Handling Missing Data by Using Stored Truth Values

G. H. Gessert
Director, Corporate MIS
Primerica Corporation
300 St. Paul Place
Baltimore, MD 21202
Telephone 301-332-3993

Abstract

This paper proposes a method for handling inapplicable and unknown missing data. The method is based on: (1) storing default values (instead of null values) in place of missing data, (2) storing truth values that describe the logical status of the default values in corresponding fields of corresponding tables. Four valued logic is used so that the logical status of the default data values can be described as, not just true or false, but also as inapplicable or unknown. This method, in contrast to the "hidden byte" approach, has two important advantages: (1) Because the logical status of all data is represented explicitly in tables, all 4-valued operations can be handled via a 2-valued data manipulation language, such as SQL. Language extensions for handling missing data (e.g., "IS NULL") are not necessary. (2) Because data fields always contain a default value (as opposed to a null value or mark), it is possible to do arithmetic across missing data and to interpret the logical status of the result by means of logical operations on the corresponding stored truth values.

Introduction

This paper proposes an alternative to the currently prevalent "hidden byte" method of handling missing data. The proposed method can be described in two parts: structure and function

Structure

For each *data table* that may contain missing data in non-key fields, there is a companion table, called a *logical status table.* The logical status table corresponds to the data table, row for row and field for field. Both tables have the same values in key fields. They differ in non-key fields: where the data table has data values, the logical status table has truth values. Both the data table and the logical status table are, however, base tables in the traditional sense and neither contains null values.

The data table records refer to "outside world" entities by names (or definite descriptions) contained in the primary key fields of each record. The logical status table records also refer to entities, but the entities that they refer to are the records in the corresponding data table. Logical status table records refer to data table records by the obvious expedient of using the primary key of the data table record as the primary key of the corresponding logical status table record. Each non-key field in a data table record contains a value that describes the entity named by the primary key. The corresponding field in the logical status table record, contains an entry that describes the logical status of the corresponding data table value. All non-key fields of the logical status table have the same domain, four truth values.

Four truth values are used to provide additional values that describe the logical status of missing data. The additional values can be used to describe data as, not just true or false, but also as inapplicable or unknown. The truth values are described below. The full significance of the rank ordering of the truth values will be described later. For the moment, consider the ranking as placing the truth values in order of "least to most true" or in order of the possibility that the data value could be true:

1. Not Applicable, written NA or 0. An NA(0) in the truth value field of a logical status table means that the corresponding data field value: (1) is a default value, (2) is 0 or a value that is not in the domain of the data field, and (3) is not applicable to the entity. The data value is neither true, nor false, nor even potentially (Maybe) true of the entity in

question. More precisely, the category of description, which the data field represents, does not apply to the entity named by the primary key. For example, suppose there is a data table referring to parts manufactured by a given company, and that there is a field called COLOR in the table heading or schema. If a record in this table refers to a software package, which has no color, a default value (for example, "none") would be placed in the data table's COLOR field, and an NA would be placed in the logical status table's COLOR field to indicate that the category COLOR does not apply to this part.

Note that NA is distinct from (and in a sense "prior to") all the other truth values in that all the others presume applicability, i.e., that the data value is in the domain.

2. Applicable and False, written AM or 1. This means that the data value: (1) is an input value (not a default), (2) is in the field's domain (is applicable), but (3) is a false description of the entity named by the primary key. This truth value might be used, for example, to handle data that failed to meet validation criteria.

3. Applicable and Maybe True, written AM or 2. This means that the data field value: (1) is a default value, (2) is in the domain of the data field (is applicable), and (3) may be a true or a false description of the entity named by the primary key. That is, the real value is unknown, but the default may turn out to be a true description of the entity.

Note that AM is between AF and AT, in that the data value may be either true or false.

4. Applicable and True, written AT or 3. This means that the value in the data field: (1) is an input value, (2) is in the domain (is applicable), and (3) is a true description of the entity named by the primary key.

Function

An input field may be missing, for two widely different reasons:

1. Oversimplification. This occurs when, in designing the database, we have not made all the distinctions that we could have made; that is, we have not defined all the types of entities (tables) that we could have defined. Thus, not all the entities of a given type (records in a given table) will have all the properties or attributes (fields) that are associated with the entity type (table). For some entities, some of the attributes will not apply. For example, if we have not distinguished between apartment dwellers and home owners, some addresses may not have an apartment number. In what follows, we will refer to this type of missing data as inapplicable missing data. This type of missing data is associated with the truth value, NA.

2. Partial Information. This occurs when not all values for all fields are known at the time the record is added to the database. For example, a new employee's office telephone number may not be known when the employee's record is added to the personnel system. We will refer to this type of missing data as unknown missing data. This type of missing data is associated with the truth value AM.

Having a systematic way of handling both types of missing data is of great practical importance. Handling inapplicable missing data is important to pragmatic database design, because it provides us with a way to be modest. We can admit that we have not made every distinction that can be made and thereby avoid having to develop data structures and systems in which 80% of the work is done to accommodate 20% of the cases. Handling unknown missing data is important, because it is very common. In addition, unknown missing data, unlike inapplicable missing data, typically requires some follow-up activity. Hence, to simplify programming and system maintenance, it is important to have a uniform way of handling unknown missing data (once) in the database, as opposed to (many times) in each application program. Analogous remarks apply to the treatment of invalid (AF) data.

The functioning of the proposed method of handling missing data can be viewed, conceptually, as a facility (or application for handling missing data), which is imbedded in the database "back-end" and provides services to all "front-end" applications. The facility links default data field values and their

corresponding truth values by means of Add, Update, and Delete rules. When input for a given field is marked as not applicable, the missing data handling facility stores the appropriate default in the data field and stores the truth value, NA, in the corresponding truth value field. Note that this approach distinguishes clearly between the input, the stored data value, and the logical status of the stored value. Similarly, when the input is marked unknown, the application stores the user's selected default domain value in the data field and stores the truth value AM in the logical status table. This implies that the data definition language provides support for domains and in particular provides for the specification of default values for each field.

Consider the following variant of the industry standard example (CJD). Suppose that the data comes in as follows:

|S(Suppliers)

S#	SNAME	STAT	CITY
s1	Smith	?	London
s2	Jones	10	Paris
s3	Balke	30	!
s4	Clark	?	London
s5	Adams	30	Athens

The "!" and "?" symbols, in this case, refer to input that is inapplicable and unknown, respectively.

The above input might be stored in the data table (S) as follows:

|S (Suppliers)

S#	SNAME	STAT	CITY
s1	Smith	10	London
s2	Jones	10	Paris
s3	Balke	30	None
s4	Clark	10	London
s5	Adams	30	Athens

In this case, inapplicable CITY values default to "None" and unknown STATus values default to "10".

Corresponding to the Supplier data table (S), there would be a Logical Status table (LS) that might look something like this:

|LS

S#	LSNAME	LSTAT	LCITY
s1	3	2,AM	3
s2	3	3	3
s3	3	3	0,NA
s4	3	2,AM	3
s5	3	3	3

Each row of the base table (S) can be understood as a shorthand way of expressing a series of assertions (statements) about the entities named by the primary key. The first row of the data table (S) would correspond to the assertions:

s1's name is Smith.
s1's status is 10.
s1's city is London.

The entries in the Logical Status table (LS) state that:

"s1's name is Smith." is AT.
"s1's status is 10" is AM.
"s1's city is London." is AT.

The Logical Status table also tells us that the sentence "Supplier s3's city is None." has the truth value, NA. That is, supplier s3 cannot be said to be located at a single city.

There are several benefits of representing missing data using defaults and stored truth values, instead of using a "hidden byte" to designate a stored value as null or marked. The term "marked", is used in the sense advocated by Codd (EFC). A mark, like a null is implemented using a "hidden byte", which is essentially a flag. If the null flag is set "on" the meaning of the field is changed; it no longer contains a data value; it contains a meta-data value indicating the logical status of the field. A mark is essentially an extended null, where the flag byte may take on an extra value so that it is possible to distinguish between an unknown "null" and an inapplicable "null". Codd argues, however, that neither nulls nor marks may be treated as data values. The benefits of the defaults-and-truth-values approach fall into three main categories:

1. Because the logical status of the data values is represented in a table, these values can be

manipulated via standard SQL. It is not necessary to complicate the data manipulation language by defining operators, such as "IS NULL" that operate on marked fields. If we wish to represent both unknown and inapplicable missing data using a "hidden byte" approach analogous to a null value, it would be necessary to extend the "IS NULL" operator to cover the added conditions "IS INAPPLICABLE" and "IS UNKNOWN".

2. It is possible to do "what if" calculations on missing data. Note that the logical status of a field depends only on the truth value in the corresponding logical status field. Any (default) data value that is AM can be replaced with any other data value that is in the domain, without altering the logical status of the field and without altering the logical status of any calculations that are done on the field. Similarly, for any NA field the truth value will be unchanged if the data value were replaced by any other value that is not in the domain. For example, one could calculate a total over all AT and AM fields. The logical status of the answer (as discussed below) would be possibly true. It might be useful to calculate a minimum and maximum possible total. This could be done easily by defining a view in which all the AM fields were replaced by the minimum or maximum values in the domain. As a further illustration of the previous point (1), this hypothetical view could be created with standard SQL.

3. It is possible to do arithmetic on missing data. Because data fields always contain a default value (rather than a null value or mark) one can always do arithmetic, in the usual way, on fields that include missing data. In particular, it is not necessary to complicate the data manipulation language by defining arithmetic on null values or marks.

For example, suppose we have a table that represents invoice detail lines, i.e., the items ordered. A field Q represents the quantity ordered, P represents the price, and F represents a fee. The rationale for the three fields is that service items may have a fee but not a price or quantity. Suppose, for example, we have an item where the values for (Q, P, F) are:

(0, 0, 5) and (NA, NA, AT)

We want the cost (C = Q*P+F) of the item to be 5, not a null (or an I-mark, indicating that the result is inapplicable). If we use marks in the data field, we might represent the same situation as:

(I, I, 5)

In this case, it is hard to escape a cost calculation of:

(I*I)+5 = I+5 = I

We may also want to make use of defaults in AM fields. For example, if the quantity is omitted (i.e., AM) but the unit price is 5 and AT, and the fee is 0 and NA; we probably want the quantity to default to 1, and want the cost to work out to 5. The final section shows how calculations like these can be handled.

Implications of the Approach

What follows describes the implications of the proposed approach to handling missing data with respect to the key concepts of the relational model.

Data Structure

One of the chief virtues of this approach is that it requires no fundamental modification of the relational data structure. Both the data table and the corresponding logical status table are base tables in the traditional sense. That is, it is not necessary to alter the definition of a domain or to extend the concept of a column in order to take into account the presence of flag bytes that indicate nulls or marks. The representation of missing data is logically clean in that missing data is handled simply as data about data, i.e., meta-data. The meta-data like all other data is represented as tables.

Entity Integrity

Entity integrity applies to both data tables and logical status tables. That is, keys must be unique and therefore can contain no missing fields. In fact, given that the keys of a data table and its corresponding logical status table are identical, entity integrity of the data table implies entity integrity of the logical status table. All primary keys are, of course, AT.

Referential Integrity
Because the representation of missing data is logically clean, it is easy to state the concept of referential integrity: If a data field is a foreign key, FK, then if its truth value is AT, there must be a record in another table with a matching primary key, PK, where FK=PK.

Update constraints
Additional integrity constraints should be enforced with this method of handling missing data. The constraints may be classified by truth value:

NA(0) - Not Applicable. The truth value field may not be updated without updating the data field. Since the data field contains a value that is not in the domain, if the truth value were updated to AT, the database would not be consistent.

Similarly, the data field may not be updated to a domain value without a corresponding update to the truth value field.

AF(1) - Applicable and False. False data might be stored in the database in a situation where the input failed validation criteria, but we still considered it advantageous to store the data as-is, rather than storing a default. This would suspend further activity on the field until the error is resolved, either by correcting the input or the validation criteria. For example, if 500, rather than 50, incorrect zip codes appear in a single day's transactions, it is likely that the error is in the editing table. In this case we want to accept the data provisionally but mark it as false. To correct the errors we do not have to re-input the data or even process a suspense file; we need only correct the validation table and change the logical status of the AF fields to AT. The key point is that handling invalid data can be accomplished using the same mechanism used for handling missing data. It is obvious that the two processes are closely connected functionally, but I know of no other proposal in which the two processes are connected formally.

If false data is stored, the following rules would apply. Since the data field, by definition, contains a domain value, an AF value may not be consistently updated to an NA value. The AF may, without contradiction be updated directly (i.e., without updating the data value) to an AM. But if an AF is update directly to an AT, the pre-update value will contradict the post-update value.

AM(2) - Applicable and Maybe True. Since the data field contains a domain value, an AM value may not be consistently updated to an NA value.

An AM value may be updated directly to an AF or an AT without contradiction. For example, we can say without contradiction both:

"The supplier s1's city is London" maybe true.
"The supplier s1's city is London" is true.

AT(3) - Applicable and True. Since the data field contains a domain value, an AT value may not be consistently updated to an NA value. An AT value may be directly updated to an AM value.

Note that the ranking of truth values from 0 to 3 can be understood precisely in terms of these constraints. NA (Not Applicable) is ranked lowest because it cannot be directly updated to any of the other (applicable) truth values, without leading to an inconsistency in the present state of the database. NA is thus anchored as the lowest, "least true" truth value. AM is ranked between AF and AT because it may be directly updated to either AF or AT without contradicting the previous state of the database.

Operators: Logic, Comparison, and Arithmetic
In what follows, it will be helpful to bear in mind that the relational operations, (SELECT, PROJECT, JOIN) are essentially 2-valued. 4-valued logic comes into play only we have to evaluate the truth value of a compound expression, the components of which may be 4-valued. For example, we have to consider 4-valued logic when the expression "STATUS = 10" or the expression "CITY = 'PARIS'" could be 4-valued and we have to evaluate the truth value of the expression "STATUS = 10 **AND** CITY = 'PARIS'". The **AND** in the above expression is a 4-valued **AND** rather than a two valued AND. Having done the 4-valued evaluation, however, we have to convert the

result into a 2-valued expression so that, based on the value, a record may be unambiguously selected to be or not to be a member of the result set. That is, while the interactions of four-valued logic may be complex, four valued logic operates only in a very limited realm of data manipulation.

Arithmetic, in the traditional sense, is defined only for fields that are applicable and true, i.e., where the logical status of the field is AT. For example, saying that A = B, in the conventional sense, applies only if the logical status of both A and B is AT. In the approach advocated here, because the data portion of a numerical field always contains at least a default value, it is always possible (if not meaningful in exactly the traditional sense) to do arithmetic or numerical comparisons on two fields. Furthermore, we can do ordinary arithmetic, without defining arithmetic on nulls or marks. The logical status of the result, however, will depend on the logical status of the individual fields and the rules for logical operations.

We will not actually define arithmetic on fields that are not AT, but we will see that it is sometimes useful do something similar. We can gain a similar effect by first applying a logical operator that transforms NA or AM values into values that are AT and then doing arithmetic on the result. To this end, it is necessary first to define the truth functional operators.

Logic

The following table illustrates the logical relationships among the truth values.

```
  |- 0 Inapplicable                0
  |
--|                  | - 0 False   1
  |                  |
  |- 1 Applicable -  | - 1 Maybe   2
                     |
                     | - 2 True    3
```

The syntax of the logic is arranged to reflect the semantics of the truth values, especially the rank ordering of truth values as defined in the first section. The distinctive feature of this particular 4-valued logic is that the "least true" value (0 or NA) is interpreted as corresponding to the concept "Not Applicable", rather than "Applicable but False". The rationale for assigning the lowest rank to Not Applicable (0) is evident from the way in which the lowest (0) value functions in the **AND** and **OR** operations. This point will be discussed more fully in connection with the function of the **AND**, **OR** and **INVERSE** operators.

The truth functions are defined, in the standard way. The term "standard" is used in the sense defined by Rescher (NR). If /a/ represents the truth value of a field "a" then:

p **AND** q = min (/p/,/q/),
p **OR** q = max (/p/,/q/), and
INV(p) = (3-/p/).

An additional **NOT** operator is defined below. Note that if the Not Applicable (0) value is dropped; **AND**, **OR**, and **NOT** correspond to the 3-valued case embodied in the SQL standard.

The following are the truth tables for **AND** and **OR**:

AND

	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	0	1	2	2
3	0	1	2	3

OR

	0	1	2	3
0	0	1	2	3
1	1	1	2	3
2	2	2	2	3
3	3	3	3	3

Two operators are defined as analogs to the 2-valued "not" operator. The following table defines the two "negation" operators along with an identity operator (**ID**).

p	**INV**(p)	**NOT**(p)	**ID**(p)
0	3	0	0
1	2	3	1
2	1	2	2
3	0	1	3

Both INV and NOT are "negation" operators, in the sense that:

ID(p) = NOT(NOT(p))
ID(p) = INV(INV(p))

The **INV** operator is a "true" inverse in the sense that, in terms of **INV**, **AND** and **OR** are dual operations. That is, De Morgan's law holds when cast in terms of **INV**:

p **AND** Q = **INV**(**INV**(p) **OR** **INV**(q))

The **NOT** operator is defined to provide a convenient analog to the 3-valued negation operator. The **NOT** operator also has the convenient property that it propagates the auxiliary truth values (0 and 2) unchanged. Note that the **NOT** operator is just the inverse (**INV**) rotated by one position (i.e., the inverse, plus 5, modulo 4).

To return to the interpretation of the truth value NA, the point of assigning the lowest rank (0) to the Not Applicable condition can be seen in relation to the operation of computing **OR** using max(/p/,/q/) and **AND** using min(/p/,/q/). The effect of regarding Not Applicable (0) as "lower than" Applicable and False, is to render the Not Applicable (0) condition ineffective in computing **OR** (the maximum) but to render the Not Applicable (0) condition "contaminating" when computing **AND** (the minimum). That is, a conjunct that is NA, contaminates the whole conjunction, and renders it NA, because NA(0) is always the minimum value.

As discussed above, the relational operations are essentially binary operations: a record is either selected or it is not. Because of this, logical operators that map a 4-valued representation of a situation into a 2-valued representation of the same situation take on special significance. Four unary operators are defined:

p	**NA**(p)	**AF**(p)	**AM**(p)	**AT**(p)
0	3	1	1	1
1	1	3	1	1
2	1	1	3	1
3	1	1	1	3

The following operators also map a 4-valued description of a situation into a 2-valued description of the same situation. They are convenient in connection with arithmetic and comparison operators.

p	**POS**(p)	**NF**(p)
0	1	3
1	1	1
2	3	3
3	3	3

These operators can be interpreted as:

POS(p) means that p is possibly true.

NF(p) means that p is not false.

These operators are useful in arithmetic computations because we frequently want to include in aggregates, values that are possibly true or all values that are not known to be false.

Syntax Preserves Semantics

Note that the **AND** and **OR** operators preserve the semantics of the additional truth values.

Consider the conjunction 'A **AND** B'. A single field, A, being NA means that the truth value alone cannot be consistently updated to make the field applicable: AF, AM, or AT. That is, since the value in the data portion of A is not in the domain, it cannot be a true (AT) or false (AF) or even a maybe true (AM) description of the entity named by the primary key. We can ask, when a conjunction 'A **AND** B' is NA, is it NA in the same sense? The expression 'A **AND** B' would not be NA in the same sense if it were possible that an update to only the logical status portion of a component field could make the conjunction applicable. According to the truth table, A **AND** B is NA only when one field is NA. As long as one field is NA (and it cannot be updated to be applicable), it will contaminate the whole conjunction -- no single update can make the conjunction applicable.

Consider also the conjunction, 'A **AND** B' relative to the meaning of AM. In the case of a single value A, the meaning of A being AM is that an update can change the truth value of A from AM to AT (and leave the data value unchanged). In this sense, A might be true (AT). We can ask, if the conjunction 'A **AND** B' is AM, does it mean the same thing, that an update might make the expression 'A **AND** B' AT? Yes. According to the truth tables, if the conjunction is AM then one of the two conjuncts is AM and the other is AM or AT. Suppose, for example, that it is A that is AM and B is AT. Then an update of A to AT would also make 'A **AND** B' AT.

Comparison

This section will discuss the comparison operation (=) in relation to the 4-valued logic

defined above. In this section, we will refer to a data value of a field A as D(A), and to the numerical value of a logical status of field A as L(A). The logical status of an expression X will, as above, be referred to as /X/. If an expression is a single field, A, then L(A) is equivalent to /A/. The notation L(A) is used to emphasize that the truth value is the truth value of a field and is part of a pair of values (D(A), L(A)). The expression /A/ is used to emphasize that the truth value is being used as part of a logical computation.

As the concept is used here, comparison is an operation that maps pairs of numbers into the truth values AT or AF. Intuitively, saying that A = B, has its conventional meaning only presupposing that A and B are both applicable and true (AT), that is, only if /A **AND** B/ = AT. If A or B is not AT, the comparison is meaningless. That is, if both fields are not AT, applying the 2-valued = operator to two fields would, strictly speaking, return an error, just as an error would result from an attempt to add two fields that are not both numeric.

What follows will propose a 4-valued extension of the 2-valued "=" and will then propose two further extensions of that concept, which will prove useful for comparing fields with missing data. For concreteness and simplicity, what follows uses the specific comparison operator that designates equality as a place holder for the general operation of comparison. The remarks apply equally to the other comparison operators (<, <=, ~=, >, >=).

Like the 2-valued equivalence operator (=), the 4-valued equivalence operator "=" (written as bold =) is also defined as an operator that maps pairs of fields into the two truth values, true (AT) or false (AF). The truth value of A = B is AF unless two conditions are met:

1. /L(A) **AND** L(B)/ = AT

2. /D(A) = D(B)/ = T = AT.

Other comparison operators are defined in an analogous way. The justification of this definition is that it preserves the sense of the 2-valued equivalence operator: the result is bi-valued (can be used as a basis for selecting records) and it is presumes that both operands are applicable and true. This 4-valued equivalence operator may also be viewed as a strong form of equivalence. That is, we might define other equivalence operators that were weaker, in the sense that they do not require that both operands be AT. In fact, 4-valued logic suggests several interesting ways in which this definition can be extend by relaxing the conditions.

In 4-valued logic, in addition to the strong form of truth, **AT**, there are also two weaker forms of truth, Possibly True (**POS**) and Not False (**NF**). As defined above, a field is Possibly True if it may be true (AM) or is actually true (AT). A field is Not False if it is not applicable (NA), it may be true (AM) or it is actually true (AT).

Using, **POS** and **NF** we can define two extended equality operators, in addition the strong form of equality (=). Put another way, in addition to the concept of actually equal (=), we can define the concepts of possibly-equal and not-unequal.

POSSIBLY EQUAL (**PQUAL**):

A **PQUAL** B if and only if **POS**(A) AND **POS**(B)

NOT-UNEQUAL (**NUQAL**):

A **NUQUAL** B if and only if **NF**(A) AND **NF**(B)

Analogous remarks hold for the other comparison operators. Note that the data values in the fields A and B (i.e., the condition D(A) = D(B)) do not come into play, because if a field is not AT, the value is a default. If a field is AM, the entry in the data portion of the field is equivalent to any other value in the domain. If a field is NA, the value is equivalent to any value that is not in the domain (0 if the field is numeric).

The point of defining these operators is, of course, to use them later in Select, Project, and Join operations. It is important to note, however, that these operators are a conceptual convenience. If the logical status information is represented explicitly in tables, it is not necessary to make the data manipulation language 4-valued to implement these

operations. Given any query cast in terms of these operators, a logically equivalent query can be written using a 2-valued data manipulation language. SQL would be adequate if it had, in addition to the aggregates MIN and MAX,the arithmetic functions min and max.

Arithmetic

Like comparison operations, arithmetic operations, in their ordinary sense, presuppose that the fields are applicable and true. Otherwise, arithmetic operations are meaningless. What follows will use the addition symbol (+) both as symbol for addition and as a place holder to stand for any arithmetic operator.

As in the case of comparison, however, we can define a 4-valued analog (+) of the 2-valued addition operator (+) and can define useful extensions of addition by making use of the notions of Possibly True, and Not False (**POS**, and **NF**). The 4-valued analog to the conventional addition operator may be defined:

A + B = (D(A) + D(B), **AT**(A) **AND** **AT**(B))

where, as above, the data value and its logical status are represented as the ordered pair (D(p), L(p)). For example, of (5, AT) + (3, AM) = (8, AM).

We can also define a Possible Sum (**POSSUM**(A,B)) and a Not False Sum (**NFSUM**(A,B)) as:

POSSUM(A,B) = (D(A) + D(B), **POS**(A) AND **POS**(B))

NFSUM(A,B) = (D(A) + D(B), **NF**(A) AND **NF**(B))

Analogous extensions can be made to the other arithmetic operators. This extended arithmetic can be used to perform some of the practical calculations on missing data that were noted in the Introduction. The means of performing these calculations will be illustrated in the following discussion of the SELECT operator.

Relational Operators

This section discusses the implications of the proposed method of handling missing data with respect to the relational operators: Select, Project, and Join. The discussion will focus on showing how the equivalent of 4-valued operations can be realized in standard SQL, by performing 2-valued operations on the logical status data. The discussion of the SELECT operation will, however, illustrate arithmetic on missing data.

Select

The select operator is realized in SQL as a statement of the form:

SELECT * FROM *tablename*
WHERE *condition*

Suppose that we have available a view that is a join of the above tables S and LS.

|SxLS

S#	SNAME	LSNM	STAT	LSTAT	CITY	LCITY
s1	Smith	3	10	2	London	3
s2	Jones	3	10	3	Paris	3
s3	Balke	3	30	3	None	0
s4	Clark	3	10	2	London	3
s5	Adams	3	30	3	Athens	3

The equivalent of the standard 2-valued theta-select is realized in the 4-valued context as follows:

SELECT *
FROM SxLS
WHERE STAT = 10 AND LSTAT = 3

The comparison operator "=", in this case is 2-valued comparison, not the 4-valued comparison operator. The AND in this example is also the 2-valued AND of ordinary SQL. This WHERE clause is equivalent to the 4-valued WHERE clause "**AT**(STAT=10)". The query could be paraphrased "select all rows where it is applicable and true that the status is 10".

The 4-valued clause that selects the cases where it is possible that the status is 10 (i.e., WHERE STAT **PQUAL** 10) is realized in standard SQL as:

```
SELECT *
FROM SxLS
WHERE (STAT = 10 AND LSTAT = 3)
OR (LSTAT =2)
```

This selection could be paraphrased, "select all rows where it is possible that the status is 10. This would include all the rows where the status is actually 10, plus (UNION, i.e., OR) all rows where there is a default value representing some domain value that could, when updated, be 10. Note that the selection of all those rows where (STAT = 10 OR LSTAT > 1), would yield, in addition to those rows where STAT = 10 and where STAT maybe 10, those rows where (NOT(STAT = 10) AND LSTAT = AT).

The cases where it is not false that the status is 10 (STAT **NUQUAL** 10) can be selected as follows:

```
SELECT *
FROM SxLS
WHERE (STAT = 10 AND LSTAT = 3)
OR (LSTAT = 2)
OR (LSTAT = 0)
```

Arithmetic

As mentioned above, strictly speaking, arithmetic is defined only for fields that are AT. Put another way, we can always do arithmetic on two fields (because at worst they contain default values), but the logical status of the result will depend on the logical status of the component fields. The logical status of the sum will be the same that of the conjunction of the two fields. If the operands are not both AT, the result will be less than AT. In some cases, this is acceptable. Doing arithmetic on missing data is essentially a matter of accepting arithmetic results that are less than true, but not actually false. In some cases, for example, we may want to include default values and calculate a sum that includes values that are possibly true (**POS**) or are not false (**NF**). Suppose we have the table below, which might represent an order for some computer equipment. In this case, Q is Quantity, P is Price, F is Fees and D is discount:.

N	Q	LQ	P	LP	F	LF	D	LD
1	24	3	4	3	0	0	1	3
2	2	3	10	3	0	0	1	3
3	24	3	1	3	0	0	2	2
4	0	0	0	0	23	3	0	0

This sort of table might come about in a situation where:

N	Description	Comment
1	Terminals	No Fee, Discounted
2	Term. Servers	No Fee, Discounted
3	Software	Discount Unknown
4	Installation	No Qty,No Discount

Suppose we want to calculate the cost of each line. Excluding missing data, we would have:

```
SELECT COST = Q*P+F
FROM ORDERxLORDER
WHERE (LQ = 3)
AND  (LP = 3)
AND  (LF = 3)
```

This would yield an empty result table. If we accept a NOT FALSE value, we would get.

```
SELECT NET_COST = Q*P+F
FROM ORDERxLORDER
WHERE NOT(LQ = 1)
AND  NOT(LP = 1)
AND  NOT(LF = 1)
```

To calculate the net cost of each line after discount, we would have to contend with the case where the discount on the software is unknown. We could include the default discount in the cost by accepting the rows where the discount is possibly true. The result would not be a true value (in the strong sense) because the actual discount is not known, but it would not be a value that was known to be false, and it would be a value that might turn out to be true.

We would certainly accept values that are less than actually true when we are doing "what if" calculations. We could, for example, estimate the total discount by including those values that are Possibly true (POS).

```
SELECT SUM(D)
FROM ORDERxLORDER
WHERE LD>1
```

We can bound this estimate by changing all the AM (2) defaults to maximum (or minimum) values and then repeating the summation. Note that the estimate will be a good one in the sense that, it will include those values that are actually true and exclude those values where it is known that no discount applies.

4-valued AND, OR, and NOT

4-valued compounds could be realized in standard SQL by making use of the fact that the 4-valued conjunction "p **AND** q" is equivalent to "min(p,q)". We would, of course, have to enhance SQL by adding the arithmetic function MIN(A,B). In doing so, however, we would not be making SQL 4-valued. The 4-valued **OR** can be represented as a maximum of two logical status values. The inverse **INV**(L(A)) can be represented as (3-L(A)). The 4-valued **NOT**(L(A)) operator can be represented as mod(4, **INV**(L(A) + 5).

Providing an adequate level of support for 4-valued logic in a relational data base management system would require only: providing support for the automatic naming the logical status fields and for appropriately named functions for MIN, MAX, (3-p), and (mod(4,INV(P) + 5)), These functions might, for example, be renamed: AND4, OR4, INV4, and NOT4 respectively.

Project

The central problem in correctly implementing the projection of a table is to eliminate duplicate rows from the result table so that, after the operation, it remains a relation. A row is a duplicate if all its fields are duplicates. But in this case, duplicate fields means having the same value in the logical status field and if that value is AT, having also the same value in the data field. For example, consider the following 4-valued projection of the view given below.

```
SELECT DISTINCT STAT
FROM SxLS
WHERE STAT = 10
```

```
|SxLS                    |
-------------------------
S# SNAME   STAT  LSTAT
-- ------- ----- -----
s1  Smith  10    2
s2  Jones  10    3
s3  Balke  30    3
s4  Clark  10    2
s5  Adams  30    3
s6  Kim    10    3
```

The desired result would be, for example, to eliminate s4 and s6, leaving :

```
STAT  LSTAT
----- -----
10    2
10    3
```

We do not want (by eliminating s1 or s2, in addition to s4 and s6) to get just the row (10, 2) or just the row (10, 3). The desired selecion can be done in standard SQL by simply including the logical status field in the target clause. For example,

```
SELECT DISTINCT STAT, LSTAT
FROM SxLS
WHERE STAT = 10
```

This will always function correctly for the following reason. If the field is AM or NA, all the defaults in the data field will be the same -- there will remain only one distinct occurrence of the field. If a field is AT, duplicates will occur only if there are truly duplicate values in the data field.

Join

The join (theta-join), which is based on a 2-valued comparison, is unaffected by missing data. That is, by definition, the comparison operation (=, for example) applies only to fields that are both AT. No missing data (or false data), with a truth value less than AT (3) will participate in the join.

For example, suppose we have these two tables, and we intend to join on CITY.

S			P		
S#	CITY	LCITY	P#	CITY	LCITY
S1	London	3	P1	London	3
S2	Paris	3	P2	Paris	3
S3	NYC	2	P3	LA	2
S4	Newark	3	P4	Paris	3

The 4-valued equivalent of the 2-valued equijoin would be equivalent to first selecting the rows where LCITY is AT (3) and then doing the standard, 2-valued equijoin over the remaining rows.

S1	London	3	P1	London	3
S2	Paris	3	P2	Paris	3
S2	Paris	3	P4	Paris	3

The logical status of P# is not shown because it must be AT.

The traditional outer join would be implemented by plugging defaults, with truth value AM (2), in the result table for all non-key fields of unmatched tuples. The left outer join, for example, would yield the above rows (from the equijoin) plus the row:

S#	CITY	LCITY	P#	LP#	CITY	LCITY
S4	Newark	3	00	0	LA	2

In this case, it is necessary to include the logical status of the key field, P#. To preserve the semantics, the default for the key field should be a value that is not in the domain, with a truth value of NA (0). We cannot use a default value that is in the domain with a truth value of AM (2), because if we did, we would be adding rows to the result that have duplicate values in the foreign key field, P#. Duplicate P# values being AM (2), however, leads to a contradiction, if we hold to the interpretation that an AM field might actually be true, i.e., that the logical status of the duplicate default P# values might at any time be updated to AT.

Thus outer joins would be implemented according to the following rule. In the missing portion of the join record that represents an unmatched tuple (1) substitute an AM default value if the field is not a foreign key; (2) substitute an NA default if the field is a foreign key.

It is, of course, possible to extend the join operation to include cases where the comparison of join fields (X theta Y) is possibly true (**POS**) or not false (**NF**). The equijoin, for example, would be extended to cases where the join fields are Possibly Equal (**PQUAL**) or Not Unequal (**NUQUAL**).

The **PQUAL**-join is potentially useful because it has properties analogous to the outer join.

The **PQUAL** join, of the two tables in the above example, would include the rows of the equijoin and in addition, the row (S3, NYC, 2) would match all four rows in P, including P3. Similarly, (P3, LA, 2) will match all the rows in S. In particular, (P3, LA, 2) will match (S4, Newark, 3). Thus, the unmatched tuple in S participates in the **PQUAL**-join, because it is a possible match.

Summary

The above analysis of the implications of the proposed approach, demonstrates that it sacrifices nothing that can be achieved by the "hidden byte" approach or by the older approach of simply using default values. The proposed approach, however, offers the particular advantages cited in the first section.

These advantages can be realized principally because the approach is a logically clean method of handling missing data. The semantics are unambiguous and are preserved by the logical operations. Operations of arithmetic, and comparison are simple and not complicated by ad-hoc constructs for treating nulls or marks. The logic of the data manipulation language remains 2-valued and uncluttered by constructs contrived for the sole purpose of handling nulls or marks.

In this approach, a field has only one function, holding a data value. In the traditional approach, a field has two functions: holding a null (or mark), and/or holding a data value. These functions are very different. Codd [EFC, p. 173] is at pains to point out that "The DBMS does not treat marks as if they were data". Yet in the traditional approach, these functions are often not clearly distinguished. If the two functions are distinguished in a particular setting, they are distinguished by the ad-hoc

means of the "hidden byte", which amounts to a flag byte used to indicate which function the field is performing at a given time.

The approach proposed here takes seriously the point that marks indicating missing data cannot be treated simply as data values. They are not treated as data, but they are treated as data about data, or meta-data. This meta-data is, however, stored in tables and can, therefore, be treated in a clear, simple, and unambiguous way.

Acknowledgement

I would like to thank my friends and colleagues who have reviewed this paper, especially Howard Rothenburg and Patricia Rouzer. I would also like to thank Paul Butterworth, and Michael Stonebreaker for their helpful discussion of these issues.

References

[CJD] C.J. Date, "Relational Database: An Overview", Chapter 1 in book entitled Relational Database: Selected Writings, Addison- Wesley, 1986

[EFC] E.F. Codd, "Missing Information", Chapter 8 in The Relational Model for Database Management, Addison-Wesley, 1990

[NR] N. Rescher, Many Valued Logic, McGraw-Hill., New York, 1969