Three-valued Boolean Logic

The Three-eyed Raven *likes it*

by Przemysław Kruglej
11-2013

przemyslawkruglej.com
przemyslaw.kruglej@gmail.com

Table of Contents

1 Shortly about chickens and roads...

Why did the chicken cross the road?

Because `1 > NULL AND cross_the_road()`...

I bet you have heard about *the* chicken and you know at least a dozen reasons why it had crossed the road. I wouldn't bet, though, that you have heard about the *three-valued logic*. And this one is interesting, it even **makes the chicken cross the road**. The answer to the above question could also be given as: Because it (the chicken) didn't short-circuit evaluate. What does it all mean? Read on.

2 The TRUE-FALSE-NULL Trio aka Three-valued Logic

Boolean expressions in PL/SQL may yield one of two possible values: `TRUE` or `FALSE`. True? Null. False, I mean! That sentence holds **only when all of the arguments of an expression are known** – and that means that neither of the arguments `IS NULL`.

Again, then. Boolean expressions in PL/SQL may yield one of three possible values: `TRUE`, `FALSE` or `NULL` and that's why the boolean logic in PL/SQL is called the "Three-valued Logic". When can a boolean expression be evaluated to `NULL`?

When you compare some value (even a `NULL`!) to a `NULL`, what do you expect? Since `NULL` can be taken as "unknown value", any comparison with something that is unknown must also yield a `NULL` result, which, again, means "result of this comparison is unknown". So, whenever any of the arguments of an expression is `NULL`, it is possible (why possible instead of always you'll understand soon enough) that the whole expression will be evaluated to `NULL`., which gives a third possible value of boolean expressions, alongside `TRUE` and `FALSE`, and those make a trio!

Let's take a look at the following example:

```
DECLARE
 v_bool BOOLEAN;
BEGIN
  v_bool := 1 > NULL;

  IF NOT v_bool THEN -- same as IF v_bool = false THEN
    dbms_output.put_line('v_bool = FALSE.');
  ELSIF v_bool IS NULL THEN
    dbms_output.put_line('v_bool IS NULL.');
  END IF;

  IF (NOT v_bool) IS NULL THEN
    dbms_output.put_line('NOT v_bool is also NULL when v_bool is NULL.');
  END IF;
END;
```

What output will you see when you execute the above block of PL/SQL code?

As I mentioned, comparing anything to `NULL` yields `NULL`, so expression `1 > NULL` evaluates to `NULL`. The expression:

```
NOT v_bool
```

Also evaluates to `NULL`, since what exactly does it mean to negate a `NULL`? The first branch of the `IF` statement is skipped, because it evaluates to `NULL`, not `TRUE`. On the other hand, condition in the next `ELSIF` branch will hold and the `v_bool IS NULL.` string will be displayed.

Since `NOT v_bool` is also `NULL`, then the second string `NOT v_bool is also NULL when v_bool is NULL.` will also be printed.

So, we will see the following output:

```
v_bool IS NULL.
NOT v_bool is also NULL when v_bool is NULL.
```

2.1 Exception to the rule

OK, I said that comparing anything to `NULL` yields `NULL`. I have to mention here the `DECODE` function, which is an exception to that rule. The `DECODE` function returns a value for corresponding expression if the first argument of `DECODE` is equal to value of that expression. `DECODE` is special, because if the first argument is `NULL`, and one of the expressions evaluates to `NULL`, then they will be treated as equal:

```
DECLARE
  v_val1 NUMBER;
  v_val2 NUMBER;
  v_val3 NUMBER;
  v_bool BOOLEAN;
BEGIN
  v_bool := v_val1 = v_val2;

  - DECODE will return:
  -   a) 1 if v_val1 is equal to v_val2
  -   b) 0 otherwise
  SELECT DECODE(v_val1, v_val2, 1, 0)
    INTO v_val3
    FROM dual;

  IF v_bool IS NULL THEN
    dbms_output.put_line('v_bool IS NULL.');
  END IF;

  IF v_val3 IS NULL THEN
    dbms_output.put_line('Result of DECODE: NULL');
  ELSE
    dbms_output.put_line('Result of DECODE: ' || v_val3);
  END IF;
END;
```

The output:

```
v_bool IS NULL.
Result of DECODE: 1
```

As you can see, normally comparison of two uninitialized variables (line `#7`) yields a `NULL` result (as proven by the `IF` statement in line `#16`). However, the `DECODE` function behaves differently – it treats two `NULL` values as equal and returns 1 instead of 0.

3 Short-circuit Evaluation

Short-circuit evaluation is the ability to skip execution and/or calculation of the second argument of an expression. It happens when the value of whole expression can be determined just by finding out the value of its first argument:

```
IF condition1 AND condition2 THEN
  -- code executed when both
  --   arguments of the above expression are TRUE
ELSE
```

```
  -- code executed when
  --   one of the arguments is either FALSE or NULL
END IF;
```

In the above example, if the value of `condition1` was either `FALSE` or `NULL`, then the `condition2` would not be evaluated at all – it would be skipped, because it is already known that the whole expression `condition1 AND condition2` **can not** be `TRUE`. Since there is an `ELSE` branch, the code associated with it would be executed. What is important to note here is that the `ELSE` branch is executed when all of the conditions in `IF..THEN..ELSIF..THEN` branches evaluate to either `FALSE` or `NULL`.

Short-circuit evaluation also works with the `OR` operator:

```
IF condition1 OR condition2 THEN
  -- code executed when either
  --   of the arguments of the above expression is TRUE
ELSE
  -- code executed when neither of the arguments is TRUE
END IF;
```

Like previously, `condition2` will not have to be evaluated when the value of `condition1` can determine the result of the whole expression, regardless of the value of `condition2`. This will happen when `condition1` will be evaluated to `TRUE`, because value of whole expression will be `TRUE` irrespectively of the value of `condition2`.

Why do I explain all of this? And where does the chicken fit in all of this?

4 Example of short-circuit evaluation not happening

What if I told that the short-circuit evaluation doesn't always take place? What if I told you it depends on the *context* of the evaluated expression? Well, I would be telling you the truth.

Before I talk about the details, let's see if the chicken will cross the road after all (and if so, will it come back?).

4.1 Quiz Time!

Given the following block of PL/SQL code, what output is going to be displayed after its execution?

```
DECLARE
  v_bool BOOLEAN;

  FUNCTION cross_the_road RETURN BOOLEAN
  AS
  BEGIN
    dbms_output.put_line('Chicken crosses the road.');
    RETURN TRUE;
  END;
BEGIN
  v_bool := 1 > NULL AND cross_the_road();

  IF 1 > NULL AND cross_the_road() THEN
    dbms_output.put_line('Chicken is back.');
  ELSE
    dbms_output.put_line('Erm... where did it go?!');
  END IF;
END;
```

Possible answers:

a) Chicken crosses the road.
   Chicken crosses the road.
   Chicken is back.

b) Chicken crosses the road.
   Chicken crosses the road.
   Erm... where did it go?!

c) Chicken crosses the road.
   Erm... where did it go?!

d) Chicken crosses the road.
   Chicken is back.

e) Erm... where did it go?!

f) None of the above, because an error will be thrown.

The answer is in the chapter that follows. If you're not sure, I can give you a hint – the key to give the correct answer lies in the truth table:

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	NULL	NULL	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	NULL	FALSE	NULL	TRUE
NULL	TRUE	NULL	TRUE	NULL
NULL	FALSE	FALSE	NULL	NULL
NULL	NULL	NULL	NULL	NULL

4.2 Truth about the truth table and the puzzle solved

The correct answer to the puzzle is c). I'm curious if that was your choice. Either way, let me walk you through what exactly is happening in that piece of code.

```
DECLARE
  v_bool BOOLEAN;

  FUNCTION cross_the_road RETURN BOOLEAN
  AS
  BEGIN
    dbms_output.put_line('Chicken crosses the road.');
    RETURN TRUE;
  END;
BEGIN
  v_bool := 1 > NULL AND cross_the_road();

  IF 1 > NULL AND cross_the_road() THEN
    dbms_output.put_line('Chicken is back.');
  ELSE
```

```
    dbms_output.put_line('Erm... where did it go?!');
  END IF;
END;
```

The most important line in that code is line `#11`. You could've expected that short-circuit evaluation will take place here since the value of the first argument (`1 > NULL`) yields `NULL`, and the execution of the `cross_the_road` function can be skipped.

As I have mentioned, whether short-circuit evaluation will be utilized, or not, depends on the context in which an expression is to be evaluated. Take a look once again at the truth table. I have marked the interesting results of boolean expressions:

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	NULL	NULL	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	NULL	FALSE	NULL	TRUE
NULL	TRUE	**NULL**	TRUE	NULL
NULL	FALSE	**FALSE**	**NULL**	NULL
NULL	NULL	**NULL**	NULL	NULL

Now, **when a value of an expression with `AND` operator is assigned to a variable, and the first argument of that expression is `NULL`, the short-circuit evaluation will not take place**. Why? Because, even though the first argument is `NULL`, the value of the *whole* expression doesn't have to be `NULL`! It may, in fact, have one of two possible values: `NULL` or `FALSE`.

As pointed out in the truth table, when the first argument is `NULL` and we use the `AND` logical operator, the value of an expression is determined by expression's second argument:

- if the second argument is either `TRUE` or `NULL`, the whole expression will be evaluated to `NULL`,
- if the second argument is `FALSE`, then the whole expression is `FALSE` as well.

I've said in the beginning of my article that, when the first argument is `NULL`, the whole expression *may* be `NULL` as well, and now you see the reason – it may be `NULL`, but it may also be `FALSE`.

The reason there are those two possible values is because when we use `AND` operator with `NULL` and `TRUE` arguments, then we can't really say whether the whole expression is `TRUE` or `NULL`. Since `NULL` means *unknown*, `TRUE` and unknown must also be unknown.

I'd also like to point out that, when `OR` operator is used, result of expression involving `NULL` and `FALSE/NULL` arguments also yields `NULL`, because we can't say if `NULL OR FALSE/NULL` is `FALSE` or `TRUE` – it is, once again, unknown.

What this all means is that when you assign a value of an expression to a variable, the short-circuit evaluation will not be utilized if the first argument evaluates to `NULL`, because to determine the value of the whole expression, PL/SQL must also calcuate the value of the second argument.

Now, let's look once again at `IF` control statement. The code associated with a branch is executed if

the corresponding condition is evaluated to `TRUE`, and only `TRUE`. Because of that, **when the first argument of an expression using `AND` operator in `IF`'s condition is `NULL`, the short-circuit evaluation is utilized –** it doesn't matter if the *whole* expression is `NULL` or `FALSE`, because PL/SQL already knows that the whole expression can not be `TRUE` and evaluation of second argument can be skipped.

This is the difference between contexts – when you assign value of an expression to a variable, the value of the **whole** expression must be calculated if the first argument is `NULL`, whereas in an `IF` statement, it is enough that the first argument will yield `NULL` to move to the next `IF` branch condition.

OK, let's get back to the code. As I have *shortly* explained, in line `#11`:

```
v_bool := 1 > NULL AND cross_the_road();
```

The short-circuit evaluation will not take place, because the first argument of the expression evaluates to `NULL` and the `cross_the_road` function will be called, and the

```
Chicken crosses the road.
```

text will be displayed.

On the other hand, in line `#13`:

```
IF 1 > NULL AND cross_the_road() THEN
```

the short-circuit evaluation **will** be utilized – the `1 > NULL` argument will be evaluated to `NULL`, the whole expression then can not be `TRUE`, so the execution of `cross_the_road` function can be skipped, so neither will the text "`Chicken crosses the road.`" be printed again nor will we see the "`Chicken is back.`" in the standard output.

Since the condition in the `IF` branch doesn't hold, the execution flow moves to the `ELSE` branch, and the:

```
Erm... where did it go?!
```

string is printed.

So, the correct answer is c) – we will see the following output:

```
Chicken crosses the road.
Erm... where did it go?!
```

4.2.1 The implication of the truth table in plain language

When an expression with `AND` operator is being evaluated and:

1. First argument is evaluated to NULL, then:
   a) The whole expression is NULL, if the second argument is either TRUE or NULL.
   b) The whole expression is FALSE, if the second argument is FALSE.
2. First argument is evaluated to FALSE, then
   a) The whole expression is FALSE, regardless of the value of the second argument.
3. First argument is evalutaed to TRUE, then:
   a) The whole expression is NULL, if the second argument is NULL.

b) The whole expression is TRUE, if the second argument is TRUE.

c) The whole expression is FALSE, if the second argument is FALSE.

When an expression with `OR` operator is being evaluated and:

1. First argument is evaluated to NULL, then:
   a) The whole expression is NULL, if the second argument is either FALSE or NULL.
   b) The whole expression is TRUE, if the second argument is TRUE.
2. First argument is evaluated to FALSE, then
   a) The whole expression is NULL, if the second argument is NULL.
   b) The whole expression is TRUE, if the second argument is TRUE.
   c) The whole expression is FALSE, if the second argument is FALSE.
3. First argument is evalutaed to TRUE, then:
   a) The whole expression is TRUE, regardless of the value of the second argument.

For each of the cases, I have put the information whether the short-circuit evaluation will take place or not in different contexts in the table below. `Yes` stands for "s*hort-circuit will take place*" and `No` means it won't:

x	y	**Assigning value of an expression to a variable**		**Expression evaluated for IF control statement**	
		`x AND y`	`x OR y`	`x AND y`	`x OR y`
TRUE	TRUE	No	Yes	No	Yes
TRUE	FALSE	No	Yes	No	Yes
TRUE	NULL	No	Yes	No	Yes
FALSE	TRUE	Yes	No	Yes	No
FALSE	FALSE	Yes	No	Yes	No
FALSE	NULL	Yes	No	Yes	No
NULL	TRUE	**No**	No	**Yes**	No
NULL	FALSE	**No**	No	**Yes**	No
NULL	NULL	**No**	No	**Yes**	No

As already mentioned, the short-circuit evaluation will not take place in an assignment when `AND` operator is used and first argument is evaluated to `NULL`, whereas, in the same situation, the short-circuit evaluation will be utilized in an `IF` statement.

5 Conclusion

As promised, the three-valued logic made the chicken cross the street (and short-circuit evaluation made it stay there).

It is important to stress the importance of the the short-circuit evaluation not taking place in some cases. You should avoid the situations when the logic in your code depends on whether the short-circuit evaluation will be utilized or not. Whether the second argument of an expression depends on the first not being `NULL` or evaluating it takes a long time, you should mind that it may not work as you would expect in the first place.

Hoping you enjoyed my article. If you have found any errors in it (even typos), you think that I haven't explained anything clearly enough or you have an idea how I could make the article better – please, do not hesitate to contact me, or leave a comment.