5

Rapid #: -3372030

Ariel IP: 157.182.199.203

Status	Rapid Code	Branch Name	Start Date
Pending	AZS	Main Library	4/23/2010 7:28:32 AM

CALL #: **QA 76.9 .D3 I5589a**

LOCATION: **AZS :: Main Library :: SCIENCE 2nd FLOOR**

TYPE:	Article CC:CCL
JOURNAL TITLE:	Very large data bases : proceedings / International Conference on Very Large Data Bases.
USER JOURNAL TITLE:	Very Large Data Bases, 7th International Conference
AZS CATALOG TITLE:	Very large data bases : proceedings / International Conference on Very Large Data Bases.
ARTICLE TITLE:	Efficient Processing of Interactive Relational Data Base Queries expressed in Logic
ARTICLE AUTHOR:	Warren
VOLUME:	
ISSUE:	
MONTH:	
YEAR:	1981
PAGES:	
ISSN:	AZS ISSN: 0253-0325
OCLC #:	8103070
CROSS REFERENCE ID:	420907
VERIFIED:	ISSN crossmatch found using OCLC number

BORROWER: **WVU :: Evansdale Library**

PATRON: **Menzies,TIm**

PATRON ID:	-
PATRON ADDRESS:	-
PATRON PHONE:	-
PATRON FAX:	-
PATRON E-MAIL:	-
PATRON DEPT:	-
PATRON STATUS:	-
PATRON NOTES:	-

RAPID ILL

System Date/Time: 4/23/2010 8:20:19 AM MST

EFFICIENT PROCESSING OF INTERACTIVE RELATIONAL DATABASE QUERIES EXPRESSED IN LOGIC

David H D Warren

Department of Artificial Intelligence
University of Edinburgh

ABSTRACT

Relational database retrieval is viewed as a special case of deduction in logic. It is argued that expressing a query in logic clarifies the problems involved in processing it efficiently ("query optimisation"). The paper describes a simple but effective strategy for planning a query so that it can be efficiently executed by the elementary deductive mechanism provided in the programming language Prolog. This planning algorithm has been implemented as part of a natural language question answering system, called Chat-80. The Chat-80 method of query planning and execution is compared with the strategies used in other relational database systems, particularly Ingres and System R.

INTRODUCTION

Is it possible to design practical computer systems which can answer questions expressed in precisely defined subsets of natural language? This is the question which motivates the research project I have been working on, in collaboration with Fernando Pereira. A major part of the problem, and the main subject of this paper, is the following. Given that one understands the natural language question, and that one has available the information needed to produce an answer, how does one go about effectively producing the answer? This is essentially a generalisation of the central problem of relational databases - how can one efficiently handle queries expressed in a non-procedural formalism that is intended to insulate the user from the details of how information is stored in the database?

Notice that I prefer not to view the problem as one of providing natural language front-ends to databases. This seems to me to be putting the cart before the horse. Rather my approach has been to investigate what "back-end" is needed to support natural language question answering.

The starting point for our work was a question answering system for a small subset of Spanish, implemented by Veronica Dahl [5] [4], following the approach advocated by Alain Colmerauer [3]. We were particularly attracted to Colmerauer's approach for the clear insight it gives into some of the essential problems of natural language question answering. In addition, it proved very easy to adapt Dahl's program to English (and to a different domain). This was due in large part to the fact that the system is implemented in Prolog [13] [21], a programming language based on first-order logic devised by Colmerauer and his colleagues around 1972. We called the adapted program "Chat".

The most serious shortcoming of Chat, we found, was that the query answering process is quite impracticable for a database of any significant size. The problem appears to be intrinsic to the semantics Colmerauer gives to what he calls "three-branched quantifiers" (3BQs). Furthermore, the semantics fails to correctly model natural language in certain cases. To tackle these problems, I decided instead to try translating 3BQ expressions directly into first-order logic (or something as close to first-order logic as possible). (The same approach has been taken by Michael McCord [8], who has independently been developing a natural language system drawing on the ideas in Chat). Using first order logic has the major advantage that the intermediate formalism is directly executable as a Prolog program. Furthermore, the first order logic formulation of a query lends itself to transformations which can improve the efficiency, corresponding to what is usually called "query optimisation". Essentially these transformations involve augmenting the logic with extra control information to produce efficient Prolog. This task is normally the responsibility of the Prolog programmer. The main topic of this paper will be to describe a simple planning algorithm which tackles the same task and produces good results.

This approach to query semantics and query processing, together with a more comprehensive English grammar written by Pereira, have been incorporated into a completely new, experimental program which we call "Chat-80". An overall account of this system will be given elsewhere [17]. Chat-80, like its predecessor, is implemented entirely in Prolog. It has been developed using the DEC-10 Prolog compiler/interpreter [19] [10].

In order to test the approach on a non-trivial domain, Chat-80 includes a database of facts about world geography. This domain has the advantage,

CH1701-2/81/0000/0272$00.75

for demonstration purposes, that the facts in the database are generally common knowledge, so it is easier to appreciate what is entailed in answering different queries. The database contains basic facts about the world's countries (over 150 of them), oceans, major seas, major rivers and major cities. The largest relation, 'borders', represents all pairs of countries, oceans or major seas which are adjacent, and contains therefore over 850 tuples. It should be emphasised that the database is itself implemented as ordinary Prolog; it therefore resides within the normal DEC-10 virtual memory.

In order to relate this approach to other work on relational databases, I shall make specific reference to Ingres [15]. Ingres has been singled out because its query languge, Quel, has a particularly simple mapping into first-order logic. Also, Ingres is a system I actually have access to.

In contrast to many current database systems, the design of Chat-80 is aimed exclusively at interactive queries, by which I mean queries that can in principle be answered in a time short enough for it to be reasonable for the user to wait for an immediate answer. Basically this rules out queries which necessarily involve a large number of accesses to very large (and therefore slow to access) relations. Natural language access does not seem so appropriate for non-interactive queries, since here it will pay the user to exercise more control over how the query is executed.

LOGIC AS A DATABASE FORMALISM

Several current relational database formalisms have a core which can be viewed as no more than a syntactic variant for a certain subset of logic. This is particularly true of the Quel formalism supported by Ingres. To illustrate this, let us consider an example used by the Ingres authors [15]:

```
range of E,M is employee
range of D is dept
retrieve (E.name)
where E.salary > M.salary
and   E.manager = M.name
and   E.dept = D.dept
and   D.floor = 1
and   E.age > 40
```

In ordinary English, this query means:

> "Which employees aged over 40 on the first floor earn more than their managers?"

The query refers to relations:

```
employee(name,dept,salary,manager,age)
dept(dept,floor)
```

This query can be expressed in logic (using a Prolog oriented syntax) as:

```
answer(E) <=
   employee(E,D,S,M,A) & A > 40 &
   dept(D,1) &
   employee(M,_,S1,_,_) & S > S1.
```

Read this as:

```
E is an answer if
   E is an employee, dept D, salary S,
                     manager M, age A, and
   A is greater than 40 and
   D is a department on floor 1 and
   M is an employee, salary S1, and
   S is greater than S1.
```

Here the identifiers starting with a capital letter, such as E, D, S, etc., are logic variables, which can be thought of as standing for arbitrary objects of the domain. Contrast this with the variables of Quel, which denote arbitrary tuples of a certain relation specified in the range statement. (Because, in this example, tuples can be uniquely identified by their first fields, it is natural for the logic variable corresponding to this field to have the same identifier as is used for the tuple variable in the Quel version). For each tuple variable in a Quel query, there is, in the logic version, a corresponding goal (also called "atomic formula"), eg.

```
dept(D,1)
```

A goal consists of a predicate, naming the range relation of the corresponding tuple variable, applied to some arguments, corresponding to the fields of this relation. Quel constraints which are identities map into an appropriate choice of variables or constants (such as '1') for certain goal arguments. This aspect tends to make the logic form of the query more concise and, it can be argued, easier to comprehend. Note the use of '_' to denote an "anonymous" variable, which is only referred to once, and which therefore does not need to be given a distinct name. Quel constraints which are inequalities map into separate logic goals. The Quel query as a whole maps into a restricted kind of implication, called a clause, where the target of the query appears as the conclusion of the implication (to the left of the '<='.

Clauses can be used not only to represent queries, but also to express the information which makes up the database itself. (It is this aspect which distinguishes what will be described here from much other work relating logic and databases).

In general a clause consists of an implication, which in the Prolog subset of logic is restricted to the form:

```
P <= Q1 & Q2 & ... Qn.
```

meaning "P is true if Q1 and Q2 and ... Qn are true", where P and the Qi may be any goals. If n = 0, we have what is called a unit clause, which is written simply as:

P.

meaning "P is true".

For example, here are some unit clauses, representing elementary facts, which serve to define which tuples make up a relation 'parent'.

```
parent(david,hugh).
parent(david,winifred).
parent(ben,david).
parent(ben,jane).
```

The first clause, for instance, may be read as:

"David has a parent Hugh".

Here we have defined a database relation by explicitly enumerating its tuples. However it is also possible to define a relation implicitly, through general rules expressed as non-unit clauses. For example, here is a definition of the 'ancestor' relation in terms of the 'parent' relation:

```
ancestor(X,Z) <= parent(X,Z).
ancestor(X,Z) <= parent(X,Y) & ancestor(Y,Z).
```

Read these clauses as:

"X has an ancestor Z if X has a parent Z".
"X has an ancestor Z if
X has a parent Y and Y has an ancestor Z".

Note that the second clause makes the definition recursive. We can think of 'ancestor' as a "virtual" relation. A pair <X,Y> belongs to the 'ancestor' relation if:

```
ancestor(X,Y)
```

is a logical consequence of the clauses which make up the database. Thus one can infer, for example, that one of Ben's ancestors is Hugh, ie.

```
ancestor(ben,hugh)
```

This use of logic clauses to define a database gives much greater power and conciseness than is available in most conventional relational database systems, including Ingres. These systems do not allow an equivalent recursive definition of the 'ancestor' relation, for example.

In fact, the logic subset we have been looking at forms the basis of a general purpose programming language, Prolog. A Prolog system is essentially a machine which can generate solutions to a problem by enumerating all instances of some goal which are valid inferences from the clauses which make up a "program". For example, if the user presents the query:

```
answer(X) <= ancestor(ben,X).
```

Prolog responds with the following list of possible values for X, representing all the ancestors of Ben that can be deduced:

```
X = david; X = jane; X = hugh; X = winifred
```

The solutions are in fact produced in exactly this order. How this takes place will now be described.

In Prolog, the ordering of clauses in a program, and the ordering of goals in the right-hand side of a clause, is important control information, which helps to determine the way a program is executed.

To execute a goal (such as 'ancestor(ben,X)' in the previous query), Prolog tries to match it against the left-hand side of some clause, by finding values for variables which make the goal and the clause "head" identical. When successful, Prolog then recursively executes the goals (if any) in the right-hand side of the clause, which will by now have been modified by the results of the matching. When no match can be found, or when there are no more goals left to execute, Prolog backtracks. That is it goes back to the goal most recently matched, undoes the effects of the match, and then seeks an alternative match.

Clauses are tried for a match in the order they appear in the program. Goals in the right-hand side of a clause are executed in the order they appear in that clause. The matching process is actually unification [12], a process which effectively produces the least possible instantiation of variables necessary to make the two goals identical.

Prolog's backtracking can be thought of as a generalised form of iteration. Thus the two clauses for 'ancestor', when used to satisfy a goal such as 'ancestor(ben,X)', give a behaviour when executed by Prolog equivalent to the following procedure:

```
To generate Zs who are ancestors of X:
    first generate Zs who are parents of X;
    then for each Y who is a parent of X:
        generate Zs who are ancestors of Y.
```

In fact, the DEC-10 Prolog compiler can compile such clauses into code which is comparable in efficiency with iterative loops in a more conventional language [18].

As a final remark, one should note that the Prolog subset of logic includes, besides the variables and elementary constants seen so far, objects which are structures. In this respect, while being similar to many other programming languages, it is a further important generalisation of most relational database formalisms. A detailed discussion of this aspect is ouside the scope of this paper.

QUERY PLANNING

It should be clear that the efficiency with which Prolog executes a query is critically dependent on the order in which goals are expressed in a conjunction. In the logic programming context, this ordering constitutes extra control information, supplied by the

programmer. The programmer uses his expert judgement, and knowledge of the predicates involved, to determine a suitable order. However, for logical expressions derived from natural language input, such control information is completely lacking (or so it seems). The purpose of query planning, therefore, is to supply the missing control information, by simulating, in a rudimentary fashion, some of the judgements an expert Prolog programmer makes.

Ordering goals in a conjunction

To see how bad an uncontrolled query expression can be, consider the following query taken from the domain of Chat-80:

> "Which countries bordering the Mediterranean border Asian countries?"

The natural language front-end translates this into:

```
answer(C) <=
   country(C) & borders(C,mediterranean) &
   country(C1) & asian(C1) & borders(C,C1).
```

Executed exactly as it stands, this query would take a time of the order of the number of Mediterranean countries multiplied by the total number of countries, ie. roughly 20*150=3000. (And if the goal ordering had happened to be the worst possible, the figure would instead be 150*150=22500). However, after planning, the query is re-ordered as:

```
answer(C) <=
   borders(C,mediterranean) & country(C) &
   borders(C,C1) & asian(C1) & country(C1).
```

and now takes a time of the same order as the number of Mediterranean country borders, ie. roughly 90. For queries which are only a little more complex than this - having say three or four variables instead of just two - the planning will generally make all the difference between an execution time which is perfectly acceptable, and one which is out of the question.

The intended objective of the planning algorithm is to minimise the number of alternatives which have to be considered during the Prolog execution. In conventional database terms, this is equivalent to minimising the number of tuple accesses. The planner attempts to meet this objective by so ordering the goals that the goal to be executed next is the one which can be expected to least increase the number of alternatives being considered, or (better still) to most decrease that number.

The planner assumes a cost function which assigns, for each predicate, and each state of instantiation of its arguments, a numeric value representing the expected factor by which such a goal would multiply the number of possibilities being considered. (For convenience, the planner in practice manipulates numbers which are the logarithms of such factors).

For example, a goal 'country(C)', where C is uninstantiated, involves generating all known countries, multiplying the number of possibilities under consideration by some 150. If however C is already instantiated, the goal is a test which will generally eliminate possibilities, and the cost factor is therefore less than 1.

By its nature, the cost function is something which can only be estimated, and there is wide scope for experiment with different methods for computing it. In practice, however, most reasonable cost functions produce very similar results. The current planner therefore uses a very simple function, which has proved adequate so far, although it could undoubtedly be improved on. The computation depends on some simple statistical information about the sizes of the relations in the database, and the sizes of the domains over which their arguments range. The cost is taken to be the total size (ie. number of tuples) of the relation corresponding to the goal predicate, divided by the product of the sizes of the domains of each instantiated argument position. Note that it is somewhat problematic to decide what constitutes the domain of a predicate argument position.

For the predicates involved in the example given above, relevant statistics might be:

predicate	size	argument domain sizes	
country	150	300	
asian	100	300	
borders	900	180	180

Thus 'borders(C,C1)' has cost:

900	if neither C nor C1 is instantiated,
900/180 = 5	if just one of C and C1 is instantiated,
900/180*180 = 1/36	if both C and C1 are instantiated.

The planning algorithm itself is comparatively simple. It selects the goal of least cost, determines which variables will become instantiated as a result of executing this goal (using certain reasonable assumptions to be discussed later), and then repeats the process for the remaining goals with those variables instantiated.

To illustrate this, the goals of our example have initial costs as follows:

```
150  country(C) &
  5  borders(C,mediterranean) &
150  country(C1) &
100  asian(C1) &
900  borders(C,C1)
```

The goal chosen is therefore 'borders(C, mediterranean)', which will result in instantiating C, to say 'c', leaving the remaining goals with costs:

```
  1/2  country(c) &
150    country(C1) &
100    asian(C1) &
  5    borders(c,C1)
```

The test 'country(c)' is selected for execution next, and this is followed by 'borders(c,C1)', which instantiates C1, to say 'c1'. The situation is now:

```
1/2  country(c1) &
1/3  asian(c1)
```

The figures indicate that 'asian(c1)' is the more restrictive of these two tests, and so this goal is chosen to be executed first. Thus the planning algorithm chooses what is probably the best order for executing the goals.

Note that the implementation (in Prolog) of the planning algorithm is such that it does not actually recompute and re-compare all the costs at every step of the planning process.

An implicit assumption of the planning process is that all tuple accesses (or, more precisely, all successful matches) are equally easy to make. For this to be at all realistic, it is necessary that the database be suitably indexed. For the Chat-80 application, each relation (of significant size) effectively has a separate index for each argument position of the predicate concerned, ie. the database is "totally inverted". (DEC-10 Prolog provides built-in indexing only on the first argument position of a predicate, but by using Prolog's "meta-logical" facilities it is possible to simulate full indexing by defining auxiliary predicates which represent the extra indexes). Of course, full indexing does make the database significantly bigger, although, since the expansion can be kept within some constant factor, say a doubling in size, this does not seem to be a major problem. However, perhaps a better approach would be to postpone creating an index until it is actually found to be needed.

Although the rationale for the planning process is most easily understood in terms of explicit relations, defined via unit clauses, the planner in fact works just as well for queries which refer to virtual relations, defined using non-unit clauses. The goals in the query are simply re-ordered exactly as before and then handed over to Prolog for execution. It is immaterial whether the relation referred to by a goal is explicit or virtual, since this will make no visible difference to the outcome of executing the goal. For example, although one may think of the tuples of the 'asian' relation as being stored explicitly, this relation is in fact defined via general rules, which, simplifying things slightly, are as follows:

```
asian(C) <= in(C,asia).
in(X,Y) <= partof(X,Y).
in(X,Z) <= partof(X,Y) & in(Y,Z).
```

These clauses constitute a piece of Prolog program which serves both to test whether something is Asian and to generate all things that are Asian. (However, for the latter purpose these clauses are rather inefficient; hence the actual definition is somewhat more complicated).

The planner is therefore only responsible for controlling the execution of the query itself; it is up to the definer of the database to make sure that any non-unit clauses he supplies will perform efficiently when executed by Prolog. In principle, it seems that the planner could also be applied to the task of automatically ordering the goals in the right-hand sides of database clauses. However the task is complicated by the fact that clauses can be used for more than one purpose, as we have seen with the definition of 'asian'.

In computing the cost function, the size of a virtual relation is just the number of distinct tuples that can be deduced to belong to that relation. Note that it is possible to make the planner deal sensibly with relations which are infinite. For example, the relation '<' (integer less than) may be taken to have size "infinity*infinity/2" with argument domain sizes each being "infinity", where "infinity" is in practice some very large number. The planner will then only allow a '<' goal to be executed when both arguments are known.

Isolating independent parts of a query

The basic deductive mechanism of Prolog has an inherent defect, shared with all systems based on the inference rule of resolution. The problem is essentially that resolution treats all goals in a conjunction as being dependent. This is fine so long as the goals share uninstantiated variables. However when two parts of a conjunction no longer share variables, they should be solved independently. Instead resolution effectively multiplies together the two tasks - steps of one task are needlessly repeated in the course of trying different ways to solve the other task.

To illustrate the problem, let us go back to the query considered earlier, with the goals appropriately ordered by the planner:

```
answer(C) <=
   borders(C,mediterranean) & country(C) &
   borders(C,C1) & asian(C1) & country(C1).
```

If Prolog executes this as it stands, then some of the solutions will be separately generated more than once, revealing the underlying inefficiency. For example, the solution 'answer(turkey)' is produced four times, the reason being that Turkey borders four different Asian countries.

Once the first goal 'borders(C,mediterranean)' has been solved, producing an instantiation such as C=turkey, the rest of the body breaks into two independent parts, neither of which share uninstantiated variables with each other or with the head of the clause. Let us indicate these parts with braces:

```
answer(turkey) <=
   {country(turkey)} &
   {borders(turkey,C1) &
      asian(C1) & country(C1)}
```

Such independent subproblems should be processed separately (ie. goals from the two tasks should not be intermingled), and no more than one solution should be produced for each subproblem.

The braces can be viewed as additional control information, showing where the normally exhaustive enumeration of possibilities can and should be overridden. Although existing Prolog implementations do not recognise braces as such, they do provide a control primitive, "cut", which Prolog programmers use (amongst other things) to achieve exactly the same effect. With the aid of the cut operator, it is easy either (a) to write in Prolog an extended Prolog interpreter for which the braces have the intended effect - the course taken in Chat-80, or (b) to translate clauses with braces into ordinary Prolog clauses. I shall therefore simply regard braces as a control primitive provided by Prolog.

The query planner takes care of producing this second kind of control information, in addition to working out the goal ordering. At every step, it checks to see whether the remaining goals can be partitioned into independent subsets. If so, each independent part is planned separately, and the result is enclosed in braces if no variable is shared with the head of the clause. The planned subproblems are ordered according to the costs of their first goals. (This, like other heuristics of the planner, is not necessarily optimal, but it produces reasonable results while simplifying the analysis). Notice that doing the extra work of recognising independent subproblems often actually reduces the complexity of the rest of the planning task, since independent subgoals can be planned separately.

For our example query, the result of this extra analysis is:

```
answer(C) <=
   borders(C,mediterranean) & {country(C)} &
   {borders(C,C1) &
      {asian(C1)} & {country(C1)}}.
```

For this example, the saving of execution time is significant but not huge. In other cases, the difference can be orders of magnitude, eg. consider the following query (describing the Indian Ocean):

> "Which is the ocean that borders African countries and (that borders) Asian countries?"

This translates into:

```
answer(X) <=
   ocean(X) &
   {borders(X,C1) &
      {african(C1)} & {country(C1)}} &
   {borders(X,C2) &
      {asian(C2)} & {country(C2)}}.
```

Leaving out the braces here would result in a combinatorial explosion where the last part of the query is needlessly repeated for each different African country that borders a given ocean.

The analysis into independent subproblems depends on the assumption that the solution to any goal is ground, ie. it contains no uninstantiated variables. This is a very reasonable assumption in the database context, though it is not true of Prolog programs in general. It amounts to forbidding the use of the "logical variable" [21].

Treatment of negation as non-provability

Many natural language queries involve (explicitly or implicitly) the concept of negation, "not". An example is:

> "Which American countries do not border the Pacific?"

This query can be translated into logic as:

```
answer(X) <=
   american(X) & country(X) &
   \borders(X,pacific).
```

However the Prolog subset of logic does not contain negation (and a proper treatment of negation is highly problematic). In the database context, it is natural to interpret questions involving negation as referring to the state of knowledge expressed in the database rather than to what is actually true in the real world. Often the assumption is that the database contains complete information about the field concerned - the "Closed World Assumption" [11]. Thus the question above would be interpreted as meaning:

> "Which American countries are not known to border the Pacific?"

with a logic translation of:

```
answer(X) <=
   american(X) & country(X) &
   \+borders(X,pacific).
```

where '\+' is to be read as a single symbol meaning "not provable" or "it cannot be shown that". It is possible to provide a partial implementation of non-provability within Prolog. The implementation only works correctly if the negated goal contains no uninstantiated free variables at the time it is executed. (There are fundamental reasons for this limitation, which I will not go into here).

The Chat-80 planning algorithm ensures that this constraint is always met. To achieve this, a negated goal is given a cost of "infinity" so long as it contains uninstantiated free variables. Otherwise it is assigned a cost of 1, (although the cost should effectively be somewhat less than 1, as the goal must function as a test). The effect for the example above is that the negative

goal is delayed until after both of the other two goals. One of them will be chosen to generate X, at which point the negative goal is potentially executable. However the negative goal is delayed until after the other goal, which, having a cost of less than 1, is considered the more restrictive test.

It is possible for the negative goal to be complex, containing a conjunction of goals, in which case these are ordered by simply applying the planner recursively. The negative goal may also contain existential quantifiers, serving to restrict the scope of certain variables, and the planner fully caters for this situation, but I will not go into further details here.

Treatment of set expressions

Besides negation, natural language queries also often involve explicit or implicit reference to the set of individuals having some property. For this and other reasons, I have proposed and implemented an extension to Prolog, which has been described in detail elsewhere [20] [2]. The extension allows goals of the form:

```
setof(X,P,S)
```

to be read as "the set of Xs such that P is provable is S, which is non-empty". P is a goal or conjunction of goals. (It may also contain existential quantifiers, as for negation). For example, the query:

```
"Which countries border (exactly) two (major)
 seas?"
```

could be translated as:

```
answer(C) <=
   country(C) &
   setof(X, borders(C,X) & sea(X), S) &
   size(S,2).
```

Notice that, here too, the query is being construed as referring to the state of knowledge expressed in the database.

An important aspect of the implementation of the 'setof' construct is that, unlike '\+', it behaves correctly in all situations. Free variables in the 'setof' expression need not be completely instantiated at the time of execution. If they are not fully instantiated, execution of the 'setof' will generate suitable instantiations, backtracking where necessary.

This allows the planning algorithm greater flexibility in choosing an efficient execution order. At present, the cost of a 'setof' is taken to be the cost of its least costly internal subgoal. Though only a crude estimate, this produces reasonable results in most cases. As for negation, the goals within a 'setof' are ordered by applying the planner recursively. The result of the planning for the above example is:

```
answer(C) <=
   setof(X, sea(X) & borders(C,X), S) &
   size(S,2) & country(C).
```

Thus the 'setof' construct itself is responsible for generating instantiations for the variable C. Given that there are only a few (major) seas (and things that border them), but many countries, this is probably the best ordering.

Experimental results

The performance of Chat-80 on some sample queries relating to the geographical database is shown in the Appendix. Generally speaking, any query in this domain that can comfortably be expressed in a single sentence of the English subset is answered in well under one second of CPU time, and normally only a small part of this time is spent in planning. Without the planning, many queries would take an unacceptably long time to answer - several minutes or more. Therefore one can say that the planning algorithm certainly pays off in practice, and, for this domain at least, it seems more than adequate.

Incidentally, it is also worth noting that the time spent in natural language analysis is typically quite small in relation to the time for planning and retrieval. Therefore cost should not be a deterrent to the use of natural language for accessing databases.

COMPARISON WITH OTHER WORK

The approach to query processing described above rests on the use of Prolog for the eventual execution of a query. However expressing queries in logic does not force us to execute them with Prolog. We are open to use any other valid inference method. Prolog was not in fact designed with relational database retrieval in mind; it was conceived purely as a programming language interpreter. How does the Prolog-based approach of Chat-80 compare with the strategies used in conventional relational database systems, and can we devise a better inference method for the database context, by lifting techniques from such systems? To answer these questions, I have looked at a number of relational database systems [7] [16] [15] [22] [1] [14] [9], especially Ingres and System R. Unfortunately there is only space for a brief discussion here.

Surprisingly (to me), I have not found convincing evidence of ways to improve on the general strategy, at least as far as interactive queries are concerned. In fact the Chat-80 query processing method, while being quite different from the strategies used in many relational database systems, is actually very similar to the strategy of System R, as recently described [14]. (This similarity was only discovered when this paper was on the point of completion!)

The basic Prolog execution method differs from that used in most relational database systems in

that there is no explicit creation of intermediate relations, with the extra sorting or indexing, and tuple accesses, that this entails. Instead answers to the entire query are produced strictly one at a time. The method can be characterised by the fact that it minimises the amount of working storage needed to process a query (which also naturally tends to reduce computation time).

Query execution by Prolog corresponds almost exactly to System R's method of "nested loops" for implementing a join, where "search arguments" (SARGs) achieve a similar effect to Prolog's pattern matching. Ingres's "tuple substitution" is somewhat similar in effect, but there the matching is not built-in. Instead the effect of the matching is achieved in a separate, preceding, "detachment" operation.

Before the introduction of the "reduction" tactic [22], Ingres's query processing strategy was in fact fairly similar in its effect to the present System R, and to the Chat-80 approach. A possible reason why the Ingres authors decided to change course may lie in the fact that Ingres mixes planning with execution, unlike System R and Chat-80. Tuple substitution therefore has the effect that planning operations have to be done on a tuple-by-tuple basis, which may well add enormously to the amount of computation that has to be done.

The query planning of System R aims to be exhaustive, whereas Chat-80 only does a very limited search by making a "best guess" at each stage. As a result, the System R algorithm takes a time which is at least exponential in the size of the query, whereas the Chat-80 algorithm, as implemented, is nearly linear (and certainly no worse than order N**2). Given that Chat-80 is dealing with interactive queries which may be quite large, and given that the present planning algorithm seems to produce quite satisfactory results, there does not seem to be sufficient justification to do a more expensive search.

The way System R estimates the size of an intermediate relation is almost identical to that used in Chat-80, except that the "selectivity factors" do not seem to give a sufficiently accurate estimate of argument domain sizes. (They ignore the possibility that an argument value may not even appear in the corresponding index, and if the corresponding index does not exist, the selectivity factor is purely arbitrary).

As we have seen, Chat-80's treatment of independent subproblems is vital for certain kinds of query. It does not appear to have any exact counterpart in System R, or indeed in any of the other systems I have looked at. It may be that the "nested subqueries" of SQL provide a means for the user to force a similar effect in System R, but this seems contrary to the philosophy of relational databases. In effect, the System R query language SQL provides two distinct ways of specifying a join, unlike Quel for example.

To sum up my conclusions on query processing strategies, I have not found an alternative strategy to Chat-80's that is guaranteed to reduce the number of tuple accesses, and in the context of current Prolog implementations, this is certainly a good measure of the cost of executing a query. Also, it seems a not unreasonable measure in general.

Experiments with running Ingres on the Chat-80 geographical database show that Ingres often accesses far more tuples than Chat-80 does on the same query. Also, for what it is worth, queries which Chat-80 answers in well under a second on a DEC-10 typically take several minutes of CPU time with Ingres on a PDP-11. Of course, although this is of interest from a user point of view, there are too many other factors involved for any immediate conclusions to drawn about query processing strategies.

A conclusion that I find difficult to avoid after reading the literature on query processing, and looking briefly into the Ingres implementation, is that query formalisms such as Quel and SQL (not to mention the algebraic formalisms) make life unnecessarily difficult for the implementor, both conceptually and in terms of implementation code. Expressing a query in logic lays bare what its essential parts are (the logic goals), what it is that links the parts together (the logic variables), and what the crucial problem is (in what order should one to attempt to satisfy the goals).

Finally, it should be noted that there is much other work relating logic and databases - Gallaire [6] gives a useful survey. A common feature in much of this work is to regard the logical formalism as separate from the database itself, and to make a distinction between database retrieval and logical deduction. Typically this view leads to systems in which deduction and retrieval are completely separate processes. In contrast, the view of this paper is that no clear line can be drawn between deduction and retrieval, or indeed between deduction and ordinary computation. Identifying deduction with retrieval clarifies common problems (such as query optimisation), and avoids unnecessary duplication of machinery (such as indexing). It also leads to a natural interface between the database and a well proven high-level programming language - Prolog.

CONCLUSION

We have seen how relational database retrieval can be viewed as a special case of logical deduction. We have seen how queries expressed in the Prolog subset of logic can be transformed into efficient Prolog code. The method can be viewed as a generalisation of techniques used in relational database systems, although it was not conceived in this way.

What is the practical importance of all this? It means that a single formalism (the Prolog subset of logic) can in principle serve both as a

practical high level programming language and as a database formalism. The difference between "program" and "database" would then be invisible to the user.

Of course existing Prolog implementations are not practical for large databases - they will only accommodate relatively small volumes of data and there are many aspects, particularly indexing, which would prove inadequate if large relations were to be handled. A promising direction for future work would be to produce a Prolog system that is equally good for both database and programming purposes.

ACKNOWLEDGEMENTS

This work was supported by a British Science Research Council grant. The Chat-80 program was written in collaboration with Fernando Pereira, and owes much to the work of Alain Colmerauer and Veronica Dahl.

REFERENCES

1. Astrahan M M and Chamberlin D D. Implementation of a structured English query language. CACM 18, 10 (Oct 1975), 580-588.

2. Byrd L, Pereira F and Warren D. A guide to Version 3 of DEC-10 Prolog. Occasional Paper 19, Dept of AI, Univ of Edinburgh, Jul, 1980.

3. Colmerauer A. Un sous-ensemble interessant du francais. RAIRO 13, 4 (1979), 309-336. [Presented under the title "An interesting natural language subset" at the Workshop on Logic and Databases, Toulouse, 1977]

4. Dahl V. Quantification in a three-valued logic for natural language question-answering systems. IJCAI-79, Tokyo, Aug, 1979, pp. 182-187.

5. Dahl V. Un systeme deductif d'interrogation de banques de donnees en Espagnol. Groupe d'Intelligence Artificielle, UER de Luminy, Universite d'Aix-Marseille II, 1977.

6. Gallaire H. Impacts of logic on data bases. Lab de Marcoussis - CGE, Marcoussis, France, 1981. [To be presented at the 1981 Conference on Very Large Data Bases]

7. Hall P A V. Optimisation of a single relational expression in a relational data base system. Report UKSC 0076, IBM UK Scientific Centre, Jun, 1975.

8. McCord M C. Using slots and modifiers in logic grammars for natural language. Report 69A-80, Dept of Computer Science, Univ of Kentucky, Oct, 1980.

9. Moore R C. Handling complex queries in a distributed data base. Tech Note 170, AI Center, SRI International, Menlo Park, Calif, Oct, 1979.

10. Pereira L M, Pereira F and Warren D H D. User's Guide to DECsystem-10 Prolog. Dept of AI, Univ of Edinburgh, 1978.

11. Reiter R. On closed world data bases. In Gallaire H and Minker J, Ed., Logic and Databases, Plenum Press, New York, 1978.

12. Robinson J A. A machine-oriented logic based on the resolution principle. JACM 12, 1 (Dec 1965), 227-234.

13. Roussel P. Prolog: Manuel de Reference et d'Utilisation. Groupe d'Intelligence Artificielle, UER de Luminy, Universite d'Aix-Marseille II, 1975.

14. Selinger P G, Astrahan M M, Chamberlin D D, Lorie R A & Price T G. Access path selection in a relational database management system. Report RJ2429, IBM Research Laboratory, San Jose, 1979.

15. Stonebraker M, Wong E, Kreps P and Held G. The design and implementation of INGRES. ACM Trans. on Database Systems 1, 3 (Sep 1976), 189-222.

16. Todd S and Verhofstad J. An optimizer for a relational database system - description and evaluation. IBM UK Scientific Centre, Feb, 1979.

17. Warren D H D and Pereira F C N. An efficient easily adaptable system for interpreting natural language queries. Dept of AI, Univ of Edinburgh, 1981. [To be submitted to IJCAI-81]

18. Warren D H D. Implementing Prolog - compiling predicate logic programs. Research Reports 39 & 40, Dept of AI, Univ of Edinburgh, May, 1977.

19. Warren D H D. Prolog on the DECsystem-10. In Michie D, Ed., Expert Systems in the Micro-Electronic Age, Edinburgh Univ Press, 1979.

20. Warren D H D. Higher-order extensions to Prolog - are they needed?. Tenth International Machine Intelligence Workshop, Cleveland, Ohio, Apr, 1981.

21. Warren D H D, Pereira L M and Pereira F. Prolog - the language and its implementation compared with Lisp. ACM Symposium on AI and Programming Languages, August, 1977.

22. Wong E and Youssefi K. Decomposition - a strategy for query processing. ACM Trans. on Database Systems 1, 3 (Sep 1976), 223-241.

APPENDIX - SAMPLE QUERIES

The Chat-80 examples below show the original English query, its logical form, the executable form after planning, and the actual answer. Also shown, preceding the corresponding output, are the separate times (in milliseconds on a DEC KL-10) for natural language analysis, for planning, and for execution. Time spent in producing output to the user is excluded.

```
Which country's capital is London?
  52 ms.  ans(C) <= country(C) & capital(C,london).
  16 ms.  ans(C) <= capital(C,london) & {country(C)}.
  16 ms.  united_kingdom.

Which European countries border Asian countries?
  72 ms.  ans(C) <= country(C) & european(C) & country(C1) & asian(C1) & borders(C,C1).
  21 ms.  ans(C) <= european(C) & {country(C)} & {borders(C,C1) & {asian(C1)} & {country(C1)}}.
 113 ms.  bulgaria, czechoslovakia, finland, greece, hungary, norway, poland, romania.

Which Asian countries border European countries?
  54 ms.  ans(C) <= country(C) & asian(C) & country(C1) & european(C1) & borders(C,C1).
  22 ms.  ans(C) <= european(C1) & {country(C1)} & borders(C,C1) & asian(C) & {country(C)}.
 112 ms.  soviet_union, turkey.

Which is the ocean that borders African countries and that borders Asian countries?
  91 ms.  ans(X) <= ocean(X) & country(C) & african(C) & borders(X,C) &
                               country(C1) & asian(C1) & borders(X,C1).
  51 ms.  ans(X) <= ocean(X) & {borders(X,C) & {african(C)} & {country(C)}} &
                               {borders(X,C1) & {asian(C1)} & {country(C1)}}.
 102 ms.  indian_ocean.

Which American countries do not border the Pacific?
  46 ms.  ans(C) <= country(C) & american(C) & \+borders(C,pacific).
  10 ms.  ans(C) <= american(C) & {country(C)} & \+borders(C,pacific).
  56 ms.  argentina, bahamas, barbados, belize, bolivia, brazil, cuba, dominican_republic, french_guiana,
          grenada, guyana, haiti, jamaica, paraguay, surinam, trinidad_and_tobago, uruguay, venezuela.

Which countries border two oceans?
  63 ms.  ans(C) <= country(C) & numberof(X,ocean(X)&borders(C,X),2).
  12 ms.  ans(C) <= numberof(X,ocean(X)&borders(C,X),2) & {country(C)}.
 699 ms.  australia, colombia, costa_rica, guatemala, honduras, indonesia, malaysia, mexico, nicaragua,
          norway, panama, south_africa, soviet_union, thailand.

How many countries does the Danube flow through?
  48 ms.  ans(N) <= numberof(C,country(C) & flows(danube,C),N).
   3 ms.  ans(N) <= numberof(C,flows(danube,C) & {country(C)},N).
  21 ms.  6.

What is the capital of each country bordering the Baltic?
  81 ms.  ans(C-X) <= country(C) & borders(C,baltic) & capital(C,X).
  12 ms.  ans(C-X) <= borders(C,baltic) & {country(C)} & capital(C,X).
  29 ms.  denmark-copenhagen, east_germany-east_berlin, finland-helsinki, poland-warsaw,
          soviet_union-moscow, sweden-stockholm, west_germany-bonn.

What are the countries bordering the Soviet Union whose population exceeds the population of the United
Kingdom?
 145 ms.  ans(C) <= country(C) & borders(C,soviet_union) & population(C,X) &
                    population(united_kingdom,Y) & exceeds(X,Y).
  41 ms.  ans(C) <= population(united_kingdom,Y) & borders(C,soviet_union) & {country(C)} &
                    {population(C,X) & {exceeds(X,Y)}}.
  28 ms.  china.

Which country bordering the Mediterranean borders a country that is bordered by a country whose
population exceeds 400 million?
 138 ms.  ans(C) <= country(C) & borders(C,mediterranean) & country(C1) & country(C2) &
                    population(C2,X) & exceeds(X,400000000) & borders(C2,C1) & borders(C,C1).
  46 ms.  ans(C) <= borders(C,mediterranean) & {country(C)} &
                    {borders(C,C1) & {country(C1)} &
                       {borders(C2,C1) & {country(C2)} &
                          {population(C2,X) & {exceeds(X,400000000)}}}}.
 195 ms.  turkey.
```