More Commentary on Missing Information in Relational Databases (Applicable and Inapplicable Information)

by

E. F. Codd

President
The Relational Institute

January 12, 1987

ABSTRACT

My earlier paper TRI/EFC-4 from The Relational Institute on this subject was published in ACM SIGMOD Record, December 1986. As a result, I have received several comments and proposed changes. The purpose of this paper is to respond to these items, while continuing to place heavy emphasis on the semantic aspects of missing information. The response is principally further explanation, but a few changes of a quite minor character are proposed. In this ·paper the reader is assumed to have read TRI/EFC-4.

The Relational Institute
Suite 106
6489 Camden Avenue
San Jose, CA 95120

Telephone: 408-268-8821

TRI Technical Report
EFC-14 / 01-12-87

1 Referential integrity

The definition of referential integrity in TRI/EFC-4, page 6, corresponds closely to my 1979 definition presented on page 400, Section 2.1 of [1]. It is as follows:

Let D be a domain from which one or more single-attribute primary keys draw their values. Let K be a foreign key, which draws its values from domain D. Every unmarked value which occurs in K must also exist in the database as a value of the primary key of some base relation.

Referential integrity defined in this way applies to pairs of simple keys only (primary key PK and foreign FK). "Simple key" in this context means a key consisting of a single attribute (or column). The question arises: "Why not apply this integrity rule to pairs of keys (primary and foreign) which happen to be compound (each of which consists of two or more attributes)?" Let us look at an example.

Suppose a database contains the following relations:

RELATION	MEANING	PRIMARY KEY
R1	suppliers	S#
R2	parts	P#
R3	the capabilities of suppliers to supply parts, including price and delivery	(S#, P#)
R4	orders for parts placed with specified suppliers, including date order placed	(S#, P#, DATE)

To avoid an extra relation and keep the example simple, assume that every order is a one-line order (that is, an order for just one kind of part) and that it is impossible for two orders with the same order date to refer to identical kinds of parts.

Suppose that each of two companies has a database of this kind. However, in company A the relation R3 is used as advisory information, and there is no requirement that every combination of (S#,P#) which appears in R4 must appear in R3. In company B, on the other hand, R3 is used as controlling information: that is, if an order is placed for part p from supplier s, there must be at least one row in relation R3 stating that p is obtainable from s, and which incidentally indicates the price and speed of delivery. Of course, there may be other rows in R3 stating that p is obtainable from other suppliers. Thus, the application of referential integrity to the combination (S#,P#) as primary key in R3 and foreign key in R4 would be inapplicable in company A, but applicable in company B.

There are two ways in which this example (and similar ones) could be handled:

1) Make the referential integrity rule applicable to all PK-FK pairs of keys (whether simple or compound) for which one key PK is declared to be primary, and the other key FK is declared to be foreign; in company B, declare the (S#,P#) combination in R4 as a foreign key, which has as its target the (S#,P#) primary key of R3; in company A, avoid altogether the declaration that (S#,P#) in R4 is a foreign key;

2) Make the referential integrity rule applicable to simple PK-FK pairs of keys only, and require the DBA to impose a referential constraint on just those compound PK-FK pairs of keys for which the constraint happens to be applicable in his company -- by specifying a user-defined integrity constraint, expressed in the relational language.

Method 2) complies with my 1979 definition of referential integrity and with my pre-1979 definition of the foreign key concept. In addition, method 2) is cleaner than 1), both in use and in DBMS implementation, because it separates the the foreign key concept from the more complicated referential integrity concept. Thus, method 2) is adopted.

Referential integrity should be implemented as far as possible as a special case of user-defined integrity, because of their similarities. For example, one such common need is to give the DBA or other authorized user the freedom to specify linguistically how the system is to react to any attempt to violate these integrity constraints, whether the constraints are referential or user-defined.

Further, it should be remembered that referential integrity is a particular application of a subset constraint, namely that the set of distinct simple FK values should be a subset of the set of distinct simple PK values drawn from the same domain. Subset constraints may, however, apply between other pairs of attributes also (e.g., non-keys and keys that are non-simple). When declared and enforced, such additional constraints then reflect either business policies or government regulations.

Subset constraints are sometimes referred to as inclusion dependencies. In TRI/EFC-4 there is a reference to an article dealing with inclusion dependencies.

2 Non-cumbersome support for predicate logic

One of the requirements for a DBMS to be fully relational is that it support at least one relational language that is comprehensive with respect to database management (see rule 5 of the 12 rules specified in TRI/EFC-6 and published in the October 14, 1985 and October 21, 1985 issues of Computerworld). "Comprehensive" in this sense includes full support for first-order predicate logic.

It is important that this language be able to express each of the required features without burdening 1) users with unnecessary complexity or 2) the system with unnecessary consumption of resources. An example of this cited in my previous articles is that the user should be able to express the join operators and relational division without using Cartesian product as an intermediate result, and without any reference whatever to access paths that may be supported below the level of the base relations.

Two additional comments are applicable at this point. It should not be necessary for any user to have to replace:

either
1) the universal quantifier FOR ALL x by a logically equivalent clause involving the existential quantifier: NOT THERE EXISTS x NOT;

or
2) the existential quantifier THERE EXISTS x by a logically equivalent clause involving the universal quantifier: NOT FOR ALL x NOT.

Item 1) is tantamount to requiring a simple, straightforward way of expressing relational division, while item 2) is a similar requirement for the various kinds of joins. Present versions of the language SQL fail to satisfy item 1) -- a serious deficiency resulting from the developers having inadequate knowledge of predicate logic, and a deficiency which has the effect of making some queries very convoluted, when expressed in SQL. An example of this is the query: given a list of parts which you would like to acquire from just one supplier (if possible), find all the suppliers, each of whom can supply every part in the list.

3 Four-valued logic

Consider an example of a combination (either by AND, or by OR) of two logical conditions used in selecting employees:

(birthdate > 50-1-1) AND/OR (commission > 1000)

Suppose that for a particular employee the first condition evaluates to the truth value "missing and applicable" and the second to "missing and inapplicable". What is the truth value of the whole condition?

Clearly, we need to examine the truth tables of four-valued logic. In the following tables (Figure 1) t stands for true, f for false, i for <u>missing and inapplicable</u>, a for <u>missing and applicable</u>. Note that t, a, i, f are actual values, and should not be regarded as marked values (see section 5 for example).

P	not P
t	f
a	a
i	i
f	t

P or Q		Q			
		t	a	i	f
P	t	t	t	t	t
	a	t	a	a	a
	i	t	a	i	f
	f	t	a	f	f

P & Q		Q			
		t	a	i	f
P	t	t	a	i	f
	a	a	a	i	f
	i	i	i	i	f
	f	f	f	f	f

Figure 1: The truth tables of four-valued logic

Note that we obtain the truth tables of the TRI/EFC-4 three-valued logic by replacing i by m and a by m (where m simply stands for missing, and the reason for anything being missing is ignored). It should be clear that four-valued logic (4VL) is more precise, but more complicated than three (3VL).

I believe that the extra complexity of 4VL is not justifiable at this time, especially as DBMS designers and DBMS users are not yet comfortable with 3VL. Therefore, for the time being, we continue to support the proposal that 3VL be built into the DBMS. External specifications of a DBMS product should permit expansion at a later time from 3VL to 4VL support without impacting users' investment in application programming (or with a minimal impact). If 4VL is built into a DBMS product and 3VL is bypassed, either it should agree with the 4VL described above, or else its departures should be defended from a technical and practical standpoint.

A repetition of the warning about 3-valued logics, which I included in [1], may be appropriate here. They can yield the truth value MAYBE for an expression that happens to be TRUE, because it happens to be a tautology. For example, find the employees, whose birth year is 1940 or prior to 1940 or after 1940. Every employee should be assigned the value TRUE for this condition, even if his birth year happens to be missing! This warning applies to other multi-valued logics. It may be necessary some time in the future for DBMS products to be equipped with detection algorithms for simple tautologies of this kind.

4 Relative strength of the two marks

In section 1.5 of TRI/EFC-4 I may not have adequately stressed that the I-mark is strictly stronger than the A-mark. Any user who is authorized to update values in an attribute (or column) is thereby permitted to change any non-missing value into an A-marked value or vice versa. However, changing any non-missing value directly into an I-marked value or vice versa is "a whole different ball game": it is prohibited by the DBMS, because it would be a direct attempt to violate the meaning of an I-mark. Thus, an I-mark is treated as if it were an integrity constraint of a special kind: namely, one applied to selected objects rather than selected object types.

The state diagram specifying permitted updates (Figure 2 of section 1.5 of TRI/EFC-4) indicates that any attempt to change a non-missing value into an I-mark or vice versa must be requested in two steps, first by making a change to an A-mark, and then by changing this A-mark into the desired end result. One of these steps necessarily involves changing an A-mark into an I-mark or vice versa, and such a step requires special (mark-type-change) authorization for the particular attribute involved.

5 Scalar functions applied to marked arguments

In this context a scalar function is a function which transforms scalar arguments into a scalar result. Consider the effect of such a function when one or more of its arguments is marked.

In general, if the strongest mark on one of its arguments is I, then the scalar result is I-marked. If, on the other hand, the strongest mark is A, then the scalar result is A-marked.

For example, let % denote any one of the arithmetic operators +,-,x,/ and let z denote an unmarked scalar argument. Then:

```
        z % a = a     z % i = i     a % z = a     i % z = i
        a % a = a     a % i = i     i % a = i     i % i = i
```

The functions NEGATION, OR, AND are not exceptions to this general rule, because of the distinction (noted in section 3) between the truth values a,i and marked truth values (A-marked, I-marked).

6 Types of marks generated by operators

The question has been raised: why do those operators which are capable of generating new marks in the result create A-marks only (never I-marks)? In this context, "new marks" means marks not simply copied into the result from one or other of the operands. The answer is that A-marks are preferred, because they are the weaker and the more flexible of the two types. Hence, they are more readily changed by users without needing any special mark-type-change authorization, unless the intent is to change an A-mark into an I-mark.

7 Temporary replacement of missing items

When a statistical or aggregate function is applied to an attribute, it is normally intended that the function be applied to every unmarked value of this attribute, converting either each scalar value into a scalar value or a set of such values into a single scalar value. With this intent (which is taken to be the default case), the marked values would be passed over and effectively ignored.

Occasionally, the intent is to include each marked value in the computational activity, by <u>temporarily</u> replacing it by a specified unmarked value (where "temporarily" means just for the execution of the pertinent command). In this case, the scalar replacements of a type A mark and a type I mark should be specified separately from one another. If either type of mark is omitted from the replacement specification, every occurrence of that type is ignored.

Notice that any replacement action specified in this way is a replacement of a marked argument of the function, not of any result the function might deliver. Moreover, the specified scalar replacement(s) must be values belonging to the domain from which that attribute draws its values, and must comply with any additional constraints which have been declared for that specific attribute. Finally, any specific replacement action applies to just one of the attributes cited in the retrieval or update command -- and, of course, several of the attributes cited may be subject to replacement actions. In general, if N attributes are cited in a relational command, there may be as many as 2N replacement actions specified in that command, two for each distinct citation.

Thus, a single pair of replacement qualifiers (one for I-marked values, one for A-marked values) for each command is generally inadequate -- it should be replaced by a pair of replacement qualifiers for each attribute cited in any pertinent command. The syntax must allow one pair to be specified for each statistical function cited, and unambiguously associate that pair with the pertinent function.

8 Language features needed

In section 1.14 of TRI/EFC-4 several minor language aspects were omitted, even though they were in most cases covered elsewhere in the report: for example, the use of the MAYBE qualifier on a condition, whenever only those items are needed for which this condition evaluates to MAYBE. Note that, in order to support this qualifier, the DBMS must be able to handle either three-valued or four-valued logic (including the truth tables).

Moreover, if the items X are needed for which the condition K evaluates either to TRUE or to MAYBE, then a command such as:

(X where K) UNION (X where K MAYBE)

should be used. Further, if the DBMS supports four-valued logic, then we need two additional qualifiers to distinguish the truth values A-MAYBE and I-MAYBE, where A-MAYBE means maybe true, maybe false, but certainly applicable; while I-MAYBE means neither true nor false, but inapplicable. Finally, note that, because the MAYBE qualifiers apply to conditions that may involve negation, OR, AND, the existential quantifier and the universal quantifier, they require that the DBMS handle three-valued or four-valued logic internally, and not put the burden on users (as the present version of SQL does).

9 Applying semantic override to commands

The semantic override qualifier was previously declared to be applicable to four carefully chosen types of commands -- those which involve comparison of values drawn from one or more pairs of different attributes. Users may accidentally apply this qualifier to other retrieval and manipulative commands. In these cases execution of the command proceeds, providing that the user is appropriately authorized, but the qualifier is ineffective, no trigger is set, and there is no error message. This represents a minor extension of the TRI/EFC-4 proposal.

10 Application of statistical functions to empty sets

This issue (raised by C. J. Date) is not directly related to the subject of missing values. Nevertheless, it was touched on in section 2.5 of TRI/EFC-4, because SQL happens to generate null as the result of applying certain statistical functions to an empty set (an unwise choice by the SQL designers). This topic is clarified in more detail here, but only with respect to the relational model (not SQL).

Section 2.5 deals with the case of applying a statistical function to a collection of sets, some of which are empty and some non-empty. We need to treat the extreme case where all the sets are empty (even if there is only one set in the collection of sets), and in such a way that all these cases behave in a consistent way. As a first step, we require that an initial value of zero be established immediately prior to the evaluation of the pertinent function against the specified sets.

If the empty set qualifier is omitted from a command, each occurrence of an empty set is ignored. However, we shall require that there be, in addition to the value returned, a trigger (known as the empty trigger) which is turned on whenever at least one set encountered in the execution of this command is empty.

Suppose that the <u>value</u> returned, whenever a statistical function is applied to a single empty set is the initial value cited above, i.e., zero. A special case needs careful attention to avoid misinterpretation of the value returned. Whenever:

1) a statistical function is cited in a command;
2) this function (for example, AVERAGE) happens to require dividing by the number of elements in the pertinent set;
3) the value returned by the function is zero for one or more of the sets;

it would normally be necessary to examine each of these sets for its possible emptiness. Such an examination would distinguish the empty set case from the case in which the statistical function happened to generate zero from elements actually encountered in the set. The reader should remember that the burden of this extra examination arises from ordinary integer arithmetic, in which dividing by zero is unacceptable. Marks in the relational model are intended to represent the fact that information in the form of a db-value is missing, and should be distinguished from the case in which the value of a function (such as division) is undefined. The burden of this extra examination is therefore NOT a consequence of the relational model.

11 Acknowledgment

I wish to thank Ron Fagin for reviewing this paper and for his comments.

12 References

The sole author of the following articles is E. F. Codd.

[1] "Extending the Database Relational Model to Capture More Meaning", ACM TODS, Vol. 4, No. 4, December 1979

TRI/EFC-4 "Missing Information in Relational Databases: Applicable and Inapplicable", February 21, 1986

TRI/EFC-6 "The Twelve Rules for Relational DBMS", May 16, 1986

Other references to articles on missing information are listed at the end of TRI/EFC-4.