How To Handle Missing Information Without Using NULL

Hugh Darwen

hugh@dcs.warwick.ac.uk
www.dcs.warwick.ac.uk/~hugh

for Warwick University, CS253

1

SQL's NULL Is A Disaster

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

Database Explorations
by C.J. Date and Hugh Darwen (2010)

2

NULL

Cause of more debate and anguish than any other Fatal Flaw.

There's even a split in the relational camp (E.F. Codd proposed "A-marks", "I-marks" and a 4-valued logic).

There's only one NULL. How many different reasons can there be for something being "missing"?

Why NULL ruins everything –
- UNION of sets, cardinality of sets.

Destruction of functional dependency theory

SQL's implementation of NULL is even worse than the best suggested by theoreticians. And it's not completely BYPASSABLE, because SQL thinks that the sum of the empty set is NULL!
Nor is it CORRECTABLE - the Shackle of Compatibility!

3

A Contradiction in Codd's Proposed Treatment

"Every relation has at least one candidate key"

"One of the candidate keys is nominated to be the primary key"

"Nulls aren't permitted in the primary key"

"Nulls *are* permitted in alternate keys"

• Consider the relation resulting from the projection of PATIENT over RELIGION, a "nullable column".

Wait a moment! What's this?

• List the candidate keys of this relation.

• Nominate the primary key.

4

Surprises Caused by SQL's NULL

1. SELECT * FROM T WHERE X = Y
   UNION
   SELECT * FROM T WHERE NOT (X = Y)
   is not equal to SELECT * FROM T
2. SELECT SUM(X) + SUM(Y) FROM T
   is not equal to
   SELECT SUM(X + Y) FROM T
3. IF X = Y THEN 'Yes'; ELSE 'No'
   is not equal to
   IF NOT (X = Y) THEN 'No'; ELSE 'Yes'

5

Why NULL Hurts Even More Than It Once Did

Suppose "*x* = *x*" returns *Unknown*

Can we safely conclude "*x* IS NULL" ?

Suppose *x* "is not the null value"?

Can we conclude "*x* IS NOT NULL"?

Not in modern SQL!

6

How *x* = *x* *Unknown* Yet *x* NOT NULL

For example:

1. *x* is ROW (1, null) - or even ROW(null, null)

   ROW(...) is a row "constructor".

   Hang on! Are you sure?

2. *x* is POINT (1,null)
   POINT(a,b) is a "constructor" for values in the user-defined data type POINT.

3. *x* is ROW (POINT(1,1), POINT(null,3))

Consequences?

7

x IS NULL (Case 1)

What does *x* IS NULL MEAN? Think you know? Well, think again!

```
CREATE TABLE T ( C1 INT, C2 ROW ( F1 INT, F2 INT ) ) ;
INSERT INTO T VALUES ( NULL, NULL ) ;
```

Query	Result Cardinality	
SELECT * FROM T WHERE C1 IS NULL	1	
SELECT * FROM T WHERE C2 IS NULL	1	
SELECT * FROM T WHERE (C1, C1) IS NULL	1	
SELECT * FROM T WHERE (C1, C2) IS NULL	1	So far, so good?
SELECT * FROM T WHERE (C2, C2) IS NULL	1	

But even this depends on our charitable interpretation of the ISO SQL standard.

8

x IS NULL (Case 2)

```
CREATE TABLE T ( C1 INT, C2 ROW ( F1 INT, F2 INT ) ) ;
INSERT INTO T VALUES ( NULL, ROW ( NULL, NULL ) ) ; -- note the difference from Case 1
```

Query	Result Cardinality	
SELECT * FROM T WHERE C1 IS NULL	1	
SELECT * FROM T WHERE C2 IS NULL	1	
SELECT * FROM T WHERE (C1, C1) IS NULL	1	
SELECT * FROM T WHERE (C1, C2) IS NULL	0	!!!
SELECT * FROM T WHERE (C2, C2) IS NULL	0	!!!

9

x IS NOT NULL

So, what does *x* IS NOT NULL MEAN?

```
CREATE TABLE T ( C1 INT, C2 ROW ( F1 INT, F2 INT ) ) ;
INSERT INTO T VALUES ( NULL, ROW ( NULL, NULL ) ) ;
```

Query	Result Cardinality	
SELECT * FROM T WHERE C1 IS NOT NULL	0	
SELECT * FROM T WHERE C2 IS NOT NULL	0	
SELECT * FROM T WHERE (C1, C1) IS NOT NULL	0	
SELECT * FROM T WHERE (C1, C2) IS NOT NULL	0	!!!
SELECT * FROM T WHERE (C2, C2) IS NOT NULL	1	!!!

10

Effects of Bad Language Design

There are general language design lessons to be learned from this tangled web, as well as lessons about NULL:

- Enclosing an expression in parens should not change its meaning.
  (C1) is not the only example in SQL. Think of "scalar subqueries".
- Great caution is needed when considering pragmatic shorthands.
  (C1, C2) IS NULL was originally shorthand for C1 IS NULL AND C2 IS NULL.
- All data types supported by a language should be "first-class", for *orthogonality*.
  ROW types were originally not first-class – could not (for example) be the declared types of columns.

11

It Could Have Been Worse ...

... if SQL had paid proper attention to degenerate cases.

SQL fails to recognise the existence of relations of degree zero (tables with no columns). These in turn depend on the existence of the 0-tuple. Suppose SQL had not made this oversight.

```
CREATE TABLE T ( C1 ROW ( ) ) ;
INSERT INTO T VALUES ( ROW ( ) ) ;
```

Query	Result Cardinality
SELECT * FROM T WHERE **NOT** (C1 IS NULL)	1
SELECT * FROM T WHERE C1 IS NULL	1

C1 "is not the null value"; also, no field of C1 "is the null value".

But it is also true that *every* field of C1 "is the null value"!

12

3-Valued Logic: The Real Culprit

Relational theory is founded on classical, 2-valued logic.

A relation *r* is interpreted as a representation of the extension of some predicate *P*.

Let *t* be a tuple with the same heading as *r*.

If tuple *t* is a member of *r*, then the proposition *P*(*t*) is taken to be TRUE; otherwise (*t* is not a member of *r*), *P*(*t*) is taken to be FALSE.

There is no middle ground. The Law of The Excluded Middle applies.

There is no way of representing that the truth of *P*(*t*) is unknown, or inapplicable, or otherwise concealed from us.

SQL's WHERE clause *arbitrarily* splits at the TRUE/UNKNOWN divide.

13

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???

14

Summary of Proposed Solution

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

15

Database Design

a. "vertical" decomposition

Decompose into 2 or more relvars 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 relvar consists of a key plus at most one other attribute.

16

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)*

17

b. Horizontal Decomposition

(*very* loosely speaking)

Principle:

Don't combine multiple meanings in a single relvar.
"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.)

18

Horizontal Decomposition of EARNS

EARNS

Id	Salary
1234	100,000
1236	70,000

SALARY_UNK

Id
1235

UNSALARIED

Id
1237

Meaning:

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

Meaning:

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

Meaning:

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

19

Horizontal Decomposition of DOES_JOB

DOES_JOB

Id	Job
1234	Lawyer
1235	Banker

JOB_UNK

Id
1236

UNEMPLOYED

Id
1237

Meaning:

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

Meaning:

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

Meaning:

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

20

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).

21

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.

22

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?

23

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!

24

Updating the Database: Solution

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

For example:

```
INSERT_TUPLE INTO CALLED { Id 1236, Name 'Cindy' } ,
INSERT_TUPLE INTO JOB_UNK { Id 1236 } ,
INSERT_TUPLE 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.

Loosely speaking: 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?

25

To Derive PERS_INFO Relation 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.

26

The New PERS_INFO

PERS_INFO

<u>Id</u>	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!

27

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
- Foreign distributed 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?

28

The End

(Appendix A follows)

29

Appendix A: Walk-through of Recomposition Query

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

30

T1: EXTEND JOB_UNK ADD ('Job unknown' AS Job_info)

JOB_UNK

Id
1236

T1

Id	Job_info
1236	Job unknown

31

T2: EXTEND UNEMPLOYED ADD ('Unemployed' AS Job_info)

UNEMPLOYED

Id
1237

T2

Id	Job_info
1237	Unemployed

32

T3: DOES_JOB RENAME (Job AS Job_info)

DOES_JOB

Id	Job
1234	Lawyer
1235	Banker

T3

Id	Job_info
1234	Lawyer
1235	Banker

33

T4: EXTEND SALARY_UNK ADD ('Salary unknown' AS Sal_info)

SALARY_UNK

Id
1235

T4

Id	Sal_info
1235	Salary unknown

34

T5: EXTEND UNSALARIED ADD ('Unsalaried' AS Sal_info)

UNSALARIED

Id
1237

T5

Id	Sal_info
1237	Unsalaried

35

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!).

36

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

37

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

38

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

39

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

40

The Very End

41