program, or a program for short, is a finite set of rules whose predicate symbols are divided into two disjoint subsets: the *base* predicates, and the *derived* predicates. The base predicates are distinguished by the fact that they do not appear in any head of a rule. An *input* to P is a relation for each base predicate. An *output* of P is a relation for each derived predicate of P. A *substitution* applied to an entity, or a sequence of entities, is the replacement of each variable in the entity by a variable or a constant. It is denoted $x1/y1, x2/y2, \ldots, xn/yn$ indicating that xi is replaced by yi. A substitution is *ground* if the replacement of each variable is by a constant. A ground substitution applied to a rule is an *instantiation* of the rule.

A *database* for P is a relation for each predicate of P. The output of P given an input I, is the set of relations for the derived predicates in the database, obtained by the following procedure, called *bottom up evaluation.*

BUE1. Start with an initial database consisting of the relations of I.

BUE2. If there is an instantiation of a rule of P such that all the ground atoms in the body are
in the database generated so far, and the one in the head is not, then:
add to the database the ground atom in the head of the instantiated rule,
and reexecute BUE2.

BUE3. Stop.

This procedure is guaranteed to terminate, and produce a finite output for any given P and I ([VEK]). The output is unique, in the sense that any order in which *bottom up evaluation* adds the atoms to the database will produce the same output. For simplicity we assume that the rules of a program are *range restricted,* i.e. every variable in the head also appears in the body. Furthermore, we assume that the rules do not have constants, and each query is to evaluate a whole relation for a predicate.

An *evaluable predicate* is an arithmetic predicate (see [BR]). Examples of evaluable predicates are sum, greater than, modulo, etc. A rule *re* is a *restricted version* of some rule r, if r and *re* have exactly the same variables, and r can be obtained by omitting zero or more evaluable predicates from the body of *re*. In other words, *re* is r with some evaluable predicates added to the body, and the arguments of these evaluable predicates are variables of r. For example, if r is:

$$S(x,y,z) :- S(w,x,y), A(w,z)$$

then one possible *re* rule is:

$$S(x,y,z) :- S(w,x,y), A(w,z), x-y=5$$

A program P_i is a *restricted version* of program P if each one of its rules is a restricted version of some rule of P. Note that P_i may have more than one restricted version of a rule r of P. To continue the above example, if P has the rule r, then P_i may have the rule *re* as well as the rule *re'*:

$$S(x,y,z) :- S(w,x,y), A(w,z), x-y=6$$

Throughout this paper, only restricted versions of a program may have evaluable predicates. The input of a program with evaluable predicates, i.e. a restricted version, is defined as before. The output is also defined as before, except that BUE2 also verifies that the substitution satisfies the evaluable predicates in the ground rule; only then the atom in the head is added to the database and BUE2 is reexecuted. For example, the substitution x/14,y/8 satisfies the evaluable predicate x-y=6, whereas the substitution x/13,y/9 does not. A predicate Q in a program P *derives* a predicate R if it occurs in the body of a rule whose head is a R-atom. Q is *recursive* if (Q,Q) is in the nonreflexive transitive closure of the "derives" relation. A program is recursive if it has a recursive predicate. A rule is recursive if the predicate in its head transitively derives some predicate in its body.

Definition: Let P be a program, T a derived predicate in P, and $P_1, \ldots, P_r$ restricted copies of P. For a derived predicate T of P, denote by T_i the relation output by P_i for T; the relation output by P is denoted T. (Observe that this is a somewhat unconventional notation, since for P_i the relation name is different than the predicate name). Predicate T is *decomposable* in P with respect to $P_1, \ldots, P_r$ if the following two conditions hold:

A. For each input I to $P, P_1, \ldots, P_r$

1. $\bigcup_i T_i \supseteq T$ (completeness), and
2. T_i and T_j are disjoint for each $i \neq j$;
   furthermore, if some derived predicate Q transitively derives T in P,
   then Q_i and Q_j are disjoint (lack-of-duplication).

and

B. For some input I to $P_1, \ldots, P_r$ each T_i is nonempty (nontriviality). []

The above definition is central to this paper, and we shall discuss it next.

Requirement A.1 states that no output is lost by evaluating the relation for T in each P_i rather than the relation for T in P; the fact that no additional output is generated is implied by the fact that each P_i is a restricted version of P. Requirement A.2 states that in the process of evaluating T, each new ground atom (or intermediate result) is computed by a unique processor. Assume that, along the lines suggested in [BR section 4], we measure the cost of evaluating the relation T, in terms of the number of new ground atoms generated in the evaluation process. Then, loosely speaking, requirement A says the following. For every input (i.e. set of base relations replicated at each processor), the evaluation by r processors is equivalent, in terms of the output produced and the total evaluation cost, to the single-processor evaluation.

The strength of requirement A enables the relaxed form of requirement B. It is enough that for "some" inputs each T_i is nonempty, since for those inputs the evaluation cost incurred by each processor is smaller than that of a single processor executing the program P. Then the evaluation of T completes sooner in the distributed case. In other words, since there is nothing to lose by distributing the computation, it is enough that we gain only in some cases to make the scheme worthwhile. However, for the decomposable predicates that we discuss in this paper, nontriviality holds for more than an isolated case input.

For instance, in the transitive closure example nontriviality holds for any input graph in which arcs exit both, even and odd nodes. Specifically, for the class of predicates that we prove decomposable in this paper, decomposability is shown using the *odd–even* predicates alone. This has two implications. First, the work performed by each processor for an arbitrary input, is roughly equal (e.g. for an arbitrary graph, the number of odd and even nodes is roughly equal). In these cases we expect the distributed evaluation to be faster than the single-processor evaluation, by a factor which is close to two, i.e. the number processors. Second, note that the *odd* and *even* predicates are a special case of the $i \bmod r$ predicates, for $r=2$. When we show that T is decomposable in P with respect to P_1 and P_2, then it will be easy for the reader to convince itself that for any r, there are restricted copies $P_1, \ldots, P_r$ such that T is decomposable in P with respect to $P_1, \ldots, P_r$. This means that the work can be divided among any number of processors. For instance, in the transitive closure example, in order to do so processor i evaluates T_i where

$$P_i.\quad T(x,y) :- T(x,z), A(z,y).$$

$$T(x,y) :- A(x,y), x = i \bmod r.$$

These facts stress the robustness of the decomposability definition.

We say that predicate T is *decomposable* in P if it is decomposable with respect to some restricted copies $P_1, \ldots, P_r$ such that $r > 1$.

A *single rule program* (see [CK]) is a DATALOG program which has a single derived predicate, denoted S in our paper, a nonrecursive rule,

$$S(x1, \ldots, xn) :- B(x1, \ldots, xn).$$

where the xi's are distinct variables, and one other, possibly recursive, rule in which the predicate symbol B does not appear.

Theorem 1: If a sirup P is nonrecursive, then its derived predicate is decomposable.

3. Sufficient Conditions for Decomposability

In this section we provide two sufficient conditions for decomposability of a general sirup. The first one is motivated by the next example, which also merits attention for the following reason. From the preceding discussion one might suspect that our notion of decomposability is equivalent to "naive" propagation of variable bindings (see introduction of [BKBR]). The latter notion means simply substituting a constant for a variable in some rules. The constant is usually taken from a query. For example, in order to find all the arcs exiting the node 2 in the transitive closure of a graph, the constant can be naively propagated into the program as follows:

$$T(2,y) :- T(2,z), A(z,y).$$

$$T(2,y) :- A(2,y).$$

It is quite clear that if a sirup is amenable to naive propagation of variable bindings, then it is decomposable. However, the reverse is not true. For example, consider the program:

$$S(x,y) :- S(y,x).$$

$$S(x,y) :- A(x,y).$$

which outputs an arc in both directions for every arc of an input graph. It is easy to see that a binding cannot be naively propagated into this program, but the sirup is decomposable; one restricted copy has the nonrecursive rule:

$$S(x,y) :- A(x,y), even(x+y).$$

and the other:

$$S(x,y) :- A(x,y), odd(x+y).$$

Our first sufficient condition for decomposability is based on the preceding observation. Next we formally define it. Assume that R is a set of atoms with each atom having a variable in each argument position. The set R is *pivoting* if there is a subset d of argument positions, such that in the positions of d:

1. the same variables appear (possibly in a different order) in all atoms of R, and
2. each variable appears the same number of times in all atoms of R.

A member of d is called a *pivot*. Note that a variable that appears in a pivot may or may not appear in a nonpivot position. The recursive rule of a sirup is *pivoting* if all the occurrences of the recursive predicate in the rule constitute a pivoting set. For example, the rule

$$S(w,x,x,y,z) :- S(u,y,x,x,w), S(v,x,y,x,w), A(u,v,z)$$

is pivoting, with argument positions 2, 3 and 4 of S being the pivots.

Theorem 2: If the recursive rule of a sirup is pivoting, then the sirup is decomposable.

Theorem 2 can be extended to general programs, not necessarily sirups, provided that we extend the pivoting definition properly. Since in this paper we concentrate on sirups, we shall be informal about general programs. A rule in a program is pivoting, if all its derived-predicate-atoms (in the head or the body) constitute a pivoting set. A program is pivoting if each one of its rules is pivoting, with the same argument positions being the pivots in all the rules; additionally it is required that the heads of rule do not have repeated variables. For example, the program

$$S(x,y,z) :- R(y,x,w), A(w,z).$$

$$R(x,y,z) :- R(x,y,w), B(w,z).$$

$$R(x,y,z):-C(x,y,z).$$

is pivoting, with positions 1 and 2 being the pivots. It can be shown that a predicate in a general program is decomposable if the rules which derive the predicate constitute a pivoting program. For example, predicate S in the program above is decomposable (add *odd–even* $(x+y)$ to the body of the third rule).

The condition of theorem 2 is not necessary for decomposability. For example, the sirup

$$S(x,x):-S(y,y),A(x,y).$$

$$S(x,y):-B(x,y).$$

is obviously not pivoting, but it is decomposable. Again, *odd–even* $(x+y)$ is added to the body of the nonrecursive rule. The intuition indicates that in this example the computation load for an arbitrary input is not evenly divided between the processors executing the two restricted versions of the program (because only the processor executing the copy with the *even* evaluable predicate can output an atom as a result of instantiation of the recursive rule). The example is unique (throughout the paper) in this respect. Expectedly, the last example motivates our next sufficient condition for decomposability of a sirup. It is defined as follows. Assume that R is a set of atoms with each atom having the same predicate symbol, Q, and a variable in each argument position. The set R is *repeating* if there are at least two argument positions of Q, i and j, such that the same variable appears in position i and position j, and this is true for each member of R (note that the variable of one member of R may be different than the variable of another). The recursive rule of a sirup is *repeating* if all the occurrences of the recursive predicate in the rule constitute a repeating set. For example, the rule

$$S(x,z,x):-S(z,z,z),S(x,x,x).$$

is repeating because of argument positions 1 and 3.

Theorem 3: If the recursive rule of a sirup is repeating, then the sirup is decomposable.

Obviously, the condition of theorem 3 is not necessary for decomposability either.

4. Linear Sirups

In this section we completely characterize the class of linear sirups with respect to decomposability. A sirup is *linear* if it is recursive, and in the body of the recursive rule there is exactly one occurrence of the recursive predicate. We also require that a linear sirup does not have repeated variables in an occurrence of the recursive predicate. The characterization of linear sirups with respect to decomposability is done by proving that the sufficient condition of theorem 2 is also necessary. We assume that the recursive rule is:

$$S(x1,...,xn):=S(Y1,...,Yn),A_1(.....),\ldots,A_k(.....).$$

where the A_i's are base predicates. Observe the notation used in this section to distinguish between two types of variables. The ones starting with a lowercase letter are logic program variables, or variables for short, as before. The ones starting with an upper case letter, e.g. $Y1$, are *metalinguistic-variables*. They denote program variables. For example, $Y1$ may denote the variable xn.

If the predicate $S(x1,...,xn)$ in a sirup P, not necessarily linear, is decomposable with respect to $P_1,...,P_r$, then we define the *home-site* of a sequence of n constants, $\bar{c} = c_1,...,c_n$. It is the S_i to which the output atom $S(\bar{c})$ belongs, if each P_i is given the input consisting of a unique atom, $B(\bar{c})$. Note that the home-site of a sequence is unique (lack-of-duplication), every sequence of n constants has a home-site (completeness), and each S_i, $1 \le i \le r$, has a sequence of constants for which S_i is the home-site. Let $\bar{c} = c_1,...,c_n$ and $\bar{d} = d_1,...,d_n$ be two sequences of constants. The ordered pair of ground atoms $<S(\bar{d}), S(\bar{c})>$ is a *one-step-derivation* if there is an instantiation of the recursive rule of P, in which the first atom is in the head and the second is in the body.

Lemma 1: If the derived predicate, S, of a linear sirup P is decomposable, and there are two sequences of constants $\bar{d} = d_1,...,d_n$ and $\bar{c} = c_1,...,c_n$ such that $<S(\bar{d}),S(\bar{c})>$ is a one-step-derivation, then the home-site of $\bar{d}$ and $\bar{c}$ is identical.

Next we discuss a procedure, called *Derive-New-Variables*(P) and given in Figure 1. We shall prove that the recursive predicate S of a linear sirup P is not decomposable, if the procedure *Derive-New-Variables*(P) halts (see step 3). Then we shall prove that it halts if the recursive rule of P is not pivoting. *Derive-New-Variables*(P) iteratively substitutes for the variables in the recursive rule of P. It starts by subscripting all variables by 1, and then at each iteration it increases the subscript of the variables and unifies the S-atom in the body with the atom in the head of the previous iteration.

Lemma 2: Let P be a linear sirup, and assume that $S(Y1,\ldots,Yn)$ and $S(Z1,\ldots,Zn)$ are two consecutive values of *Last-Deriv* in the execution of the procedure *Derive-New-Variables*(P). Furthermore, assume that there is a ground substitution ρ of the program variables in the sequence $S(Y1,\ldots,Yn), S(Z1,\ldots,Zn)$, resulting in the sequence of ground atoms $S(c_1,\ldots,c_n), S(d_1,\ldots,d_n)$. Then the pair $<S(d_1,\ldots,d_n), S(c_1,\ldots,c_n)>$ is a one-step-derivation.

Lemma 3: If for a linear sirup P, the procedure *Derive-New-Variables*(P) halts, then P is not decomposable.

Proof: Based on Lammas 1 and 2.

Lemma 4: If the recursive rule of a linear sirup P is not pivoting, then *Derive-New-Variables*(P) halts.

Theorem 4: A linear sirup is decomposable if and only if its recursive rule is pivoting.

Proof: (if) Special case of Theorem 2. (only if) Immediate from Lemmas 3 and 4. □

5. Simple Chain Programs

A simple chain program is a recursive sirup in which: (a) all the predicates are binary, (b) the argument positions in the left hand side of the recursive rule have distinct variables, and these variables appear in the first argument position of the first atom in the body, and in the last argument position of the last atom, respectively, (c) all the argument positions in the body of the recursive rule have distinct variables, except that the first argument position of the second atom has the same variable as the last argument position of the first atom, the first argument position of the third atom has the same variable as the last argument position of the second atom, etc.

For example, the following is a simple chain program:

$$S(x,y) :- A(x,z_1), S(z_1,z_2), S(z_2,z_3), C(z_3,z_4), D(z_4,y)$$

$$S(x,y) :- B(x,y).$$

Derive-New-Variables(P).

1. *Last-Rec-Rule* := The recursive rule of P with all the variables given the subscript of one.
2. *Last-Deriv* := $S(x1_0,\ldots,xn_0)$
3. Do until none of the variables of the atom in *Last-Deriv* is equal to one of the variables $x1_0,\ldots,xn_0$.
4. Assume that *Last-Rec-Rule* $= S(x1_i,\ldots,xn_i) :- S(Y1,\ldots,Yn), A_1(\ldots),\ldots,A_k(\ldots)$ and *Last-Deriv* $= S(Z1,\ldots,Zn)$.
   Let *Last-Deriv* := The atom in the head of the rule obtained by applying the substitution $Y1/Z1,\ldots,Yn/Zn$ to *Last-Rec-Rule*.
5. Let *Last-Rec-Rule* := *Last-Rec-Rule* with the subscript of the variables increased by one.
6. END;

Figure 1

---

where the A, B, C, D are base relations. A simple chain program is *regular* if in its recursive rule there is one occurrence of the predicate S and this occurrence is the first or the last in the body of the recursive rule. Note that a simple chain program is pivoting if and only if it is regular.

Theorem 5: A simple chain program P is decomposable if and only if it is regular.

6. Future Work

We shall continue the work on decomposability in several directions. One of them is to extend the characterization of decomposable predicates to other sirups first, e.g. typed (see [K]), and then to general logic programs. Another direction is to determine whether decomposition implies that the work can be evenly divided among the processors, as we have seen that can be done using the *mod* predicate. For this purpose a notion of *fair* decomposition should be defined. Another topic which merits attention is minimizing communication when evaluating nondecomposable predicates in a distributed environment. We feel that the work on decomposability should also be helpful in this area. More specifically, observe that the method proposed in this paper to partition the load in evaluating decomposable predicates, can be applied to nondecomposable ones as well; however in that case communication among the processors is necessary. The question is, how does the amount of necessary communication compare in different partitioning schemes. Finally, we shall mention that we intend to study the relationship between the class of decomposable programs and the programs in the complexity class NC.

References

[AP] F. Afrati and C. H. Papadimitriou "The Parallel Complexity of Simple Chain Queries", *Proc. 6th ACM Symp. on PODS*, pp. 210-213, 1987.

[BKBR] C. Beeri, P. Kanellakis, F. Bancilhon, R. Ramakrishnan "Bounds on the Propagation of Selection into Logic Programs", *Proc. 6th ACM Symp. on PODS*, pp. 214-226, 1987.

[BMSU] F. Bancilhon, D. Maier, Y. Sagiv, J. Ullman "Magic Sets and Other Strange Ways to Implement Logic Programs", *Proc. 5th ACM Symp. on PODS*, pp. 1-15, 1986.

[BR] F. Bancilhon and R. Ramakrishnan "An Amateur's Introduction to Recursive Query Processing", *Proc. SIGMOD Conf.* pp. 16-52, 1986.

[CK] S. S. Cosmodakis and P. C. Kanellakis "Parallel Evaluation of Recursive Rule Queries", *Proc. 5th ACM Symp. on PODS*, pp. 280-293, 1986.

[I] Y. E. Ioannidis "Bounded Recursion in Deductive Databases", TR UCB/ERL M85/6, UC Berkeley, Feb. 1985.

[K] P. C. Kanellakis "Logic Programming and Parallel Complexity", *Proc. ICDT '86, International Conference on Database Theory*, Springer-Verlag Lecture Notes in CS Series, no. 243, pp. 1-30, 1986.

[MW] D. Maier and D. S. Warren "Computing with logic: Introduction to logic programming," Benjamin Cummings, 1987.

[N1] J. F. Naughton "Data Independent Recursion in Deductive Databases", *Proc. 5th ACM Symp. on PODS*, pp. 267-279, 1986.

[N2] J. F. Naughton "One-Sided Recursions", *Proc. 6th ACM Symp. on PODS*, pp. 340-348, 1987.

[NS] J. F. Naughton and Y. Sagiv "A Decidable Class of Bounded Recursions", *Proc. 6th ACM Symp. on PODS*, pp. 227-236, 1987.

[U] J. D. Ullman "Database Theory: Past and Future", *Proc. 6th ACM Symp. on PODS*, pp. 1-10, 1987.

[VEK] M. H. Van Emden and R. A. Kowalski "The Semantics of Predicate Logic as a Programming Language", *JACM* 23(4) pp. 733-742, 1976.