Table of Contents

We keep coming back to nulls. We will start with a review of testing for nulls and how nulls work with other tests. Then we will look at a MySQL extension.

We have the following table which we used previously.

```
select *
from a_testbed.z_tst_nulls;
+--------+------------+---------+-----------+
| col_id | col_string | col_int | col_float |
+--------+------------+---------+-----------+
|      1 | abc        |      10 |    10.567 |
|      2 | abc        |    NULL |    20.222 |
|      3 | NULL       |      30 |      NULL |
|      4 | NULL       |    NULL |      NULL |
|      5 |12          |      12 |      12.2 |
+--------+------------+---------+-----------+
```

1. Testing for nulls

Demo 01: Use the operators Is Null and Is Not Null. These are ANSI standard operators and work with any data type

```
select col_id, col_int
from a_testbed.z_tst_nulls
where col_int is null;
+--------+---------+
| col_id | col_int |
+--------+---------+
|      2 |    NULL |
|      4 |    NULL |
+--------+---------+

select col_id, col_int
from a_testbed.z_tst_nulls
where col_int is not null;
+--------+---------+
| col_id | col_int |
+--------+---------+
|      1 |      10 |
|      3 |      30 |
|      5 |      12 |
+--------+---------+
```

2. Tests and null values

Demo 02: Nulls do not pass other tests. The null values in col_int are not greater than 15 and they are not equal to 15 and they are not more than 15. They are just null.

```
select col_id, col_int
from a_testbed.z_tst_nulls
where col_int > 15
;
```

```
+--------+---------+
| col_id | col_int |
+--------+---------+
|      3 |      30 |
+--------+---------+

select col_id, col_int
from a_testbed.z_tst_nulls
where col_int > 15 or col_int <= 15;
+--------+---------+
| col_id | col_int |
+--------+---------+
|      1 |      10 |
|      3 |      30 |
|      5 |      12 |
+--------+---------+
```

3. The null safe equality operator

MySQL provides an extension called the null safe equality operator <=>.

Demo 03: Suppose we set up a variable and assign it a value

```
set @tst_1 = 10;
Query OK, 0 rows affected (0.00 sec)
```

Demo 04: Then we use that variable against our table using the regular = operator.

```
select col_id, col_int
from a_testbed.z_tst_nulls
where col_int = @tst_1
;
+--------+---------+
| col_id | col_int |
+--------+---------+
|      1 |      10 |
+--------+---------+
```

Demo 05: Then we test using the <=> operator and get the same result.

```
select col_id, col_int
from a_testbed.z_tst_nulls
where col_int <=> @tst_1
;
+--------+---------+
| col_id | col_int |
+--------+---------+
|      1 |      10 |
+--------+---------+
```

Demo 06: Now we use a variable which has not been initialized and is therefore null.

The first query uses the regular = operator and the two nulls do not equal each other. This is the standard correct meaning for comparing nulls

```
select col_id, col_int
from a_testbed.z_tst_nulls
where col_int = @tst_2
;
```

```
Empty set (0.00 sec)
```

Demo 07: But if we test with the <=> operator we get the rows returned where both of the operands are null

```
select col_id, col_int
from a_testbed.z_tst_nulls
where col_int <=> @tst_2;
+--------+---------+
| col_id | col_int |
+--------+---------+
|      2 |    NULL |
|      4 |    NULL |
+--------+---------+
```

Demo 08: What does the null-safe equality operator return? Remember that `@tst_1` has the value 10 and `@tst_2` is null.

```
select col_id, col_int, col_int <=> @tst_1, col_int <=> @tst_2
from a_testbed.z_tst_nulls;
+--------+---------+--------------------+--------------------+
| col_id | col_int | col_int <=> @tst_1 | col_int <=> @tst_2 |
+--------+---------+--------------------+--------------------+
|      1 |      10 |                  1 |                  0 |
|      2 |    NULL |                  0 |                  1 |
|      3 |      30 |                  0 |                  0 |
|      4 |    NULL |                  0 |                  1 |
|      5 |      12 |                  0 |                  0 |
+--------+---------+--------------------+--------------------+
```

NULL-safe equal. This operator performs an equality comparison like the = operator, but returns 1 if both operands are NULL, and 0 if one operand is NULL. If neither operand is null, then the null-safe operator does the regular equality test. Remember that 1 is treated as True and 0 as false.

Row with col_id =1.
    Col_int is 10 and the test against @tst_1 returns 1 since the two values are the same.
    The test against @tst_2 returns 0 since only one of the operands is null

Row with col_id =3.
    Col_int is 30 and the test against @tst_1 returns 0 since the two values are not the same.
    The test against @tst_2 returns 0 since only one of the operands is null

Row with col_id =2 and 4.
    Col_int is null and the test against @tst_1 returns 0 since only one of the operands is null.
    The test against @tst_2 returns 1 since both of the operands are null

So essentially, this operator treats two nulls as if they are the same. This is not the rule in standard SQL so take care when using this approach if you are working in multiple dbms.

4. Should you use this operator?

This is a decision you will face many times. This operator is a MySQL extension. If you are writing sql that you need to move between various dbms databases, then you would want to avoid these extensions. If you are writing code that will be used only on MySQL databases then you might consider these extensions if they make your code easier to understand (therefore easier to maintain) and/or more efficient (because this dbms implements this feature more efficiently than the standard approach). In that case it would be a good idea to comment your code. You do not want the maintenance programmer to "fix" your code by replacing that funny <=> with the more common = operator.

The next question that you need to consider with some extensions is should you be writing SQL code that goes against the traditional rdbms standards- in this case that two nulls are not to be treated as equal. Again this should be commented if you use the extension.

There are traditionalists who want all SQL to work as close to the standard as possible and these people do not appreciate the attitude that you should use all extensions because they are "developer friendly". Then there are people who want the fastest and most flexible way to implement applications that do what they need done. Tempers and language flare!