

2

MENTATION PAGE

Form Approved
OMB No. 0704-0188

AD-A228 873

is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this form, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Avenue, S.W., Washington, DC 20543.

REPORT DATE

3. REPORT TYPE AND DATES COVERED

Final Report, 1 Aug 88 to 31 Jul 90

4. TITLE AND SUBTITLE

RESEARCH INTO THE DESIGN AND IMPLEMENTATION OF KNOWLEDGE
BASE SYSTEMS

5. FUNDING NUMBERS

AFOSR-88-0266
61102F 2304/A7

6. AUTHOR(S)

Jeffrey D. Ullman

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Stanford University
Department of Computer Science
Stanford, CA 943058. PERFORMING ORGANIZATION
REPORT NUMBER

AFOSR-R- 00 1106

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFOSR/NM
Bldg 410
Bolling AFB DC 20332-644810. SPONSORING/MONITORING
AGENCY REPORT NUMBER

AFOSR-88-0266

11. SUPPLEMENTARY NOTES

DTIC

ECTE

NOV 16 1990

S B C D

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release;
distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The general goal of the work has been to develop the techniques needed to process queries, expressed as logic programs, efficiently. A system called NAIL was developed, by mid-1989, to test out our ideas. It was fully declarative, which we found an interesting challenge, and its implementation exposed a number of issues that lead to important new ideas and research. However, the full declarativeness proved too much of a burden in writing some applications that we hoped would be facilitated by a logic/database language, and NAIL was abandoned in favor of a new language, called GLUE, that is logical, but that allows for controlled-flow, sets as data values, aggregation operators such as sums of average. NAIL now serves as the view facility for GLUE, and we are in the process of writing a NAIL-to-GLUE translator that will offer both the nondeclarative capabilities of GLUE and the declarative capabilities of NAIL, whichever is more appropriate in a given situation. (KB)

14. SUBJECT TERMS

15. NUMBER OF PAGES

7

16. PRICE CODE

17. SECURITY CLASSIFICATION
OF REPORT

UNCLASSIFIED

18. SECURITY CLASSIFICATION
OF THIS PAGE

UNCLASSIFIED

19. SECURITY CLASSIFICATION
OF ABSTRACT

UNCLASSIFIED

20. LIMITATION OF ABSTRACT

SAR

90 11 15 068

RESEARCH INTO THE DESIGN AND IMPLEMENTATION OF KNOWLEDGE-BASE SYSTEMS

REPORT ON AFOSR GRANT 88-0266

For the two years Aug., 1988—July, 1990

Jeffrey D. Ullman, PI

The general goal of the work has been to develop the techniques needed to process queries, expressed as logic programs, efficiently. A system called NAIL was developed, by mid-1989, to test out our ideas. It was fully declarative, which we found an interesting challenge, and its implementation exposed a number of issues that lead to important new ideas and research. However, the full declarativeness proved too much of a burden in writing some applications that we hoped would be facilitated by a logic/database language, and NAIL was abandoned in favor of a new language, called GLUE, that is logical, but that allows for control-flow, sets as data values, and aggregation operators such as sum or average. NAIL now serves as the view facility for GLUE, and we are in the process of writing a NAIL-to-GLUE translator that will offer both the nondeclarative capabilities of GLUE and the declarative capabilities of NAIL, whichever is more appropriate in a given situation. The current status of the GLUE/NAIL implementation is as follows.

1. Geoff Phipps is the principal designer of GLUE, and has a "quick-and-dirty" implementation, in which GLUE programs are translated to Prolog and executed. Details of the language appear in Phipps et al. [1990]. There are a number of novel solutions to common problems in logic languages. For example, we store sets by name, rather than value, an approach that has made set processing awkward in logic languages like LDL or Prolog. There is a diction (caret) for returning relations from function calls in a way that prevents them from dangling. Eventually, Phipps will rewrite the translator to produce IGLOO, an intermediate form that will be executed more efficiently than the Prolog output.
2. Marcia Derr is working on the implementation of IGLOO, the intermediate form. She has ideas for dynamic creation of index structures and ordering of subgoals that will make GLUE execution much faster than other logic languages.
3. Ashish Gupta is working on the translation of NAIL, the fully-declarative view facility, into GLUE. The problems he encounters involve how general NAIL rules can be. For example, when negated subgoals and/or aggregation are allowed, sometimes the rules do not have a natural meaning, or it becomes impossible to apply the "magic sets" transformation (described below) to produce an efficient GLUE program from NAIL rules.
4. David Chang and Kathleen Fisher, two undergraduates, are working on applications programs in GLUE. This work has been a significant help in focusing the design of the language. Chang's work involves economic theory, in particular deducing dependence relationships among parameters of economic activity, and Fisher's involves simulating and controlling the flow of automobiles through a grid of streets and traffic lights.

It is interesting to compare the performance of a program written by Chang in LDL before the early GLUE implementation became available, with his later implementation in GLUE. The problem was a simple one: matrix inverse by Gaussian elimination. His LDL

program took one hour to compile on a SUN-3. His GLUE program compiles in a minute on a slower machine. Moreover, adjusting for differences in machine speed, the GLUE version executes faster, even though it is really executing Prolog code. And we have not yet begun to get serious about efficiency, being happy at this stage to experiment with language dictions.

Magic-Sets

This technique for rewriting rules is probably the most important idea that has come up in the study of logical query optimization. It allows us to take a set of logical rules and a query, and rewrite the rules so that they can be evaluated bottom-up (by forward chaining, or reasoning from rule body to rule head) as fast as the best top-down (backward-chaining, or goal-directed search, as in Prolog) algorithm. Moreover, the rules need be rewritten only once for each query form, (pattern of bound and free arguments in a predicate), rather than for each query. Since bottom-up evaluation is used, we are not subject to the pitfalls of top-down evaluation, such as left-recursive loops. However, since the new rules simulate a top-down derivation (in much the way an LR parser simulates an LL parser in the LR parser's sets of items), we are not subject to the typical bottom-up problem of proving too many facts that are irrelevant to the query.

Example 1: If presented with the transitive closure rules (using Prolog notation)

```
path(X,Y) :- arc(X,Y).
path(X,Y) :- arc(X,Z) & path(Z,Y).
```

and the query $path(0, W)$, i.e., "what nodes W can I reach from node 0?", one possible magic-sets transformation would construct the rules

```
m_path(0).
m_path(Z) :- m_path(X) & arc(X,Z).

path(X,Y) :- m_path(X) & arc(X,Y).
path(X,Y) :- m_path(X) & arc(X,Z) & path(Z,Y).
```

The predicate m_path ("magic path") represents the set of nodes v that are relevant, in the sense that a Prolog-like search starting with goal $path(0, W)$ would call a subgoal $path(S, T)$, where S is a set of nodes including v . Then, m_path is used in the last two rules to restrict the set of $path$ facts that we are allowed to infer. Note that the third and fourth rules above are the original rules with the magic predicate added to the bodies. $\square$

The original magic-sets idea was an early result of the NAIL project, dating from 1986. A recent contribution to the theory, supported by the grant is Ullman [1989], which shows that for datalog (Horn-clauses with no function symbols), one can do a magic-sets transformation that needs no nonground tuples, and yet is guaranteed to perform at least as well as any top-down method.

The idea has been extended to allow constraints on arguments (the original techniques assumed only that an argument could be bound to a set of constants, not constrained to an infinite set, as by $X > 5$). The paper by Mumick, Finkelstein, Pirahesh, and Ramakrishnan [1990a] describes this advance. Mumick, Pirahesh, and Ramakrishnan [1990] shows how



or	☒
	☐
	☐
y Codes	
and/or	
special	

A-1

magic-sets extends to relations in which duplicates are significant (i.e., you must count the number of times a tuple appears) and/or certain kinds of aggregation as well.

Mumick, Finkelstein, Pirahesh, and Ramakrishnan [1990b] tells of the implementation of magic sets in IBM's experimental Starburst DBMS and gives some experimental evidence that the method is superior not only for recursive queries, for which it was intended, but for nonrecursive queries as well. A further extension by Ken Ross is described in the section on well-founded semantics.

Beyond Magic-Sets

There are certain situations in which specialized techniques give better results than magic-sets for rules to which they apply. Several years ago, Jeff Naughton began a study of left- and right-linear recursions, under the project. These methods applied initially to rules that were similar to transitive closure. Recently, the techniques have been generalized to provide linear-time algorithms for a large class of linear rules (that is, rules with only one recursive subgoal), in Naughton, Ramakrishnan, Sagiv, and Ullman [1989a], and even to some rules that are nonlinear, in Naughton, Ramakrishnan, Sagiv, and Ullman [1989b].

In a different vein, Sagiv [1990] has explored the technique of *envelopes*, which are similar to magic-sets, in that they attempt to restrict the set of facts that one must deal with during the answering of a query. However, there are examples where envelopes give better results than magic-sets, as well as examples where the opposite is true. Moreover, Sagiv shows how one can combine envelopes and magic sets, or apply the enveloping transformation several times, to get successive improvements. That is, we can apply several transformations to rules, one at a time, and get better rules at each step. This area is intriguing, and not yet fully developed.

Logic with Negated Subgoals

When we try to implement logical rules with negated subgoals, we run up against the problem that there usually exists more than one minimal fixed point, and we need to select one such as the "meaning" of the rules.

Example 2: Consider the rules

$$\begin{aligned} r(X) &:- p(X) \ \& \ \text{not } q(X). \\ s(X) &:- p(X) \ \& \ \text{not } r(X). \end{aligned}$$

Suppose that p is defined by a database relation and currently contains the tuples 1 and 2. Also, q is a database relation and currently has only the tuple 1. Then one minimal model of these rules and data is $r = \{2\}$ and $s = \{1\}$. However, there is another minimal model, where $r = \{1, 2\}$ and $s = \emptyset$. The first of these is intuitively the "correct" model, since there does not seem to be any explanation for why we should believe $r(1)$ is true. $\square$

The first attempt to resolve this ambiguity was by restricting to "stratified" rules, where negation could not be wrapped inside a recursion. In 1986, Allen Van Gelder, working on NAIL, and independently, Apt, Blair, and Walker at IBM, defined *stratified logic programs* to be those for which we can partition their predicates into *strata*, such that a rule for a stratum i predicate can only involve unnegated predicates of stratum i

and lower, and negated predicates only of lower strata. For instance, the rules of Example 2 are stratified, with p and q in stratum 0, r in stratum 1, and s in stratum 2.

To find the fixed point of a stratified program, we evaluate predicates in order of their strata. Thus, in Example 2, we first evaluate $r = \{2\}$, and only then do we evaluate s , using this value of r in the second rule, to get $s = \{1\}$. That is the intuitively correct fixed point.

However, there are examples of logic programs in which recursion and negation are intimately connected, and these programs are not stratified. A key step was taken by Van Gelder, Ken Ross, and John Schlipf in 1988. They proposed the *well-founded semantics*, which gives a reasonable three-valued (true, false, unknown) model for any logic program. The intuitive idea is that there are two ways in which we can draw conclusions.

1. Ordinary deduction. If we instantiate a rule (replace variables by constants), and we find that all the unnegated subgoals of the body are known to be true and all the negated subgoals are known to be false, then we may conclude that the head is true.
2. *Unfounded sets*. Suppose we have a collection U of instantiated predicates such that every instantiated rule with a member of U in the head has the property that either
 - a) There is a fatal problem with some subgoal: either a positive subgoal is known false or a negative subgoal is known true, or
 - b) There is a positive subgoal of the rule that is in U .

Then there is no way we could ever prove a member of U by (1), because some other member of U would have to be proved first. Under the well-founded semantics, we conclude every p in U is false.

Example 3: An instructive example is the rule

$\text{win}(X) \text{ :- move}(X,Y) \ \& \ \text{not win}(Y).$

That is, a stalemate position, with no possible move, is a loss, and one can win by selecting a move that puts your opponent in a losing position. The well-founded semantics does exactly what we intuitively expect. If best play by both sides from board X leads to a win for the first player, then $\text{win}(X)$ is made true. If the first player is forced to lose, then $\text{win}(X)$ is made false. And if best play leads to an infinite sequence of moves, with no resolution, then $\text{win}(X)$ will not be found true or false, and it is given the value "unknown."
□

There has been some significant development of the well-founded semantics, with an eye toward discovering efficient algorithms for answering queries according to this interpretation of rules. For instance, we would like to ask of the rule in Example 3, the query $\text{win}(43)$, i.e., "is board 43 a win?" and get the answer quickly. Ross [1989a] gave such a top-down evaluation algorithm.¹ Ross [1989b] extends the idea to disjunctive rules, where the head can be the "or" of two or more atoms. Also, Ross [1990a] attempts to incorporate not only negation, but aggregation (functions like min or average applied to columns of a relation). For a subclass called *modularly stratified*, he shows that it is possible to

¹ Note that the original paper on well-founded semantics does not give an algorithm for the general case, just a definition.

apply a magic-sets transformation to rules with aggregation and get efficient evaluation thereby. Finally, Ross [1990b] examines what happens when the well-founded semantics and also the generalized stable-model semantics of Gelfond and Lifschitz are applied to HiLog programs.²

Applications of Conjunctive Query Containment

Conjunctive queries are the bodies of datalog rules, that is, the logical “and” of atoms. Any datalog program and query can be expanded into an infinite set of conjunctive queries whose union is the answer to the query. Sagiv, supported by the project in 1987, showed how one can test whether the result of a given conjunctive query is contained in the result of a datalog program and query. This test was applied in Ramakrishnan, Sagiv, Ullman, and Vardi [1989] to give a test for commutativity of linear rules (rules with at most one recursive subgoal in their bodies) that is more general than any previously considered. Incidentally, the motivation for studying rule commutativity is that when two linear rules r_1 and r_2 for a predicate p commute, one can evaluate p by first closing the database under r_1 and then under r_2 . Using some evaluation strategies like “counting,” we can reduce exponential work to polynomial work.

Plambeck [1990] looks at the general question of algebraic equivalences among rules, of which commutativity is one important example. He uses some known semigroup theory to characterize all algebraic identities that can hold for a collection of rules.

Another problem attacked by this theory is “ZYT-linearizability.”³ Here, we are asked to take a nonlinear recursion with two occurrences of the recursive subgoal in one nonrecursive rule, and replace one of the occurrences of the recursive predicate by the basis predicate. The motivation for doing so is that linear recursions are often easier to evaluate than nonlinear recursions. Conversely, if we are executing rules in parallel, we can often get speedup by replacing a linear recursion by an equivalent nonlinear recursion; i.e., apply the ZYT transformation in reverse.

Example 4: The simplest example is bilinear transitive closure,

```
path(X,Y) :- arc(X,Y).
path(X,Y) :- path(X,Z) & path(Z,Y).
```

which is equivalent to the linear recursion of Example 1. That is, we replace the first *path* subgoal by the identical subgoal with predicate *arc*, the basis predicate. $\square$

Saraiya [1989] gave a polynomial-time test for ZYT-linearizability in the case that the recursive rule does not have two subgoals with the same nonrecursive predicate. Saraiya [1990a] gives efficient tests for a number of properties of rules with nonrepeating names among the nonrecursive subgoals, while Saraiya [1990b] shows that it is not possible to extend these results significantly, since the problem becomes $\mathcal{NP}$ -hard. Ramakrishnan, Sagiv, Ullman, and Vardi [1989] gives the broadest known test for ZYT-linearizability that does not limit the form of the nonrecursive subgoals.

² HiLog is a logic proposed by Chen, Kifer, and Warren, that incorporates sets and other second-order logical features in a language with a first-order semantics.

³ After W. Zhang, C. T. Yu, and D. Troy, who first proposed an algorithm to decide this question.

I/O Complexity of Transitive Closure

When computing a large transitive closure, one that cannot fit in main memory, the critical cost is often the amount of data that must be moved between secondary and main storage. We can parametrize the problem by letting n be the number of nodes, e the number of arcs, and s the number of facts that will fit in main memory at one time. We assume $n < s < e < n^2$, because otherwise the best algorithm is obvious. About three years ago, there was published in SIGMOD an algorithm for handling transitive closure using n^4/s I/O, that is, moving $O(n^4/s)$ facts across the boundary between main and secondary memory. Interestingly, way back in 1974, Coffman and McKellar published in JACM an algorithm taking only $n^3/\sqrt{s}$ I/O, which suggests that people are not learning from history.

Ullman and Yannakakis [1990] looked at the inherent complexity of the problem to see if there were even better algorithms. We showed, however, that $n^3/\sqrt{s}$ I/O is optimal for dense graphs ($e = O(n^2)$), as long as the algorithm is "standard," in the sense that it only infers path facts like $path(A, B)$ if it has the three arc or path facts involving nodes A , B , and some third node C in main memory simultaneously.

We then looked at how well you could do if you assumed the graph is sparse, i.e., e much less than n^2 . There, we show that I/O equal to $n^2\sqrt{e/s}$ is sufficient. Moreover, any standard algorithm requires this much I/O.

REFERENCES

Mumick, I. S., S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan [1990a]. "Magic conditions," *Proc. Ninth ACM Symposium on Principles of Database Systems*, pp. 314-330.

Mumick, I. S., S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan [1990b]. "Magic is relevant," *ACM SIGMOD International Conf. on Management of Data*, pp. 247-258.

Mumick, I. S., H. Pirahesh, and R. Ramakrishnan [1990]. "The magic of duplicates and aggregates," *Intl. Conf. on Very Large Databases*, Aug., 1990.

Naughton, J. F., R. Ramakrishnan, Y. Sagiv, and J. D. Ullman [1989a]. "Efficient evaluation of left-linear and right-linear rules," *ACM SIGMOD International Conf. on Management of Data*, pp. 235-242.

Naughton, J. F., R. Ramakrishnan, Y. Sagiv, and J. D. Ullman [1989b]. "Argument reduction through factoring," *Proc. International Conference on Very Large Data Bases*, pp. 173-182.

Phipps, G. et al. [1990]. "Glue-Nail definition," unpublished language manual, Dept. of CS, Stanford Univ., Stanford CA, July, 1990.

Plambeck, T. [1990]. "Semigroup techniques in recursive query optimization," *Proc. Ninth ACM Symposium on Principles of Database Systems*, pp. 145-153.

Ramakrishnan, R., Y. Sagiv, J. D. Ullman, and M. Y. Vardi [1989]. "Proof tree transformation theorems and their applications," *Proc. Eighth ACM Symposium on Principles of Database Systems*, pp. 172-181.

Ross, K. A. [1989a]. "A procedural semantics for well founded negation in logic programs," *Proc. Eighth ACM Symposium on Principles of Database Systems*, pp. 22-33.

Ross, K. A. [1989b]. "The well-founded semantics for disjunctive logic programs," *Proc. First Intl. Conf. on Deductive and Object-Oriented Databases*, Kyoto.

Ross, K. A. [1990a]. "Modular stratification and magic sets for datalog programs with negation," *Proc. Ninth ACM Symposium on Principles of Database Systems*, pp. 161-171.

Ross, K. A. [1990b]. "On negation in HiLog," unpublished memorandum, Stanford Univ., Dept. of CS.

Sagiv, Y. [1990]. "Is there anything better than magic," to appear in *Proc. NACLP*, Oct., 1990.

Saraiya, Y. [1989]. "Linearizing nonlinear recursions in polynomial time," *Proc. Eighth ACM Symposium on Principles of Database Systems*, pp. 182-189.

Saraiya, Y. [1990a]. "Polynomial time program transformations in deductive databases," *Proc. Ninth ACM Symposium on Principles of Database Systems*, pp. 132-144.

Saraiya, Y. [1990b]. "Hard problems for simple logic programs," *ACM SIGMOD International Conf. on Management of Data*, pp. 64-73.

Ullman, J. D. [1989]. "Bottom-up beats top-down for datalog," *Proc. Eighth ACM Symposium on Principles of Database Systems*, pp. 140-149.

Ullman, J. D. [1990]. "The theory of deductive databases," *IEEE COMPCON*, pp. 496-502, March, 1990.

Ullman, J. D. and M. Yannakakis [1990]. "The input/output complexity of transitive closure," *ACM SIGMOD International Conf. on Management of Data*, pp. 44-53.

Doctoral Theses Supported by Grant

1. Thane Plambeck, "Semigroups and Transitive Closure in Deductive Databases," completed Aug., 1990.
2. Yatin Saraiya, "Transformations on Logical Recursions," final oral exam July, 1990.
3. Kate Morris, "Subgoal Order for Query Optimization in Logic Databases," final oral exam July, 1990.