****

This material is planned for inclusion in a forthcoming book by C. J. Date

****

A NOTE ON THE RELATIONAL CALCULUS

by

C. J. Date

PO Box 2647, Saratoga
California, USA 95070

408/867-0695

August 1989

Abstract

We examine a logical anomaly in Codd's relational calculus [1], according to which queries can occasionally return somewhat surprising results.

INTRODUCTION

We start by considering a simple example. Suppose we have a database containing three union-compatible relations X, Y, and Z, each with just one attribute called A. Suppose also that relations X and Y each contain just one tuple, each of which in turn contains the single scalar value "a", and relation Z contains no tuples at all (refer to Fig. 1).

X	A
	a

Y	A
	a

Z	A

Fig. 1: Sample data values

Now let tx, ty, and tz be tuple variables ranging over relations X, Y, and Z, respectively, and consider the following query (i.e., relational calculus expression):

```
tx.A  WHERE  EXISTS ty
             ( ty.A = tx.A )
       OR    EXISTS tz
             ( tz.A = tx.A )
```

The intuitive interpretation of this query is "Find those values in relation X that exist in either relation Y or relation Z (or both)." Given the sample data of Fig. 1, the result intuitively expected is "a" -- or, more precisely, it is a relation of degree one, containing just one tuple, which in turn contains the single scalar value "a" (see Fig. 2, part a). The result actually produced, however, is a relation of degree one containing no tuples at all! -- see Fig. 2, part b.

a. Expected result

A
a

b. Actual result

A

Fig. 2:
a. Result intuitively expected
b. Result actually produced

The reason for this somewhat surprising state of affairs is explained in the sections that follow.

Aside: For people unfamiliar with the relational calculus, we show a SQL version of the query

also:

```
SELECT TX.A FROM X TX
WHERE  EXISTS
     ( SELECT * FROM Y TY
       WHERE TY.A = TX.A )
OR     EXISTS
     ( SELECT * FROM Z TZ
       WHERE TZ.A = TX.A ) ;
```

Or, still more intuitively (using the SQL trick that allows the name of a relation to be used as an implicit tuple variable that ranges over the relation in question, together with SQL's rules for implicit name qualification):

```
SELECT A FROM X
WHERE  EXISTS
     ( SELECT * FROM Y
       WHERE Y.A = X.A )
OR     EXISTS
     ( SELECT * FROM Z
       WHERE Z.A = X.A ) ;
```

Two points in connexion with this SQL version, however:

1. First, the SQL version may actually not be equivalent to the original calculus version, as will subsequently be made clear. Indeed, this fact is part of the overall message of this short paper.

2. Second, we are assuming (in both the SQL version and the original calculus version) that the system is supporting conventional two-valued logic. If instead it is supporting three-valued logic, then the SQL version is definitely not equivalent to the original calculus version, as has been clearly demonstrated elsewhere [2].

End of aside.

In order to explain what is happening in the example, it is first necessary to digress for a moment and elaborate on exactly what it is that constitutes a relational calculus expression. It is of course well known that relational calculus is an applied form of predicate calculus, tailored for use with relational databases. A relational calculus expression is really a special case of a set definition. It takes the general form

target WHERE qualification

where "target" identifies the relations and attributes from which the result set (more precisely, result relation) is to be constructed, and "qualification" identifies a condition to be satisfied by each tuple of that result relation. To be a little more specific:

1. "target" is a comma-separated list of elements, each of the form

variable . attribute

where "variable" is the name of a tuple variable and "attribute" is the name of an attribute of the corresponding relation;

2. "qualification" is a well-formed formula (abbreviated wff, pronounced "weff"), defined in accordance with the following production rule:

```
wff ::=   term
        | NOT wff
        | term AND wff
        | term OR  wff
        | IF term THEN wff
        | EXISTS variable (wff)
        | FORALL variable (wff)
```

Note: For present purposes, there is no need to define "term" any further; it can be regarded as essentially a simple comparison, such as "ty.A = tx.A", or else a wff in parentheses. Also, of course, there are several more rules governing the correct formulation of "target"s and "qualification"s, and further rules that allow the dropping of redundant parentheses. We assume that these rules are basically well understood and omit the details here.

PRENEX NORMAL FORM

Now, it is a fact that any wff of the predicate calculus (as opposed to the relational calculus) can always be converted into an equivalent wff in what is called prenex normal form ("equivalent" in that it has the same denotation or meaning). A wff is in prenex normal form if and only if all the quantifiers appear unnegated and at the left-hand end. For example, the predicate calculus wff

```
EXISTS x ( x > 5 ) OR
EXISTS y ( y > 2 )
```

is equivalent to the prenex normal form wff

```
EXISTS x ( EXISTS y ( x > 5 OR
                      y > 2 ) )
```

(Note: We are specifically interested in this paper in wffs of the form EXISTS ... OR EXISTS ... .) More generally, if

1. p(x) is a wff in which y does does not occur as a free variable, and

2. q(y) is a wff in which x does not occur as a free variable

(where by "free variable" we mean a variable that is not bound, i.e., not quantified), then the wff

```
EXISTS x ( p(x) ) OR
EXISTS y ( q(y) )
```

is equivalent to the prenex normal form wff

```
EXISTS x ( EXISTS y ( p(x) OR
                      q(y) ) )
```

The validity of this assertion, and hence the validity of the transformation of the original wff into prenex normal form, can easily be shown by means of truth tables.

THE MEANING OF RELATIONAL CALCULUS EXPRESSIONS

We return now to the relational calculus per se. In reference [1], Codd gives a reduction algorithm by which any expression of the relational calculus can be transformed into an expression of the relational algebra. That transformation is then effectively used as a basis for defining the meaning of the original calculus expression (i.e., the meaning of the calculus expression is defined to be identical to that of the corresponding algebraic expression).

The very first step in Codd's reduction algorithm is as follows [1]:

"Convert [the qualification part of the expression] to prenex normal form ... without expanding the range-coupled quantifiers" (my italics).

In this step, according to Codd,

"the range-coupled quantifiers behave just like ordinary quantifiers" [1].

As these two quotes suggest, the quantifiers of relational calculus are not identical to the quantifiers of ordinary predicate calculus. Instead, they are "range-coupled" quantifiers. Loosely speaking, a range-coupled quantifier is a quantifier that includes a definition of the range (i.e., set of permitted values) of its associated bound variable; in other words, it is shorthand for a predicate calculus wff of the form

```
Qx ( x IN X )
```

where Q is a normal quantifier, x is a bound variable, X is a relation, and "x IN X" is a term meaning "x is a member (i.e., tuple) of X." For example, the relational calculus wff

```
EXISTS x ( p(x) )
```

is really shorthand for the predicate calculus wff

```
EXISTS x ( x IN X AND p(x) )
```

Let us examine once again the sample relational calculus expression from the beginning of this paper:

```
tx.A  WHERE  EXISTS ty
             ( ty.A = tx.A )
      OR     EXISTS tz
             ( tz.A = tx.A )
```

We concentrate on just the qualification part, namely the relational calculus wff

```
1. EXISTS ty ( ty.A = tx.A ) OR
   EXISTS tz ( tz.A = tx.A )
```

Let us agree to refer to this expression as "Expression 1."

Expanding out the two range-coupled quantifiers to show the range definitions, we obtain the predicate calculus wff (Expression 2):

```
2. EXISTS ty ( ty IN Y AND
                  ty.A = tx.A ) OR
   EXISTS tz ( tz IN Z AND
                  tz.A = tx.A )
```

("EXISTS ty" and "EXISTS tz" are now genuine quantifiers, not range-coupled quantifiers.) The prenex normal form equivalent of this wff in predicate calculus is as follows (Expression 3):

```
3. EXISTS ty ( EXISTS tz (
   ( ty IN Y AND ty.A = tx.A ) OR
   ( tz IN Z AND tz.A = tx.A ) ) )
```

In relational calculus, however, since "range-coupled quantifiers behave just like ordinary quantifiers" in the conversion process, the prenex normal form "equivalent" of Expression 1 is Expression 4:

```
4. EXISTS ty ( EXISTS tz (
                ( ty.A = tx.A ) OR
                ( tz.A = tx.A ) ) )
```

Here the quantifiers are range-coupled. Expanding them out, we obtain Expression 5:

```
5. EXISTS ty ( ty IN Y AND
   EXISTS tz ( tz IN Z AND (
                ( ty.A = tx.A ) OR
                ( tz.A = tx.A ) ) ) )
```

which is obviously syntactically different from Expression 3. In fact, it is semantically different also. In particular, if relations X, Y, and Z are as shown in Fig. 1, then for the sole possible value of tuple variable tx, Expression 3 evaluates to true but Expression 5 evaluates to false (as the reader may readily confirm).

The first step in the reduction algorithm has thus changed the semantics of the original expression! -- and it is this fact that accounts for the counterintuitive nature of the result of the original relational calculus query.

Of course, the anomaly just described is presumably due to a mere oversight in Codd's definition of the reduction algorithm. It looks as if Codd was making a tacit assumption that the (qualification) expression under consideration did not contain any "OR"s -- especially since he goes on to form the Cartesian product of the ranges of all pertinent tuple variables, which is not appropriate in the presence of "OR"s. (To see why this is so, the reader is invited to consider the example addressed in the body of this paper once again. The Cartesian product of relations X, Y, and Z is clearly empty.)

ACKNOWLEDGMENTS

I would like to thank Charley Bontempo of IBM and Nat Goodman of Codd and Date Inc. for several helpful discussions. I would also like to thank Relational Technology Inc. for giving me the opportunity to check the original example on INGRES (using QUEL), and Colin White for trying out the SQL version for me on ORACLE. For the record, INGRES/QUEL did indeed return an empty relation, and ORACLE/SQL returned a relation containing the single value "a" -- which shows, if nothing else, that ORACLE's dialect of SQL is at least not just a syntactic variation on Codd's original relational calculus.

Note added in final version: After I had prepared the original version of this paper, I discovered that the error in the reduction algorithm had already been pointed out in an earlier paper by Klug [3]. Readers are referred to that paper for an alternative proof of the equivalence of the calculus and the algebra.

REFERENCES

1. E. F. Codd: "Relational Completeness Of Data Base Sublanguages." In R. Rustin (ed.), Data Base Systems, Courant Computer Science Symposia Series, Vol. 6, Prentice-Hall (1972).

2. C. J. Date: "EXISTS Is Not "Exists"! (Some Logical Flaws In SQL)." In C. J. Date, Relational Database Writings 1985 - 1989, Addison-Wesley (1989, to appear).

3. Anthony Klug: "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions." JACM 29, No. 3 (July 1982).

*** End of Paper ***