Optimizing Rules in Production System Programs

Toru Ishida
NTT Communications and Information Processing Laboratories
1-2356, Take, Yokosuka-shi, 238-03, Japan

Abstract

Recently developed production systems enable users to specify an appropriate ordering or a clustering of join operations. Various efficiency heuristics have been used to optimize production rules manually. The problem addressed in this paper is how to automatically determine the best join structure for production system programs. Our algorithm is not to directly apply the efficiency heuristics to programs, but rather to enumerate possible join structures under various constraints. Evaluation results demonstrate this algorithm generates a more efficient program than the one obtained by manual optimization.

1 Introduction

The efficiency of production systems rapidly decreases when the number of working memory elements becomes larger. This is because, in most implementations, the cost of join operations performed in the match process is directly proportional to the square of the number of working memory elements. Moreover, the inappropriate ordering of conditions causes a large amount of intermediate data, which increases the cost of subsequent join operations.

To cope with the above problem, ART [Clayton, 1987], YES/OPS [Schor *et al.*, 1987] and other production systems have introduced language facilities which enable users to specify an appropriate ordering or a clustering of join operations. The following are major heuristics, which are known for creating an efficient join structure.

a) Place restrictive conditions first.
b) Place volatile conditions last.
c) Share join clusters among rules.

Heuristic a) and c) are also known as the heuristics of optimizing conjunctive queries in AI and database areas [Smith *et al.*, 1985; Warren, 1981; Jarke *et al.*, 1984]. On the other hand, heuristic b) is peculiar to production systems. Since the three heuristics often conflict with each other, there is no guarantee that a particular heuristic always leads to optimization. Thus, without an optimizer, expert system builders have to proceed through a process of trial and error.

There are two more reasons for the development of the production system optimizer.

1. *To enable expert system users to perform optimization:*
   The optimal join structure depends on the execution statistics of production system programs. Thus, even if the rules are the same, results of the optimization may differ when different working memory elements are matched to the rules. For example, the optimal join structure for a circuit design expert system depends on the circuit to be designed. This means that the optimization task should be performed not only by expert system builders but also by expert system users. The optimizer can help users to tune expert systems to their particular applications.
2. *To improve efficiency without sacrificing maintainability:*
   Production systems are widely used to represent expertise because of their maintainability. However, optimization sometimes makes rules unreadable by reordering conditions only to reduce execution time. To preserve the advantages of production systems, source program files to be maintained must be separated from optimized program files to be executed. Using the optimizer, users can improve efficiency without sacrificing maintainability by generating optimized programs each time rules are modified.

This paper describes an optimization algorithm to minimize the total cost of join operations in a production system program. Optimization is performed based on execution statistics measured from earlier runs of the program. All rules are optimized together so that join operations can be shared by multiple rules. Our approach is not to directly apply the efficiency heuristics to the original rules, but rather to enumerate possible join structures and to select the best one. The basic methodology is to find effective constraints and to use those constraints to cut off an exponential order of possibilities. The evaluation results demonstrate the algorithm generates a more efficient program than the one optimized by the expert system builder himself.

2 Basic Definitions and Concepts

Before describing our approach in detail, a brief overview of production systems and their topological transformation will be given. We will use an OPS5-like syntax [Forgy, 1981] for the reader's convenience, and assume the reader's familiarity with the RETE match algorithm [Forgy, 1982].

2.1 Production System

A *production system* is defined by a set of *rules* or *productions*, called the *production memory (PM)*, together with a database of assertions, called the *working memory (WM)*. Assertions in the WM are called *working memory elements (WMEs)*. Each rule consists of a conjunction of *condition*

elements, called the *left-hand side (LHS)* of the rule, along with a set of actions called the *right-hand side (RHS)*.

The RHS specifies information which is to be added to or removed from the WM when the LHS is successfully matched with the contents of the WM. There are two kinds of condition elements: *positive condition elements* that are satisfied when there exists a matching WME, and *negative condition elements* that are satisfied when no matching WME is found. *Pattern variables* in the LHS are consistently bound throughout the positive condition elements.

The *production system interpreter* repeatedly executes the following cycle of operations:

1. *Match*:
   For each rule, determine whether the LHS matches the current environment of the WM.
2. *Conflict Resolution*:
   Choose exactly one of the matching instances of the rules according to some predefined criterion, called a *conflict resolution strategy*.
3. *Act*:
   Add to or remove from the WM all assertions as specified by the RHS of the selected rule.

In the RETE algorithm, the left-hand sides of rules are transformed into a special kind of data-flow network. The network consists of *one-input nodes*, *two-input nodes*, and *terminal nodes*. The one-input node represents an *intra-condition test* or *selection*, which corresponds to an individual condition element. The two-input node represents an *inter-condition test* or *join*, which tests for consistent variable bindings between condition elements.

When a WME is added to or removed from the WM, a *token* which represents the action is passed to the network. First, the intra-condition tests are performed on one-input nodes; then matched tokens are stored in *alpha-memories*, and copies of the tokens are passed down to successors of the one-input nodes. The inter-condition tests are subsequently executed at two-input nodes. The tokens arriving at a two-input node are compared against the tokens in the memory of the opposite side branch. Then, paired tokens with consistent variable bindings are stored in *beta-memories*, and copies of the paired tokens are passed down to further successors. Tokens reaching the terminal nodes activate corresponding rules.

2.2 Topological Transformation

Examples of various join structures in which condition elements are variously clustered are shown in Figure 1. Since the join operation is commutative and associative, an LHS which consists of only positive condition elements can basically be transformed to any form. For example, in Figure 1, nodes *a*, *b* and *c* can be placed in any order, but if *MEA* [Forgy, 1981] is used as a conflict resolution strategy, node *s* cannot change its position. Therefore, in this case, 24 possible join structures, i.e. an exponential order of join structures, can exist.

When negative condition elements are present, there are a number of constraints in the transformation of a given LHS into an equivalent one. Since the role of negative condition elements is to filter tokens, they cannot be the first condition element, and all their pattern variables have

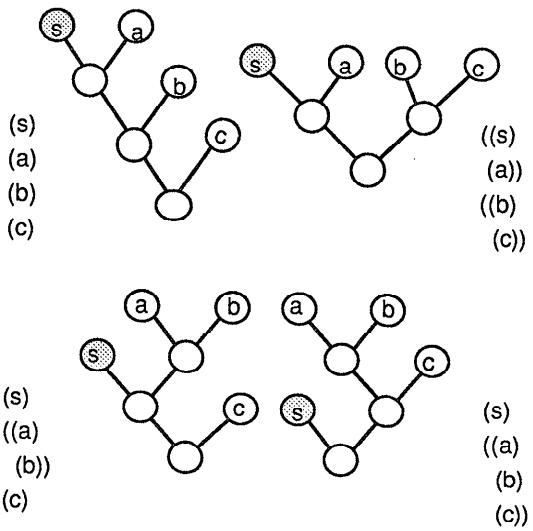


Figure 1: Join structure variations

to be bound by preceding positive condition elements. To simplify the following discussion, however, we ignore the detailed topological transformation constraints. We also do not treat the non-tree-type join topology, such as $(s)((a)(b))((a)(c))$, where two *a*'s share the same one-input node.

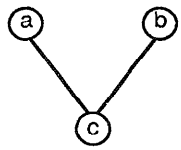
3 Cost

3.1 Parameters

The cost model of join operations is shown in Figure 2. Networks are bounded by their lowest nodes and are said to be *join structures of those particular nodes*. For example, the network shown in Figure 2, is a join structure of node *c*. There are five parameters associated with each node. These are *Token(n)*, *Memory(n)*, *Test(n)*, *Cost(n)*, and *Ratio(n)*.

Token(n) indicates the running total of tokens passed from a node *n* to successor nodes. *Memory(n)* indicates the average number of tokens stored in an alpha- or a beta-memory of *n*. *Test(n)* indicates the running total of inter-condition tests at *n*. A consistency check of variable bindings between one arriving token and one token stored in a memory is counted as one test. *Cost(n)* indicates the total cost of inter-condition tests performed in the join structure of *n*. The cost function is defined later. *Ratio(n)* indicates the ratio of how many inter-condition tests are successful at *n*.

Before optimization, a production system program is executed once, and the values of one-input node parameters are determined. In the process of optimization, various join structures are created and evaluated. The values of the thus created two-input node parameters are calculated each time using the equations defined as follows.



When b is a positive condition element:

Ratio(c) = Ratio(b)
Test(c) = Token(a) * Memory(b) + Token(b) * Memory(a)
Token(c) = Test(c) * Ratio(c)
Memory(c) = Memory(a) * Memory(b) * Ratio(c)
Cost(c) = Cost(a) + Cost(b) + Test(c)

When b is a negative condition element:

Ratio(c) = Ratio(b)
Test(c) = Token(a) * Memory(b) + Token(b) * Memory(a)
Token(c) = Token(a) * Ratio(c)
Memory(c) = Memory(a) * Ratio(c)
Cost(c) = Cost(a) + Cost(b) + Test(c)

Figure 2: Cost model

3.2 Parameters for One-Input Nodes

Let b be a one-input node. Token(b), Memory(b) and ratios at all two-input nodes are measured once. Then the ratio at the two-input node whose right predecessor is b is set to Ratio(b); i.e., Ratio(b) holds an approximate ratio of how many inter-condition tests are successful at the direct successor of b. Test(b) and Cost(b) are always set at 0, because no join operations are performed at one-input nodes.

3.3 Parameters for Two-Input Nodes

Let c be a two-input node joining two nodes, a and b, as shown in Figure 2. Note a and b are either one- or two-input nodes. Then, the following equations express the parameters of two-input nodes.

1. *Test(c):*
   When tokens are passed from the left, the number of tests performed at c is represented by Token(a)*Memory(b), and when from the right, Token(b)*Memory(a). Thus, Test(c) is represented by Token(a)*Memory(b)+Token(b)*Memory(a).
2. *Memory(c)* and *Token(c):*
   When the right predecessor node is a negative one-input node, Memory(c) is represented by Memory(a)*Ratio(c), otherwise by Memory(a)*Memory(b)*Ratio(c). Similarly, when the right predecessor node is a negative one-input node, Token(c) is represented by Token(a)*Ratio(c), otherwise by Test(c)*Ratio(c). This is because the negative condition element filters tokens passed from the left predecessor node.
3. *Ratio(c):*
   The accurate value of Ratio(c) is difficult to know, because it depends on the correlation between tokens to be joined. We use Ratio(b) for an approximate value of Ratio(c), even when the join structure is different from the measured one. Various techniques have been developed to refine the ratio, but these will not be discussed in this paper due to space limitations.
4. *Cost(c):*
   In general, the local cost at c can be represented by $C1$*Test(c)+$C2$*Token(c), where $C1$ and $C2$ are appropriate constants. In this paper, we use $C1$=1 and $C2$=0 in order to set a clearly defined goal: *reducing the number of inter-condition tests.* Thus Cost(c) is represented by Cost(a)+Cost(b)+Test(c). However, the constants should be adjusted to the production system interpreters. For example, for OPS5, $C1$ may be large, because join operations are executed in a nested-loop structure. For [Gupta *et al.*, 1987], on the other hand, $C2$ may be large, because hash tables are used.

4 Optimization Algorithm

4.1 Outline of the Algorithm

As described above, efficiency heuristics cannot be applied independently, because the heuristics often conflict with one another. For example, applying one heuristic to speed up some particular rule destroys shared join operations, slowing down the overall program [Clayton, 1987]. On the other hand, since there exists an exponential order of possible join structures, a simple generate-and-test method cannot handle this problem. Our approach is to generate join structures under various constraints, which reduce the possibilities dramatically. An outline of the algorithm is shown in Figure 3. The key points are as follows.

1. Sort rules according to their cost, measured from earlier runs of the program. Optimize rules one by one from higher-cost rules. This is done to allow higher-cost rules enough freedom to select join structures. (See multiple-rule optimization, described later.)
2. Before starting the optimization of each rule, the following nodes are registered to the *node-list* of the rule: one-input nodes, each of which corresponds to a condition element of the rule, and *pre-calculated two-input nodes*, which are introduced to reduce search possibilities and to increase sharing join operations. The details of pre-calculated two-input nodes are described later.
3. In the process of optimizing each rule, two-input nodes are created by combining two nodes in the node-list. The created nodes are registered in the node-list if the same join structures have not already been registered. The algorithm chooses newer nodes to accelerate creating a complete join structure of the rule. Constraints proposed later are used to reduce the number of possibilities.
4. After creating all possible join structures, select the lowest-cost complete join structure.

```
clear the rule-list;
push all rules to the rule-list;
sort the rule-list in descending order of cost;
for r from the first rule to the last rule of the rule-list;
  clear the node-list;
  push all one-input nodes of r to the node-list;
  let k be the number of one-input nodes;
  append pre-calculated two-input nodes to the node-list;
  for i from the second node to the last node of the node-list;
    for j from the first node to the i-1th node of the node-list;
      if all constraints are satisfied then do;
        create a two-input node n to join i and j;
        calculate parameters of n;
        push n to just after the max(i,k)th node of the node-list
      end
    end
  end
  find the lowest-cost complete join structure;
  generate an optimized version of r
end
```

Figure 3: Outline of the optimization algorithm

4.2 Constraints for Reducing Possibilities

The following constraints are used for reducing possible join structures.

4.2.1 Minimal-Cost Constraint

The *minimal-cost constraint* prevents the creation of a join structure whose cost is higher than that of the registered one. More formally, let *Conditions(n)* be a set of condition elements included in the join structure of n. Conditions(n)=$\{n\}$, when n is a one-input node. The constraint prevents creating n, if

$\exists m \in$ *node-list*
such that Conditions(n)$\subseteq$Conditions(m), and
Cost(n)$\geq$Cost(m).

In contrast, if n is in the node-list and m is created, then n is removed from the node-list. The minimal-cost constraint guarantees optimality, if the tree-type join topology is assumed.

To take advantage of the minimal-cost constraint, it is important to create large and low-cost join structures in the early stages. In our system, two-input nodes in the original rule are registered as the pre-calculated nodes. Using this technique, we can prevent creating a join structure, whose cost is higher than the original one.

4.2.2 Connectivity Constraint

The *connectivity constraint* prevents an inter-condition test with no shared variable, which produces a full combination of tokens to be joined. More formally, let p and q be one-input nodes, and *Variables(n)* be a set of pattern variables appearing in Conditions(n). The constraint prevents the creation of a two-input node to join n and m, if

(i) Variables(n)$\cap$Variables(m)=$\emptyset$, and
(ii) $\exists$ $p,q \notin$ Conditions(n)$\cup$Conditions(m)
such that Variables(n)$\cap$Variables(p)$\neq\emptyset$, and
Variables(m)$\cap$Variables(q)$\neq\emptyset$.

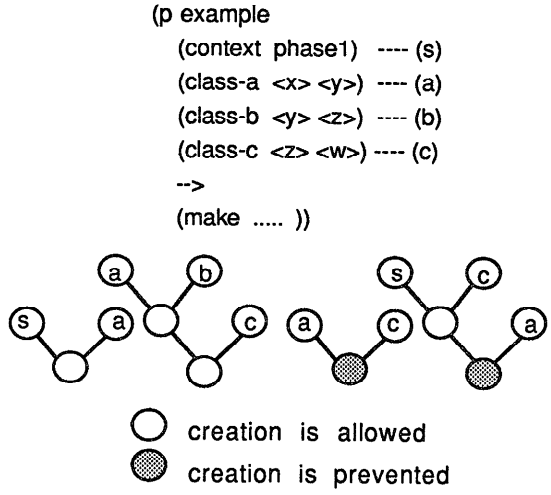


Figure 4: Example of the connectivity constraint

In the example shown in Figure 4, the connectivity constraint prevents joining a and c, because there is no shared variable (thus (i) is satisfied), and there remains a possibility of avoiding such a costly operation, if a and b or b and c are joined first (thus (ii) is satisfied). On the other hand, joining s and a is not prevented, though there is no shared variable. This is because, sooner or later, s will be joined with some node without a shared variable anyway (thus (ii) is not satisfied).

4.2.3 Priority Constraint

Based on the execution statistics, it may be possible to prioritize the one-input nodes. The *priority constraint* prevents creating two-input nodes joining lower-priority nodes while higher-priority nodes can be joined. More formally, let p and q be one-input nodes, and $p \succ q$ indicates that p has a higher priority than q. The constraint prevents the creation of a two-input node to join n and m, if

$\exists$ $p \notin$ Conditions(n)$\cup$Conditions(m),
$\exists$ $q \in$ Conditions(m)
such that (i) $p \succ q$, and
(ii) Variables(n)$\cap$Variables(p)$\neq\emptyset$, or
Variables(n)$\cap$Variables(m)=$\emptyset$.

At present, we define $p \succ q$ only when Token(p)$\geq$Token(q) and Memory(p)$\geq$Memory(q). We introduced (ii) to avoid the situation where joining n and p is prevented by the connectivity constraint while joining n and m is prevented by the priority constraint. The connectivity and the priority constraints can significantly reduce the search possibilities, but sacrifice the guarantee of optimality.

Table 1: Optimization Results

Rule No.	Condition elements	Number of inter-condition tests			Created two-input nodes
		Original	Manual	With Optimizer	
1	21	46432	47888	22396	179
2	18	29548	27244	494	198
3	17	28548	27244	494	92
4	22	25513	7813	144	236
5	21	25513	7813	144	94
6	18	10322	3749	3749	552
7	17	10322	3749	3749	194
* 8	15	9966	9966	12539	228
9	7	9830	1180	1180	99
10	6	9278	630	630	20
11	17	8566	8566	4640	116
12	7	7656	520	760	44
13	6	7544	408	648	22
* 14	23	3160	4616	13780	76
:	:	:	:	:	:
Total 33	Average 14.2	Total 241329	Total 159517	Total 75002	Average 131.8
CPU time (Normalized)		1.00	0.69	0.53	

- The cost of shared nodes is divided by the number of sharing rules.
- The number of tests of marked rules is increased by optimization. However, this does not mean the optimization failed. For example, Nos.1 and 14 share many nodes. The sum of the costs of the two rules has been decreased considerably.

Table 2: Effectiveness of constraints

Rule No.	Condition elements	Number of created two-input nodes		
		Minimal-cost	Minimal-cost Connectivity	Minimal-cost Connectivity Priority
1	21	>1000	680	179
2	18	>1000	438	198
3	17	>1000	162	92
4	22	>1000	887	236
5	21	>1000	139	94
6	18	>1000	>1000	552
7	17	>1000	>1000	194
8	15	>1000	513	228
9	7	121	99	99
10	6	22	20	20
11	17	>1000	382	116
12	7	68	44	44
13	6	24	22	22
:	:	:	:	:

- The priority constraint is applied only when the number of condition elements is greater than 10.

4.3 Multiple-Rule Optimization

Sharing join operations by multiple rules reduces the total cost of a program. We use the following techniques to increase the sharing opportunities.

1. When creating a two input-node n, we assume that n will be shared by all rules which contain Conditions(n). We reduce the value of Cost(n) based on this prediction: the cost is recalculated by dividing the original cost by the number of rules which can share the node.
2. When optimizing each rule, sharable existing two-input nodes are registered in the node-list of the rule as pre-calculated two-input nodes. This time, costs of those two-input nodes are set to 0, because no cost is required to share existing nodes.

Using the above techniques, multiple-rule optimization can be realized without an explosion of combinations. Rules are optimized one by one, but the result is obtained as if all rules are optimized at once.

5 Evaluation

We have implemented an optimizer applicable to OPS5-like production systems. The optimizer reads a program and its execution statistics, then outputs the optimized program. In our system, the overhead of statistics measurement is less than 5%. We apply the optimizer to a real-world production system program, a circuit design expert system currently under development at NTT Laboratories [Ishikawa *et al.*, 1987]. This program consists of 107 rules, which generate and optimize digital circuits. In this evaluation, the approximate number of WMEs representing a circuit is 300 to 400.

There were two reasons why this program was selected as our benchmark. First, the program includes many large rules consisting of more than 20 condition elements. The program is thus not a mere *toy* for evaluating our constraint-based approach to cope with the combinatorial explosion. The second and main reason is that the program was optimized by the expert system builder himself. He spent two to three days optimizing it manually.

The result of optimizing the main module of the program, which consists of 33 rules, is shown in Table 1. The total number of inter-condition tests was reduced to 1/3, and CPU time to 1/2. Perhaps the most important thing to note is that the optimizer produces a more efficient program than the one obtained by manual optimization.

The optimization time is directly proportional to the square of the number of created two-input nodes. The effectiveness of the constraints is shown in Table 2. Without the minimal-cost constraint, it is impossible to optimize rules which contain more than 10 condition elements. The connectivity and the priority constraints also demonstrate significant effects. Currently, the initial version of the optimizer takes somewhat more than 10 minutes on a Symbolics work station to optimize the circuit design program. For average production system programs, in which the number of condition elements is 5 or so, optimization is usually completed in a few minutes.

6 Related Work

The TREAT algorithm [Miranker, 1987] optimizes join operations dynamically. The method is called *seed-ordering*, where the changed alpha-memory is considered first, and the order of the remaining condition elements is retained. Since the overhead cannot be ignored for run-time optimization, sophisticated techniques such as those described here cannot be applied.

Compile-time optimization has been studied for conjunctive queries in AI and database areas [Smith *et al.*, 1985; Warren, 1981; Jarke *et al.*, 1984]. Various heuristics are investigated to determine the best ordering of a set of conjuncts. The SOAR reorderer [Scales, 1986] attempts to directly apply those heuristics to the optimization of production rules. However, we found that applying them to production rules often fails to produce better join structures.

Most of the previous studies on optimizing conjunctive queries are based only on statistics about sizes of the WM. Since production systems can be seen as programs on a database, statistics about changes in the WM (*program behavior of production systems*) should also be considered. This makes optimization of production rules more complex than that of conjunctive queries.

The connectivity and the priority constraints proposed in this paper are respectively based on the *connectivity* and the *cheapest-first heuristics* described in [Smith *et al.*, 1985]. However, the program behavior forces changes in the usage of those heuristics. In previous works, the heuristics are used to directly produce semi-optimal queries. In this paper, we modify the heuristics to be a slightly weaker or less limiting, and use them as constraints to reduce the possibilities.

Many papers have also been published on the subject of parallel matching [Gupta *et al.*, 1987] and parallel firing [Ishida *et al.*, 1985] of production system programs. Since many of these studies have assumed the RETE pattern matching, the optimization algorithm proposed here is also effective in the parallel execution environment.

7 Conclusion

We have explored an optimization algorithm for production system programs. Applying the algorithm to a design expert system demonstrates that the algorithm produces a better program than one optimized by expert system builders. The complexity of optimization increases when the number of rules and working memory elements becomes larger. We believe the algorithm will release both expert system builders and users from time-consuming optimization tasks.

Acknowledgment

The author wishes to thank Yuzou Ishikawa for providing a circuit design expert system for this study, and Ryohei Nakano, Kazuhiro Kuwabara and Makoto Yokoo for their participation in helpful discussions.

References

[Brownston *et al.*, 1985] L. Brownston, R. Farrell, E. Kant and N. Martin. *Programming Expert System in OPS5: An Introduction to Rule Based Programming.* Addison-Wesley, 1985.

[Clayton, 1987] B. D. Clayton. *ART Programming Tutorial, Volume Three: Advanced Topics in ART.* Inference Corp, 1987.

[Forgy, 1981] C. L. Forgy. *OPS5 User's Manual.* CS-81-135, Carnegie Mellon University, 1981.

[Forgy, 1982] C. L. Forgy. A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17-37, 1982.

[Gupta *et al.*, 1987] A. Gupta, C. L. Forgy, D. Kalp, A. Newell and M. Tambe. Results of Parallel Implementation of OPS5 on the Encore Multiprocessor. CS-87-146, Carnegie Mellon University, 1987.

[Ishida *et al.*, 1985] T. Ishida and S. J. Stolfo. Towards the Parallel Execution of Rules in Production System programs. In *Proceedings of International Conference on Parallel Processing*, pages 568-575, 1985.

[Ishikawa *et al.*, 1987] Y. Ishikawa, H. Nakanishi and Y. Nakamura. An Expert System for Optimizing Logic Circuits. In *Proceedings of the 34th National Convention of Information Processing Society of Japan (in Japanese)*, pages 1391-1392, 1987.

[Jarke *et al.*, 1984] M. Jarke and J. Koch. Query Optimization in Database Systems. *Computing Surveys*, 16(2):111-152, 1984.

[Miranker, 1987] D. P. Miranker. TREAT: A Better Match Algorithm for AI Production Systems. In *Proceedings AAAI-87*, pages 42-47, 1987.

[Scales, 1986] D. J. Scales. Efficient Matching Algorithms for the SOAR/OPS5 Production System. STAN-CS-86-1124, Stanford University, 1986.

[Schor *et al.*, 1986] M. I. Schor, T. P. Daly, H. S. Lee and B. R. Tibbitts. Advances in RETE Pattern Matching. In *Proceedings AAAI-86*, pages 226-232, 1986.

[Smith *et al.*, 1985] D. E. Smith and M. R. Genesereth. Ordering Conjunctive Queries. *Artificial Intelligence*, 26:171-215, 1985.

[Warren, 1981] D. H. D. Warren. Efficient Processing of Interactive Relational Database Queries Expressed in Logic. In *Proceedings of the 7th VLDB*, pages 272-281, 1981.