

HOW TO NOT SAY "NIL" -
IMPROVING ANSWERS TO FAILING QUERIES IN DATA BASE SYSTEMS *

Jürgen M. Janas
Hochschule der Bundeswehr München
Fachbereich Informatik
Werner-Heisenberg-Weg 39
D-8014 Neubiberg, Germany

If a data base query fails, i.e., if the required information is not available, then current data base management systems normally report this failure but do not provide a possible deeper reason for this failure. In this paper, a method is introduced by means of which for any query expressed in a predicate calculus based language the actual reasons for a failure may be detected; this is done by recursively inferring related queries from a failing query until no more failing queries are found, the queries obtained by this process then determine the desired answer to the original query. The method may be combined with an arbitrary query evaluation strategy and requires a minimum of additional computation. It is a good means to clear the dialog between a user and the data base system of unnecessary queries and thus contributes to the usefulness and efficiency of the user interface.

1. INTRODUCTION

The relationship between a question and the appropriate answer(s) to this question has recently been investigated from several points of view; we only mention [3], [5], [10]. In this paper, we shall be concerned with a subproblem, namely how to find an appropriate answer to a failing data base query. We shall illustrate the purpose of this paper by the following example: Suppose somebody had asked for "all employees in the shoe department who are younger than 30 and earn more than \$25.000" and the query failed, i.e., the answer was something like "nil", "Ø", or "there are none". If the user is seriously interested in the answer because s/he needs it for some decision s/he has to make, then s/he probably goes on asking a series of questions like the question about "all employees younger than 31 and earning more than \$24.000" which all might be answered by "nil" as well; thus s/he will finally ask for a list of "all employees in the shoe department together with their age and salary" where another "nil" would indicate that there is no shoe department at all. Clearly it would be desirable that the system reports in its answer

to the very first question that - in contradiction to the user's expectations - there is no such department. This becomes even more demanding since the number of questions which a user will have to put in order to discover the origin of a failure will rapidly increase as data base or queries become more complex.

In this paper, we shall propose a method by means of which the reasons for the failure of a user's query may be reported to the user as the answer to his query. In section 2, we shall introduce a non-procedural predicate calculus based formal language which will be used to represent queries and answers throughout the rest of this paper. In section 3, we define how the so-called predecessors of a query (which are queries themselves) are inferred from a query; furthermore we show how recursive substitution of failing queries by their predecessors leads to an informative answer to the user's original query. In section 4, we shall discuss how the suggested method may be implemented and we shall demonstrate how it may be improved by considering the integrity constraints the data base is subject to. In a concluding section, we shall compare our approach to related research.

Throughout this paper, all examples refer to the following data base according to the relational model of data (cf. [1]):

* This research was carried out while the author was still at the Institut für Informatik der Technischen Universität München.

EMP (NAM, SAL, AGE, DN)
 DEPT (DNO, DNAME)
 CAR (LIP, OWN, COL)

An employee EMP has a name NAM, a salary SAL, the age AGE and works for a department DEPT which is identified by a department number DNO (or DN respectively) and which has a department name DNAME. Moreover there are cars CAR with a licence plate LIP, a color COL, and an owner OWN who is an employee identified by his name.

2. THE QUERY LANGUAGE

2.1 Conventions

Throughout this paper, we shall use the following notation:

- R and R' are non-empty relations ($R \neq R'$)
- A, A1, and A2 are attributes of R
- A' is an attribute of R'
- v is an attribute value of A'
- x and y are tuple variables for relations ($x \neq y$)
- ω is a relational operator, i.e.,
 $\omega \in \{ =, \neq, <, >, \leq, \geq \}$

For convenience, we assume that all attributes in the data base are named differently also if they are from different relations.

2.2 Terms

The elementary constituents of our query language are terms, they are defined by the following axioms:

- (T1) Any string of characters χ where
 $\chi = (x.A \omega v)$
 is called a term.
- (T2) Any string of characters χ where
 $\chi = (x.A1 \omega x.A2)$
 is called a term.
- (T3) Any string of characters χ where
 $\chi = (x.A \omega y.A')$
 is called a term.
- (T4) A string of characters is a term only if this is by reason of (T1), (T2), or (T3).

For a term χ , let $F(\chi)$ denote the set of variables which are free in χ ; clearly $F(\chi) = \{x, y\}$ in the case of (T3) and $F(\chi) = \{x\}$ in the case of (T1) or (T2). Terms according to (T3) also will be called join terms, all other terms will be called simple terms if they have to be distinguished. Note that for any tuple variable occurring in a term, the relation it stands for may be uniquely determined because all attributes are named differently.

2.3 Expressions

Terms may be combined to form expressions by

applying the following axioms:

- (E1) If χ is a term then χ is an expression.
- (E2) If α and β are expressions then
 $\chi = (\alpha \wedge \beta)$ and $\chi = (\alpha \vee \beta)$
 are expressions.
- (E3) If α is an expression and $x \in F(\alpha)$ then
 $\chi = \exists x \alpha$ and $\chi = \forall x \alpha$
 are expressions.
- (E4) A string of characters is an expression only if this is by reason of a finite number of applications of (E1) through (E3).

$F(\chi)$ may be expanded to expressions χ in a natural way: If χ is an expression according to (E2) then $F(\chi) := F(\alpha) \cup F(\beta)$ and if χ is an expression according to (E3) then $F(\chi) := F(\alpha) \setminus \{x\}$. Given an expression χ , we shall refer to the variables from $F(\chi)$ as to T-variables (target), whereas all other variables occurring in χ will be called Q-variables (qualification).

2.4 Well-Formed Sets

Expressions form the basis of our query language, but not every expression according to section 2.3 leads to a meaningful query; therefore, we restrict our further considerations to a subclass of expressions, namely the so-called connected expressions. An expression χ is called a connected expression if and only if for any Q-variable x in χ there are $n+1$ join terms $l_0, l_1, \dots, l_n$ ($n \geq 0$), n Q-variables

$x_1, \dots, x_n$ and a T-variable y , such that:

$$F(l_0) = \{x, x_1\} \wedge F(l_1) = \{x_1, x_2\} \wedge \dots \\ \dots \wedge F(l_{n-1}) = \{x_{n-1}, x_n\} \wedge F(l_n) = \{x_n, y\}$$

Informally speaking, for any Q-variable of a connected expression there is a path of join terms which "ties" this Q-variable to a T-variable.

Let now χ be a connected expression with $F(\chi) \neq \emptyset$ and $x_1, x_2, \dots, x_n$ the elements of $F(\chi)$ in an arbitrary but fixed order, then

$$\{(x_1, x_2, \dots, x_n) \mid \chi\}$$

is called a well-formed set. A well-formed set may be interpreted as the following data base query: "Find all those combinations of tuples from the relations specified by the variables $x_i \in F(\chi)$ for which the expression χ holds." Thus the well-formed set

$$\{x \mid (\exists y ((y.DNO = x.DN) \wedge (y.DNAME = 'shoe')) \\ \wedge ((x.AGE < 30) \wedge (x.SAL > 25.000)))\}$$

corresponds to the query "Find all employees in the shoe department who are younger than 30 and earn more than 25.000."

2.5 Properties of Well-Formed Sets

The query language formed by the well-formed sets corresponds to a great extent to the data sublanguage ALPHA (cf. [2]). The fact that well-formed sets cannot be used to ask for single attributes but only for complete tuples is not really a restriction but rather a tribute to a more convenient notation. The absence of explicit negation from the expressions does not mean a deficiency either, since the language contains also the inverse operator of any of its relational operators (cf. 2.1).

Our restriction to connected expressions seems to be of some consequence for the expressive power of the language; but in fact, queries formed from unconnected expressions do not seem to make much sense as one can see from the following example:

```
{x | ((x.AGE > 30) ∧ ∃y (y.DNAME = 'shoe'))}
```

The answer to this query would be either a list of employees older than 30 or "nil" depending on whether there is a shoe department or not; thus, if the answer to this query is "nil" one does not know whether there is no shoe department or whether there are no employees older than 30 and this seems to be a strong argument for excluding unconnected expressions from the formation of well-formed sets. Still we have to mention that for most (not all) unconnected expressions there are equivalent connected ones; this is illustrated by the following query which is semantically equivalent to the above one but formed from a connected expression:

```
{x | ((x.AGE > 30) ∧ ∃y ((y.DNAME = 'shoe') ∧  
((x.DN = y.DNO) ∨ (x.DN ≠ y.DNO))))}
```

3. THE PREDECESSORS OF A QUERY

3.1 Predecessors of Natural Language Questions

If somebody asked for "all employees who are younger than 30 and own a red car" and there are no such employees then one possible situation is that there are employees who are younger than 30 and own a car but none of these cars is red and that there are red cars but all of them are owned by employees older than 30. A user who obtains "nil" as the answer to the above question will be likely to interpret the answer in this way. Still there are several other situations which might have led to "nil":

- (R1) There are no employees who own a red car.
- (R2) There are no employees who are younger than 30 and own a car.
- (R3) There are no employees who own a car.
- (R4) There are no red cars.
- (R5) There are no employees younger than 30.

We neglect the pathological situations where

there are no employees or no cars because in this case the question would have resulted from a wrong understanding of the structure of the data base.

Each of the situations (R1) through (R5) implies a negative answer to the user's question; moreover for each of these situations there is a question which corresponds to it in a natural way:

- (S1) Find all employees who own a red car.
- (S2) Find all employees who are younger than 30 and own a car.
- (S3) Find all employees who own a car.
- (S4) Find all red cars.
- (S5) Find all employees younger than 30.

Such questions are called the predecessors of the original question. In general, a question Q is called a predecessor of a question R if a negative answer to Q implies a negative answer to R.

A predecessor of a question may have predecessors itself and one can easily see that these have to be elements of the set of predecessors of the original question; thus the set of predecessors of a question is partially ordered. In the above example (S5) is a predecessor of (S2), (S4) is a predecessor of (S1), and (S3) is a predecessor of both (S1) and (S2). We make use of this property in the following definition: A question Q is called an immediate predecessor of a question R if Q is a predecessor of R but not predecessor of any of R's predecessors. The reason why we are concerned with these immediate predecessors is that stepwise substitution of a failing query by its predecessors finally leads to the actual reason for the failure like (R1) - (R5) in the example.

3.2 Predecessors of Well-Formed Sets

The questions from section 3.1 may be expressed as well-formed sets:

- (U0) {x | ((x.AGE < 30) ∧ ∃y ((y.OWN = x.NAM) ∧ (y.COL = 'red')))}
- (U1) {x | ∃y ((y.OWN = x.NAM) ∧ (y.COL = 'red'))}
- (U2) {x | ((x.AGE < 30) ∧ ∃y (y.OWN = x.NAM))}
- (U3) {x | ∃y (y.OWN = x.NAM)}
- (U4) {y | (y.COL = 'red')}
- (U5) {x | (x.AGE < 30)}

Replacing a question by one of its predecessors here mainly means removing one term from the query's expression. On the other hand, not every removal of a term from a query results in a predecessor of that query; thus (y.OWN = x.NAM) must not be removed from (U0) since the resulting expression would not be connected. Now we are going to introduce rules which specify how the predecessors of a well-formed set are obtained, i.e., which of the terms of a well-

formed set may and which may not be removed in order to yield a predecessor. Predecessors of well-formed sets correspond to a large extent to the immediate predecessors of natural language questions but they are not equivalent: The negative answer to a predecessor of a well-formed set does not necessarily imply the negative answer to the original query. Moreover predecessors are only similar to the related concepts "presumption" (cf. [6]) and "presupposition" (cf. [8]).

The above example showed that a join term may be removed from a query expression only if this does not "disconnect" the query expression; therefore, the first rule for obtaining predecessors is:

(X1) A join term must not be removed if both its free variables occur in other terms of the expression too.

This is the reason why $(y.OWN=x.NAM)$ must not be removed from (U0) but may be removed from (U2) thus yielding (U5). There is one exception to this rule, namely:

(X2) A join term may always be removed if there is a "parallel" join term in the expression which connects the same variables.

To give an example of this rule we add an attribute DRIVER to the CAR relation where a DRIVER is an employee identified by his name; we then might ask:

$$\{x | ((x.AGE < 30) \wedge \exists y ((y.OWN=x.NAM) \wedge (y.DRIVER=x.NAM) \wedge (y.COL='red'))))\}$$

i.e., "find all employees who are younger than 30 and own and drive a red car"; according to (X2) we may remove $(y.OWN=x.NAM)$.

Simply deleting join terms may have the effect that we obtain expressions without T-variables and hence no well-formed sets. Therefore we have to make sure that the resulting expressions always contain at least one free variable; this is achieved by the following rule:

(X3) Whenever we remove a join term which contains a Q-variable and a T-variable which does not occur anywhere else in the expression the Q-variable is made a T-variable.

A Q-variable is made a T-variable by removing the quantifier of that variable from the expression. An example of the application of (X3) is the way in which (U4) is derived from (U1).

3.3 How to Handle Disjunctions

So far we were only concerned with queries without disjunctions. When considering disjunction, we have to distinguish two situations, namely whether there is a disjunction at the top level of the query (i.e., the query ex-

pression is $\chi=(\alpha \vee \beta)$) or somewhere else in the query.

(X4) If a query is disjunctive at the top level of its query expression, then it has two predecessors which correspond to the two constituents of the disjunction.

Thus a query like:

$$\{x | (((x.COL='red') \wedge (x.OWN='Smith')) \vee ((x.COL='blue') \wedge (x.OWN='Brown'))))\}$$

i.e., "find all cars which are either red and owned by Smith or blue and owned by Brown" has the following predecessors:

$$\{x | ((x.COL='red') \wedge (x.OWN='Smith'))\}$$

$$\{x | ((x.COL='blue') \wedge (x.OWN='Brown'))\}$$

A negative answer to one of these predecessors does not imply a negative answer to the original query, rather a negative answer to the original query is equivalent to negative answers to both predecessors. Note, however, that a deeper reason for the failure of the original query, e.g., "there are no red cars and there is no Mr. Brown" would be detected by forming the predecessors of the predecessors.

If there is a disjunction at a deeper level of the query expression, then we need two more rules which further restrict the terms which may be removed from the query.

(X5) A simple term which is a constituent of a disjunction may be removed only if the other constituent is a simple term too.

(X6) If one constituent of a disjunction has been removed then the other constituent of that disjunction has to be removed together with it if it is a simple term.

Application of (X5) and (X6) is demonstrated by the following example:

$$\{x | \exists y ((x.NAM=y.OWN) \wedge (((y.COL='red') \vee (y.COL='blue')) \vee (y.COL='white'))))\}$$

$(x.COL='red')$ may be removed because of (X5), then because of (X6) both other simple terms are removed too; so the only predecessors of this query are:

$$\{x | \exists y (x.NAM=y.OWN)\}$$

$$\{y | (((y.COL='red') \vee (y.COL='blue')) \vee (y.COL='white'))\}$$

Summarizing we can say that the predecessors of a well-formed set are obtained either by application of (X4) or - for every term of the query expression - by trying to remove it under consideration of (X1), (X2), (X3), (X5), and (X6). It can be shown that if B is a well-formed set representing a query which fails, then the answer " $B=\emptyset$ " may be substituted by the answer

$$B_1=\emptyset \wedge B_2=\emptyset \wedge \dots \wedge B_n=\emptyset$$

where B_i are those predecessors of B which are empty themselves. The advantage of this latter answer is that it is more informative than the original one because each B_i is either a super-

set of B or a set whose emptiness implies the emptiness of B; if B is disjunctive at the top level then both answers are equivalent. Since each B_i is a well-formed set and thus a query itself, we may apply the process of answer substitution recursively until each B_i in the answer has either non-empty predecessors only or no predecessors at all. For an example, we have to refer to [4].

4. IMPLEMENTATION

4.1 The Elementary Algorithm

In this section, we shall investigate how the predecessors of a given query may be computed. We start by taking the point of view that the evaluation of a query and the computation of the predecessors of that query are kept completely apart from each other; thus, as long as a query does not fail nothing has to be done. In the case of failure, the following algorithm will provide the appropriate answer according to 3.3. The data structures the algorithm operates on are three sets whose elements are well-formed sets; there is one set (PA) containing possible answers to the query, one set (DA) containing definite answers to the query, and an auxiliary set (AS). At the beginning PA contains the failing query whereas DA and AS are empty. The basic idea of the algorithm is to recursively replace the elements of PA by their predecessors as long as there are any and to move them to DA otherwise. In detail the algorithm looks like this:

- (A1) Choose an element α from PA and put the predecessors of α into AS (α is the next element which either will be moved to the definite answers or will be replaced by its empty predecessors).
- (A2) If AS is empty then go to (A4) (If there are no more predecessors of α then we have to test whether α is a definite answer or not).
- (A3) Remove an element β from AS; if $\beta \neq \emptyset$ then immediately go to (A2) otherwise remove α from PA - if possible -, insert β in DA and then go to (A2) (As long as a predecessor of α is non-empty, nothing has to be done, otherwise the predecessor replaces α in the set of possible answers).
- (A4) If α is contained in PA then remove it from PA and insert it in DA (If α is still a possible answer then it was not replaced by any of its predecessors and therefore becomes a definite answer).
- (A5) If PA is non-empty then go to (A1), otherwise stop (The procedure does not terminate as long as there are still possible answers).

It can be shown that DA is non-empty after the algorithm has terminated and that it contains exactly those well-formed sets B_i which form the appropriate answer

$$B_1 = \emptyset \wedge B_2 = \emptyset \wedge \dots \wedge B_n = \emptyset$$

to the user's query.

4.2 Performance Improvements

The above algorithm is a pretty obvious method but on the other hand it does not work very efficiently. One of the drawbacks of the method as it stands is that a well-formed set may be computed more than once if it is a predecessor of more than one possible answer. A simple way to overcome this redundancy is to keep a list of already computed predecessors together with the information whether they are empty or not. A better solution would be to support the algorithm by a strategy which makes sure that no well-formed set has to be evaluated more than once. An example for such a strategy would be to choose always that well-formed set with the "longest expression" in (A1); then no well-formed set could be inserted in PA again after it has already been removed from PA.

If we abandon the (naive) point of view that query evaluation and computation of predecessors are kept apart from each other, the algorithm may be improved considerably: Note that before our algorithm can start the whole query has already been evaluated, namely in order to find out that it fails. Evaluation of a query consists of a sequence of elementary operations such as projections, joins, and selections whose results are sets of tuples corresponding to certain well-formed sets. It is easy to see that - no matter what query evaluation strategy is applied - these well-formed sets occur somewhere among the predecessors of the failing query. Therefore, if information about the well-formed sets which already have been computed by the query evaluator is made available to the above algorithm, then the number of predecessors which have to be tested for emptiness will be further reduced.

4.3 Considering Integrity Constraints

Roughly speaking, integrity constraints are restrictions which the data base is subject to in order to be in a well-defined state. An example of an integrity constraint on our data base would be that for every tuple of the relation CAR there has to be a tuple of the relation EMP such that the values of their attributes OWN and NAME are identical. Such integrity constraints may be used to simplify the predecessor structure of a given query; we illustrate

this by means of the following example:

$$B = \{x \mid ((x.COL = 'red') \wedge \exists y((y.NAM = x.OWN) \wedge (x.AGE < 30)))\}$$

i.e., "find all red cars owned by employees who are younger than 30." According to section 3 the predecessors of B are B1 and B2, those of B1 are B3 and B4, and those of B2 are B4 and B5

$$B1 = \{x \mid \exists y((y.NAM = x.OWN) \wedge (y.AGE < 30))\}$$

$$B2 = \{x \mid ((x.COL = 'red') \wedge \exists y(y.NAM = x.OWN))\}$$

$$B3 = \{y \mid (y.AGE < 30)\}$$

$$B4 = \{x \mid \exists y(y.NAM = x.OWN)\}$$

$$B5 = \{x \mid (x.COL = 'red')\}$$

Now, if every car has to have an owner then obviously B4 is the set of all cars and therefore cannot be empty; moreover B2 is identical to the set of all red cars, namely B5. Consequently, we would like to obtain B1 and B5 as the predecessors of B and B3 as the only predecessor of B1. Integrity constraints may be used to remove certain join terms from the predecessor queries, we thereby reduce the average number of predecessors which have to be checked for emptiness in the case of failure and hence further improve the efficiency of our procedure. For a more thorough discussion of this point we have to refer to [4].

5. CONCLUSION

The work described in this paper is closely related to the research reported in [9] and in [7]. In [9], a formal query language called HI-IQ for a CODASYL-type data base is used to express queries and to compute deeper reasons for a possible failure of a query. Still, HI-IQ supports a much narrower range of queries than well-formed sets do; moreover the procedure that computes the predecessors of a query does not obtain all possible predecessors of an arbitrary query. The method for obtaining an appropriate answer to a failing query will on average require more computation than the procedure suggested here because small deviations of the user's expectations from the actual data require more computation than greater deviations do. Finally, the method proposed in [9] does not make any use of integrity constraints during the computation of predecessors.

In [7], the problem of how to find an appropriate answer to a failing query is investigated for natural language queries: A natural language query is translated into a so-called meta query language (MQL) which is a graphical representation of the query still close to natural language; from this MQL representation the predecessors of a query are computed. A query represented in MQL then has to be translated into the actual query language before it may be evaluated. Contrary to that approach, we

have shown in this paper that informative answers to failing data base queries may be obtained from a purely formal query representation and that clues provided by natural language are not actually needed for this task. Nevertheless, we have to mention that our method will gain its full strength only if the informative answers will be rephrased in natural language.

REFERENCES

- [1] Codd, E.F. "A Relational Model of Data for Large Shared Data Banks." Communications ACM 13:6 (1970) 377-387
- [2] Codd, E.F. "A Database Sublanguage Founded on the Relational Calculus." In Proc. 1971 SIGFIDET Workshop. ACM, New York, 1971, pp. 35-68
- [3] Hobbs, J.R. and Robinson, J.J. "Why Ask?" Technical Note 169, SRI international, 1978
- [4] Janas, J.M. "Towards More Informative User Interfaces." In Proc. 5th Int. Conf. on Very Large Data Bases. Rio de Janeiro, Brasil, 1979
- [5] Joshi, A.K. and Weischedel, R. "Computation of a Subclass of Inferences: Presupposition and Entailment." AJCL microfiche no. 63 (1977)
- [6] Kaplan, S.J. "On the Difference between Natural Language and High Level Query Languages." Proc. ACM'78. Washington, D.C., 1978
- [7] Kaplan, S.J. "Cooperative Responses from a Portable Natural Language Data Base Query System." PhD thesis, Dept. of Computer and Information Sciences, University of Pennsylvania, 1979
- [8] Keenan, E.L. "Two Kinds of Presupposition in Natural Language." In Studies in Linguistic Semantics (C.J. Fillmore and D.T. Langendoen, eds.). Holt, Rinehart, and Winston, New York, 1971
- [9] Lee, R.M. "Conversational Aspects of Database Interactions." In Proc. 4th Int. Conf. on Very Large Data Bases. Berlin, Germany, 1978
- [10] Lehnert, W.G. "The Process of Question Answering." Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1978