JULY 1994

$3.95
$4.95 Canadian

# DATABASE Programming & Design

**Data Warehouse: The Road to Success**

**Ten Steps to Database Migration**

**Date & McGoveran: How to Avoid Data Redundancy**

**Business Process Reengineering: The Challenge to IS**

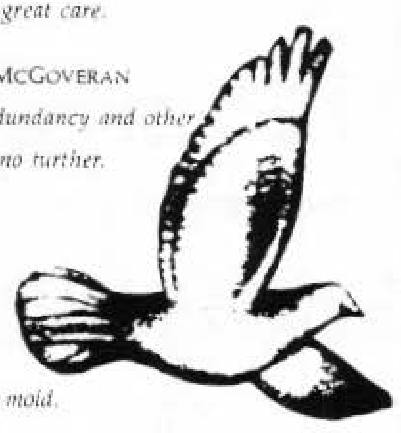
A MILLER FREEMAN PUBLICATION

07
0 70989 35537 8

VOL. 7 NO. 7

# DATABASE Programming & Design

JULY 1994



**DATABASE PROGRAMMING & DESIGN** (ISSN 0895-4518) is published monthly except in September, which is semi-monthly and contains the DATABASE PROGRAMMING & DESIGN Special Client/Server Issue, by Miller Freeman, Inc., 600 Harrison St., San Francisco, CA 94107, (415) 905-2200. Please direct advertising and editorial inquiries to this address. For subscription inquiries, call (800) 289-0169 (outside U.S. (303) 447-9330). **SUBSCRIPTION RATE** for the U.S. is $47 for 13 issues (basic). Canadian/Mexican orders must be prepaid in U.S. funds with additional postage at $10 per year. Canadian GST Permit #124513185. All other countries outside the U.S. must be prepaid in U.S. funds with additional postage at $15 per year for surface mail or $40 per year for air mail. **POSTMASTER:** Send address changes to DATABASE PROGRAMMING & DESIGN, P.O. Box 53481, Boulder, CO 80322-3481. For quickest service call toll-free (800) 289-0169 (in Colorado or outside the U.S. (303) 447-9330). Please allow six weeks for change of address to take effect. SECOND CLASS POSTAGE paid at San Francisco, CA 94107 and at additional mailing offices. **DATABASE PROGRAMMING & DESIGN** is a registered trademark owned by the parent company, Miller Freeman Inc. All material published in **DATABASE PROGRAMMING & DESIGN** is copyrighted © 1994 by Miller Freeman Inc. All rights reserved. Reproduction of material appearing in DATABASE PROGRAMMING & DESIGN is forbidden without permission. 16mm microfilm, 35mm microfilm, 105mm microfiche and article and issue photocopies are available from University Microfilms International, 300 N. Zeeb Rd., Ann Arbor, MI 48106, (313) 761-4700.

BY C. J. DATE AND DAVID McGOVERAN

*Redundancy, update anomalies, and other integrity problems are the bane of the relational database designer's existence. Here's an "intuitive" solution*

# A New Database Design Principle

IN THIS ARTICLE, we will present and describe a new—and, we believe, very fundamental—database design principle. Like the well-known principles of normalization, our principle can be seen in part as a systematic means of avoiding redundancy, and thereby avoiding certain update anomalies that might otherwise occur. Although the principle is very simple, we have never seen it appear in published form; in fact, our experience suggests that the principle is quite often violated in practice. We will briefly explore some of the consequences of this type of violation.

## THE LOVES-HATES EXAMPLE

We begin with a simple example. Consider the following database:

```
CREATE DOMAIN PERSONS ... ;

CREATE BASE TABLE LOVES
     ( X DOMAIN ( PERSONS ),
       Y DOMAIN ( PERSONS ) ... ) ;

CREATE BASE TABLE HATES
     ( X DOMAIN ( PERSONS ),
       Y DOMAIN ( PERSONS ) ... ) ;
```

The intended semantics are, of course, that the row <x,y> appears in LOVES only if "x loves y" is true, and the row <x,y> appears in HATES only if "x hates y" is true.

*Note:* Throughout this article, we use a modified form of conventional SQL syntax, for reasons of simplicity and explicitness.

Now suppose the row <Romeo, Juliet> is to be inserted into this database. The user responsible for the INSERT will presumably insert the row into base table LOVES. *Note carefully, however, that he or she could just as easily have inserted the row into base table* HATES *instead.* It is only because the user had some additional knowledge—namely, the knowledge that "Romeo loves Juliet" is true—that he or she decided to insert the row into LOVES and not into HATES. Since that additional knowledge is not known to the

DBMS, the "decision procedure" for deciding whether a given row should be inserted into LOVES or HATES is likewise not known to the DBMS. In other words, part of the meaning of the database is effectively *concealed from the system.*

Note, moreover, that it is not just the DBMS that is affected by this concealment. *Exactly the same information is being concealed from other users as well.* That is, given the same row <Romeo,Juliet>, another user will be just as unable to decide which of LOVES and HATES the row is to go into if he or she does not have the necessary extra information (namely, that "Romeo loves Juliet" is true). In other words, suppose we are told that for a certain pair of persons *x* and *y* either "*x* loves *y*" is true or "*x* hates *y*" is true. In general, then, we will only be able to tell which of these two possibilities is in fact the case by looking to see which of the two base tables the row <x,y> appears in (assuming we are not privy to the extra information).

Lest the reader object that the "extra information" is in fact transparently obvious since it is represented by the *names* of the two base tables involved (LOVES and HATES), let us now rename those two tables HATES and LOVES, respectively. Now can you tell which of "*x* loves *y*" and "*x* hates *y*" is true? The answer is, of course, that you can tell only if you know that HATES "means" loves and LOVES "means" hates!

Before attempting to draw any conclusions, let us move on to examine another example.

## THE EMPLOYEES EXAMPLE

Suppose we have a database concerning employees, in which every employee has a (unique) employee number, EMP#; a name, ENAME; a department number, DEPT#; and a salary, SALARY. Furthermore, suppose that we decide (for some reason—the precise reason is not important for the moment) to represent employees by *two* base tables, EMPA and EMPB, where:

☐ EMPA contains rows for employees in department D1;

☐ EMPB contains rows for employees who are *either* not in department D1 *or* have a salary in excess of 33K.

**EMP**

| EMP# | ENAME | DEPT# | SALARY |
|---|---|---|---|
| E1 | Lopez | D1 | 25K |
| E2 | Cheng | D1 | 42K |

**EMPB**

| EMP# | ENAME | DEPT# | SALARY |
|---|---|---|---|
| E2 | Cheng | D1 | 42K |
| E3 | Finzi | D2 | 30K |
| E4 | Saito | D2 | 45K |

**FIGURE 1.** *Base tables EMPA and EMPB (first version): sample values.*

See Figure 1 for some sample values.

The reader will surely agree that this design is bad. But why exactly is it bad? We can gain some insight into this question by considering the following scenario. Suppose first of all that we start off with an empty database (that is, base tables EMPA and EMPB both contain no rows at all). Suppose next that we are asked to insert information regarding employee E2 (name Cheng, department D1, salary 42K) into this database. We construct the row:

<E2, Cheng, D1, 42K>

But which base table do we put it in? The answer, obviously, must be *both* (as suggested by Figure 1). It must be both, because the new row satisfies both (a) the criterion for membership in EMPA (the department number is D1), and (b) the criterion for membership in EMPB (the salary is greater than 33K). After all, if the row were to be inserted into just one of the two base tables, the question is which one? There are no grounds, except arbitrary ones, for choosing either table over the other.

*Aside:* In fact, if we decide to put the row into just one of the tables—say EMPA and not EMPB—we could be accused of a *contradiction*. For the appearance of the row in EMPA would mean that Cheng works in department D1 and earns 42K, while the simultaneous nonappearance of the row in EMPB would mean that Cheng either does not work in department D1 or does not earn more than 33K. *End of aside.*

Thus, we see that one reason the design is bad is that it leads to *redundancy*. The very same information is represented twice, in two distinct base tables.

Of course, it is fairly easy to see what causes the redundancy in this particular example. To be specific, a certain *lack of independence* exists between the two base tables EMPA and EMPB, inasmuch as "their meanings overlap" (it is possible for the same row to satisfy the membership criterion for both). Lack of independence between objects is generally to be avoided if possible, because it implies that changes to one object will require changes to the other as well. For instance, a DELETE on one table might require a DELETE on another (as is indeed the case in our EMPA-EMPB example, if we wish to delete the information for employee E2).

Thus, it looks as if a good design principle might be "Don't have tables whose meanings overlap"—and indeed, so it is. Before we can make this principle more precise, however, we must examine the question of table meaning in greater depth. Note in particular that the principle as just—very loosely—articulated is not sufficient in itself to explain what is wrong with the LOVES-HATES example discussed earlier.

## INTEGRITY CONSTRAINTS[10]

To discuss what tables mean, we must first digress for a moment to consider the general issue of *integrity constraints*. For this discussion, it is convenient to classify such constraints into three kinds: namely, column constraints, table constraints, and database constraints,[7] as follows:

☐ A *column* constraint states that the values appearing in a specific column must be drawn from some specific domain. Consider base table EMPB from the previous section. The columns of that table are subject to the following column constraints:

```
e.EMP# IN EMP#_DOM
e.ENAME IN NAME_DOM
e.DEPT# IN DEPT#_DOM
e.SALARY IN US_CURRENCY_DOM
```

Here *e* represents an arbitrary row of the table and EMP#_DOM, NAME_DOM, and so on, are the names of the relevant domains.

☐ A *table* constraint states that the rows of a specific table must satisfy some specific condition, where the condition in question refers *solely* to the table under consideration—that is, it does not refer to any other table, nor to any domain. For example, here are two table constraints for base table EMPB:

```
1. e.DEPT# ≠ 'D1' OR e.SALARY > 33K

2. IF e.EMP# = f.EMP#
   THEN e.ENAME  = f.ENAME
    AND e.DEPT#  = f.DEPT#
    AND e.SALARY = f.SALARY
```

The first of these constraints is self-explanatory. The second constraint says that if two rows *e* and *f* have the same EMP# value, then they also have the same ENAME value, the same DEPT# value, and the same SALARY value—in other words, they are the same row. (In other words, EMP# is a candidate key. Naturally we assume that all tables do have at least one candidate key!—that is, duplicate rows are not permitted.)

Incidentally, note that we are speaking of table constraints in general, not just base table constraints. The point is, *all* tables, base or otherwise, are subject to table constraints, as we have discussed in detail elsewhere.[5] For present purposes, however, it is indeed base table constraints in particular that are our primary interest; for the remainder of this article, therefore, we will take the unqualified term "table constraint" to mean a base table constraint specifically, barring explicit statements to the contrary.

☐ A *database* constraint states that the overall database must satisfy some specific condition, where the condition in question can refer to—or, more precisely, *interrelate*—as many tables as desired. For example, suppose the database shown in Figure 1 were extended to include a departments table, DEPT. Then the referential constraints from EMPA and EMPB to DEPT would both be database constraints (they would both in fact refer to exactly two tables).

## THE QUESTION OF MEANING

Now we can get back to our discussion of what tables (and indeed databases) *mean*. The first point is that every table—be it a base table, a view, a query result, or whatever—certainly does have an associated meaning. And, of course, users must be aware of these meanings if they are to use the database correctly and effectively. For example, the meaning of base table EMPB is something like the following:

*"The employee with the specified employee number (*EMP#*) has the specified name (*ENAME*), works in the specified department (*DEPT#*), and earns the specified salary (*SALARY*). Furthermore, either the department number is not D1 or the salary is greater than 33K (or both). Also, no two employees have the same employee number."*

Formally, this "meaning" is an example of what is called a *predicate*, or a truth-valued function—a function of four arguments, in this particular case. Substituting values for these arguments is equivalent to *invoking* the function (or "instantiating" the predicate), thereby yielding an expression that evaluates to either *true* or *false*. For example, the following substitution:

```
EMP# = 'E3'
ENAME = 'Finzi'
DEPT# = 'D2'
SALARY = '30K'
```

yields the value *true*. By contrast, the substitution:

```
EMP# = 'E3'
ENAME = 'Clark'
DEPT# = 'D2'
SALARY = '25K'
```

yields the value *false*. And at any given time, of course, the table contains exactly those rows that make the predicate evaluate to *true* at that time.

It follows that if, for example, a row is presented as a candidate for insertion into some table, the DBMS should accept that row only if it does not cause the corresponding predicate to be violated. More generally, the predicate for a given table represents the *criterion for update acceptability*

**Vendors must incorporate proper domain support**

for that table—that is, it constitutes the criterion for deciding whether or not some proposed update is in fact valid (or at least plausible) for the given table. In other words, such a predicate corresponds to what we earlier referred to as the *membership criterion* for the table in question.

In order for it to be able to decide whether or not a proposed update is acceptable for a given table, therefore, the DBMS must be aware of that table's predicate. Now it is, of course, not possible for the DBMS to know *exactly* what the predicate is for a given table. In the case of base table EMPB, for example, the DBMS has no way of knowing *a priori* that the predicate is such that the row <E3,Finzi,D2,30K> makes it *true* and the row <E3,Clark,D2,25K> does not; it also has no way of knowing exactly what certain terms appearing in that predicate mean (such as "works in" or "earns"). However, the DBMS certainly *does* know a reasonably close approximation to that predicate. To be specific, it knows that, if a given row is to be deemed acceptable, all of the following must be true:

- ☐ The EMP# value must be a value from the domain of employee numbers.
- ☐ The ENAME value must be a value from the domain of names.
- ☐ The DEPT# value must be a value from the domain of department numbers.
- ☐ The SALARY value must be a value from the domain of U.S. currency.
- ☐ Either the DEPT# value is not D1 or the salary is greater than 33K (or both).
- ☐ The EMP# value is unique with respect to all such values in the table.

In other words, for a base table such as EMPB, the DBMS knows all the integrity constraints (column and table constraints) that have been declared for that base table. Formally, therefore, we can *define* the (DBMS-understood) "meaning" of a given base table to be the logical AND of all column and table constraints that apply to that base table (and it is this meaning that the DBMS will check whenever an update is attempted on the base table in question). For example, the formal meaning of base table EMPB is:

```
  e.EMP# IN EMP#_DOM AND
  e.ENAME IN NAME_DOM AND
  e.DEPT# IN DEPT#_DOM AND
  e.SALARY IN US_CURRENCY_DOM AND
( e.DEPT# ≠ 'D1' OR e.SALARY > 33K ) AND
( IF e.EMP# = f.EMP#
  THEN e.ENAME  = f.ENAME
   AND e.DEPT#  = f.DEPT#
   AND e.SALARY = f.SALARY )
```

We will refer to this expression—let us call it PE—as *the table predicate* for base table EMPB.

*Aside:* To repeat an observation, note how the remarks from the previous article in this series serve to point up once again the fundamental importance of the relational *domain* concept. The relational vendors should be doing all within their power to incorporate proper domain support into their DBMS products. It is perhaps worth pointing out too that "proper domain support" does *not* mean support for the very strange construct called "domains" in the SQL standard. *End of aside.*

To return to the main thread of our discussion: As indicated, for the DBMS to be able to decide whether or not a given update is acceptable on a given table, the DBMS must be aware of the table predicate that applies to the table in question. Now, the DBMS certainly is aware of the relevant predicate in the case of a base table, as we have just seen. But what about *derived* tables—for example, what about views? What is the table predicate for a derived table?

Clearly, we need a set of rules such that if the DBMS knows the table predicate(s) for the input(s) to any relational operation, it can deduce the table predicate for the output from that operation. Given such a set of rules, the DBMS will then know the table predicate for all possible tables.

It is in fact quite easy to state

**EMP**

| EMP# | ENAME | DEPT# | SALARY |
|---|---|---|---|
| E1 | Lopez | D1 | 25K |
| E2 | Cheng | D1 | 42K |

**EMPB**

| EMP# | ENAME | DEPT# | SALARY |
|---|---|---|---|
| E3 | Finzi | D2 | 30K |
| E4 | Saito | D2 | 45K |

**FIGURE 2.** *Base tables EMPA and EMPB (second version): sample values.*

such a set of rules—they follow immediately from the definitions of the relational operators. For example, if A and B are any two type-compatible tables[2] and their respective table predicates are PA and PB, then the table predicate PC for table C, where C is defined as A INTERSECT B, is obviously (PA) AND (PB); that is, a row *r* will appear in C if and only if it appears in both A and B—that is, if and only if PA(*r*) and PB(*r*) are both true. So if, for example, we define C as a view and try to insert *r* into that view, *r* must satisfy both the table predicate for A and the table predicate for B, or the INSERT will fail.[2]

Here is another example: The table predicate for the table that results from the following *restriction* operation:

```
T WHERE condition
```

is (PT) AND (condition), where PT is the table predicate for T. For example, the table predicate for EMPB WHERE SALARY < 40K is:

```
( PE ) AND ( SALARY < 40K )
```

where PE is the table predicate for EMPB as defined earlier.

We will leave the table predicates corresponding to the other relational operators as an exercise for the reader.

We conclude this section by remarking that—although it is somewhat irrelevant to the main theme of this article—the overall database has a formal meaning too, just as the individual base tables do. The meaning of the database—the *database predicate* for that database—is essentially the logical AND of all individual table predicates for base tables in that database, together with all database constraints that apply to that database.

## OVERLAPPING MEANINGS

Now we can pin down what we mean when we say that the meanings of two tables overlap. Let A and B be any two tables, with associated table predicates PA and PB, respectively. Then the meanings of A and B are said to *overlap* if and only if some row *r* can be constructed such that PA(*r*) and PB(*r*) are both true.

Given this definition, our new design principle is now precise: *Within a given database, no two distinct base tables should have overlapping meanings.*

However, two very important corollaries of the principle are perhaps not immediately obvious, and in any case are worth stating explicitly. The first has to do with *isomorphic tables*. For the purposes of this article, we define two tables, A ( A1, ..., An ) and B ( B1, ..., Bn ), to be *isomorphic* if and only if a one-to-one correspondence exists between the columns of A and the columns of B, say A1:B1, ..., An:Bn, such that in each pair of columns Ai:Bi (i = 1, ..., n), the two columns are defined on the same domain. *Note:* Two tables that are type-compatible are certainly isomorphic, but tables can be isomorphic without necessarily being type-compatible (because type-compatibility as we define it requires the two tables to have identical column names).[3]

Here, then, is the first corollary:

☐ *Two tables cannot possibly have overlapping meanings if they are not isomorphic.*

It follows that our design principle applies specifically to isomorphic tables. However, we caution the reader that the tables in question are not necessarily base tables! Refer to the section "Important Clarification" later in this article.

The second corollary follows:

☐ *When a given row r is presented for insertion into the database, the DBMS should be able to decide for itself which table (if any) row r belongs to.*

In other words, the process of inserting a row can be regarded as a process of inserting that row *into the database* (rather than into some specific table), provided the design principle is followed.

*Note:* On being informed of this point, the reader might very well respond by saying "So what?"—relational languages always require the user to specify the target table on an INSERT, so what is the advantage of having the system be able to figure out what the target table is? One answer to this question is that the specified target table might be a *view*. Consider an INSERT into a view, V, defined as the union of two tables, A and B. As we have discussed in detail elsewhere,[5] it is very desirable that the system be able to decide for itself which of A and B the new row belongs to.

## THE EXAMPLES REVISITED

Now let us revisit the examples discussed earlier in this article. First of all, the EMPA-EMPB design is clearly bad, since the meanings of the two tables clearly overlap. But suppose we were to redefine those tables as follows:

☐ EMPA contains rows for employees in department D1.

☐ EMPB contains rows for employees not in department D1.

See Figure 2 for some sample values.

The meanings of the tables now do not overlap. However, *the design is still bad if the DBMS is not aware of that fact;* that is, if the table predicates for the two tables *as stated to the DBMS* do not include the terms

```
... AND e.DEPT# = 'D1' ... /* for EMPA */
... AND e.DEPT# ≠ 'D1' ... /* for EMPB */
```

then the meanings still overlap as far as the DBMS is concerned. In other words, the word "meaning" in our design principle refers specifically to the "meaning" as it is understood by the DBMS—of course—and *not* necessarily to the meaning as understood by the user.

What about the LOVES-HATES example? Well, here are the table predicates for the design as originally given:

```
r.X IN PERSONS AND r.Y IN PERSONS
/* for LOVES */
```

**EMPX**

| EMP# | ENAME | DEPT# |
|---|---|---|
| E1 | Lopez | D1 |
| E2 | Cheng | D1 |
| E3 | Finzi | D2 |
| E4 | Saito | D2 |

**EMPY**

| EMP# | ENAME | SALARY |
|---|---|---|
| E1 | Lopez | 25K |
| E2 | Cheng | 42K |
| E3 | Finzi | 30K |
| E4 | Saito | 45K |

**FIGURE 3.** *Base tables EMPX and EMPY: sample values.*

```
r.X IN PERSONS AND r.Y IN PERSONS
/* for HATES */
```

(where r is an <x,y> row). These two predicates are identical, of course, and therefore most certainly do overlap!" In fact, the example is not really different in kind from the second EMPA-EMPB example.

Here, by contrast, is a revised design that does not violate our design principle:

```
CREATE DOMAIN PERSONS ... ;

CREATE DOMAIN L_OR_H VALUES
       ( 'loves', 'hates' ) ;

CREATE BASE TABLE LOVES
     ( X DOMAIN ( PERSONS ),
       R DOMAIN ( L_OR_H ),
       Y DOMAIN ( PERSONS ) ... ) ;

CREATE BASE TABLE HATES
     ( X DOMAIN ( PERSONS ),
       R DOMAIN ( L_OR_H ),
       Y DOMAIN ( PERSONS ) ... ) ;
```

The table predicates are now as follows (and should be so defined to the DBMS):

```
r.X IN PERSONS AND r.Y IN PERSONS
               AND r.R = 'loves'
/* for LOVES */

r.X IN PERSONS AND r.Y IN PERSONS
               AND r.R = 'hates'
/* for HATES */
```

To insert the information that Romeo loves Juliet, it is now necessary to insert the row <Romeo,loves,Juliet>. Note, incidentally, that nothing is stopping us from inserting the row <Romeo,hates,Juliet> as well! In fact, the two three-column base tables LOVES and HATES might as well now be replaced by a single base table that is the union of the two. (*Exercise for the reader:* What would the corresponding table predicate be?) An analogous remark applies to the second EMPA-EMPB example.

## IMPORTANT CLARIFICATION

So far, we can sum up the message of this article as follows: Whenever your database design includes two distinct base tables that are isomorphic, be sure that the DBMS-understood meanings of those two tables do not overlap. This rule (or discipline) is easy to state and apply, and it would be nice if matters stopped right there. Unfortunately, however, we have so far overlooked one important ramification. Consider the tables EMPX and EMPY shown in Figure 3.

It should be clear that Figure 3's design is bad, because of the redundancy it implies. Here, however, the overlap in meaning occurs, not between the two tables EMPX and EMPY, but rather between the two *projections* of those tables over EMP# and ENAME. Clearly, therefore, we must extend our design principle to deal with such a situation, as follows:

*Let* A *and* B *be any two base tables in the database. Then there must not exist nonloss decompositions of* A *and* B *into* A1, A2, ..., Am *and* B1, B2, ..., Bn *(respectively) such that two distinct projections in the set* A1, A2, ..., Am, B1, B2, ..., Bn *have overlapping meanings.*

By the term "nonloss decomposition" (of some given table), we mean a decomposition of that table—according to the well-known principles of normalization—into a set of projections such that the given table is equal to the join of those projections.

*Note:* This refined version of our design principle in fact subsumes the original version, because one "nonloss decomposition" of any given table T is the set of projections consisting of just the "identity projection" T itself. In other words, if we agree to refer to tables that have projections whose meanings overlap as having meanings that *partially* overlap, then *total* overlap is just a special case (that is, two tables that have totally overlapping meanings certainly have partially overlapping meanings, *a fortiori*).

## CONCLUDING REMARKS

To close, let us point out some important consequences of the preceding discussions:

1. Readers might be tempted to think that our new design principle is very obvious and is really just common sense. And in a way they would be right. But the principles of normalization (such as third-normal form, and so forth) are likewise "obvious and just common sense." The point is, however, that the principles of normalization take these common sense ideas and *provide a precise, accurate characterization of those intuitive concepts.* In a similar manner, our new design principle provides a precise, accurate characterization of certain additional intuitive concepts.

2. We have not only encountered genuine database designs in which the principle has been flouted (despite the fact that it is "really just common sense"), we have also encountered database practitioners and database "experts" who have expressly recommended flouting that principle. Indeed, we have probably all seen designs such as the following:

```
ACTIVITIES_88 ( ENTRY#, DESCRIPTION,
 AMOUNT, NEW_BALANCE )
ACTIVITIES_89 ( ENTRY#, DESCRIPTION,
 AMOUNT, NEW_BALANCE )
ACTIVITIES_90 ( ENTRY#, DESCRIPTION,
 AMOUNT, NEW_BALANCE )
ACTIVITIES_91 ( ENTRY#, DESCRIPTION,
 AMOUNT, NEW_BALANCE )
ACTIVITIES_92 ( ENTRY#, DESCRIPTION,
 AMOUNT, NEW_BALANCE )
ACTIVITIES_93 ( ENTRY#, DESCRIPTION,
 AMOUNT, NEW_BALANCE )
```

and so forth—in which activities for different years are kept in different tables.

3. In a design such as the one that was just illustrated, part of the (informal, user-understood-but-not-DBMS-understood) meaning of the database is *encoded in the table names.* Such a design can thus be seen as violating Codd's "Information Principle," which can be stated as follows:

*All information in the database must be cast explicitly in terms of values in tables and in no other way.*

Our design principle—at least, the part of it that recommends against the use of names in order to carry meaning—can thus be seen as a corollary (although not a very obvious one) of Codd's Information Principle.[9]

4. Note that adherence to our design principle has the consequence that if `A` and `B` are any two type-compatible base tables, then it will be true for all time that:

| | |
|---|---|
| `A UNION B` | is a disjoint union |
| `A INTERSECT B` | is empty |
| `A MINUS B` | is equal to `A` |

5. Adherence to our principle also has the very desirable consequence that the rules for updating union, intersection, and difference views work very well and never produce "surprising results."[5]

6. One final *and very important* remark: The new design principle is equally applicable to the design of what might be called "individual user databases"—that is, an individual user's perception (as defined by views and/or base tables) of some underlying shared database. In other words, such an "individual user database" ought not to include any views and/or base tables whose meanings overlap (even partially), for essentially all of the same reasons that the shared database ought not to include any base tables whose meanings overlap (even partially). ▮

*The authors would like to thank Hugh Darwen, Fabian Pascal, and Paul Winsberg for their helpful comments on earlier drafts of this article.*

## REFERENCES

1. Codd, E. *The Relational Model for Database Management: Version 2.* Addison-Wesley, 1990.

2. Date, C. *An Introduction to Database Systems*, 6th edition. Addison-Wesley, to appear in September 1994.

3. Date, C. According to Date. "A Matter of Integrity," (in three parts). *Database Programming & Design* 6(10): 15-18, October 1993; 6(11):15-18, November 1993; and 6(12): 19-21, December 1993.

4. Date, C. "A Contribution to the Study of Database Integrity," *Relational Database Writings 1985-1989.* Addison-Wesley, 1990.

5. Date, C., and D. McGoveran. "Updating Union, Intersection, and Difference Views," *Database Programming & Design*, 7(6): 46-53, June 1994.

6. This classification is slightly different (but not dramatically so) from that previously given in references [3] and [4].

7. Type-compatibility is usually referred to as *union-compatibility*. We prefer our term for reasons that are beyond the scope of the present discussion.

8. The same would still be true if we renamed the `X` and `Y` columns (say) `L1` and `L2` in `LOVES` and `H1` and `H2` in `HATES` (the tables would still be isomorphic). Consider what would happen on an attempt to insert a row into the union of the two tables.

9. In further discussion of his Information Principle, Codd points out that names too are "cast in terms of values": "Even names are represented as character strings in [certain tables that] are normally part of the built-in database catalog" (reference [1], page 31). This fact is something of a red herring, however; in no way does it invalidate our new design principle.

10. Portions of the "Integrity Constraints" section of this article appeared in a different form in reference [5].

**C. J. Date is an independent author, lecturer, and consultant, specializing in relational database systems.**

**David McGoveran is president of Alternative Technologies (Boulder Creek, California), a relational database consulting firm founded in 1976.**