The Industry's New Power Players • Exploring Repository Standards

# DATABASE Programming & Design

## Building on Today's Good Models

## Legacy Systems Migration: Risks and Rewards

## The Magic Behind DB2's Access Path Selection

## From Model to Relational DBMS Design

MARCH 1994
$3.95
$4.95 Canadian

VOL. 7 NO. 3

# DATABASE Programming & Design

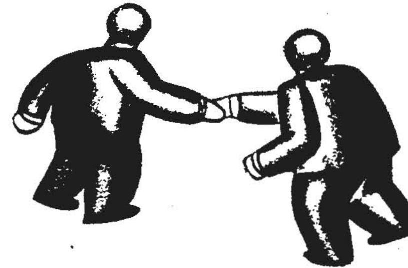
MARCH 1994

**Building on Today's Good Models** — 26 — ADRIENNE TANNENBAUM *Can we take our existing models and make them better? And will CASE help—or hurt?*

**The Magic of Materialization** — 32 — CAROLYN DOW, JANET RIZNER, AND DOUG STACEY *Know DB2's access path magic, and you'll perform tricks of your own.*

**From Model to Database** — 46 — MARK M. DAVYDOV *Transforming ER models into relational DBMS designs is a rough passage; here's a solution.*

**Nothing From Nothing: The Conclusion** — 54 — DAVID MCGOVERAN *Our final installment describes how designers can handle nulls without many-valued logic.*



## DEPARTMENTS

EDITOR'S BUFFER — 7 — *Database World greets new power players.*

ACCESS PATH — 11 — *A reader counters Codd and Date.*

DATABASE DESIGN — 13 — *Seeing the forest—and the trees.*

ACCORDING TO DATE — 19 — *If only we used tables with no columns . . .*

CLIENT/SERVER FORUM — 23 — *Migrating the legacy: Risks and rewards.*

ENTERPRISE VIEW — 67 — *Checking the status of repository standards.*

PRODUCT WATCH — 70 — *Connectivity, CASE, client /server, and more.*

DATABASE PROGRAMMING & DESIGN (ISSN 0895-4518) is published monthly, except in October, which is semi-monthly and contains the DATABASE PROGRAMMING & DESIGN Buyer's Guide, 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 DAVID McGOVERAN

*Database designers and users need to represent missing information: Can it be done without resorting to many-valued logic? Yes—here's how*

# Nothing from Nothing Part IV: It's in the Way that You Use It

IN PART III OF THIS four-part series of articles, I reviewed and categorized the main reasons that both database designers and users find themselves wanting the support of a many-valued logic. In this (the final) part of the series, I propose a number of methods by which database designers can handle missing information without the need for many-valued logic.

The emphasis is on using logical design principles to eliminate the need for nulls. Logical design is defined here as the database design that is inferrable from logical concerns alone. This design is not affected by physical implementation issues such as denormalization, index creation, or space allocation (which are properly the physical design's domain), nor is it affected by the manner in which data is perceived by particular users (the conceptual design). It is my position that all design should first be logical, with the physical design deviating from an augmentation of the logical design only where no other alternative exists. The reasons for deviating from the logical design should be documented and the impact on data integrity thoroughly assessed.

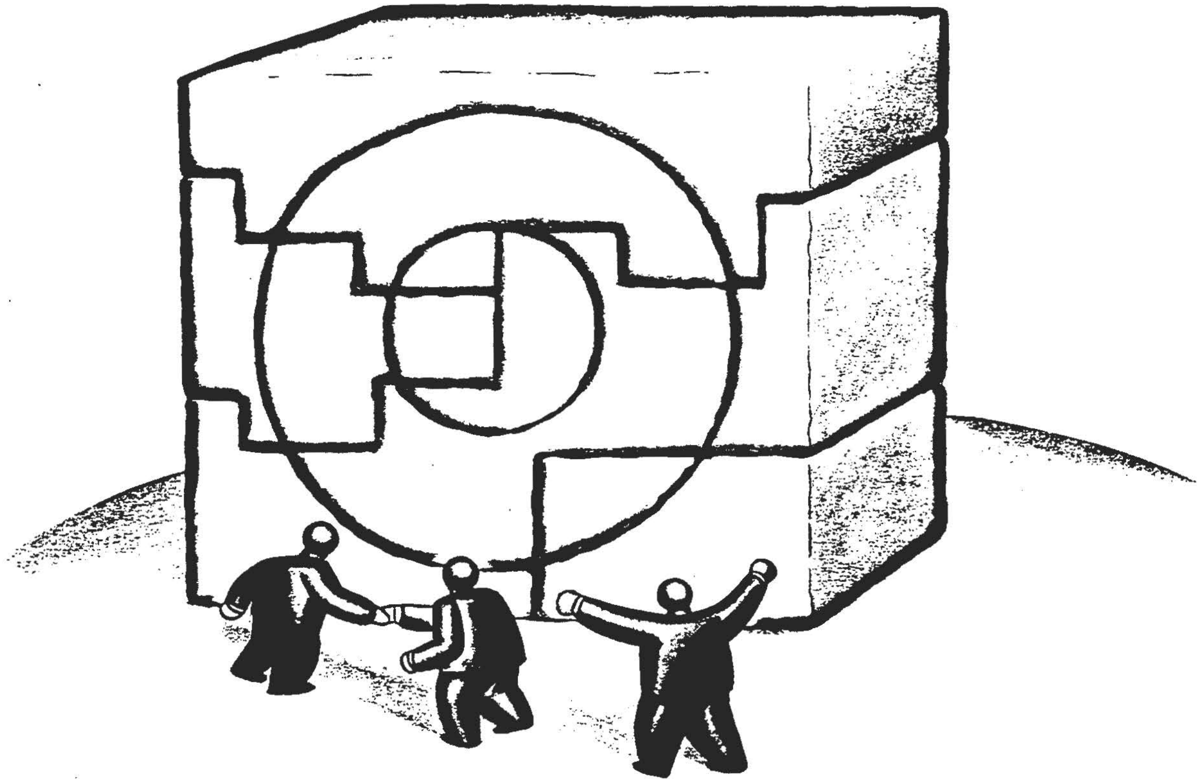
I will invoke several database design principles in this article. An important one is captured in what I call the "Knowledge Principle." Just as the Information Principle requires that all information be captured by values in columns, the Knowledge Principle precludes disinformation by stating: "All columns must contain values that convey knowledge about the universe of discourse (and may not contain metadata)."

Both the Knowledge Principle and the Information Principle extend trivially to the system catalog, which contains the database's structure. Thus, we record data about what we know, and do not attempt to catalog what we do not know.

The following sections will address each of the key sources of missing information, describing different techniques for handling each. These logical design techniques will eliminate the need for nulls and many-valued logic, but are *not* intended to be implemented separately; they comprise a systematic solution to the problem of missing information. In overview, the techniques proposed are:

- ☐ Separating metadata from data: part of the solution to conditional information
- ☐ Association tables: the solution to conditional relationships
- ☐ Enforcing relation predicate uniqueness: the solution to an apparent case of conditional properties (conditional entities)
- ☐ Modeling subtypes: the solution to conditional properties and conditional operations
- ☐ Abstract entities: converts some apparently conditional relationships to ordinary relationships and handles conditional constraints
- ☐ Meaningful defaults: eliminates nulls as defaults, without confusing data and metadata.

Because these techniques lead to a more accurate representation of the application domain, the resulting logical data design inherently contains more structure. On one hand, this structure will reduce the amount of application code needed to support database access, and the SQL statements used will be more meaningful and easier to write. On the other hand, SQL's weaknesses will be manifested, sometimes requiring that a larger num-



ber of SQL statements be written, and physical design may sometimes be less optimal. To meet these concerns, I propose a few enhancements to relational products:

☐ User-extensible audit trails: declarative support for user-defined metadata

☐ Multitable and computed indexes: better support for association tables

☐ Set operation support: automatic support via set operations for types and subtypes

☐ Declarative relation predicates: automatic recognition and distinction of tables based on relation predicates, of which type and subtype support is a special case

☐ Surrogate keys: better support for some conditional properties.

## CONDITIONAL INFORMATION

Last month in Part III, I introduced the term "conditional information" as a catch-all phrase for various types of missing information encountered during data entry. In addition to handling "data entry nulls," designers may have to take into account metadata about missing information captured in legacy or nonrelational databases. Given SQL's implementation of undifferentiated nulls, this metadata is often lost when the nonrelational database is migrated to a relational one. On one hand, data entry nulls were found in Part III to reflect conditional properties, constraints, relations, and so on, depending on the null's type. These causes are addressed separately in subsequent sections. On the other hand, data entry nulls also reflect an attempt to record information about the data entry process.

The classifications of missing information into "value not applicable," "value undefined," "value temporarily unknown," and so on, serve to record information about why the data is missing; they give information about the data (or metadata). For example, "value temporarily unknown" tells us that the data entry operator believes a value exists that is only temporarily unavailable. Until the value is actually obtained, it remains possible that the data entry operator is incorrect in this belief; the value may not be applicable or may not even exist. Similar comments hold with respect to the other meanings of null. Let's examine why we might want such information.

We might, for example, want to know the data entry operator's opinion about missing information so that we can, at a later time, locate the instances of "value temporarily unknown." We could then actively seek the real value to replace these unknowns. A similar reasoning would apply to instances of "value not supplied" and "value rejected." However, this activity is distinct from the activity of modeling the state of our knowledge about the entities represented in a logical design. It is the task of modeling our knowledge about the data collection and entry process.

Distinguishing between data about the application and "data about the data" (that is, metadata) in this way makes it easier to understand how we can improve our efforts. We can model the data and metadata more carefully, thereby obtaining more utility than possible using nulls and many-valued logic. To illustrate this point trivially, note that we often keep track of metadata (such as exceptions and errors) in application processing. Why should we not do so in the database as well? And, just as

ARTWORK BY: BARTON STABLER

**EMP**

| E_ID | FNAME | LNAME |
|---|---|---|
| 1 | MINNIE | MOUSE |
| 2 | BILL | CLINTON |
| 3 | DAFFY | DUCK |
| 4 | HILLARY | CLINTON |

**META_DATA**

| TABLE | PK_VALUE | COLUMN | STATUS |
|---|---|---|---|
| EMP | 2 | SPOUSE | TEMP. UNK. |
| EMP | 3 | SPOUSE | N/A |

**FIGURE 1.** *Keeping metadata in lookup tables.*

exception processing should be separated from normal processing, such information should be maintained in separate "exception tables" and not interspersed with unexceptional production data.

A system catalog is an example of metadata, but generally is not related to the specifics of data entry. The metadata we are concerned with is most commonly encountered in audit trails; we can capture the data's source, the data entry operator, time of entry, and so on. Extending this concept to capture metadata about missing information seems natural. To accomplish this task, we must capture the table identifier, the primary key of the row involved, a code classifying the data entry operator's belief, and an identifier for the attribute involved (but not a column value). This metadata can be maintained in separate lookup tables (see Figure 1). In order to implement this approach properly, we will also have to support subtypes I will describe in the "Types and Subtypes" section. While this solution can be implemented manually, shouldn't RDBMS products support audit trails in which the information to be captured is user-definable?

You might complain that this solution constitutes planning for additional information about every value in the database. In reality, however, rows in which nulls fulfill a functional purpose are the exception when the database is designed along the principles outlined in this article. Few database administrators actually scan their databases for nulls, attempting to replace them with real values. Likewise, few applications provide any meaningful information about nulls to data entry operators; a field containing a null in the database is usually displayed as empty or some other noninformaive value, such as the string "NULL." No information is conveyed as to the database null's type.

However, make a note that if RDBMS products supported differentiated nulls as metadata (that is, nulls affected neither DBMS logic nor the user's view of data), something much like the current physical implementation of nulls could be used to track metadata about missing information. For users who already have tables containing nulls, I recommend creating views that implement the database design principles I will describe. These views will not contain nulls and should be used for all data manipulation. Then, if you wanted to obtain metadata, you could access the base tables containing nulls (see Figure 2).

## CONDITIONAL RELATIONSHIPS

Recall from Part III that conditional relations among entities are those in which the entities do not always participate; that is, some instances of at least one of the entities do not obey the relation. In general, these relationships are characterized as *0/m:0/n* relations, with *0/m:n*, *0/1:n*, and so on, being special cases. A common approach to modeling conditional relations involves using foreign keys. Where the instance does not participate, the foreign key is then set to null. This approach treats all instances uniformly, imposing a table structure that represents noninformation for some rows and violates the Knowledge Principle.

For example, the traditional employees table EMP often contains a foreign key M_ID to its primary key E_ID, identifying the employee's manager (Figure 3). Although we often think of every employee as having a manager, obviously this is not true. At least one employee will not have a manager, which is often modeled by inserting a null in place of a proper foreign key value (shaded row in Figure 3).

The solution is to model all the conditional relations using association tables. An association table is one in which the entire primary key consists of foreign keys to other tables, thereby capturing an association among two or more entity instances. By creating an association table, EMP_MGR, consisting of E_ID and M_ID, we can remove the M_ID foreign key from the original employee table (see Figure 4). With this design, no row in any table need have any nulls in the foreign key columns. Instead, a particular employee's lack of a manager is modeled correctly as the lack of a row in the association table.

This procedure also breaks referential cycles, thereby eliminating the various performance and design problems associated with them.[1] Likewise, modeling relationships via an association table is often easier for users to understand and control. For example, if we wish to represent a *0/1:4* rela-

**EMP**

| E_ID | FNAME | LNAME | SPOUSE |
|---|---|---|---|
| 1 | MINNIE | MOUSE | MICKEY |
| 2 | BILL | CLINTON | <null> |
| 3 | DAFFY | DUCK | <null> |
| 4 | HILLARY | CLINTON | <null> |

RESTRICT/PROJECT TO →

**EMP_1**

| E_ID | FNAME | LNAME | SPOUSE |
|---|---|---|---|
| 1 | MINNIE | MOUSE | MICKEY |

**EMP_2**

| E_ID | FNAME | LNAME |
|---|---|---|
| 2 | BILL | CLINTON |
| 3 | DAFFY | DUCK |
| 4 | HILLARY | CLINTON |

**FIGURE 2.** *Projection views on null-bearing tables.*

**EMP**

| E_ID | FNAME | LNAME | M_ID |
|---|---|---|---|
| 1 | MINNIE | MOUSE | 2 |
| 2 | BILL | CLINTON | 4 |
| 3 | DAFFY | DUCK | 2 |
| 4 | HILLARY | CLINTON | <null> |

**FIGURE 3.** *EMP table with nulls.*

tionship, the cardinality constraint can be placed on the association table in a rather straightforward manner (assuming the DBMS supports multirow constraints with aggregate functions):

```
(NOT EXIST (SELECT MGR_ID, COUNT(EMP_ID)
   FROM EMP_MGR HAVING COUNT (EMP_ID)
   <> 4 GROUP BY MGR_ID)
```

Because association tables eliminate so many anomalies (such as "special case" handling of referential cycles, difficult integrity constraint enforcement, and restrictions on constraint definition) performance often improves when association tables replace the embedded foreign key approach. Certainly tuning becomes easier: Since access to all tables is by primary key, we need not be concerned about foreign key index creation and maintenance. The optimizer may become more efficient and predictable as well. Of course, performance losses are a potential disadvantage that must be addressed. By using simple surrogate keys (especially integers) in place of nonsimple primary keys, we can keep association tables quite small, fully indexed, and often cached. Note also that the foreign key index in the original EMP table can now be dropped, improving table update performance.

Indexing an association table improves performance, but requires some care. Several options exist. A single B-tree index on the primary key may suffice. This option requires that the leading index keys be specified in queries so they are known in most lookups. A small number of B-tree indexes with different index key orders may be appropriate otherwise. If the DBMS vendor supports index-only join strategies, separate indexes on each of the foreign keys will permit either key to be unknown, and the additional index access would represent little additional overhead. Ideally, DBMS vendors would support cross-table indexes for this purpose. These indexes could sometimes implement the association table itself and could even be a better implementation of foreign keys (for example, Computer Associates' CA-DB uses this method for managing referential integrity).

Although it is certainly possible to use foreign keys for *1:m* relationships, it seems more reasonable to use association tables uniformly in logical design and treat the more traditional approach as a performance optimization. Such a uniform approach to representing relationships could greatly improve database understanding and eliminate many anomalies that are due to the standard foreign key approach. With proper support for association tables in RDBMSs, any conceivable performance penalties for their use will most likely be eliminated. In the meantime, you may be surprised to find that they actually improve performance.

## CONDITIONAL ENTITIES

Even if we create an association table in place of a traditional foreign key implementation of a conditional relationship, we may possibly be overlooking the source of other problems. The very existence of a null in place of a value for a self-referential foreign key (for example, the shaded row of Figure 3) makes it clear that rows represent two distinct kinds of entities. Indeed, they have very different relation predicates because they have different defining properties. I refer to entities with multiple relation predicates as conditional entities: We might say that every employee is conditionally also a manager.

Therefore, in the traditional employees table, each row potentially represents two rather different kinds of employees since some employees are managers and each manager is probably (though not necessarily) an employee. This fact becomes immediately apparent if we recognize that managers have defining properties not shared with nonmanager employees. The columns representing such attributes are often assigned nulls for rows representing nonmanager employees. Nonmanagerial employees are not the same kind of entity as managers. (As if we didn't already know that!)

Instead of trying to capture the employee and manager entities in a single table, they should be modeled with two distinct tables: an employees table, EMP, with primary key E_ID; and a managers table, MGR, with primary key M_ID. Anything else violates an important design principle: Each table should have a single relation predicate specifying the represented entity's defining properties.[2] An association table representing the relationship between them can then be established, having a primary key composed of two foreign keys, E_ID and M_ID.

## CONDITIONAL PROPERTIES

Whenever a value in a non-key base table column is optional (that is, the database designer permits it to be null), the column represents a conditional property or meaning criterion. Such columns indicate

**EMP**

| E_ID | FNAME | LNAME |
|---|---|---|
| 1 | MINNIE | MOUSE |
| 2 | BILL | CLINTON |
| 3 | DAFFY | DUCK |
| 4 | HILLARY | CLINTON |

+

**EMP_MGR**

| E_ID | M_ID |
|---|---|
| 1 | 2 |
| 2 | 4 |
| 3 | 2 |

**FIGURE 4.** *EMP_MGR association table.*

**VEHICLES**

| VIN | MAKE | MODEL | COLOR |
|---|---|---|---|
| 1 | FORD | ESCORT | GREEN |
| 2 | PONTIAC | GRAND | RED |
| 3 | PORSCHE | PRIX | <null> |
| 4 | CHRYSLE | CARRERA | <null> |

**FIGURE 5.** *Vehicles multitable.*

**(A) PAINTED_VEHICLES**

| VIN | MAKE | MODEL | COLOR |
|---|---|---|---|
| 1 | FORD | ESCORT | GREEN |
| 2 | PONTIAC | GRAND PRIX | RED |

**(B) UNPAINTED_VEHICLES**

| VIN | MAKE | MODEL |
|---|---|---|
| 1 | PORSCHE | CARRERA |
| 2 | CHRYSLER | LEBARON |

**FIGURE 6.** *Differentiating tables by relation predicates.*

that multiple entity types are being represented in a single table. Each of these entities should be given a separate table—they have distinct relation predicates.

For example, if a table describing vehicles contains a column describing the color of paint on the vehicle, this column will be null for vehicles that have not been painted (Figure 5). In the case of painted vehicles, the relation predicate is similar to "vehicles identified by vehicle identification number (VIN) have paint color COLOR(VIN) and other properties P(VIN)." By contrast, the rows representing vehicles that have not been painted have a relation predicate such as "vehicles identified by VIN have properties P(VIN)."

Following the simple design principle that entities are differentiated by their relation predicates leads to the solution depicted in Figure 6. Of course, entities with large numbers of meaning criteria might cause the database design to become quite complicated, with the design principle leading to a combinatorial explosion of new tables. However, in my experience, few such tables exist, and the gains of creating additional tables far outweigh the inconvenience. This concern would be properly addressed through DBMS support for types and subtypes, which I will describe in the "Types and Subtypes" section.

## CONDITIONAL OPERATORS

Even if the relational operands do not contain nulls, the result of a conditional operator may create nulls. As with base tables, the appearance of nulls in a derived table is an indication that it represents multiple entities. Each of these entities has a distinct relation predicate.

For example, take the outer join of the employees table with itself on the M_ID foreign key (see Figure 7). Most rows in the result table contain information about the employee followed by information about the employee's manager (probably also an employee). These rows have a relation predicate similar to "the employee identified by E_ID has properties P(E_ID) and manager identified by M_ID with properties P(M_ID)." By contrast, rows for which there is no manager have a relation predicate similar to "the employee identified by E_ID has properties P(E_ID) and such that a manager identified by M_ID with properties P(M_ID) for that employee does not exist."

Conditional operations can and perhaps should be simulated as the union of the results of multiple ordinary SELECTs—one for each unique relation predicate. Of course, this approach can become quite cumbersome if more than a few relation predicates are involved in the desired result, which is one of the reasons why users are asking for outer join support in SQL. Unfortunately, it is incorrect since the multiple entities in the result are presented as though they were the same type.

The manual solution is to query each entity type separately, thus being deliberately cognizant of the differences in meanings of the rows. A more automatic solution would involve true support for set operations, as contrasted with the restricted versions implemented by the relational algebra. For example, a set union permits arbitrary sets as operands, while the relational union requires that all the sets be type compatible. When a set union is performed, the result's type is not the same as either operands' type (it is a supertype). Thus, the union of a "set of apples" and a "set of oranges" is a "set of apples and oranges," or perhaps a "set of fruit." As with conditional properties, this capability would be properly addressed through DBMS support for types and subtypes—which I will describe next.

## TYPES AND SUBTYPES

Each table should have a unique relation predicate, thereby representing one and only one entity. As noted previously, a table containing nulls indicates multiple relation predicates and, logically, multiple tables. These tables should be broken out by successively restricting the table and projecting away the null-bearing columns.[3] The various tables that can be created by this kind of projection are each supertypes of the source table. For example (see Figure 6a), vehicles not differentiated by paint color (because paint color is

**USUAL_RESULT**

| E_ID | M_ID | MGR_LNAM |
|---|---|---|
| 1 | 2 | CLINTON |
| 2 | 4 | CLINTON |
| 3 | 2 | CLINTON |
| 4 | <null> | <null> |

VERSUS

**RESULT_TABLE_1**

| E_ID | M_ID | MGR_LNAM |
|---|---|---|
| 1 | 2 | CLINTON |
| 2 | 4 | CLINTON |
| 3 | 2 | CLINTON |

\+

**RESULT_TABLE_2**

| E_ID |
|---|
| 4 |

**FIGURE 7.** *Multitable result of an outer join.*

not a relevant property) are more general than painted vehicles (Figure 6b). Note that, unlike ordinary projection, projecting away a null-bearing column does not append an EXISTS to the relation predicate: It is meaningless to refer to "VIN where some color COLOR(VIN) exists in the domain of colors" if color is not a relevant property for this vehicle.

Ideally, DBMS vendors would support supertypes, types, and subtypes directly. It should not be necessary to create supertypes explicitly by user-written projection operations: The lack of a value for a property should automatically imply an appropriate modification of the relation predicate. If the DBMS kept track of these supertypes automatically, it could present multiple relations to the user with appropriate identification of the relation predicate on demand. Furthermore, conditional operators would then be understood as operations on multiple relationships (masquerading as single relationships) and having a multiple entity result (again masquerading as a single entity).

The possibility of multiple entity results suggests extensions to the relational model to support more general versions of the relational set operators. In particular, relational union is a restricted version of set union. I propose that the system should automatically create multiple tables in the output, grouping like rows together by performing the "restrict and project away nulls" operation for users. This capability would help users distinguish between the entity types and recognize their interrelationships. In effect, such set operations would be the multitable result versions of existing relational operations, which users must simulate manually today using multiple SQL statements. Whether multitable operands should be permitted deserves additional and careful consideration. For the time being, the operation should be reserved for output.

Whereas the relational model per se does not permit the union of two distinct types (such as apples and oranges), the outer union does, as does my proposal. The difference is that outer union creates a bizarre kind of combined entity in which every apple is given orange properties and every orange is given apple properties. My proposal creates the expected collection of types in which apples and oranges retain their identity and the user is not encouraged to misidentify the types of the resulting entities. Similar remarks apply to the extension of the other set of operations.

Some readers might be concerned regarding complexity that is due to the numbers of tables in a result collection. However, such complexity is at once bounded and unlikely to be very great in the majority of queries. In most cases, few entities are involved in a query result: Even the more complex outer joins usually produce fewer than four entities. Certainly the complexity is no greater than what already exists in processing with nulls since, essentially, it is only the presentation of results that changes! Users need not be burdened with attempting to collect rows having nulls in the same columns, and understanding the result is more straightforward.

**STOCK_SALES**

| STOCK_ID | BUYER | SELL_PRICE | SELL_DATE | SHARES |
|---|---|---|---|---|
| 1 | ROCKEFELLER | 100.00 | 10/12/91 | 5000 |
| 2 | HEARST | 400.00 | 12/01/93 | 10000 |

**STOCK_BUYS**

| STOCK_ID | SELLER | BUY_PRICE | BUY_DATE | SHARES |
|---|---|---|---|---|
| 1 | J.P. MORGAN | 100.00 | 10/12/91 | 5000 |

**PENDING_STOCK_BUY**

| STOCK_ID | STATUS | SHARES |
|---|---|---|
| 2 | SOLD_SHORT | 10000 |

**FIGURE 8.** *Satisfying the selling short conditional constraint.*

## CONDITIONAL CONSTRAINTS

Conditional constraints, such as those implied by "selling short" (see Part III), involve an anticipated ability to comply with some constraint at some time in the future. Unlike normal constraints, it may be hard to specify the conditions that spell out when conditional constraints are to be satisfied. In a sense, conditional constraints represent an attempt to enforce the database's desired states, whereas ordinary constraints represent enforcement of the database's necessary states. Conditional constraints require special design considerations if the common solution involving nulls (for example, creating an entry in a STOCK_BUYS table with nulls in place of attribute values) is to be avoided.

The solution is to create status or bookkeeping tables. The tables represent the abstract instances that would "balance the books." For example, consider a simplified version of selling short. As noted in Part III, the requirement that stock sales balance stock purchases in any transaction is a conditional constraint. In place of a special entry in the STOCK_BUYS tables, we would create a PENDING_STOCK_BUYS table. For each instance of selling short, we would insert a row in this table. The table would contain those columns that would constrain any future purchase and satisfy the conditional constraint. It is sometimes useful to include a general STATUS column in the table as well. The conditional constraint would then be rewritten to examine the STOCK_SALES, STOCK_BUYS, and PENDING_STOCK_BUYS tables (Figure 8).

## ABSTRACT ENTITIES

Some occurrences of nulls are due to a perceived lack of an appropriate entity. This problem arises when the database designer insists on modeling only concrete entities when, in fact, the business process relates to concrete and abstract entities. In fact, the problem is most prevalent when some instances of an entity are concrete and some are abstract. Suppose that every employee is supposed to be as-

signed to a department so that the EMP table contains a DEPT# column (ignoring for the moment that this might be a foreign key). However, from time to time a new hire may be entered into the employees table before being assigned to a department. Typically, this problem is handled by assigning a null to DEPT#.

A bit of careful thought will show that this problem is frequently an artifact of simplistic conceptual design. Although the employee has not been explicitly assigned to any department, some aspect of the company is likely to perform the function of a department for the employee nonetheless. You can discover this abstract instance's identity by analyzing the business function of departments and the business processes that affect employees.

For example, a department may serve as the collective unit for a group of employees. Typically, the manager of an employee's department has hiring and firing authority and may sign off on paychecks. Sometimes the department determines the physical location where the employee is to report at the beginning of work. The human resources department might be one entity that meets these qualifications for the unassigned employee, but as an abstract instance distinct from the concrete instance to which the human resources personnel are assigned. The trick is to assign the abstract entity instance a unique department identifier and place it in the appropriate domain.

## SURROGATE KEYS

In Part III, I pointed out that sometimes conditional information involves the primary key. When it is possible for a data entry operator to fail to enter a value for any presumed "candidate key," no primary key can be selected without violating the entity integrity rule. This problem can be solved by creating an artificial identifier, called a surrogate key, to serve as the primary key.

For example, the EMP table originally contained FNAME and LNAME columns (first name and last name), but not E_ID. It is possible that an employee is always identified

**We can eliminate the need to put nothing in our databases**

uniquely by the combination of a first and last name, but sometimes the first or last name is missing. If this case occurs, creating a system-assigned E_ID eliminates the problem by converting FNAME and LNAME to conditional properties.

## MEANINGFUL DEFAULTS

As noted in Part III, Date has recommended a systematic use of defaults as being better than using nulls.[4] While my article should make it quite clear that I believe better logical design would eliminate nulls, I also believe that systematic use of defaults is an essential part of the solution and agree with much of Date's proposal. Although the details are much too involved to expound here, Date recommends (among other things), the following:

- ☐ DBMSs should support the declaration of user-defined defaults for columns.
- ☐ Various built-in functions should be defined to support defaults, including IS_DEFAULT (returning TRUE if and only if its column value operand contains the default value defined for the column), and DEFAULT (returning the defined default value of its column name operand).

I would add the following caveats and embellishments to this proposal:

- ☐ Nulls should never be used in place of default values.
- ☐ Defaults should always be meaningful values from the domain. That is, they should not be artificially created values (including such values as "N/A" or "UNKNOWN"). Otherwise, default values no longer convey any knowledge about the entities to which they apply, which violates the Knowledge Principle.
- ☐ Default values should be defined only in the system catalog and not stored in the table itself. This approach requires the ability to mark a column as having the default value. Then, if the system administrator or DBA chooses to change the default for a column, the table need not be rewritten.
- ☐ It should be possible to force "marked as default" columns to take on the value specified as the default and be unmarked (this operation is trivial in SQL).
- ☐ During data entry, users (including data entry operators) should be required to distinguish between an entry of a default value and entry of a value that happens to be the same as the declared default. For example, it must be possible to distinguish between a value of "BLUE" and a default that happens to be set to "BLUE."
- ☐ We should distinguish between data entry defaults and database defaults. Data entry defaults are usually associated with a particular application and quite possibly a particular data entry operator or even a session. Their purpose is to save the data entry operator time and effort by "pre-entering" a field's most probable value. By contrast, the purpose of database default values is to reflect the "standard" value of an attribute for all possible instances. For example, if we had a table containing information about rooms, we might want to set the database default ceiling color to white, under the hypothesis that this color was the most likely of any ceiling's color. If a particular data entry operator were entering information about rooms in a particular building that tended to have light gray ceilings, however, the data entry default for ceiling color would be different from the database default and would most likely be set by the data entry operator and maintained in the application. In fact, it is possible to have an entire hierarchy of default values, but only the one at the "root" of the hierarchy corresponds to the database default value.

While database default values can and should be utilized to represent our "best guess" about missing information, they should not be used for conditional properties, relationships, constraints, or operations. In such cases, the assertion of a value where none exists conveys false information

about the universe of discourse.

## CONCLUSIONS

By capturing missing information structurally and information about information as the separate class it is, we eliminate the need to put nothing in our databases. Between the various design methods and the new presentation of conditional operators outlined in this article, it would seem that no need exists to place (or permit the automatic creation of) nulls in a database. As a result, there is no need to support many-valued logic in DBMSs.

These changes have a number of positive influences:

- ☐ The DBMS can rely on classical logic, and in doing so, meet the goals we established for a DBMS.
- ☐ Performance can improve since the entire power of classical logic can be brought to bear on the problem of optimization, and because the many anomalies due to the existence of nulls do not have to be treated.
- ☐ Integrity can be enforced more directly without having to worry about restrictions (such as disallowing cascade delete on a self-referencing table).
- ☐ The database's structure will be easier to understand and, therefore, queries easier to write.
- ☐ The results of relational operations will be more consistent with expectations and easier to interpret.
- ☐ Meanings of tables (base and derived) will not depend on whether or not nulls could or actually do exist in the database.
- ☐ Without the need for null support, the SQL standard could be simplified, as could several applications.

In this series of articles we have (1) reviewed the motivations for using a formal logical system as a foundation for a DBMS; (2) translated certain important properties of logical systems into database terminology, thereby motivating the need for database folks to understand logic; (3) examined the use of many-valued logics as a formal foundation of DBMSs and found them wanting; (4) reviewed the key reasons that users and database designers seem to require DBMS support of a many-valued logic; and (5) provided a set of alternatives that meet user requirements without doing violence to the logical properties we desire from a DBMS. While most of these alternatives can be implemented through careful database design and use, a couple require additional support by vendors. In particular, vendors must improve support for default values and add support for types and subtypes, domains, proper set operations (union, intersection, difference, and so on),[5] and relation predicates. ■

*I would like to thank Chris Date and Hugh Darwen for their helpful comments and criticisms on this series. I would also like to apologize to Billy Preston (again) and Eric Clapton for the abuse of their song titles.*

## NOTES AND REFERENCES

1. Date, C. J. "A Note on One-to-One Relationships," in *Relational Database: Writings 1985-1989*, Addison-Wesley, 1990.
2. A relation predicate states all the necessary conditions for row membership in a table that is independent of other tables. There is much to be said about the definition and importance of relation predicates, a problem I address elsewhere.
3. I recently became aware of a similar treatment by Codd [7], which he refers to as "Conceptual Normal Form." Codd uses projection to create supertypes, eliminating all but some arbitrarily determined percentage of nulls. By contrast, I treat all null-bearing rows as evidence of multiple relation predicates in the table and, therefore, an indication of bad table design.
4. Date, C. J. "The Default Values Approach to Missing Information" in *Relational Database: Writings 1989-1991*, Addison-Wesley, 1992.
5. The relational model will have to be extended to handle collections of tables in single operations, while guaranteeing that results are the same as from some sequence of relational operations. This is potentially a big job and should be tackled cautiously.
6. Codd introduced association tables, surrogate keys, and types/subtypes [8].
7. Codd, E. F. "A Practical Approach to Combining Two or More Relational Databases," *The Relational Journal*, 2(3), June/July 1990.
8. Codd, E. F. "Extending the Database Relational Model to Capture More Meaning," reprinted in *Readings in Database Systems*, M. Stonebraker, ed., Morgan Kaufmann, 1988.

**David McGoveran is president of Alternative Technologies (Boulder Creek, California), a relational database consulting firm founded in 1976. He has authored numerous technical articles and is also the publisher of the "Database Product Evaluation Report Series."**


# Database Benchmark

## Neal Nelson has a new database benchmark.

You can learn which DBMS is fastest by running the database benchmark on a given machine with several different database packages.

You can find out which computer is best for you by running the database benchmark with your preferred DBMS on several different machines.

You can see how much the tuning process helps by running the database benchmark before and after hardware or software tuning.

You can measure how a system degrades under load by running the benchmark at gradually increasing user load levels.

You can run the benchmark on a standalone basis or as part of a mix with other Neal Nelson remote terminal emulation scripts that perform word processing, spreadsheet, electronic mail, statistical, program development, and Unix tasks.

This new benchmark performs inserts, updates, deletes, updates to key fields, sequential queries, range queries, nested queries, wild card queries, lists, 2, 4, 6 and 10 table joins, aggregates and sorts.

The Neal Nelson database benchmark is written in the Structured Query Language and can be run without modification for Sybase, Oracle, Informix and other SQL databases.

Learn the whole story about database performance by contacting:

NEAL NELSON & ASSOCIATES

330 North Wabash, Chicago, Illinois 60611 Telephone (312) 755-1000

**CIRCLE 15 ON READER SERVICE CARD**