# How To Handle Missing Information Without Using NULL

Hugh Darwen
hd@thethirdmanifesto.com
www.TheThirdManifesto.com

First presentation: 09 May 2003, Warwick University

Updated 27 September 2006

"Databases, Types, and The Relational Model: The Third Manifesto", by C.J. Date and Hugh Darwen (3rd edition, Addison-Wesley, 2005), contains a categorical proscription against support for anything like SQL's NULL, in its blueprint for relational database language design. The book, however, does not include any specific and complete recommendation as to how the problem of "missing information" might be addressed in the total absence of nulls. This presentation shows one way of doing it, assuming the support of a DBMS that conforms to The Third Manifesto (and therefore, we claim, to The Relational Model of Data). It is hoped that alternative solutions based on the same assumption will be proposed so that comparisons can be made.

# Dedicated to Edgar F. Codd 1923-2003

Ted Codd, inventor of the Relational Model of Data, died on April 18th, 2003. A tribute by Chris Date can be found at http://www.TheThirdManifesto.com.

# SQL's "Nulls" Are A Disaster

No time to explain today, but see:

Relational Database Writings 1985-1989
by C.J.Date with a special contribution by H.D. ( as Andrew Warden)

Relational Database Writings 1989-1991
by C.J.Date with Hugh Darwen

Relational Database Writings 1991-1994
by C.J.Date

Relational Database Writings 1994-1997
by C.J.Date

Introduction to Database Systems
(8th edition ) by C.J. Date

**Databases, Types, and The Relational Model:**
***The Third Manifesto***
by C.J. Date and Hugh Darwen

In RDBW 1985-1989: Chapter 8, "NOT Is Not "Not!", Chapter 9, "What's Wrong with SQL", and Chapter 13, "EXISTS Is Not "Exists!", by CJD; Chapter 19, "The Keys of the Kingdom", and Chapter 23, "Into the Unknown", by HD.

In RDBW 1989-1991: "Oh No Not Nulls Again", Chapter 19, "Watch Out for Outer Join", and Chapter 32, "Composite Foreign Keys and Nulls", by CJD; Chapter 20, "Outer Join with No Nulls and Fewer Tears", by HD.

In RDBW 1991-1994: Chapter 9, "Much Ado about Nothing" and Chapter 10, "A Note on the Logical Operators of SQL", by CJD.

In RDBW 1994-1997: Part III, "The Problem of Missing Information" (6 chapters)

# Case Study Example

PERS_INFO

| Id | Name | Job | Salary |
|---|---|---|---|
| 1234 | Anne | Lawyer | 100,000 |
| 1235 | Boris | Banker | ? |
| 1236 | Cindy | ? | 70,000 |
| 1237 | Davinder | ? | ? |

Meaning (a predicate):

The person identified by *Id* is called *Name* and has the job of a *Job*, earning *Salary* pounds per year.

BUT WHAT DO THOSE QUESTION MARKS MEAN???

That predicate is at best approximate. It might be appropriate if it weren't for the question marks. "The person identified by 1234 is called Anne and has the job of a Lawyer, earning 100,000 pounds per year" is a meaningful instantiation of it (by substituting the values given by the first row shown in the table), but "The person identified by 1236 is called Cindy and has the job of a ?, earning 70,000 pounds per year", does not make sense.

What is the data type of Job? What is the data type of Salary? Good questions! In SQL those question marks moght indicate the presence of nulls, in which case the data type of Job might be called VARCHAR(20) and that of Salary DECIMAL(6,0). But NULL isn't a value of type VARCHAR(20), nor of type DECIMAL(6,0).

# Summary of Proposed Solution

1. Database design:
   a. "vertical" decomposition
   b. "horizontal" decomposition
2. New constraint shorthands:
   a. "distributed key"
   b. "distributed foreign key"
3. New database updating construct:
   "multiple assignment" (table level)
4. Recomposition by query
   to derive (an improved) PERS_INFO
   when needed

I also comment on the extent to which the proposed solutions are available in today's technology.

# 1. Database Design

a. “vertical” decomposition

Decompose into 2 or more tables by *projection*

Also known as *normalization*.
Several degrees of normalization were described in the 1970s:
1NF, 2NF, 3NF, BCNF, 4NF, 5NF.

The ultimate degree, however, is 6NF: “irreducible relations”.
(See “Temporal Data and The Relational Model”,
Date/Darwen/Lorentzos, 2003.)

A 6NF table is a key plus at most one other column.

“Vertical”, because the dividing lines, *very* loosely speaking, are between columns. It is fundamental that the table to be decomposed can be reconstructed by joining together the tables resulting from the decomposition.

Ultimate decomposition can be thought of as reducing the database to the simplest possible terms. There should be no conjunctions in the predicates of the resulting tables. Vertical decomposition removes “and”s. (The relational JOIN operator is the relational counterpart of the logical AND operator.) We shall see later that horizontal decomposition removes “or”s.

Note that the proposal does not require the whole database design to be in 6NF.

# Vertical Decomposition of PERS_INFO

CALLED

| Id | Name |
|---|---|
| 1234 | Anne |
| 1235 | Boris |
| 1236 | Cindy |
| 1237 | Davinder |

Meaning:

The person identified by *Id* is called *Name*.

DOES_JOB

| Id | Job |
|---|---|
| 1234 | Lawyer |
| 1235 | Banker |
| 1236 | ? |
| 1237 | ? |

Meaning:

The person identified by *Id* does the job of a *Job*.

EARNS

| Id | Salary |
|---|---|
| 1234 | 100,000 |
| 1235 | ? |
| 1236 | 70,000 |
| 1237 | ? |

Meaning:

The person identified by *Id* earns *Salary* pounds per year.

BUT WHAT DO THOSE QUESTION MARKS MEAN? *(reprise)*

The predicates for DOES_JOB and EARNS are still not really appropriate.

The purpose of such decomposition is to isolate the columns for which values might sometimes for some reason be “missing”. If that Job column never has any question marks in it, for example, then the idea of recombining DOES_JOB and CALLED into a three-column table might be considered. By “never has any question marks”, we really mean that at all times every row in CALLED has a matching row in DOES_JOB and every row in DOES_JOB has a matching row in CALLED (and no row in either table has a question mark).

# b. Horizontal Decomposition

(*very* loosely speaking)

## Principle:

Don't combine multiple meanings in a single table.
"Person 1234 earns 100,000", "We don't know what person 1235 earns", and "Person 1237 doesn't have a salary" are different in kind.

The suggested predicate, "The person identified by *Id* earns *Salary* pounds per year", doesn't really apply to every row of EARNS.

Might try something like this:

Either the person identified by *Id* earns *Salary* pounds per year,
or we don't know what the person identified by *Id* earns,
or the person identified by *Id* doesn't have a salary.

Why doesn't that work?

We will decompose EARNS into one table per *disjunct.*
(DOES_JOB would be treated similarly. CALLED is okay as is.)

"Horizontal", *very* loosely speaking, because the dividing lines are between rows. But some columns are lost on the way, too (see next slide).

When a sentence consists of two main clauses connected by "or", those two main clauses are called disjuncts. If they were connected by "and", they would be called "conjuncts". "Or" is disjunction, "and" conjunction.

## Horizontal Decomposition of EARNS

EARNS

| Id | Salary |
|---|---|
| 1234 | 100,000 |
| 1236 | 70,000 |

Meaning:

The person identified by *Id* earns *Salary* pounds per year.

SALARY_UNK

| Id |
|---|
| 1235 |

Meaning:

The person identified by *Id* earns something but we don't know how much.

UNSALARIED

| Id |
|---|
| 1237 |

Meaning:

The person identified by *Id* doesn't have a salary.

Nothing wrong with the predicates now! And we have reduced the salary part of the database to the simplest possible terms. Yes, some of the complicated queries get more difficult now, because we might have to combine these tables back together again, but the simple queries, such as "How much salary does each person (who has a known salary) earn?" and "Who earns no salary?" become trivial.

Regarding SALARY_UNK and UNSALARIED, note:

1. They cover only two distinct reasons why salary information for an employee might be "missing". If there are other reasons, this approach would require more tables, one for each distinct reason. Another possible reason that springs to mind is, "We don't even know whether the person identified by *id* is paid a salary."

2. Some people cavil at the possible proliferation of relvars. Why might that be a problem? In any case, we would argue that a design based on ultimate decomposition is a sensible starting point. When you consider all the constraints that must apply to that design, then you might that some of those constraints are most easily implemented by some kind of recombination.

3. SALARY_UNK and UNSALARIED could be combined into a two-column table SALARY_MISSING ( Id, Reason ). But then you have to decide what the data type of the Reason column is. Note the trade-off: for each one-column table to be recombined, you have to decide what the corresponding value for the Reason column is. So the single two-column table approach is in some ways no less complex than the multiple one-column table approach.

## Horizontal Decomposition of DOES_JOB

DOES_JOB

| Id | Job |
|---|---|
| 1234 | Lawyer |
| 1235 | Banker |

Meaning:

The person identified by *Id* does *Job* for a living.

JOB_UNK

| Id |
|---|
| 1236 |

Meaning:

The person identified by *Id* does some job but we don't know what it is.

UNEMPLOYED

| Id |
|---|
| 1237 |

Meaning:

The person identified by *Id* doesn't have a job.

Same process, *mutatis mutandis*, as for EARNS.

## What We Have Achieved So Far

What started as a single table (PERS_INFO) is now a *database (sub)schema* (let's call it PERS_INFO again), consisting of:

CALLED ( Id, Name )

DOES_JOB ( Id, Job )

JOB_UNK ( Id )

UNEMPLOYED ( Id )

EARNS ( Id, Salary )

SALARY_UNK ( Id )

UNSALARIED ( Id )

Next, we must consider the constraints needed to hold this design together (so to speak).

## Constraints for New PERS_INFO database

1. No two CALLED rows have the same Id. (Primary key)
2. Every row in CALLED has a matching row in either DOES_JOB, JOB_UNK, or UNEMPLOYED.
3. No row in DOES_JOB has a matching row in JOB_UNK.
4. No row in DOES_JOB has a matching row in UNEMPLOYED.
5. No row in JOB_UNK has a matching row in UNEMPLOYED.
6. Every row in DOES_JOB has a matching row in CALLED. (Foreign key)
7. Every row in JOB_UNK has a matching row in CALLED. (Foreign key)
8. Every row in UNEMPLOYED has a matching row in CALLED. (Foreign key)
9. Constraints 2 through 8 repeated, *mutatis mutandis*, for CALLED with respect to EARNS, SALARY_UNK and UNSALARIED.

WE NEED SOME NEW SHORTHANDS TO EXPRESS 2, 3, 4 AND 5.

The slide shows the constraints pertaining to Job information. A similar set of constraints pertaining to Salary information is needed.

## Proposed Shorthands for Constraints

1. Id is a *distributed key* for (DOES_JOB, JOB_UNK, UNEMPLOYED).
   This addresses Constraints 3, 4 and 5.
2. Id is a *distributed key* for (EARNS, SALARY_UNK, UNSALARIED).
3. Id is a *foreign distributed key* in CALLED, referencing (DOES_JOB, JOB_UNK, UNEMPLOYED).
   This addresses Constraint 2.
4. Id is a *foreign distributed key* in CALLED, referencing (EARNS, SALARY_UNK, UNSALARIED).

Plus regular foreign keys in each of DOES_JOB, JOB_UNK, UNEMPLOYED, EARNS, SALARY_UNK, UNSALARIED, each referencing CALLED.
(Might also want UNEMPLOYED to *imply* UNSALARIED – how would that be expressed?)

So, now we have a schema and constraints. Next, how to add the data and subsequently update it? Are the regular INSERT/UPDATE/DELETE operators good enough?

A key value is one that must exist no more than once in the table for which the key is defined.

A foreign key value is one such that, if it exists in the table for which the foreign key is defined (the referencing table), then it must exist as a key value in the referenced table.

A “distributed key” value is one that must exist as a key value in no more than one of the tables over which the constraint in question is defined.

A “foreign distributed key” value is one such that, if it exists in the referencing table, then it must exist in exactly one of the referenced tables.

(These definitions are a bit loose!)

## Updating the Database: A Problem

How can we add the first row to any of our 7 tables?

Can't add a row to CALLED unless there is a matching row in DOES_JOB, JOB_UNK or UNEMPLOYED and also a matching row in EARNS, SALARY_UNK or UNSALARIED.

Can't add a row to DOES_JOB unless there is a matching row in CALLED. Ditto JOB_UNK, UNEMPLOYED, EARNS, SALARY_UNK and UNSALARIED.

*Impasse!*

The SQL standard has an unsatisfactory solution, "deferred constraint checking", allowing intermediate database states in a transaction to be inconsistent. This approach is explicitly outlawed by *The Third Manifesto*, which requires the database to be consistent at all *atomic statement* boundaries (as opposed to transaction boundaries).

## Updating the Database: Solution

"Multiple Assignment": doing several updating operations in a single "mouthful".

For example:

INSERT_ROW INTO CALLED ( Id 1236, Name 'Cindy' ) ,
INSERT_ROW INTO JOB_UNK ( Id 1236 ) ,
INSERT_ROW INTO EARNS ( Id 1236, Salary 70000 ) ;

Note very carefully the punctuation!

This triple operation is "atomic". Either it all works or none of it works.

Operations are performed in the order given (to cater for the same target more than once), but intermediate states might be inconsistent and are not visible.

So, we now have a *working* database design. Now, what if the user wants to derive that original PERS_INFO table from this database?

I've invented an INSERT_ROW operator just for this presentation. It might or might not be a good idea for a database language.

DBMS support for multiple assignment is prescribed by *The Third Manifesto*.

## To Derive PERS_INFO Table from PERS_INFO Database

```
WITH (EXTEND JOB_UNK ADD 'Job unknown' AS Job_info) AS T1,
     (EXTEND UNEMPLOYED ADD 'Unemployed' AS Job_info) AS T2,
     (DOES_JOB RENAME (Job AS Job_info)) AS T3,
     (EXTEND SALARY_UNK ADD 'Salary unknown' AS Sal_info) AS T4,
     (EXTEND UNSALARIED ADD 'Unsalaried' AS Sal_info) AS T5,
     (EXTEND EARNS ADD CHAR(Salary) AS Sal_info) AS T6,
     (T6 { ALL BUT Salary }) AS T7,
     (UNION ( T1, T2, T3 )) AS T8,
     (UNION ( T4, T5, T7 )) AS T9,
     (JOIN ( CALLED, T8, T9 )) AS PERS_INFO :
PERS_INFO
```

Q.E.D.

Evaluation of this expression entails evaluation of the expression, "PERS_INFO" (just a name!), following the WITH clause. Evaluation of PERS_INFO entails evaluation of the expression as which it is defined, "JOIN ( CALLED, T8, T9 )". The name CALLED is part of our database definition. The names T8 and T9 are defined in earlier elements of the WITH clause.

Appendix A works through these steps one at a time.

This expression is (mostly) in a language called Tutorial D, defined in "Foundation for Future Database Systems: *The Third Manifesto*" and also (incompletely) in "Introduction to Database Systems" (7th edition onwards).

I have added prefix JOIN and prefix UNION here, to allow more than two operands in each case. Tutorial D as currently defined has only infix versions of these operators. UNION ( a, b, c ) is equivalent to ( a UNION b ) UNION c, not that it matters where you put the parentheses, of course. Similarly, JOIN ( a, b, c ) is equivalent to ( a JOIN b ) JOIN c, and again it doesn't matter where you put the parens.

Some of this expression is a bit like "outer join". A respectable variety of outer join might be considered for inclusion in the language as a shorthand to make such queries easier to write. ("Respectable" means as in Chapter 20 of *Relational Database Writings, 1989-1991*.)

# The New PERS_INFO

PERS_INFO

| Id | Name | Job_info | Sal_info |
|---|---|---|---|
| 1234 | Anne | Lawyer | 100,000 |
| 1235 | Boris | Banker | Salary unknown |
| 1236 | Cindy | Job unknown | 70,000 |
| 1237 | Davinder | Unemployed | Unsalaried |

LOOK – NO QUESTION MARKS, NO NULLS!

Remember that this table is only the result of a query that might need to be executed on a regular basis, perhaps for those periodic reports that need to be made available to auditors and the like. It is not intended to be updatable. If this bothers you, I would ask if you have a good reason for wanting it to be updatable.

## How Much of All That Can Be Done Today?

- Vertical decomposition: can be done in SQL
- Horizontal decomposition: can be done in SQL
- Primary and foreign keys: can be done in SQL
- Distributed keys: can be done in (awful) longhand, if at all
- Distributed foreign keys can be done in (awful) longhand, if at all
- Multiple assignment: hasn't caught the attention of SQL DBMS vendors, but Alphora's D4 supports it.
- Recomposition query: can be done but likely to perform horribly. Might be preferable to store PERS_INFO as a single table under the covers, so that the tables resulting from decomposition can be implemented as mappings to that. But current technology doesn't give clean separation of physical storage from logical design.

Perhaps something for the next generation of software engineers to grapple with?

A caveat concerning multiple assignment:

We got the definition wrong in the "Foundation for Object-Relational Databases: *The Third Manifesto*"(1998).

We changed it but got it wrong again in the second edition, "Foundation for Future Database Systems: *The Third Manifesto*" (2000).

We changed it but got it wrong again in "Temporal Data and The Relational Model" (2003).

We are changing it again in the third edition, "Types, Relations and The Relational Model: *The Third Manifesto*" (to appear in 2005).

The End

(Appendix A follows)

## Appendix A:
## Walk-through of Recomposition Query

We look at each step in turn, showing its effect.

T1: EXTEND JOB_UNK ADD ‘Job unknown’ AS Job_info

JOB_UNK

| Id |
|---|
| 1236 |

T1

| Id | Job_info |
|---|---|
| 1236 | Job unknown |

The result of EXTEND consists of each row in the operand, with one or more added columns whose values are calculated from given formulae. Here, the only such formula is a simple character string literal.

T2: EXTEND UNEMPLOYED ADD ‘Unemployed’ AS Job_info

UNEMPLOYED

| Id |
|---|
| 1237 |

T2

| Id | Job_info |
|---|---|
| 1237 | Unemployed |

T3: DOES_JOB RENAME (Job AS Job_info)

DOES_JOB

| Id | Job |
|---|---|
| 1234 | Lawyer |
| 1235 | Banker |

T3

| Id | Job_info |
|---|---|
| 1234 | Lawyer |
| 1235 | Banker |

The result of a RENAME is a copy of the operand, with one or more columns being renamed.

T4: EXTEND SALARY_UNK ADD ‘Salary unknown’ AS Sal_info

SALARY_UNK

| Id |
|---|
| 1235 |

T4

| Id | Sal_info |
|---|---|
| 1235 | Salary unknown |

T5: EXTEND UNSALARIED ADD ‘Unsalaried’ AS Sal_info

UNSALARIED

| Id |
|---|
| 1237 |

T5

| Id | Sal_info |
|---|---|
| 1237 | Unsalaried |

T6: EXTEND EARNS ADD CHAR(Salary) AS Sal_info

EARNS

| Id | Salary |
|---|---|
| 1234 | 100,000 |
| 1236 | 70,000 |

T6

| Id | Salary | Sal_info |
|---|---|---|
| 1234 | 100,000 | 100,000 |
| 1236 | 70,000 | 70,000 |

Salary and Sal_info differ in type. Sal_info contains character representations of Salary values (hence left justified!).

T7: T6 { ALL BUT Salary }

T6

| Id | Salary | Sal_info |
|---|---|---|
| 1234 | 100,000 | 100,000 |
| 1236 | 70,000 | 70,000 |

T7

| Id | Sal_info |
|---|---|
| 1234 | 100,000 |
| 1236 | 70,000 |

We use curly braces to denote the relational PROJECT operator. Inside the braces, we can either write a list of all the columns of the operand that are to be included in the result, or a list, following “ALL BUT”, of the operand columns to be discarded.

Remember that, in general, any duplicate rows resulting from loss of columns are discarded (of course!). Here there are no duplicates.

T8: UNION ( T1, T2, T3 )

= ( T1 UNION T2 ) UNION T3

T1

| Id | Job_info |
|---|---|
| 1236 | Job unknown |

T2

| Id | Job_info |
|---|---|
| 1237 | Unemployed |

T3

| Id | Job_info |
|---|---|
| 1234 | Lawyer |
| 1235 | Banker |

T8

| Id | Job_info |
|---|---|
| 1234 | Lawyer |
| 1235 | Banker |
| 1236 | Job unknown |
| 1237 | Unemployed |

The result of a UNION consists of each row that exists in at least one of its operands. Any row that appears in more than one operand appears only once in the result (of course!).

T9: UNION ( T4, T5, T7 )

= ( T4 UNION T5 ) UNION T7

T4

| Id | Sal_info |
|---|---|
| 1235 | Salary unknown |

T5

| Id | Sal_info |
|---|---|
| 1237 | Unsalaried |

T7

| Id | Sal_info |
|---|---|
| 1234 | 100,000 |
| 1236 | 70,000 |

T9

| Id | Job_info |
|---|---|
| 1234 | 100,000 |
| 1235 | Salary unknown |
| 1236 | 70,000 |
| 1237 | Unsalaried |

PERS_INFO: JOIN ( CALLED, T8, T9 )

CALLED

| Id | Name |
|---|---|
| 1234 | Anne |
| 1235 | Boris |
| 1236 | Cindy |
| 1237 | Davinder |

T8

| Id | Job_info |
|---|---|
| 1234 | Lawyer |
| 1235 | Banker |
| 1236 | Job unknown |
| 1237 | Unemployed |

T9

| Id | Sal_info |
|---|---|
| 1234 | 100,000 |
| 1235 | Salary unknown |
| 1236 | 70,000 |
| 1237 | Unsalaried |

PERS_INFO

| Id | Name | Job_info | Sal_info |
|---|---|---|---|
| 1234 | Anne | Lawyer | 100,000 |
| 1235 | Boris | Banker | Salary unknown |
| 1236 | Cindy | Job unknown | 70,000 |
| 1237 | Davinder | Unemployed | Unsalaried |

Each row in the result of a JOIN is the result of taking exactly one row from each operand and combining them together. Rows that are thus combined must have equal values for each "matching" column. Every combination satisfying these conditions does appear in the result. Columns having the same name are matching columns. Here there is just one matching column, Id.

(If there are no matching columns at all, then the result consists of every row that can be constructed by taking one row from each operand and combining them together.)

The Very End