CS253: HACD.1 How to Handle Missing Information Without Using NULL

(notes keyed to lecture slides)

1. How To Handle Missing Information Without Using NULL

"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.

2. SQL's NULL Is A Disaster

The chapters in question, in the *Database Writings* books mentioned on this slide are:

- 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)
- In Database Explorations, Part IV, "Missing Information" (7 chapters)

3. NULL

Codd had nothing to do with nulls back in 1970. The proposal referred to here is from his book "The Relational Model for Database Management: Version 2" (Addison-Wesley, 1990).

Assuming that NULL stands for "something should appear here but we do not know what that is", what is the cardinality of the set {1, NULL}? Indeed, is {1, NULL} even a set? If it is a set, what might be the membership predicate for that set?

Some of the errors in SQL can be avoided by avoiding the language constructs that introduce them. For example, it is possible to use SQL in such a way that no table in or derived from the database will contain more than one appearance of the same row. (Of course, NULL gets in the way of our normal understanding of "the same", so it might be argued that even this observation is subject to some arm-waving.)

The Shackle of Compatibility refers to that property of computer languages whereby errors in the language definition cannot be corrected because of the existence of applications that depend on the existence of those errors.

4. A Contradiction in Codd's Proposed Treatment

Mind you, we even question the distinction between "primary key" and "alternate key". Codd himself noted that the choice of primary key was arbitrary, not based on logic. It is surprising, therefore, that he perceived any *logical* difference in concept here.

As for that projection, assume that at least two patients have NULL appearing as their religion. How many rows with NULL for RELIGION should appear in the result of SELECT DISTINCT RELIGION FROM PATIENT? Note "*should* appear". We know that the SQL result contains just one such row. How can SQL have come to that conclusion, except by assuming that NULL=NULL? But SELECT DISTINCT RELIGION FROM PATIENT WHERE RELIGION = RELIGION doesn't include a row with NULL for RELIGION!!!

5. Surprises Caused by SQL's NULL

Explanations:

1. If either X or Y "is the null value" (as the SQL standard puts it, without defining what it means), then "X = Y" evaluates to SQL's third truth value, *Unknown*. So does "NOT (X = Y)". So rows that have NULL for either X or Y (or both) fail both WHERE clauses and therefore do not appear in the union.
2. SUM(X) gives the some of all the non-null X values in T. SUM(Y) gives the sum of all the non-null Y values in T. SUM(X+Y) gives the sum of the results of evaluating "X+Y" for each row of T. X+Y evaluates to NULL if either X or Y "is the null value".
3. The THEN branch is invoked only when the IF condition evaluates to *True* and not when it evaluates to *Unknown* or *False*.

6. Why NULL Hurts Even More Than It Once Did

We have to consider the cases where *x* is in some sense composite. Examples are on the next slide.

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

In general, a comparison evaluates to *Unknown* when any component of either operand "is the null value", however deeply nested that component is within some structure.

The case of ROW expressions is further complicated by the way IS NULL and IS NOT NULL are defined. They are defined to operate on rows so that they can be used on lists of expressions, as shown on the next few slides ...

8. *x* IS NULL (Case 1)

Warning: the notes on this slide are likely to hurt your brain!

The standard uses the term "is the null value" without defining it. It is not clear what "is the null value" means, because "x IS NULL" can be true when it is not the case that x "is the null value".

All this might seem reasonably clear but surprises are coming up, and here's why: The expression "C1" in the first example query is not an expression of type INTEGER! Instead, because it is immediately followed by "IS NULL", it is defined to be an expression of type ROW (*f1* INTEGER), where *f1* is an unspecified (!) field name. (The unspecified field name is strange but not relevant to the issue at hand.)

In fact, "C1", in this special context only, is short for "ROW (C1)". So is "(C1)". You might wonder, then, what "((C1))" is short for. Is it ROW ((C1)), which means the same as ROW (C1),

or is it ROW (ROW (C1)), which doesn't? Amazingly, by an almost unfathomable piece of special-casing, the SQL standard makes ROW ((C1)) mean the same as ROW (C1). (You are not expected to remember this!)

Here is the definition of IS [NOT] NULL given in the SQL standard (as of November 2005)

8.7 <null predicate>

Function

Specify a test for a null value.

Format

```
<null predicate> ::=
  <row value predicand>[1] <null predicate part 2>
<null predicate part 2> ::=
  IS [ NOT ] NULL
```

Syntax Rules

None.

Access Rules

None.

General Rules

1) Let *R* be the value of the <row value predicand>.

2) Case:

   a) If *R* is the null value, then "`R IS NULL`" is *True*.

   b) Otherwise:

      i) The value of "`R IS NULL`" is

      Case:

         1) If the value of every field in *R* is the null value, then *True*.

         2) Otherwise, *False*.

      ii) The value of "`R IS NOT NULL`" is

      Case:

         1) If the value of no field in *R* is the null value, then *True*.

         2) Otherwise, *False*.

NOTE 204 — For all *R*, "*R* IS NOT NULL" has the same result as "NOT *R* IS NULL" if and only if *R* is of degree 1.

Note very carefully what NOTE 204 is trying to tell us. For example, if X "is the null value" and Y is not, then ROW (X, Y) IS NOT NULL is *False* because Case b)ii)1) fails and therefore Case b)ii)2) applies. However, NOT (ROW (X, Y) IS NULL) is *True* because ROW (X, Y) IS NULL is *False* because Case b)i)1) fails and so Case b)i)2) applies.

[1] A <row value predicand> is any expression denoting a row consisting of at least one value. It is specified in general by writing ROW (x_1, ..., x_n), but note carefully that, for historical reasons, the key word ROW is optional. And in the case where n=1 and x_1 is not itself a row, so are the parens. So you can write, for example, (C1, C2) instead of ROW (C1, C2) and you can write just C1 for ROW (C1). Note, however, that C1 denotes a value that is itself a row, then C1 IS NULL is not short for ROW (C1) IS NULL—the operand of IS NULL is just the row denoted by C1. [This footnote is not for the faint-hearted!]

9. *x* IS NULL (Case 2)

The reason for the last two lies in the fact that "is the null value" and "IS NULL" are not equivalent. "C2 IS NULL" = *True*, because C2.F1 and C2.F2 are both "the null value" and so Case b)i)1) of the SQL standard's definition applies; but C2 is not "the null value".

10. *x* IS NOT NULL

Note that (C1,C2) IS NULL and (C1, C2) IS NOT NULL are both *False*. The first is *False* because at least one of C1 and C2 is not "the null value"; the second is *False* because at least one *is* "the null value".

11. Effects of Bad Language Design

Here is an example containing a scalar subquery:

```
SELECT StudentName
FROM ExamMarks
WHERE Score = ( SELECT MAX ( Score ) FROM ExamMarks )
```

The expression inside the parens is a query and therefore returns a table. And yet the WHERE clause is apparently comparing this table with a number, which doesn't make sense. The parens around the query tell SQL to treat it as denoting the only column value in the only row in the result of the query.

This treatment by SQL caused great problems when the language need to be extended to support "nested tables" (tables appearing as column values inside other tables).

12. It Could Have Been Worse ...

Recall that "for all *x*, *P*(*x*)" is equivalent to "there does not exist *x* such that NOT(*P*(*x*))".

13. 3-Valued Logic: The Real Culprit

No notes.

14. Case Study Example

That predicate is at best approximate. It might be appropriate if it weren't for the question marks in the data. "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 might 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). In fact, NULL isn't even a value, because it is not possessed of that essential property whereby it compares equal with itself.

15. Summary of Proposed Solution

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

16. Database Design

a. "vertical decomposition"

I use the term "Vertical" because the dividing lines, *very* loosely speaking, are between columns. It is fundamental that the relvar to be decomposed can be reconstructed by joining together the relvars 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 relvars. 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.

17. Vertical Decomposition of PERS_INFO

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

The purpose of such decomposition is to isolate the attributes 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 relvar of degree 3 might be considered. By "never has any question marks", we really mean that at all times every tuple in CALLED has a matching tuple in DOES_JOB and every tuple in DOES_JOB has a matching tuple in CALLED (and no tuple in either relvar has a question mark).

In fact, when we have two or more non-key attributes, we are using a very convenient shorthand for the constraint I have just alluded to. The constraint is to the effect that each tuple in any relvar in the 6NF decomposition is matched by a corresponding tuple in each other relvar in that decomposition (where "corresponding" tuples are tuples having the same key value).

18. b. Horizontal Decomposition

"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.

19. Horizontal Decomposition of EARNS

There's 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 relvars, 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 binary relvar SALARY_MISSING { Id, Reason }. But then you have to decide what the data type of the Reason attribute is. Note the trade-off: for each unary relvar to be recombined, you have to decide what the corresponding value for the Reason attribute is. So the single binary relvar approach is in some ways no less complex than the multiple unary relvar approach.

20. Horizontal Decomposition of DOES_JOB

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

21. What We Have Achieved So Far

Some people express great concern at the proliferation of relvars necessitated when the decomposition approach is taken to such extremes. They complain about (a) the loss of convenience and (b) the loss of performance. But loss of convenience in what respect(s)? And compared with what? The same questions apply to loss of performance. Some tasks are very easy under this approach. Performance depends of course on the DBMS being used, but some tasks, such giving somebody a salary rise, surely cannot be subject to poorer performance resulting from adopting this approach rather than some other approach.

We need to be specific about the tasks that concern us, regarding convenience and performance, and we need to prioritise those tasks, to make well-informed database design decisions.

22. Constraints for New PERS_INFO database

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

23. Proposed Shorthands for Constraints

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

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

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

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

(These definitions are a bit loose!)

24. Updating the Database: A Problem

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).

25. Updating the Database: Solution

I’ve invented an INSERT_TUPLE operator just for this presentation. It might or might not be a good idea for a database language. In SQL and **Tutorial D** the source for an INSERT is a table

(SQL) or relation (**Tutorial D**), so if you want to "insert a row" you have to insert a one-row table (SQL) or one-tuple relation (**Tutorial D**).

DBMS support for multiple assignment is prescribed by *The Third Manifesto*. The precise definition is rather complex but the complexities need not bother us here.

The point of "not visible" is that the expression denoting the source of an assignment might in general include an invocation of a user-defined function, and that UDF might access the database. We think it is essential for all users at all times to be able to assume safely that the database is consistent.

26. To Derive PERS_INFO Relation from PERS_INFO Database

This expression is written in **Tutorial D**, the language taught in our module CS252, "Fundamentals of Relational Databases".

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 on 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.

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*.)

27. The New PERS_INFO

Remember that this relation 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.

28. How Much of All That Can Be Done Today?

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 changed it again in the third edition, "Databases, Types, and The Relational Model: *The Third Manifesto*" (2005). It remains to be seen if we got it right this time!

29. The End

No notes.

30. Appendix A: Walk-through of Recomposition Query

No notes.

31. T1: EXTEND JOB_UNK ADD 'Job unknown' AS Job_info

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

32. T2: EXTEND UNEMPLOYED ADD 'Unemployed' AS Job_info

No notes.

33. T3: DOES_JOB RENAME (Job AS Job_info)

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

34. T4: EXTEND SALARY_UNK ADD 'Salary unknown' AS Sal_info

No notes.

35. T5: EXTEND UNSALARIED ADD 'Unsalaried' AS Sal_info

No notes.

36. T6: EXTEND EARNS ADD CHAR(Salary) AS Sal_info

No notes.

37. T7: T6 { ALL BUT Salary }

We use curly braces to denote relational *projection*. Inside the braces, we can either write a list of all the attributes of the operand that are to be included in the result, or a list, following "ALL BUT", of the operand attributes to be discarded.

Remember that, in general, any duplicate tuples resulting from loss of attributes are discarded (of course!). Here there would be no duplicates anyway.

38. T8: UNION (T1, T2, T3)

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

39. T9: UNION (T4, T5, T7)

No notes.

40. PERS_INFO: JOIN (CALLED, T8, T9)

Each tuple in the result of a JOIN is the result of taking exactly one tuple from each operand and combining them together. Tuples that are thus combined must have equal values for each "matching" attribute. Every combination satisfying these conditions does appear in the result. Attributes having the same name (and type) are matching attributes. Here there is just one matching attribute, Id. (If there are no matching attributes at all, then the result consists of every tuple that can be constructed by taking one tuple from each operand and combining them together.)

41. The Very End

End of Notes