VOL. 7 NO. 1

DATABASE Programming & Design

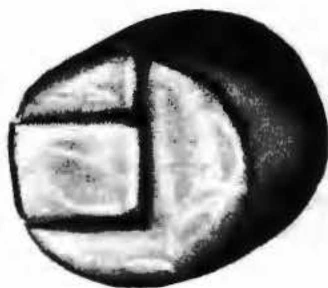
JANUARY 1994

All for One, One for All | **24** | PEGGY WATT
How does database fit into the emerging groupware paradigm—and will it work?

Plug 'n Play Data: Standards First | **34** | JOHN VAN DEN HOVEN
If ODBC, SAG, and IDAPI are still alphabet soup to you, here's help sorting it out.

Building Your Own SQL Generator | **45** | DON BURLESON
For do-it-yourselfers, all it takes is a dose of SQL, some REXX, and a little know-how.

Classical Logic: Nothing Compares 2 U | **54** | DAVID MCGOVERAN
Are you sure you want your RDBMS associating with many-valued logic?



DEPARTMENTS

EDITOR'S BUFFER | **5** | *Users: Get the message or kill the messenger?*

ACCESS PATH | **9** | *Massively parallel processing redux.*

DATABASE DESIGN | **11** | *Data Architecture: What goes where?*

ACCORDING TO DATE | **15** | *Why duplicate rows in SQL are a mistake.*

CLIENT/SERVER FORUM | **21** | *Software must play catch-up to hardware.*

DESKTOP DATABASE | **62** | *The goods on KnowledgeWare's Flashpoint.*

ANNUAL INDEX | **66** | *A subject index of 1993 articles and columns.*

PRODUCT WATCH | **71** | *Development products crowd the field.*

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

If the true power of the relational model is its foundation in classical logic, why do RDBMSs use multivalued logic—and betray that strength?

Classical Logic: Nothing Compares 2 U

IN PART I OF THIS series last month, I provided a brief tutorial on logic for database practitioners. Now, having laid the foundation, we may move on to see why the current handling of missing information using many-valued logic is misguided. Although numerous many-valued logic systems exist, it is my central thesis that *no many-valued logic will be suited to a DBMS's needs*. By contrast, the propositional logic meets all the objectives I presented in Part I (see the sidebar, "Objectives," page 60).

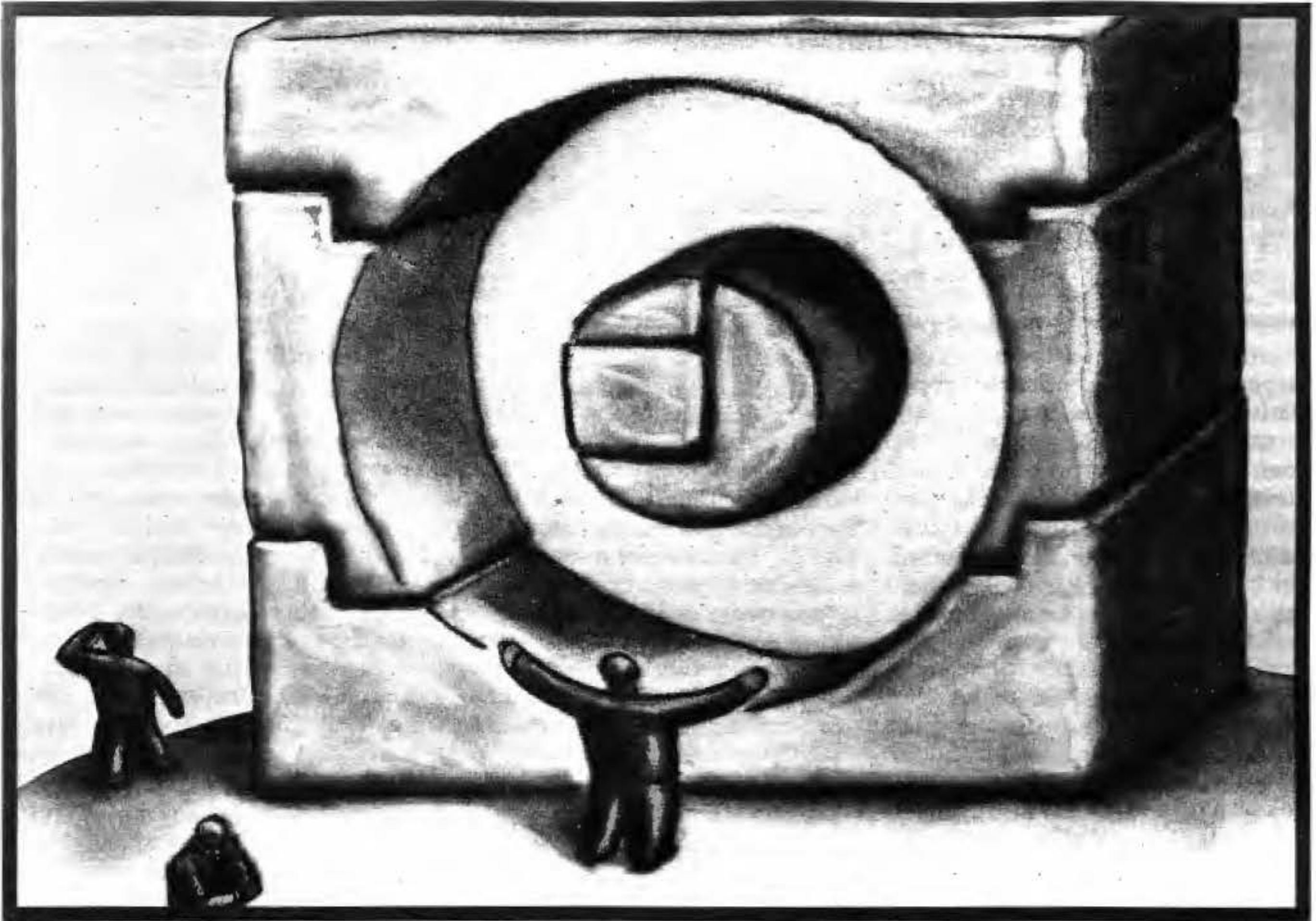
To understand this thesis, we must determine whether the properties of a many-valued logical system can meet the objectives set out in Part I for a DBMS as a logical system. To simplify our quest, we will examine only the properties of many-valued propositional logics. We may confine our examination in this manner for three reasons. First, we can understand any many-valued predicate logic as a generalization of a corresponding many-valued propositional logic; and so, certain problems with propositional logic version carry over to the predicate logic version. Second, real databases are finite; at the worst, only a very restricted version of the first-order predicate calculus (one without infinite domains and values) must be used. To reiterate a point (one I did not elaborate on in Part I), the kinds of expressions permitted via real database queries are limited. Specifically, only a fixed number of types of variables and a fixed number of possible values for each of those types, and well-formed formulas (wffs) of some maximum length are supported (ever write a query the DBMS found too big to parse?). Therefore, only some finite, possibly large number of propositions can ever be expressed as queries. Third, the formal investigation of many-valued predicate logic is immature; comparatively few such investigations exist.

C. J. Date has written extensively about the problems associated with E. F. Codd's (and SQL's) version of many-valued logic. The discussion here is more general. Codd's so-called four-valued logic (4VL) (and even worse, the "three valuedness" of SQL) is ill-defined, as I will show later in this article. For this reason, I cannot address formal arguments against any specific formal flaws! My arguments are forced to be general and apply to formal and informal problems that arise in using any many-valued logic for database work.

LEAVING NOTHING

To begin the examination, it is important to understand the similarities and differences of many-valued logics. Most three-valued systems are identical in their definitions of "AND," "OR," and "NOT," but differ with respect to the truth table definitions for other connectives, which connectives they take as primitive, and which rules of inference apply.

For purposes of informal explanation, a three-way classification of propositional many-valued logics will be useful, letting us take a divide-and-conquer approach



ARTWORK BY: BARTON STABLER

to investigating the use of many-valued logics for database use. The classification scheme is based on the following reduction procedure: allow the components of every wff in the logic to take only "TRUE" or "FALSE" as the truth values, and compare the resulting logical system to the propositional logic. The classes, slightly nonstandard and informal, follow:

☐ *Fragment*. A many-valued logic will be classified as a fragment if, under the reduction procedure, it reduces to a fragment of the propositional logic, by which we mean that some propositional logic connectives or rules of inference are missing, or some propositional logic theorems or tautologies no longer hold.

☐ *Extension*. A many-valued logic will be classified as an extension if it reduces to the propositional logic under the reduction procedure.

☐ *Deviant*. A many-valued logic will be classified as a deviant if it is so different from the propositional logic that it cannot be understood either as an extension or a fragment. A number of well-known (and often referenced) many-valued logics cannot be treated either as extensions or fragments of the propositional logic. These entirely different logical systems (the deviants) do not satisfy the familiarity objective (see the sidebar), and are either not truth functionally complete or have difficult-to-understand semantics. I will come back to this type of many-valued logic shortly.

FRAGMENTS

Fragments are generally not truth functionally complete and, in addition, require that users understand which portions of the propositional logic do not apply. This fact means that fragments necessarily violate the familiarity objective. To see why, we must understand a special type of one-place many-valued logic operator that is called the Slupecki T-function. A T-function is a one-place connective that converts every possible truth value to some particular one of the nontrue, nonfalse truth values (that is, to "UNKNOWN" in a three-valued logic [3VL]. All many-valued logics must contain the T-function to be truth functionally complete; but, adding this connective will at best convert the fragment into an extension, and possibly into a deviant. A meaningful interpretation of this function for database use is hard to imagine; this fact alone makes it unreasonable to expect a many-valued logic to meet our needs. The inclusion of the T-function and certain tautologies that may be implied by it creates a logical system that clearly violates the familiarity objective.

EXTENSIONS

The familiarity objective requires that the truth tables for the various many-valued propositional connectives of an extension reduce to those connectives for the two-valued propositional logic under the reduction procedure (refer to Figure 1). Clearly, this property is highly desirable in a logic used by a DBMS. However, a many-valued

AND

P \ Q	T	U	F
T	T	U	F
U	U	U	F
F	F	F	F

OR

P \ Q	T	U	F
T	T	T	T
U	T	U	U
F	T	U	F

NOT

P	NOT P
T	F
U	U
F	T

FIGURE 1. *Reduction of three- to two-valued logic.*

logic can reduce to classical propositional logic *if and only if* it is not truth functionally complete! By definition, every connective in a truth functionally complete logical system must be expressible, either directly (as a primitive operator) or indirectly (by composing primitive connectives). As already noted, the T-function must be expressible in truth functionally complete many-valued logics. Unfortunately, it has no counterpart in two-valued logics even under the reduction procedure. Therefore, either the familiarity objective or the truth functional completeness objective is violated by extensions.

EXTENSIONS AND DEVIANTS

The familiarity objective is actually more difficult to satisfy than the previous information would indicate. Both extensions and deviants violate the uniform interpretability objective. Under the reduction procedure, few many-valued logics preserve the tautologies and rules of inference most commonly relied upon by database users to reason with queries (see Figure 2). (Of course, users may not realize how much they depend on these rules!)

We must consider two implications: First, violated propositional tautologies must never be used —implicitly or explicitly—when working with a database using such a logic. Thus, extensions and deviant logics are less intuitive than the propositional logic, and, for practical purposes at least, are less deductively powerful as well. Second, permissible rules of inference (ones used by the user and the optimizer) must be sensitive to whether or not the database permits nulls and/or actually contains nulls.

If the DBMS does not permit nulls, we can use the familiar propositional logic and never even need to learn the many-valued logic. But, if nulls could exist in the database, the many-valued logics rules of inference must be used from the beginning. The meanings of query results are then definite as long as nulls do not actually appear in the database, and the uniformly interpretable objective can be preserved.

Once nulls are permitted to appear, even in tables not accessed in a particular query, all queries become indefinite in meaning (excluding the rather bizarre possibility that the table containing the null has no relationship whatsoever to the tables accessed). One null, anywhere in the database, changes the meaning of all related tables, violating the uniformly interpretable objective! This violation occurs because we can no longer think of the accessed tables as though they simply contain rows representing facts about the universe of discourse; each row now represents a fact having relationships to missing information.

For example, consider the parts-suppliers database in Figure 3. If the database did not contain the shaded row in the Suppliers relation, the results of all queries would have a definite and fairly intuitive meaning. But with the shaded row permitted, the very meaning of parts and suppliers changes! In particular, parts are no longer definitely located in a

Law of Detachment	P & (P → Q) → Q
Modus tollendo tollens	-Q & (P → Q) → -P
Modus tollendo ponens	-P & (P V Q) → Q
Law of Simplification	P & Q → P
Law of Adjunction	P & Q → P & Q
Law of Hypothetical Syllogism	(P → Q) & (Q → R) → (P → R)
Law of Exportation	[P & Q → R] → [P → (Q → R)]
Law of Importation	[P → (Q → R)] → [P & Q → R]
Law of Absurdity	[P → Q & -Q] → -P
Law of Addition	P → (P V Q)
Law of Double Negation	P → --P
Law of Contraposition	(P → Q) ↔ (-Q → -P)
DeMorgans's Laws	-(P V Q) → (-P & -Q)
	-(P & Q) → (-P V -Q)
Commutative Laws	P & Q → Q & P
	P V Q → Q V P
Law of Equivalence for Implication and Disjunction	(P → Q) ↔ (-P V Q)
Law of Negation for Implication	-(P → Q) ↔ P & -Q
Laws for Bicondional Sentences	(P ↔ Q) ↔ (P → Q) & (Q → P)
	(P ↔ Q) ↔ (P & Q) V (-P & -Q)
Laws of the Excluded Middle	P V -P
Law of Contradiction	-(P & -P)

Legend: For space considerations,
↔ means "BI-IMPLIES" (logical equivalence)
→ means "IMPLIES"
& means "AND"
- means "NOT"
V means "OR"

FIGURE 2. *Some useful tautologies of two-valued propositonal logic.*

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris

SP

S#	P#	QTY
S1	P1	300
S2	P1	300
S2	P2	400

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London
P2	Bolt	Green	17	<null>

FIGURE 3. *Troublesome rows in the Parts-Suppliers database.*

known city. And since suppliers are defined as supplying parts, by extension they are no longer suppliers of parts located in a known city. Thus, querying the suppliers table *S*—which contains no nulls—results in a fundamentally different kind of answer when the parts table *P* is allowed to have nulls in one of its columns.

Now consider the distinction between the shaded row being permitted and actually appearing in the database. Whereas the row being permitted changes the meaning of the entities represented by tables, the actual appearance of the row changes the meaning of a query result even when the row is deliberately excluded! For example, suppose we try to follow a "no nulls" discipline and want to see only those suppliers "unaffected" by the shaded row. To select these suppliers, I must first presume the existence of a relationship to rows similar to the shaded one, and then use this relationship to exclude affected suppliers. In pseudo-SQL, something like "SELECT * FROM S MINUS (SELECT * FROM S, SP, P WHERE S.S# = SP.S# AND SP.P# = P.P# AND CITY IS NOT NULL)" is required. If the shaded row does not exist, these suppliers provide, if anything at all, the type of parts that might or might not be definitely located in a known city. However, if the shaded row appears in the database, this same list of suppliers definitely does not supply the specific parts indefinitely located or locatable! As strange as it seems, when a row similar to the shaded one appears in the database, the results to our "null avoiding" queries become more definite regarding the indefinite.

Readers might object that I have chosen a particular interpretation of null to illustrate these problems, but I invite them to consider other interpretations as an exercise. DBMS use of many-valued logics requires teaching an entirely new way of thinking to all your database designers, developers, and users. The cost of this approach is hard to assess in practice. It is at odds with the goals we set out to satisfy with an RDBMS.

What about the truth functional completeness and the deductive completeness objectives? Sometimes we can make a many-valued logic truth functionally complete by adding a new axiom or connective, such as the T-function mentioned earlier, to the set of axioms and primitive connectives. However, this approach has at least one of three undesirable consequences: producing theorems that have no counterpart in two-valued logic, making the system inconsistent, or making the system incomplete. Indeed, based on work by the logician Rose,[6] avoiding the first possible consequence forces us to choose between the other two outcomes. In particular, as long as the system does not contain certain types of undesirable theorems (having no counterpart in the propositional logic and, therefore, violating the familiarity objective), Rose showed that either the new axiom makes the system inconsistent, or the new axiom is a tautology of the propositional logic (in other words, something we intuitively thought was already true, but actually was not).

The second possible consequence (inconsistency) is clearly undesirable since it means that every wff becomes a tautology, even one that would otherwise be considered a contradiction. (In an inconsistent system, you can prove anything.) Suppose an SQL SELECT was issued against a database man-

	T	F
T	T	T
F	T	T

	T	F
T	F	F
F	F	F

	T	F
T	F	T
F	F	T

	T	F
T	T	F
F	T	F

	T	F
T	F	T
F	T	T

	T	F
T	T	F
F	T	T

	T	F
T	T	T
F	F	T

	T	F
T	T	T
F	T	F

	T	F
T	F	F
F	T	T

	T	F
T	T	T
F	F	F

	T	F
T	F	T
F	T	F

	T	F
T	T	F
F	F	T

	T	F
T	T	F
F	F	F

	T	F
T	F	T
F	F	F

	T	F
T	F	F
F	T	F

	T	F
T	F	F
F	F	T

FIGURE 4. *The 16 two-place connectives of two-valued logic.*

Truth Values	One-place Connectives	Two-place Connectives
2	4	16
3	27	19,683
4	256	4,294,967,296
n	n^n	n^{n^2}

FIGURE 5. *Number of connectives versus number of truth values.*

aged by a DBMS based on such a system. Regardless of the predicate in the **WHERE** clause, this predicate would be treated as "**TRUE**" for all column values tested, and would therefore never restrict the result set!

For the third consequence (the new axiom is a tautology) to be applicable, the system cannot be an extension of the propositional logic (since this approach requires adding a many-valued tautology and will result in an inconsistent system). Therefore, it is either a fragment (and subject to the problems discussed earlier for such many-valued logics) or a deviant.

MORE ROPE, PLEASE!

Suppose we are willing to violate the truth functional completeness objective, under the assumption that the theorems that cannot be expressed (due to missing means of representing some connectives) are, in some sense, obscure. Perhaps we are even willing to violate part of the familiarity objective under the assumption that learning new tautologies and rules of inference is not an excessive task. Even so, a many-valued logic introduces further undesirable complexities. These complexities include the number of connectives, the number of meaning assignments for connectives, meanings of query results, arbitrariness in the number of truth values, loss of deductive power, and unusual/nonintuitive semantics. I'll briefly discuss each of these complexities.

The *number of connectives* in a logic depends combinatorially on the number of permissible truth values. In the familiar two-valued logics, 16 possible two-place connectives exist (refer to Figure 4). The number of connectives grows rapidly with the number of truth values (refer to Figure 5). For a 3VL, 19,683 two-place connectives exist, as compared to the 16 of ordinary two-valued logic.[6]

Of course, even in the two-valued propositional logic we do not normally need to remember or use all of these connectives explicitly: a few suffice to express all the others, which is the essential importance of truth functional completeness. Likewise, we do not need to memorize all possible connectives in a many-valued logic if the primitive set is truth functionally complete. The number of connectives required can be very few.[1]

But if, as assumed, the system is not truth functionally complete, users must be prepared to use and understand all 19,683 dyadic connectives (in the three-valued case) to express a query! Such complexity is beyond most users' grasp; not only would they find it frustrating, but the user will probably make mistakes, using the wrong connective for a desired result. This same complexity applies to the optimizer's design and the amount of code required to implement it.[2]

In addition to increases in syntactic complexity due to the number of connectives, the number of distinct *meaning assignments for connectives* increases as well. As noted earlier, any truth value can be treated as true-like (that is, designated), false-like (antidesignated), or neither (undesignated); these distinctions are necessary for identifying tautologies and contradictions in many-valued logics. For example, in a 3VL, three distinct one-place connectives could be called negation. With the additional complexity of unknown being designated, antidesignated, or undesignated, the number of possible meaning assignments for "negation" expands to nine (see Figure 6)! This complexity violates one of the motivations for using a

P	NOT P
+T	F
U	U
- F	T

P	NOT P
+T	F
+U	U
- F	T

P	NOT P
+T	F
- U	U
- F	T

P	NOT P
+T	F
U	F
- F	T

P	NOT P
+T	F
+U	F
- F	T

P	NOT P
+T	F
- U	F
- F	T

P	NOT P
+T	F
U	T
- F	T

P	NOT P
+T	F
+U	T
- F	T

P	NOT P
+T	F
- U	T
- F	T

Legend: "+" signifies designated "–" signifies antidesignated

FIGURE 6. *Possible meaning assignments of three-valued "NOT."*

logical system in the first place. Surely users do not wish to work with such a DBMS.

If the users and designers of a database do not agree on the meanings of query results, confusion is inevitable and results in a loss of data integrity—users will eventually update the database in ways that violate the intended, but unenforceable, data meaning. To assign truth values to propositions or arguments (the process of defining an intended interpretation), the database's designer must have a consistent understanding of what each truth value means (our uniformly interpretable objective). This meaning must be understandable to users and consistent with the connectives and rules of inference. Although the meaning of individual truth values (as used, for example, in the relational model) may appear to be reasonable, they can have nonintuitive or incorrect consequences. Codd categorizes these problems as being either "mildly incorrect" (meaning an expression is evaluated as unknown when it is actually either true or false) or "severely incorrect" (meaning an expression is evaluated as true or false when it is actually unknown).[7] Either way, the possibility of an incorrect response from the DBMS means "don't trust the DBMS!" It is equivalent to saying: "When you use a calculator, sometimes 1 + 1 = 2 and sometimes it doesn't, so check it yourself."[7] (If this is the case, why even use the calculator?)

The *number of truth values* in a many-valued logic can be arbitrary, in that the number required cannot be established definitively. If users think of "UNKNOWN" as intermediate between "TRUE" and "FALSE" in a 3VL, no intuitive reason exists to stop at three truth values. In fact, some motivations exist for immediately extending the number of truth values. For example, Codd suggests a four-valued approach with unknown and inapplicable. What if we need to insert a row in a table, but don't know if the missing value is properly described as the "UNKNOWN" truth value or the "INAPPLICABLE" truth value? This problem leads to the need for a fifth value. Where does the process end?[3,8]

No many-valued logic will be suited to a DBMS's needs

Two-valued logical systems can sometimes be uniformly extended to handle an arbitrary number of truth values, assuming that properties such as completeness are not important. However, as the number of truth values increases, the number of tautologies in these systems generally decreases. Since tautologies are among the essential tools of deduction, this process results in a practical, if not formal, loss in deductive power. And let us not forget the implications for optimizers: This DBMS component not only offers performance improvements, but also enables data independence! Among other things, an optimizer that uses many-valued logic is less likely to recognize the equivalence (via a suitable semantic transformation) of two expressions, and is less likely to be able to reduce a complex expression to a simpler one (via rules of inference and tautologies) than one using standard two-valued logic. This result would not be a problem were it not for the particular tautologies that are often affected by many-valued logics.[6,8] For example, "(P IMPLIES P) BI-IMPLIES ((NOT P) OR P)," although intuitively is always "TRUE," is not a tautology in some many-valued logics!

The impact of this loss in deductive power is serious. Most optimizers effectively give up when faced with many-valued logic, making no semantic transformations whatsoever. Some even fail to use an index if the indexed columns can contain nulls, whether they actually do or not! Certainly, this reduction in deductive power makes it much more difficult for users to reason toward a desired answer using a sequence of queries. The poorer the optimizer in this regard, the more the user must "optimize by hand," carefully selecting the exact manner in which a query should be expressed (two expressions are not likely to be equivalent except for specific values of arguments). And this situation means the user must understand the logical system very well and be willing to give up logical data independence.

Although interesting from a formal perspective, the many-valued logic proposed by Codd (and related proposals by Vassiliou, Lipski, and Biskup) leaves much to be desired from the perspective of understandable semantics. In particular, as elaborated by Grahne,[13] each occurrence of an A-mark (applicable but unknown) in a table can be seen as a shorthand for a set of tables, each obtained by substituting a permissible value for the A-mark. To construct understandable queries in such a system, the user must somehow keep in mind all the possible substitutions. Although these formal systems may be interesting, such semantics can make them nonintuitive and error-prone. In an informal poll I conducted of approximately 30 database designers and administrators, all of them expressed amazement at this interpretation and felt that it was unacceptable.

OTHER SYSTEMS

The most common versions of many-valued logic are variations on other systems, such as those developed by Lukasiewicz, Post, and Kleene. Variations of Lukasiewicz's systems are sometimes referred to as the basis for SQL's 3VL. While this supposition cannot be true,[4] it is worth examining the properties of the Lukasiewicz systems. Lukasiewicz systems are not truth functionally complete (so the system would not be able to verify some facts using the available operators), nor are they natural extensions of the classical propositional logic. Certain tautologies of the propositional logic cease to be true in the Lukasiewicz systems, and, conversely, certain tautologies of the Lukasiewicz systems have no counterpart in the propositional logic. In our terminology, they are deviants.

Lukasiewicz's were intended to treat contingent (especially future contingent) propositions as meaning "temporarily unknown." The "UNKNOWN" in his 3VL is similar to Codd's A-marks. For example,

the truth value of "It will rain tomorrow." would be "UNKNOWN," but would eventually be determined as either "TRUE" or "FALSE." Therefore, the Lukasiewicz "UNKNOWN" is a temporary placeholder for a standard truth value. These various facts about Lukasiewicz systems eliminate them from further consideration: They are not candidates for use as a DBMS's logical system.

We can prove that some many-valued logics are truth functionally complete (all are consistent by definition if at least one truth value is undesignated), but have semantics clearly inappropriate for a DBMS. Here are a few examples: In Post's systems, truth valuations apply only to sets of propositions (that is, sets of rows), each individual proposition having a classical truth valuation, rather than the individual propositions. Kleene had in mind the truth valuations of propositions involving mathematical functions undefined for certain ranges of predicate values. The concept of undefined is similar to the purpose of Codd's I-marks. Bochvar created a system with a set of "internal" truth tables and a set of "external" truth tables, treating unknown as "undecidable" or "meaningless." This system is similar to SQL in the sense that SQL effectively returns false (the external system) to the user when the answer is unknown (the internal system), but Bochvar's systems are dissimilar in other respects.

"RELATIONAL'S" 3VL

So far, the arguments I have presented are generic; they apply to many-valued logics generally. However, the problems raised cannot be fixed. Indeed, formal logicians do not perceive them as problems that must be fixed! Although my thesis is that no many-valued logic is suitable for the a DBMS's needs, I feel compelled to point out a few problems that apply specifically to the 3VL and 4VL described by Codd and the 3VL implemented in SQL.

☐ As I noted at the beginning of this article, this discussion of the problems associated with using a many-valued logic in a RDBMS was forced to be general, because the 3VL used in Codd's version of the relational model is not completely defined. The situation is even worse in SQL, in part because the definition is only implicit (rules of inference, axioms, and primitive connectives are not specified)! In particular, the system is definitely not a Lukasiewicz system, nor is it one defined by Post, Kleene, or Bochvar. What, exactly, is this logical system's definition?

☐ The rules of inference are unspecified. We can assume that, since subqueries are supported, a limited rule of substitution is supposed to hold. What about other standard rules such as *modus ponens* (if "P IMPLIES Q" and "P," then "Q")? A many-valued logic has multiple forms of this rule (two for 3VL). If modus ponens is supposed to hold, it is important to say which of the forms are intended. Similar concerns apply to other rules of inference such as *modus tollens* and DeMorgan's Laws.

☐ Although most many-valued logics are based on an extension to the propositional logic, the relational model is supposedly based on first-order predicate logic. Certainly SQL defines the "EXISTS" quantifier and, so long as nulls are excluded, the "FORALL" quantifier can be simulated. Unfortunately, no discussion of a many-valued first order predicate logic exists in the relational model, nor of how the relational model fares without appeal to first-order predicate logic. How the formal system should treat quantifiers, and what special rules of inference apply is left largely to our imagination. At best, we know that both

Objectives

A DBMS SHOULD BE A logical system that is uniformly interpretable, expressively complete, deductively complete, consistent, truth functional, truth functionally complete, familiar, and, ideally, decidable. Intuitively, you can understand each of these objectives by keeping the following in mind:

■ Familiar: The common understanding of the truth values, connectives, rules of inference, and accepted tautologies should remain valid. In other words, the user should not have to learn an unfamiliar or nonintuitive logical system that contains surprising theorems and tautologies or denies commonly held rules of inference, so that errors of usage become more likely to occur.

■ Uniformly interpretable: The intended interpretation of every symbol, truth value, and query should be unambiguous, irrespective of the database's state.

■ Truth functional: A query's evaluation (a wff) can proceed mechanically from the evaluation of its components; similarly, queries of arbitrary complexity can be written and understood from an understanding of the connectives alone.

■ Truth functionally complete: The set of initially given operations and connectives in the query language suffice to express any logical connective definable via a truth table. Thus, for every fact in the universe of discourse, a truth-valued expression will exist to determine whether this fact is represented in the database.

■ Expressively complete: All queries that are meaningful in the application's context can be expressed and all relevant facts about the application environment can be captured in the database.

■ Deductively complete: Every fact represented by the database, either implicitly or explicitly, can be obtained via a query.

■ Consistent: The result of every query represents facts that can be inferred from the database.

■ Decidable: Although not strictly required, a decidable and consistent system allows a query to be checked via an algorithm to determine if it is (1) a tautology (since every theorem in a consistent system is a tautology—in this case every row would satisfy the predicate), (2) a contradiction (in which case no rows could ever satisfy the predicate), or (3) neither.

—by David McGoveran

the relational model and SQL treat "EXISTS" as a finite interation of "OR" and so, in practice and as long as nulls are not permitted, the logical system is at best the finite version of the first-order predicate calculus mentioned in the beginning of this article.[5]

WHAT SHOULD WE DO?

The criticisms of many-valued logics in this article as they apply to use in DBMSs have simple, practical consequences. Based on these results, I recommend adherence to the following guidelines:

- ☐ Avoid nulls and many-valued logic.
- ☐ Do not use SQL operations such as outer join and outer union, which create nulls.
- ☐ Until you can implement these first two actions, review the meanings of queries and query results carefully. The more complex the query, the more important this step is.
- ☐ Lobby vendors to drop support for nulls and many-valued logic from their products.
- ☐ Ask vendors to make full use of first-order predicate calculus in their optimizers.
- ☐ Demand that DBMS vendors place high priority on the goals and objectives outlined early in this article. To this end, they must recommend against the use of many-valued logic in their products, and must oppose it in the SQL standard.
- ☐ Demand that, until vendors can comply with these guidelines, they supply a configuration option that disables the use of nulls and many-valued logic at the system level.

To summarize somewhat glibly, the key conclusion readers should draw from this technical discussion is that "nothing" is to be gained from "nothing"; nothing compares to the two-valued approach. In fact, a great deal of knowledge, power, usability, performance, and maintainability is at risk if many-valued logic is used in a DBMS. Apply Occam's Razor: Eliminate all the nothing from your databases.

In next month's installment, I will propose a list of the main motivations for including nulls (both I-marks and A-marks) in a

Nothing compares to the two-valued approach

database and discuss their validity. Some of these motivations are valid; this conclusion, along with the conclusion that many-valued logic is inappropriate, leaves us in a dilemma. This dilemma will be addressed in Part IV.

The author would like to thank Chris Date, Hugh Darwen, and Ron Fagin for their helpful comments and criticisms. I would also like to apologize to Billy Preston (again) and Sinead O'Connor for the abuse of their song titles.

NOTES AND REFERENCES

1. In fact, a single two-place connective W will suffice for all many-valued logics. In a system with n truth values, if the truth values are represented by the natural numbers from 1 to n, the binary connective W is defined as: /P W Q/ = (1 + [max (/P/,/Q/) % n] where /R/ is the truth valuation of the truth-valued expression R and % is the modulo operation. [6, p. 65]

2. Contrary to Codd's position on this issue, the number of distinct logical connectives in the absence of a specified number of primitive connectives that satisfy truth functional completeness is not comparable to the infinite number of distinct arithmetic functions that can be defined in ordinary arithmetic. Instead, this number is properly compared to the number of distinct arithmetic operations, of which very few in arithmetic exist (such as addition and multiplication) from which an infinite number of arithmetic functions can be defined.

3. Codd replied to this criticism [7] saying, essentially, that an I-mark is a "catch-all" and thus terminates the process. This statement is incorrect. Such systems require a mechanical procedure by which the system can determine which wffs should be evaluated as corresponding to the truth value for a simple predicate with one I-marked variable versus one with an A-marked variable. At the very least, if A-marks are to be distinguished from I-marks and vice-versa, such a procedure is implied by the distinction. Thus Codd's kind of 4VL is not substantially different from the kind of 4VL described by Date [8] and susceptible to the same troublesome semantics.

4. Lukasiewicz gave "IMPLIES" and "NOT" as his primitive connectives, deriving "OR" and "AND" from them. His definition of "IMPLIES" is different from that used in the propositional or predicate calculi, which define "P IMPLIES Q" as "NOT P OR Q" (see Figure 7). In fact, Lukasiewicz's version of "IMPLIES" cannot be derived from the definitions of "NOT," "AND," and "OR." This is because "NOT," "AND," and "OR" each preserve "UNKNOWN" from the inputs (and therefore so do any combinations of these), whereas Lukasiewicz version of IMPLIES does not. Since SQL does not define IMPLIES, claims that "it is based on a variant of Lukasiewicz's 3VL" must be false!

5. It would be good if this logical system were the intended one; such a system has the desirable properties of being both complete and decidable.

6. Rescher, N. *Many-Valued Logic*, McGraw-Hill, 1969, pps. 63 and 166.

7. Codd, E. F., and C. J. Date. "Much Ado About Nothing," *Database Programming & Design*, 6(10): 46-53, October 1993.

8. Date, C. J. "NOT is Not NOT!" in *Relational Database: Writings 1985-1989*, Addison-Wesley Publishing Co., 1990.

9. Suppes, P. *Introduction to Logic*, Wadsworth, C. 1957.

10. Codd, E. F. "A Relational Model of Data for Large Shared Data Banks," reprinted in *Readings in Database Systems*, M. Stonebraker, ed., Morgan Kaufmann, 1988.

11. Codd, E. F. "Extending the Database Relational Model to Capture More Meaning," reprinted in *Readings in Database Systems*, M. Stonebraker, ed., Morgan Kaufmann, 1988.

12. Bolc, L., and P. Borowik. *Many-Valued Logics 1: Theoretical Foundations*, Springer-Verlag, 1992.

13. Grahne, G. *The Problem of Incomplete Information in Relational Databases*, Springer-Verlag, 1991.

14. Delong, H. *A Profile of Mathematical Logic*, Addison-Wesley Publishing Co., 1970.

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."

MATERIAL IMPLICATION

P \ Q	T	U	F
T	T	U	F
U	T	U	U
F	T	T	T

LUKASIEWICZ IMPLICATION

P \ Q	T	U	F
T	T	U	F
U	T	T	U
F	T	T	T

FIGURE 7. *Three-valued material implication versus Lukasiewicz implications.*