ORACLE TECHNOLOGY NETWORK

TECHNOLOGY: SQL

As Published In
ORACLE MAGAZINE
July/August 2005

Nulls: Nothing to Worry About

By Lex de Haan and Jonathan Gennick

Avoid problems with three-valued logic.

Called null values in the ISO SQL standard, nulls are anything but values. Nulls are markers indicating the complete lack of a value. They lead to three-valued logic, which is confusing to work with, and that confusion often leads the unwary down the path of writing `SELECT` statements that return wrong results. This article highlights some of the pitfalls you'll encounter and gives advice for avoiding them.

Nulls in Scalar Expressions

Typically, the result of any scalar expression involving a null will itself be a null. The first query in Listing 2, against the data in Listing 1, demonstrates this by generating a report showing the impact of raising all salaries by US$1,000. You can see right away in Listing 2 that neither Adams nor Morle have any value at all for their new salary.

Code Listing 1: Our example schema

```
TABLE: DEPT_M

DEPTNO      DNAME          LOC
_______     _______        _______________
10          HQ             UTRECHT
20          SALES          MUNISING
30          MANUFACTURING  NOVOSIBIRSK

TABLE: EMP_M

EMPNO       ENAME          JOB            MGR       SAL      COMM     DEPTNO
_______     _____________  ____________   _____     _____    _______  ________
100         NORGAARD       PRESIDENT                5000               10
122         LEWIS          SALESREP        120      1100
199         GENNICK                                 2200               10
111         DE HAAN        CLERK           110      2000
112         MILLSAP        SALESREP        110      1250     1400      20
110         ADAMS          MANAGER         100      1700               20
120         KOLK           MANAGER         100      2450               10
113         MCDONALD       SALESREP        110      1500               20
121         WOOD           CLERK           120      1300               10
130         MORLE          CLERK           100                         10
```

From the perspective of the database server, nulls can have only one context- and datatype-independent meaning: "information missing." Any further interpretation of nulls might be very human and intuitive, but the database server treats all nulls the same way, regardless of where they come from. The database server cannot add US$1,000 to null and must simply return null as the result.

Yet humans do need to deal with nulls, and they have the ability to ask questions that the database server cannot. What does it mean to have a null salary? Does it mean that salary doesn't apply? Or does it mean that an employee has "no salary" in the sense that that person's salary is US$0? Or does it mean that a value would be applicable, but we simply don't know that value? Or could it mean any of the foregoing, depending on which employee we are talking about? Although it is sometimes necessary, you can begin to see that human interpretation of nulls can be quite dangerous.

Perhaps most important is the question of what it means from a business standpoint to increase a null salary by US$1,000. It's not enough to deal with nulls in a technical vacuum. You must step back and ask the right business questions. Only after understanding both the underlying data model and the business intent of the query at hand are you ready to tackle the query's handling of nulls.

One approach to handling nulls in scalar expressions is to substitute a real value anytime a null might occur. To this end, `COALESCE` can be very helpful. If you determine that in addition to raising all salaries by US$1,000, the business also wants to grant US$1,000 salaries to those currently without a salary, you can use the `COALESCE` function to treat nulls as zero. The

second query in Listing 2 passes two arguments, the value from the SAL column and then a zero, to COALESCE. The function returns the first non-null argument as its result.

Code Listing 2: Null in, null out, and the result of COALESCE

```
SELECT EMPNO, ENAME, SAL, SAL + 1000 FROM EMP_M;

EMPNO       ENAME         SAL     SAL+1000
_______     _________     _____   ___________
112         MILLSAP       1250    2250
110         ADAMS
120         KOLK          2450    3450
130         MORLE
  ...

SELECT EMPNO, ENAME, SAL,
        COALESCE(SAL,0) + 1000 FROM EMP_M;

EMPNO      ENAME        SAL       COALESCE(SAL,0)+1000
_______    _________    _____     ____________________________
112        MILLSAP      1250      2250
110        ADAMS                  1000
120        KOLK         2450      3450
```

Oracle Database supports several functions that are similar to COALESCE. These include NVL2, NULLIF, and NVL. (Take time to read about these functions in the Oracle SQL Reference manual.) We recommend COALESCE over NVL, because COALESCE can handle more than just two arguments and it's part of the SQL standard. When COALESCE isn't enough, you may be able to find refuge in CASE expressions, which are also covered in Oracle SQL Reference.

Nulls in Boolean Expressions

Nulls make themselves felt in particularly subtle ways in Boolean expressions, such as those you might write for the WHERE clause of a query. Boolean expressions normally result in TRUE or FALSE, but nulls introduce a third possible result of Boolean expressions: UNKNOWN. Note that NULL is not the same as UNKNOWN:

- SAL + NULL results in NULL. (This is a scalar expression.)
- SAL < NULL results in UNKNOWN. (This is a Boolean expression.)

Listing 3 shows the ultimate consequence of three-valued logic. Even though COMM is being compared with itself, the database treats any nulls in a context-independent manner: Any comparison to null results in UNKNOWN, and queries return only those rows for which the WHERE clause evaluates to TRUE.

Code Listing 3: WHERE COMM = COMM is not equal

```
SELECT * FROM EMP_M WHERE COMM = COMM;

EMPNO      ENAME        JOB          MGR     SAL     COMM       DEPTNO
_______    _________    __________   _____   _____   _______    ________
112        MILLSAP      SALESREP     110     1250    1400       20
110        ADAMS        MANAGER      100             1700       20
```

You can often use IS NULL or IS NOT NULL in Boolean expressions to avoid UNKNOWN results. Consider the problem of listing employees with commissions less than US$1,500. You could begin by writing:

```
SELECT * FROM EMP_M
WHERE COMM < 1500;
```

And you'd soon discover (we hope) that employees such as Norgaard and Lewis, who have null commissions, would be omitted from the list. Assuming that you wanted to interpret a null commission as "no commission," you could broaden your WHERE clause by adding an IS NULL condition:

```
SELECT * FROM EMP_M
WHERE (COMM < 1500)
   OR (COMM IS NULL);
```

Be careful here! Does a null commission mean "no commission," or might it mean "commission unknown"? The answer depends on your application, and there might not be a clear answer at all. Don't automatically add `IS NULL` predicates on nullable columns to your queries. Doing so will lead to errors just as surely as failing to think about the possibility of nulls to begin with will do so. Whether to include the `IS NULL` condition in our example is actually a business decision, not a technical one.

When writing `WHERE` clauses, it's common to link several predicates by using the operators `AND, OR`, and `NOT`, as we've just done. The truth tables in Figure 1 show how these operators handle different combinations of `TRUE, FALSE`, and `UNKNOWN` operands.

NOT			
	TRUE	FALSE	UNKNOWN
NOT	FALSE	TRUE	UNKNOWN
AND			
	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN
OR			
	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Figure 1: Truth Table

Nulls in CHECK Constraints

When you are evaluating a `WHERE` clause, `UNKNOWN` leads to the same end result as `FALSE`—the row is rejected. Yet in a `CHECK` constraint, `UNKNOWN` leads to the same end result as `TRUE`—the row is accepted. This is because constraints raise violations only if their Boolean expressions evaluate to `FALSE`. It's why the following constraint definition allows nulls in the `DEPTNO` column, although it might suggest otherwise:

```
CHECK (DEPTNO
   IN (10, 20, 30))
```

Here is a good way to think about all this. The action of `WHERE` and `HAVING` clauses is to pass those rows for which expressions evaluate to `TRUE`. The action of a `CHECK` constraint is to reject rows for which expressions are `FALSE`. In all cases, no action, neither pass nor reject, is taken in the `UNKNOWN` case.

Nulls and Joins

Listing 4 demonstrates that outer joins represent one situation in which the database engine will generate nulls for you, on the fly, even if your database doesn't contain a single null to begin with. The row for department 30 is generated by the outer-join operation, and the employee columns in that row are initialized to null.

Code Listing 4: Outer joins generate nulls

```
SELECT  E.EMPNO, E.ENAME, E.DEPTNO, D.DEPTNO, D.DNAME
FROM EMP_M E RIGHT OUTER JOIN DEPT_M D ON E.DEPTNO = D.DEPTNO;

EMPNO      ENAME          DEPTNO        DEPTNO     DNAME
_______    ____________   ________      ________   ____________________
100        NORGAARD       10            10         HQ
...
130        MORLE 10       10                       HQ
                                        30         MANUFACTURING
```

Another subtle issue with nulls and outer joins is that it matters from which table you return a join column. Note that one of the two `DEPTNO` values for the last row in Listing 4 is null. The null `DEPTNO` is in the employee table's `(EMP_M) DEPTNO` column, whereas the non-null `DEPTNO` value comes from the department table `(DEPT_M)`. When writing an outer join, give careful consideration to the table from which you retrieve join columns.

Nulls in Summarized Data

Scalar expressions yield null if any operand is null, but nulls are ignored by aggregate functions. This behavior is specified by the SQL standard, but it can still lead to some very surprising and unintuitive query results. Look at Listing 5. Apparently, if you want to derive the sum of two or more columns containing numeric values, there is a difference between first adding them up horizontally and then vertically, and adding them up vertically first.

Code Listing 5: SUM(A+B) is not the same as SUM(A)+SUM(B)

```
SELECT SUM(SAL+COMM), SUM(SAL)+SUM(COMM) FROM EMP_M;

SUM(SAL+COMM)             SUM(SAL)+SUM(COMM)
_____________________     ___________________________
2650                      19900
```

Why the difference? It's because the result of `SAL + COMM` is non-null only for those rows in which both `SAL` and `COMM` are non-null. Thus, only those rows contribute to the result of `SUM(SAL+COMM)`. The result of `SUM(SAL)+SUM(COMM)`, on the other hand, manages to include all non-null values from both columns.

We wish we could provide a nice, neat, tidy bit of advice for dealing with nulls in summarized data. The best we can tell you, however, is to carefully think through the possibility of nulls and to make a conscious decision about the results you want in your summary.

"NOT IN" versus "NOT EXISTS"

Compare the two queries (and their results) in Listing 6. In each query, we are trying to retrieve all employees without subordinates. Obviously, the results are different—so it is impossible that they are both correct. An interesting question is: Which query is correct? And which result did you expect, based on our `EMP_M` demo table? You probably expected the second result, and therefore you probably would point at the first query as being wrong, right? Perhaps you have an idea of how to "fix" the first query to make it return the "correct" result.

Code Listing 6: Who has no subordinates?

```
SELECT E1.ENAME FROM EMP_M E1
WHERE E1.EMPNO NOT IN
      (SELECT E2.MGR FROM EMP_M E2);

No rows selected.

SELECT E1.ENAME FROM EMP_M E1
WHERE NOT EXISTS
      (SELECT E2.* FROM EMP_M E2
       WHERE E2.MGR = E1.EMPNO);

ENAME
_____________
DE HAAN
WOOD
MILLSAP
LEWIS
MCDONALD
GENNICK
MORLE
```

But wait a minute! We have a convincing argument to show that the first query is correct and the second one is wrong! Lewis, Gennick, De Haan, Millsap, McDonald, Wood, Morle—any one of them could be the manager of Norgaard. Thus, we cannot truly be certain that any of them have no subordinates.

What is subtle about Listing 6 is that the programmer may not have consciously intended to apply any sort of business interpretation to nulls. `NOT IN` and `NOT EXISTS` are often thought of as being interchangeable, but they really aren't—not quite—not when it comes to nulls. The intermediate result of the subquery in the first example in Listing 6 contains a null. The database engine cannot be sure that `E1.EMPNO` is not equal to null, and thus no rows evaluate to `TRUE`. The second example tests for the existence of rows having a specific `MGR` value. Rows where `MGR` is null aren't even considered. When writing `NOT IN` conditions, always be sure you take the time to think about the *`X` `NOT IN (...,NULL,...)`* case. When writing `NOT EXISTS`, consider whether the business really requires the results you'd get from the NOT IN version of the

query. And remember, choosing the path to go down is a business decision. Given the scenario posed in Listing 6, we would go back to our business client to discuss the ramifications of the two different solutions.

Watch for the Empty Set!

As soon as you use aggregate functions in your SQL, you must be aware that empty sets might come into play. Listing 7 shows how aggregate functions react to empty sets. Apparently, `COUNT` returns zero whereas `AVG`, `SUM`, `MAX`, and `MIN` return null. This behavior does make a certain amount of sense. If you have no values to count, it's fair to say that you have zero values, whereas you can't really come up with, say, a maximum value without at least one value from which to choose. We can make a reasonable argument that `SUM` should return zero instead of null, that the sum of no values is zero, but Oracle's implementation of the behavior we describe here is fully compliant with the SQL standard.

Code Listing 7: Aggregate functions and the empty set

```
SELECT COUNT(EMPNO),AVG(EMPNO),SUM(EMPNO),MAX(EMPNO),MIN(EMPNO)
FROM EMP_M
WHERE 1 = 2;

COUNT(EMPNO)          AVG(EMPNO)          SUM(EMPNO)          MAX(EMPNO)          MIN(EMPNO)
____________________  ____________________  ____________________  ____________________  ____________________
0
```

Listing 8 shows two solutions to the problem of listing employees who are paid more than any sales rep in department 10. How is it possible that the two results are different? The first query retrieves (in the subquery) the maximum salary of all sales reps from department 10 and then compares all employee salaries, one by one, with that maximum salary. The second query does almost the same thing, the only difference being that each employee salary from the outer query is now compared against all salaries from department 10, as opposed to just the highest salary.

Code Listing 8: Who has a higher salary?

```
SELECT E1.ENAME FROM EMP_M E1
  WHERE E1.SAL >
        (SELECT MAX(E2.SAL) FROM EMP_M E2
         WHERE E2.DEPTNO = 10 AND E2.JOB = 'SALESREP');

No rows selected.

SELECT E1.ENAME FROM EMP_M E1
  WHERE E1.SAL > ALL
        (SELECT E2.SAL FROM EMP_M E2
         WHERE E2.DEPTNO = 10 AND E2.JOB = 'SALESREP');

ENAME
------------------
NORGAARD
LEWIS
ADAMS
...
MORLE

10 rows select
```

The empty set plays an important role here, because if you give our `EMP_M` table a closer look, you will see that department 10 has no sales reps. Therefore, the `MAX` function of the first query returns a null and the `WHERE` clause of the main query results in the value `UNKNOWN` for all employees. On the other hand, the second query returns all employees—because any salary is greater than all salaries in an empty set. Note that even Adams and Morle show up in the result, even though they both have a null salary. We must hasten to say that this is not a bug but fully expected behavior in accordance with the SQL standard. The lesson to take away here is to always, always ask yourself the question: "What if the aggregate function is applied against an empty set?"

Next Steps

READ more about
LogMiner
DBMS_LOGMNR package

Watch out for Nothing

Be vigilant for possible nulls when writing and processing your SQL statements. Remember the tools available to help you

work with nulls, including `CASE`, `COALESCE`, `IS NULL`, `IS NOT NULL`, `NULLIF`, `NVL`, and `NVL2`. Be aware of the third value when working with nulls in Boolean expressions—`UNKNOWN`—and remember that your business can look at nulls in different ways.

Lex de Haan (lex.de.haan@naturaljoin.nl) is an author and lecturer. He studied applied mathematics at the Technical University in Delft, The Netherlands; worked for Oracle from 1990 to 2004; and is a member of the OakTable Network. Jonathan Gennick (Jonathan@Gennick.com) is an experienced Oracle professional who enjoys writing on SQL topics. He wrote the SQL Pocket Guide and the Oracle SQL*Plus Pocket Reference, both from O'Reilly Media.

Send us your comments