VOL. 7 NO. 2

DATABASE Programming & Design

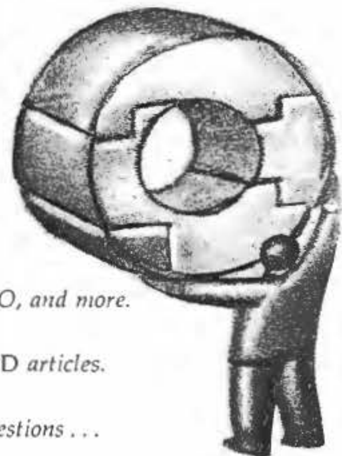
FEBRUARY 1994

Why We Need Object-Oriented Systems | **26** | CHARLES F. BOWMAN
Slow to success, OO database could catch fire as software systems reach greater complexity.

24 Ways to Improve Database Performance | **32** | PEGGY VIEHMAN
DB2 database performance keeping you up at night? Here's what you can do to improve it.

Can't Lose What You Never Had | **42** | DAVID MCGOVERAN
"Nothing From Nothing" continues with the myriad multivalued logic solutions for nulls.

Data Mining: Tapping the Mother Lode | **50** | LISA LEWINSON
Buried in historical data is critical information; a new wave of tools is trying to unearth it.



DEPARTMENTS

EDITOR'S BUFFER	7	*Taler De database? Ya; DB2, OO, and more.*
ACCESS PATH	11	*Readers respond to recent* DBPD *articles.*
DATABASE DESIGN	13	*When questions create more questions . . .*
ACCORDING TO DATE	19	*Answers to C. J. Date's recent puzzlers.*
CLIENT/SERVER FORUM	23	*DBMS integrity: Therein lies the rub.*
DESKTOP DATABASE	58	*Reviewing 1993's PC database developments.*
ENTERPRISE VIEW	60	*How Access and Paradox fare for modeling.*
SQL UPDATE	67	*SQL Access Group and the CLI wars.*
PRODUCT WATCH	70	*Never a lull in the database market.*

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. Canadian/Mexican orders must be prepaid in U.S. funds with additional postage at $6 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

Null support in RDBMS applications implies multivalued logic support—with all its attendant problems. What are designers really looking for?

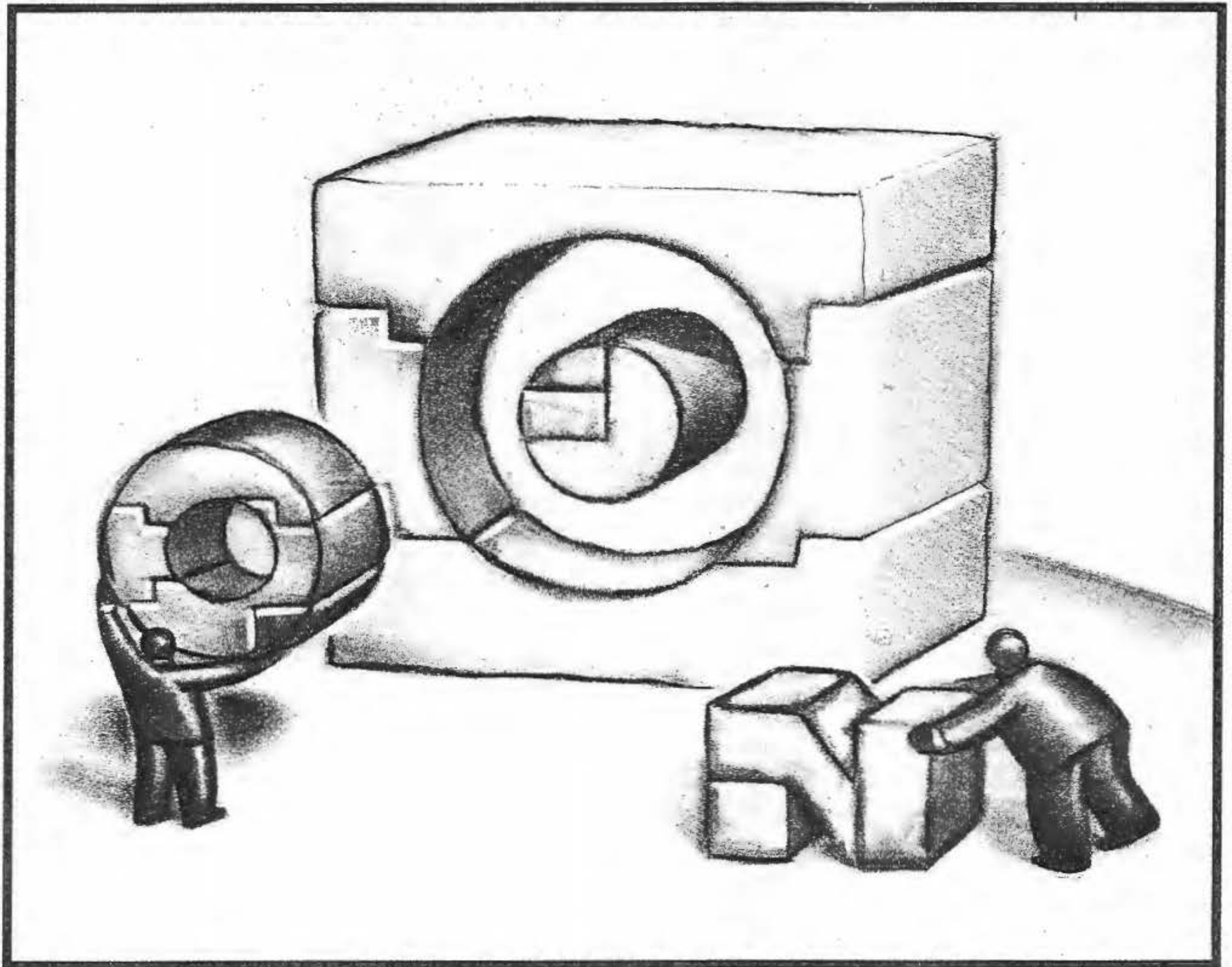
Nothing from Nothing Part III: Can't Lose What You Never Had

PROFESSIONAL DBMS users play a variety of roles: database administrator, database designer, application developer, and end user. When using a DBMS, I doubt any consider whether or not the DBMS supports a many-valued logic. (Indeed, if the products themselves are any evidence, I doubt DBMS makers examine this issue either.) Professional DBMS users often question whether the DBMS and database design support nulls, however. For most users, "SQL nulls" have become a catchall means of addressing a variety of problems. Even end users, who used to be interested in application support of "don't know" or "not applicable" responses during data entry, now treat this issue as a question of null support.

But why exactly are professional users requiring "null support" even if—though they may not realize it—this requirement implies that they want support for many-valued logics? Last month in Part II, we considered why many-valued logics were inappropriate as a DBMS's foundation. In Part III this month, we will examine the key reasons database designers and users find themselves wanting the support of a many-valued logic vis-a-vis null support.

For this analysis to make any sense, it is important to review our understanding of a database's semantics. First, a database design implements a model or representation of some portion of the world of experience called the "universe of discourse." It defines the permissible facts that can be represented, of which only some are made "active" by storing data in rows in relational tables. For these permissible facts *not* represented by rows in the database, the closed-world assumption permits us to say that they are "false."

Careful consideration of your own database application will demonstrate that the existence or absence of a row in the database represents a statement about your knowledge of the application domain. Specifically, a row R in a ta-



ble representing a predicate *P(x)* means that "we know that *P(R)* is true," while its absence means "we know that *P(R)* is false." Under the closed-world assumption, an impermissible set of column values (that is, those values not within the defined universe of discourse) cannot be used to form a predicate: the result would not be a well-formed formula.

Given this understanding of database semantics, it is my position that every appearance of a null in a database represents some form of *conditioned knowledge.* By conditioned knowledge, I mean that a precise expression of knowledge (such as the choice of a particular value in the place of a null) is determined by some condition that cannot be satisfied. For example, when we permit a data entry field to be "not applicable," the mere appearance of this field on the screen could be conditioned on whether or not a value is applicable; that is, some condition must exist that, if satisfied, would determine whether the data entry operator would be asked for a field value.

We have long known that some facts about the world are conditional, in the sense that "some, but not all, instances of *x* have property *P*." We understand conditional relationships, such as "most instances of *x* have relationship *R(x, y)* to *y* for some instances of *y*." Indeed, it was such considerations that led to the introduction of the so-called necessity and possibility quantifiers, thus creating modal logic. In the remainder of this article, we will examine various kinds of conditioned knowledge that lead to the appearance of nulls in the database.

I will first examine the types of conditioned knowledge that encourage database designers to specify columns as permitting nulls, including:

- ☐ Conditional relationships
- ☐ Conditional properties
- ☐ Conditional operations
- ☐ Conditional constraints.

I will then examine the various types of nulls that data entry

ARTWORK BY: BARTON STABLER

operators might invoke (called "conditional information" here), and relate these nulls to the kinds of conditionality handled by database designers. In the absence of maliciousness, carelessness, or ignorance (as due to, for example, lack of training), these situations, inclusively, account for the appearance of nulls in a database and, therefore, for the perceived need for many-valued logics.

CONDITIONAL RELATIONSHIPS

We can characterize a relationship among types of entities in many ways. The number of entities involved in the relationship is called its *degree*. When the relationship is not computed by a function or transformation, it is usually specified as a simple mapping among entities of the types. Mappings are often characterized by the ratio of the numbers (the cardinalities) of each entity participating in the relationship. For example, a mapping of degree two is often characterized via the notation *n:m*, which means that *n* entities of one type have the designated relationship to *m* entities of a second type. Note that the notion of a *n:0*, *0:m*, or *0:0* relationship conveys no positive information since it says that the relation holds for *no* entities of one of the types.

Perhaps the mapping most familiar to relational database designers is the "parent-child" type of relationship or, more precisely, the *1:m* (one to one-or-more [or many]) relationship. For each instance of the "parent" entity, zero or more instances of the child entity exist. Such a relationship is generally modeled via a foreign key in each row of the child relation having the value of the primary key of the corresponding row in the parent relation. By coincidence, this technique also works for the *1:0/m* (one to zero-or-more) relationship (the *1:1* and *1:0/1* relationships are special cases of the *1:0/m* relationship). Of course, it is a little strange to think of a "parent" who has no "children": How can something be a parent by virtue of a relationship to nonexistent children?

Suppose that not every child instance has a corresponding parent, which is the *0/1:0/m* (meaning zero-or-one to zero-or-more) relationship. We will refer to such relationships as "conditional" because it represents a situation in which not all instances of any one of the involved entities are related to some instance of the other entity. For example, consider the case of real children generally. Not all children have identifiable parents: it is an unfortunate fact that the parents of some children are forever unknown due to the inhumanities of wars, kidnappings, or other causes. Similarly, it is not uncommon to find a conditional relationship modeled using the foreign key approach, with the special case of zero references (as in the child that has no parent) being modeled by entering a null in place of a foreign key value.

Nulls represent some form of conditioned knowledge

CONDITIONAL PROPERTIES

In logic, an entity type (or class) is said to have defining properties and meaning criteria.[1] A candidate instance must satisfy all the defining properties to be of the entity type. By contrast, the candidate might satisfy only some of the meaning criteria: Any individual meaning criterion counts only as evidence that the candidate is of the entity type. However, the exhaustive disjunction of all meaning criteria is a defining property.

For example, one defining property of the insect entity type is that it must have six legs. However, having wings is a meaning criterion: Some insects have wings and others do not. As another example, the definition of a poem abounds with meaning criteria: some poems have meter, others rhyme, still others may use metaphor, and so on. Conditional properties are often modeled by permitting nulls in the columns representing the meaning criteria. Thus, a table containing descriptions of insects might have a column characterizing wing types that would be set to null if the particular insect did not have wings. Similarly, a table about poems might have a column to specify the poem's meter that would be set to null if the poem had no meter. By extension, a compound foreign key may be entered as partially null when a part of the referenced primary key is, in fact, a meaning criteria.

CONDITIONAL OPERATIONS

Various operations supported in relational DBMS products operate on multiple types of operands. For example, whereas the join operates on two relations that are related in a particular manner, the outer join operates on two conditionally related relations. Similarly, the outer union operates on two relations that only conditionally satisfy the union compatibility relationship. In a sense, then, the operands of these operators are conditionally defined. I will refer to such operations as *conditional operations*.

Because the relationship among operands is not uniform for all instances of the operands, the result of a conditional operator is not a uniquely defined relation. Specifically, the result of an outer join or outer union does not, in general, have a unique relation predicate. Instead, it is a collection of possible relations: one relation for rows without nulls, plus one for each extant combination of columns containing nulls.

Suppose we had an employees table EMP and a managers table MGR with primary keys E_ID and M_ID, respectively, both from the same domain. An outer (equi-) join of these tables on E_ID and M_ID and returning E_ID and M_ID would typically return at least two tables: one having the relation predicate "employee E_ID with properties P(E_ID) managed by manager M_ID" and one with the simpler re-

lation predicate "employee E_ID with properties P(E_ID)." Since this situation cannot be directly modeled in the relational algebra, the various result relations are made uniform by creating extra columns containing nulls. These nulls are of the type "value is the empty set."

CONDITIONAL CONSTRAINTS

For some constraints, the time at which they must be satisfied cannot be stated in advance; they are neither at statement completion time (immediate) or commit time (deferred). As such, they are neither state nor transition constraints. For example, an investor "selling short" implies a commitment to buy the sold stock at some future, unspecified time. As such, selling short involves a conditional integrity constraint (balancing the amount of stock "sold short" with the amount purchased) that is satisfied at a time dependent on any event or other condition.

Because conditional constraints imply the existence of entity instances that would satisfy the required constraint, these instances are sometimes modeled by including special entries in the database in advance of the time at which the constraint is satisfied in reality. This approach creates entity instances for which the values of certain properties cannot be known. The missing information is often modeled with nulls, to be replaced at some later time with values. In the interim, the conditional constraint is satisfied by programming it to accept either real values or nulls.

For example, a stock trade might be modeled by a transaction that inserts into a STOCK_SALES table a row containing the stock identifier, its sell price, date, and recipient, and into a STOCK_BUYS table the stock identifier, buy price, date, and seller. Selling short would then insert the appropriate sell information, but would insert a row for the stock identifier in the STOCK_BUYS table, setting the buy price, date, and seller columns to null until some later time. The constraint would be written to accept the existence of such a row, effectively deferring the real constraint check indefinitely.

What are users trying to convey when they use nulls?

CAPTURING NOTHING: CONDITIONAL INFORMATION

Having considered the various data modeling issues that contribute to nulls, we are left with situations in which the database designer must anticipate incomplete data entry. What are end users trying to convey when they enter a null into a field during data entry? C. J. Date has listed the more common possible meanings that may be attributed to a null.[2] Say we had a DBMS capable of distinguishing all these flavors of null. Let's examine the use of each in turn.

Value not applicable. For example, a data entry form may contain fields for an employee's name and spouse's name. If the employee has no spouse, the user may enter "N/A" for "not applicable" or may simply skip the field, leaving it empty. The program, in turn, may enter the spouse's name into the database as a null. However, some thought shows that no entry into the database should have been made unless an integrity constraint requires employees to have spouses. In that case, the spouse's name would be required; failure to enter a value from the domain of possible spousal names would be a constraint violation. The "value not applicable" is the data entry operator's way of handling either a conditional relationship or a conditional property.

Value unknown (temporarily). Say the same data entry form is used, but this time the user simply does not know the name of the spouse. The employee has a spouse, but the name has not been obtained. In other words, the field is "applicable, but value temporarily unknown." This situation may occur quite legitimately; it is often the case that not all necessary information for a given task is gathered at one time. Of course, not all information relating to a task can be deferred. For example, some unique designation of the employee is essential, even if this designation happens to be an arbitrary, unique employee identification number. In the case of the spousal name, a need exists to recognize the existence of a spouse without necessarily knowing the spouse's name.

"Value unknown" is another way data entry operators handle a conditional property, while at the same time conveying the belief that a value will be known at some time in the future. Unfortunately, the "value unknown" kind of null does not quite succeed; it captures the existence of a relationship, but does not capture the fact that a unique designation for the spouse exists. In particular, it cannot treat two occurrences of the "value unknown" designation as the same and all others as different. For example, if the form also had a place for children of the spouse (as, for example, those from a different marriage), it would be very difficult to capture this information in a database without resorting to repeating groups. Few people would make this mistake on paper: some means would be invented to establish which "value unknown spouse name" had which children!

Value does not exist. Suppose that employees normally have social security numbers, but one particular employee does not. Although it might be reasonable to expect that the employee would eventually get a social security number, some foreign employees might never be able to obtain them. Thus, the data entry operator might know that the value belonging in the social security number field not only was not known, but it would never even be assigned. "Value does not exist" is another way of handling a conditional property, while conveying the belief that the value can never be known.

Value undefined. Some fields are defined in such a way that the appropriate value is "undefined" in certain circumstances. In particular, consider a field defined as the quotient of two numbers, such as the percentage of departmental sales revenues contributed by a particular salesperson. If the departmental sales revenues are zero (at the beginning of the sales period, for example), this number is undefined. This case might be due to bad design: The value is not entirely functionally dependent on either the salesperson or the department. Instead, it is a value derived from two other values, one of which is functionally dependent on salesperson and the other on department. "Value undefined" is one data entry version of a conditional constraint.

Value not valid. Say the value that a data entry operator enters violates a constraint, such as a domain constraint. We might want to record that such errors in the information gathering process have occurred. "Value not valid" is another data entry version of a conditional constraint.

Value rejected. The system may reject a value that the data entry operator knows is correct, and imply that a change is required to the domain definition or some other constraint. In other words, it is possible that the system will inform the data entry person that it cannot accept a value that is known to be correct, implying that the data entry person has detected a system design error. "Value rejected" is another data entry version of a conditional constraint, but with the added attempt to convey a belief that the constraint being violated is incorrect.

Value not supplied. Sometimes a value is not supplied during data entry—which often occurs when the data collection process is combined with the data entry process. It can also occur when the entered data is collected from uncooperative users or unreliable sources. For example, the U.S. Census survey contains certain optional questions that some residents of the U.S. do not wish to answer. It is also possible that the data entry operator chooses not to enter a particular field. Two cases must be

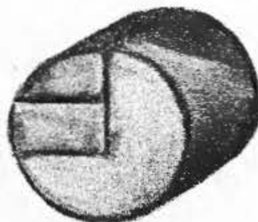
Motivations for allowing nulls in a database are sometimes valid

treated when data is not supplied; either the missing information is the value of a property (and so a way of handling conditional properties or conditional relationships), or it is the value of an identifier for the entity. In the latter case, we have the problem of capturing information about an improperly identified entity, possibly violating primary key discipline. Next month, in Part IV, we will see that this situation represents a database design error, and will reduce the problem of handling conditional properties.

The importance of understanding different flavors of null should not be underestimated. Not only do they appear in new databases through data entry, but we often find each of them as types of missing information in legacy databases. The process of migrating to a relational database requires identifying and handling each possible case.

DEFAULTS: AN ALTERNATIVE?

It is unfortunate that most commercial RDBMSs make it far easier to specify the SQL NULL as a default than to specify a meaningful default value for a column. As a result, SQL NULLs are sometimes used as an improper substitute for defaults. Indeed, the relationship between defaults and nulls is so intertwined that Date has suggested a systematic use of defaults as an alternative for all types of null,[3] a position with which I have some sympathy.



Certainly a systematic use of defaults would result in a far better DBMS than the many-valued logic alternative I criticized in Part II. However, I do not believe systematic defaults can or should be used in place of every possible occurrence of nulls. An alternative and more restricted systematic use of defaults will be among the proposed solutions to nulls in Part IV.

SUMMARY

Ralph Waldo Emerson once said (*Journals*, 1866) "If I cannot brag of knowing something, then I brag of not knowing it." By accident and by design, database practitioners often find themselves in this unfortunate position. The existence of a null in a database is ultimately a statement about what we do not know, about something that is not part of the defined (and hopefully agreed upon) universe of discourse for a particular database.

A database designer's motivations for incorporating or allowing nulls in a database are sometimes valid, representing a valiant attempt to deal with conditional relationships, properties, operations, constraints, and information. This conclusion, along with the conclusion of Part II that many-valued logic and, therefore, nulls are not an appropriate solution to the "missing information problem," leaves us in a dilemma. As a conclusion to this series, next month in Part IV I will propose a set of solutions to this dilemma.

The author would like to thank Chris Date and Hugh Darwen for their helpful comments and criticisms. Also, apologies to Billy Preston (again) and The Allman Brothers for the abuse of their song titles.

REFERENCES

1. Olson, R. G. *Meaning and Argument: Elements of Logic*, Harcourt, Brace, & World, 1969.
2. Date, C. J. "NOT is Not NOT!" in *Relational Database Writings 1985-1989*, Addison-Wesley, 1990.
3. Date, C. J. with Hugh Darwen. "The Default Values Approach to Missing Information" in *Relational Database Writings 1989-1991*, Addison-Wesley, 1992.

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."