CA's dBFast 2.0 • Can IBM's DB2 Family Crack a Tough DBMS Market?

DATABASE Programming & Design

Missing Values: Why False Logic Endangers Relational Database

GUI Builder Tools: The Lowdown on Hot Products

From Centralized to Distributed: Rethinking Database Administration

Repository Standards Revisited

12

0 71896 48946 0

DECEMBER 1993
$3.95
$4.95 Canadian

STARTER

BY DAVID McGOVERAN

Success of relational database is endangered by a misunderstanding of its true foundation—the ageless logic behind the relational model

Nothing from Nothing

(Or, What's Logic Got to Do With It?)

THE RELATIONAL model may not be dead, but it suffers from incapacitating wounds at the hands of database vendors, standards, and benchmark committees (not necessarily distinct from vendors), and "experts" who insist on explaining what they never bothered to understand. Of course, not all the blame can be laid on these innocents; despite the genius of many of his insights, Dr. E. F. Codd must take some responsibility for the confusion that surrounds relational model issues—in particular, the confusion over many-valued logics.

This article is Part I of a four-part series. The series addresses a crisis (and a *scandal*) of the relational model: the use of many-valued logic as a mechanism for handling missing information. Before embarking on our journey into the problems with many-valued logic, it is necessary to consider what a logical system is and how it relates to DBMS implementation. The goal of Part I is to provide a foundational knowledge of logic for database uses.

Part II will use the terminology defined here to explain why many-valued logic approaches are not appropriate for practical database use. Part III will examine the pragmatic motivations that compel database designers to use "nulls," or any other indicator of missing information.[1] Therefore, we are left with a conundrum: If many-valued logics are inappropriate for handling missing information, and database designers have legitimate reasons for wanting to represent this missing information, what can we use in place of many-valued logic? Part III sets the stage for—

ARTWORK BY: BARTON STABLER

and Part IV provides—a comprehensive answer to this question, with practical solutions for the most common situations in which nulls appear in current database implementations.

This series is neither as formal or complete as I would like to make it. I must pass over known issues due to space limitations; I am also certain that formal issues and developments exist of which I am unaware. Nonetheless, I have tried to assure myself (through extensive research and study) that the discussion of key issues is accurate, and that the reader will be prepared to understand the essential problems associated with real-world database use of many-valued logics discussed in Part II. The concepts are difficult: Be patient; take it in carefully and slowly.

Vocabulary:
- Variables: any legal SQL truth-valued expression (for example, "colA > colB")
- Grouping indicators: "(", ")"
- Truth values: "TRUE," "UNKNOWN," "FALSE"
- Connectives: "NOT," "AND," "OR" (defined for two-valued logic)

Formation Rules:
- The rules for creating syntactically correct SELECTs (and, by extension, syntactically correct INSERTs, DELETEs, and UPDATEs).

Axioms:
- No explicit axioms, although each row in the database may be considered an axiom.

Rules of Inference:
- A restricted rule of substitution; equivalent expressions may generally be substituted, one for the other, and a subquery may appear in place of certain expressions.

FIGURE 1. *SQL (without nulls) as a logical system. Loosely speaking, we can consider SQL a logical system.*

DBMS GOALS

A DBMS should provide shareable, reusable, and efficient services for the definition, capture, organization, and manipulation of data. The DBMS should do it in a way that ensures the data's integrity, regardless of user actions or system failures. An additional goal drives much of what differentiates a relational DBMS (RDBMS) from other DBMSs: Changes to the data should not require changes to applications, and vice-versa. Adherence to this goal minimizes the amount of code that must be written for a given application, and reduces application maintenance. Curiously, however, this goal is often forgotten. This goal has had a strong impact on the features needed in an RDBMS.

If applications are to be independent of data organization and access methods, the data definition and manipulation language must be declarative (or "nonprocedural"). Otherwise, a change to organization or access methods (such as those required as the database grows or is optimized for performance) will require a compensating change in the application. In addition, much of the data access code found in structured applications involves sorting and selecting data, typically using numerous control loops. If this code can be moved out of individual applications and shared by all applications, we gain in areas of performance optimization and maintenance. However, if applications developers must choose from several data access routines, the gains will not be very large. For example, having one data access routine for each data structure would defeat the purpose.

This situation suggests that we should minimize the number of distinct data access routines, at least as seen by applications. Therefore, we need a special language (the data sublanguage or "query language") using the smallest number of operations. At odds with this goal is the concern that the language be capable of expressing every possible request for the intended set of applications (including ad hoc queries, unimplemented applications, and more). This latter concept is known formally as *expressive completeness.*

Another obstacle to minimizing data access routines is that the most efficient access method *does* depend on physical data organization. Thus, the question is: How can we map a few data access routines to the possibly many routines, each optimized for the actual organization and type of data being accessed? The answer is automatic data access code optimization. Such optimization requires knowledge of data's physical organization. And if the database is changing rapidly, this knowledge must be up-to-the-minute. So, where in a business system's structure is this information available? Where should shared data access code be maintained, and where should this optimization take place? The obvious answer is: in the DBMS.

This line of argument leads us to a fundamental problem: Which algorithms should the DBMS optimizer (the code that performs optimization) use? Techniques used by compilers for procedural code optimization are relatively straightforward; for the most part, they do not alter the algorithm as it was coded by the developer. We need a different kind of optimization, one that substitutes an equivalent but more efficient algorithm (a data access method) for a standard data access routine. This concept implies that the optimizer is able to identify algorithm equivalence and evaluate the relative cost of the available algorithms; it must have provably correct rules by which it determines algorithm equivalence.

FORMAL LOGICAL SYSTEM

All of these considerations lead us to one conclusion: The query language should be an implementation of a formal logical system. To understand what is possible, let's briefly examine the components and properties of formal logical systems.

We must begin with some definitions, which will be used extensively as we go on. A logical system consists of four types of ob-

jects (see Figure 1):

☐ Vocabulary. A collection of symbols used unambiguously to represent truth-valued variables (such as "P," "Q," and "x"), grouping indicators (such as "(", ")", and "|"), truth values ("T" for true or "F" for false), and connectives (such as "OR," "AND," and "NOT").

☐ A collection of *formation rules* for governing the creation of "well-formed formulas" (wffs—pronounced "wiffs").

☐ A collection of *axioms*. The set of wffs that is given initially, each of which is guaranteed to be "true." Ideally, the axioms should all be *independent*; that is, it will not be possible to prove any given axiom from the others.

☐ A collection of *rules of inference* (also called deductions) by which a new wff may be derived from existing wffs. This new wff is called a *theorem*. A finite sequence of wffs, each of which is either an axiom or can be inferred from an earlier wff via a rule of inference, is called a *proof* and, in particular, is a proof of the final wff.

The definition of a logical system is meant to be applied in a purely mechanical fashion: that is, the process of determining whether an object (expression) belongs to the vocabulary, is a wff, or is an axiom, cannot be based on judgment or on the outcome of some random event. Determining whether or not a rule of inference has been properly applied must also be purely mechanical.

The connectives are often defined in terms of truth tables. A truth table is a tabular representation of a formation rule and, possibly, certain rules of inference that specify the compound's truth value, given the components' truth values (see Figure 2). We say that a set of connectives is *independent* if it is not possible to express the truth table for any given connective in terms of the truth tables for the other connectives. For example, the set of truth tables consisting of the SQL connectives and the connective (shown in Figure 3) called *material implication* is not independent, since material implication has the same truth table as "(NOT P) OR Q." Readers can show this fact by substitution using the truth tables in Figure 2 for "NOT" and "OR." In fact, "AND," "OR," and "NOT" also fail the test of independence (see the exercises at the end of the article).

When the truth value of a wff can be evaluated in a mechanical manner from the truth values of its components, a logical system is said to be truth functional. We will use this important concept later.

Each possible choice for the objects that make up a logical system results in a different logical system. We could show some logical systems to be equivalent in some sense, but only in trivial cases (usually involving differences of representation—for example, if Greek letters are used in place of English for symbols, or if we swap the identification of axioms and theorems). Not all logical systems are equally powerful. For a particular logical system, a rule that assigns a truth value to every truth-valued variable in a set of wffs is called an *interpretation* of that set of wffs. The following is an example of an interpretation:

Given wffs: "P AND Q," "P OR Q"
Truth value assignments: "P" is "TRUE," "Q" is "FALSE"

Note that in some logical systems (including, in particular, the system supposedly underlying SQL), a wff may consist of truth-valued expressions containing variables that are not themselves truth-valued, but whose assigned values imply the truth or falsity of the expression. The interpretation of such expressions then includes the values assigned to these non-truth-valued variables.

AND

P\Q	T	F
T	T	F
F	F	F

OR

P\Q	T	F
T	T	T
F	T	F

NOT

P	NOT P
T	F
F	T

FIGURE 2. *Two-valued truth tables for SQL connectives.*

IMPLIES

P\Q	T	F
T	T	F
F	T	T

BI-IMPLIES

P\Q	T	F
T	T	F
F	F	T

FIGURE 3. *Two-valued material implication and material equivalence. Note that these are not directly supported in SQL!*

For example, a relational table (simply "table" hereafter) defines a truth-valued expression, with each column representing a variable. Consider the Suppliers table of the familiar Parts-Suppliers database. The table consists of a supplier number (S#), a supplier name (SNAME), a status (STATUS), and a city location (CITY), and defines the expression: "There exists a supplier with S# = w AND SNAME = x AND STATUS = y AND CITY = z, where w, x, y, and z are variables." The row <'S1','Smith','20','London'> appearing in the table represents the expression, with these values substituted for *w*, *x*, *y*, and *z*, respectively. If values for *w*, *x*, *y*, and *z* are substituted in the expression and these values actually appear as a row in the table, the expression is said to evaluate to TRUE. Otherwise, it must evaluate to FALSE.

Perhaps the most familiar logical system is the (two-valued) *propositional calculus*. This system treats each variable as a proposition that must be evaluated independently of all others, and each such variable can be assigned exactly one value in a proof. The usual set of independent connectives consists of the familiar "AND", "OR", and "NOT", with "IMPLIES" (implication) and "BIMPLIES" (truth-value equivalence)[2] being defined in terms of these base connectives in a manner called *material implication* and *material equivalence* (see Figure 3). Among its rules of inference are:

☐ The rule of *substitution* (given Q BIMPLIES R and P IMPLIES Q, the rule of substitution tells us that "P IMPLIES R")

☐ The rule of *modus ponens* (if P IMPLIES Q is true and "P" is true, then "Q" is true).

A logical system is generally intended for some practical use. For convenience, we will say that our understanding of the subject of that practical use (for example accounting) is an *informal theory* (that is, some informal set of rules, requirements, descriptions, and so forth) regarding the particular subject or process. The informal theory's scope is sometimes called the *universe of discourse*. The practical use of a logical system is implemented by assigning meaning to

the elements of the vocabulary. From this meaning, it should be possible to assign to each truth-valued variable an obvious truth value. The resulting interpretation is said to be the *intended interpretation*. For example, if we designed a database with an ACCOUNTS table, our intended interpretation of the column ACCOUNT NUMBERS would be the set of permissible account numbers for the actual business; we would not intend database users to substitute PRODUCT NUMBER for ACCOUNT NUMBER, a substitution that would always make false the expression that the ACCOUNTS table represents.

We try to set up a logical system in such a way that (a) any interpretation that makes all of the axioms true also makes all of the theorems true (*correctness*); (b) all statements of the informal theory can be expressed in the system (*expressive completeness*), and (c) any truth expressible in the system is provable (*deductive completeness*). In other words, we intend that the system's set of true expressions —under any interpretation that makes the axioms true—will be identical to the set of provable expressions or theorems. A system is said to be *truth functionally complete* if, given a set of connectives defined by truth tables, we can express all possible truth tables by various combinations of the given truth tables. As will be seen, this property is extremely important.

When a logical system has more than two truth values, it is said to be a *many-valued logic*. Generally, at least one truth value is called "TRUE" and another is called "FALSE," reflecting our commonly held understanding of these words. The meaning of the other values depends on the logical system's intended interpretation. To understand the role played by these other values better, logicians classify them as being "true-like," "false-like," or neither. A truth value is said to be *designated*, *anti-designated*, or *undesignated*, according to whether it is treated as true-like, false-like, or neither. This subtlety is necessary for many-valued logics in which the notions of "degrees of truth" and "degrees of falsity" may be intended.

Within any logical system, a *tautology* is a wff that always evaluates to a true-like truth value. For example, according to our usual understanding of two-valued logic, "P OR (NOT P)" is always true, regardless of whether "P" is true or false. Similarly, a contradiction is a wff that evaluates to a false-like truth value regardless of assignment of truth values to its components. As with the example of a tautology, according to our usual understanding of two-valued logic, "P AND (NOT P)" is always false, regardless of whether "P" is true or false. Note that the axioms of a logical system are tautologies under the intended interpretation.

Many-valued logic approaches aren't correct for practical database use

In a correct logical system (that is, one with the property of correctness), if every wff that is a theorem is also a tautology, logicians say that the system is *consistent*; the theorems provable from the axioms are always tautologies. Otherwise, they say the system is *inconsistent*. The logician's concepts of consistency and inconsistency are related to, but distinct from, the common notion of being inconsistent (that is, contradictory). Logicians call our common notion *negation inconsistency*.

If every tautology in a correct system is guaranteed to be provable, we say the system is *deductively complete*. A system is deductively complete in a *strong sense* if no wff can be added to its axiom set that would be independent of the other axioms. In fact, we could then prove that a new independent axiom could only make a consistent system inconsistent. We say a system is *decidable* if an algorithm exists by which we could determine whether or not an arbitrary wff is a theorem.

PROPOSITIONAL CALCULUS

You are now equipped to understand the next section. Our task is to consider a few key properties of the propositional calculus. The propositional calculus is deductively complete, negation consistent (it is impossible for a wff and its negation to be true), and decidable. Despite these positive properties, the propositional calculus is expressively weak. In particular, it is not capable of recognizing that two propositions share a common subject. Deduction involving statements of this nature must be considered outside the intended interpretation. For example, the Greek Stoics recognized the problem with the following invalid argument, called The Nobody:

Premise: If someone is here, then he is not in Rhodes.
Premise: Someone is here.
Conclusion: Therefore it is not the case that someone is in Rhodes!

This line of reasoning would be valid, of course, if we replaced the word "someone" by the name of an individual, say "Ted." But the word "someone" is ambiguous: It refers to a specific individual part of the time and to some nonspecific individual the remainder of the time. The propositional calculus cannot help us determine what is wrong because it has no way of representing the concept of propositions that share a common subject.

PREDICATE CALCULUS

The *first-order predicate calculus* is an extension of the propositional calculus which, among other things, is intended to handle such concepts. In order to grapple with the problem, it adds the notion of arguments (formally called "predicate variables"), much as algebra extends the concepts of arithmetic by adding variables. An argument is interpreted by assigning it a particular value from a domain of possible values. (We have already seen such arguments in the description of a table representing a truth-valued expression.) A *predicate* is a statement that the argument possesses a certain property; a predicate without uninterpreted arguments performs essentially the same function as a proposition, and is truth-valued. A predicate may have zero or more arguments, and these arguments may be shared among numerous predicates. Note that all occurrences of an argu-

ment must uniformly take on the same value for any given interpretation. Thus, in the compound predicate "x is red AND x is angry," it is not permissible to replace the first "x" with "Joe" and the second "x" with "Jim."

Given arguments, it is possible to introduce *quantifiers.* Two quantifiers of the first-order predicate calculus are particularly important: the *existential* (a claim that at least one value of the argument has the specified properties) and the *universal* (a claim that all values of the argument have the specified properties). The first-order predicate calculus, unlike the propositional calculus, is capable of handling infinite domains, such as the domain of natural numbers. The existential "EXISTS x" may be thought of (but only informally) as the propositional connective "OR" iterated over the possibly infinite domain of the argument x and the truth value "FALSE." Similarly, the universal "FORALL x" may be thought of (informally) as the propositional connective "AND" iterated over the possibly infinite domain of the argument x and the truth value "TRUE."

Note that any attempt to write an algorithm for "EXISTS" over infinite domains must fail, since the evaluation might never terminate (we might never find the one value that makes the predicate true). Likewise, any attempt to write an algorithm for "FORALL" must fail, since the evaluation cannot terminate if FORALL is to range over a variable having an infinite domain (we might never find the one value that makes the predicate false). Therefore, it is important to understand that the predicate calculus "EXISTS" and "FORALL" quantifiers are not in general identical to simple iterated propositional "OR" and "AND," respectively. This substitution is permissible only when the number of possible values each quantified argument can take is guaranteed to be finite.

The first-order predicate calculus is deductively complete (although not in the strong sense) and consistent. It is not decidable; that is, no algorithm exists for determining whether a wff is a first-order predicate calculus theorem or not. However, if a restricted

Insist that RDBMS vendors meet these logical objectives

version of the first-order predicate calculus is created in which only a finite number of arguments is possible in any expression, and in which the domains for these arguments are guaranteed to be finite (that is, have fewer than some given—possibly very large—number of elements), then this finite version is decidable.

Note that these restrictions change the quantifiers' intended meaning and reduce certain rules of inference to the corresponding rules of inference for the propositional calculus. This fact will be important in the discussion that follows. Although we commonly say that the relational model is built on the first-order predicate calculus, in practice any implementation of the relational model will be more like the finite version described here (that is, at any point-in-time, any real database will have a finite number of tables, columns, domains, actual values, and so on).

IMPLEMENTATION

When designing a relational database, we define a set of predicates (the defining predicates for tables), which we call *relation predicates.* A permissible row in a table has values that satisfy the appropriate domain constraints and relation predicate. Each permissible row in these tables represents a true instance of the relation predicate. The resulting set of true propositions may be understood as the system's axioms. In effect, we assert that the rows represent the true instances of the relation predicates; they are the system's intended interpretation.

The normalization process is an algorithm intended to remove redundancy given the relational operations; that is, to make certain that any table predicate is not actually composed of multiple, independent relation predicates. The process thus contributes to (but does not guarantee) the axiom set's independence. A table's attributes may be understood as arguments, which are functions over the defining domains.

By establishing a set of database domains (with domain constraints), we effectively constrain the universe of discourse. Domains, along with column and table constraints, are used to implement the relation predicate for each table. As such, if the domains, in conjunction with the relational operators, do not suffice to express the facts of interest regarding the application, the system cannot be expressively complete.

When certain rows are inserted into a particular table in the database, we are making the implicit claim that the substitution of these values for the appropriate arguments in this table's relation predicate results in a fact (that is, true proposition). We may regard each such row as a premise from which conclusions may be drawn using the formal axioms and rules of inference. If a row could exist in a given table, but does not (that is, the values are legal and would otherwise satisfy the relation predicate), the meaning of this absence depends on whether we want to take a closed- or an open-world interpretation.

The *closed-world interpretation* states that both the DBMS and the user may infer that the predicate corresponding to an absent row is "FALSE." The *open-world interpretation* states that the status of the predicate for that same row is "FALSE OR UNKNOWN." For example, suppose that we have a table MANAGERS with columns EMP# and MGR#, having the relation predicate, "The manager of employee number EMP# is employee number MGR#," and that the row containing <368, 126> does not appear in the table. The closed-world interpretation says that we may assume that the statement, "The manager of employee number 368 is employee number 126" is "FALSE." By contrast, the open-world interpretation says that we can only assume this statement is "FALSE OR UNKNOWN." For definiteness, in the remainder of this article I will assume the closed-world interpretation.

When you write a query, you

are attempting to produce a wff (a predicate in its own right). It is the parser's job to verify that the query you write is a wff—that it is syntactically correct. Think of the optimizer as using the rules of inference, various axioms, and various provable theorems to produce a set of equivalent wffs, each of which uses only operations with associated physical access methods. Each of the rows returned by the DBMS represents a tuple of values which, on proper substitution into the predicate, result in propositions that evaluate as true (that is, facts) in the logical system. The result set is the system's provable theorem (see Figure 4). The important point to remember is: Whenever you write a multistatement transaction or application, issue a sequence of decision-support queries, or embed a subquery in a query (or decompose it into a sequence of queries), you are using the axioms and rules of inference of the logical system to prove theorems!

Given the familiar parts suppliers database, prove the theorem:

S	S#	SNAME
	S1	Smith
	S2	Jones

P	P#	PNAME
	P1	Nut
	P2	Bolt

SP	S#	P#	QTY
	S1	P1	300
	S1	P2	200
	S2	P1	300

Theorem: The quantity (QTY) of the part (PNAME) named 'Nut' available from supplier named (SNAME) Smith is 300.

Proof:

1. By axiom (the row in S):
SELECT S# FROM S WHERE SNAME = 'Smith';
2. By axiom (the row in P):
SELECT P# FROM P WHERE PNAME = 'Nut';
3. By axiom (relation predicate for SP):
SELECT QTY FROM SP WHERE (TRUE) AND (TRUE);
4. By substitution:
SELECT QTY FROM SP
WHERE S# = (SELECT S# FROM S WHERE SNAME = 'Smith')
AND P# = (SELECT P# FROM P WHERE PNAME = 'Nut');
5. By substitution (from the rows in 1 and 2):
SELECT QTY FROM SP WHERE S# = 'S1' AND P# = 'P1';
6. By substitution (the row in SP): 300

FIGURE 4. *Querying as proving a theorem.*

OBJECTIVES

It is clear that certain properties are desirable of any logical system on which a DBMS is based. In particular, the 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, we can understand each of these objectives as follows:

☐ *Familiar.* The common understanding of the truth values, connectives, rules of inference, and accepted tautologies should remain valid. That is, the user should not have to learn an unfamiliar or nonintuitive logical system, which contains surprising theorems and tautologies or which denies commonly held rules of inference so that usage errors are 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.* The evaluation of a query (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, there will be a truth-valued expression to determine whether the fact is represented in the database.

☐ *Expressively complete.* All queries that are meaningful in the context of the application 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 would have the advantage that a query could be checked via an algorithm to determine if it were a tautology (since every theorem in a consistent system is a tautology; in this case, every row would satisfy the predicate), a contradiction (in which case no rows could ever satisfy the predicate), or neither.

LOGIC AND THE DATABASE

Hopefully, Part I of this series will have made the relationship between formal logic and databases a bit clearer for database practitioners. I have stated the goals that make using a logical system as the basis of database management advantageous, and presented the desirable properties of this logical system in database terminology. At the very least, I would hope that database professionals will now be able to use some of these logical concepts in evaluating a relational DBMS's strengths and weaknesses. While errors of implementation are sometimes to blame, the cause of many performance, data integrity, and maintenance problems lie in much more serious design flaws involving the failure to capitalize on the logical foundation of relational theory. With a little practice, logical concepts will prove useful in identifying such database and application design problems. You can start by insisting that RDBMS vendors meet the logical objectives outlined here.

In Part II of this series, I will examine many-valued logics in the light of these objectives. We will find these logics lacking, and therefore unsuitable for representing and manipulating partial knowledge in a database. ■

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 and Tina Turner for the abuse of their song titles.

NOTES & REFERENCES

1. By "null," we mean an argument value placeholder—not a value—that typically forces the relation predicate to be neither "TRUE" nor "FALSE." Note that it is distinct from the "UNKNOWN" truth value and can be of various kinds ("applicable but unknown," "inapplicable," and so on). SQL's NULL is a particularly bad implementation of such a placeholder.
2. We use BI-IMPLIES in place of the usual logical EQUIVALENTS to improve readability.
3. Suppes, P. *Introduction to Logic*, Wadsworth, 1957.
4. 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.
5. Codd, E. F. "Extending the Database Relational Model to Capture More Meaning," reprinted in *Readings in Database Systems*, M. Stonebraker, ed., Morgan Kaufmann, 1988.
6. Delong, H. *A Profile of Mathematical Logic*, Addison-Wesley Publishing Co. Inc, 1970.
7. Massey, G. *Understanding Symbolic Logic*, Harper & Row, 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."