Table of Contents

When we come to testing strings - char and varchar- we need to look more closely at how these types work in MySQL. Treatment of strings and case-sensitivity and trailing spaces in strings can differ with different dbms and also within versions of MySQL. For some of this, we will need to do some testing and also refer to the MySQL manuals. This material can get quite confusing. Read through this and try some experiments to see some of the issues with strings in MySQL.

1. Case sensitivity

In your a_testbed database , create a small table

```
Create table a_testbed.z_tst_case (
  col_id int primary key
, col_1 varchar(10) not null
);
```

Insert a few rows with differing case for col_1.

```
Insert into a_testbed.z_tst_case values (1, 'zebra');
Insert into a_testbed.z_tst_case values (2, 'ZEBRA');
Insert into a_testbed.z_tst_case values (3, 'Zebra');
Insert into a_testbed.z_tst_case values (4, 'ZEBra');
```

Demo 01: Filter for 'zebra' and all of the rows match.

```
select  col_id, col_1
from a_testbed.z_tst_case
where col_1 = 'zebra';
+--------+-------+
| col_id | col_1 |
+--------+-------+
|      1 | zebra |
|      2 | ZEBRA |
|      3 | Zebra |
|      4 | ZEBra |
+--------+-------+
```

With our default character set and collation, string comparison is case insensitive. Suppose you do not want that behavior. You can specify a binary operator in the test; using the binary operator means that the comparison is done on a byte by byte comparison

Demo 02: Forcing a case sensitive test.

```
select  col_id, col_1
from a_testbed.z_tst_case
where binary col_1 = 'zebra';
+--------+-------+
| col_id | col_1 |
+--------+-------+
|      1 | zebra |
+--------+-------+
select  col_id, col_1
from a_testbed.z_tst_case
where binary col_1 = 'ZEBra';
+--------+-------+
| col_id | col_1 |
+--------+-------+
|      4 | ZEBra |
+--------+-------+
```

You can use binary on the either part of the comparison.

```
select  col_id, col_1
from a_testbed.z_tst_case
where col_1 = binary 'ZEBRA';
+--------+-------+
| col_id | col_1 |
+--------+-------+
|      2 | ZEBRA |
+--------+-------+

select  col_id, col_1
from a_testbed.z_tst_case
where binary col_1 IN( 'ZEBRA', 'Zebra');
+--------+-------+
| col_id | col_1 |
+--------+-------+
|      2 | ZEBRA |
|      3 | Zebra |
+--------+-------+
```

Demo 03: Another feature of using binary is that trailing spaces are significant.

Without binary

```
select  col_id, col_1
from a_testbed.z_tst_case
where col_1 = 'zebra  ';
+--------+-------+
| col_id | col_1 |
+--------+-------+
|      1 | zebra |
|      2 | ZEBRA |
|      3 | Zebra |
|      4 | ZEBra |
+--------+-------+
```

With binary

```
select  col_id, col_1 from a_testbed.z_tst_case where binary col_1 = 'zebra  ';
Empty set (0.00 sec)

select  col_id, col_1 from a_testbed.z_tst_case where col_1 = binary 'zebra  ';
Empty set (0.00 sec)
```

2. Storing Char and Varchar data

I want to start this with the demos. Create the following table in your a_testbed database.

Demo 04: Create table z_tst_strings

```
create table a_testbed.z_tst_strings(
   col_id int primary key
 , col_1 char(10)
 , col_2 varchar(10)
 );
```

Col_1 is of the char type which is a fixed length column - in this case length 10. When you insert data into this column, it is end-padded with spaces to a length of 10.

Col_2 is of the varchar type which means it can store up to 10 characters of data and it is not end-padded with spaces.

We are going to insert the string 'zebra' into these two columns with a variety of leading and trailing spaces.

Look at the following inserts and try to figure out what will be stored with each row. Think about this before you do the inserts. Then execute these insert queries.

Demo 05: Inserts for z_tst_strings

```
-- no explicit spaces
   insert into a_testbed.z_tst_strings values (1, 'zebra',      'zebra');
-- two trailing spaces for col_2
   insert into a_testbed.z_tst_strings values (2, 'zebra',      'zebra  ');
-- two leading spaces for col_2
   insert into a_testbed.z_tst_strings values (3, 'zebra',      '  zebra');
-- explicit spaces for col_1 to a length of 10
   insert into a_testbed.z_tst_strings values (4, 'zebra     ', 'zebra');
-- explicit spaces for col_1 and col_2 to a length of 10
   insert into a_testbed.z_tst_strings values (5, 'zebra     ', 'zebra     ');
```

Demo 06: Now display the data

```
select  col_id, col_1, col_2
from a_testbed.z_tst_strings
;
+--------+-------+------------+
| col_id | col_1 | col_2      |
+--------+-------+------------+
|      1 | zebra | zebra      |
|      2 | zebra | zebra      |
|      3 | zebra |   zebra    |
|      4 | zebra | zebra      |
|      5 | zebra | zebra      |
+--------+-------+------------+
```

You can see the leading spaces for col_2 in row 3. You might be suspicious of the column width used for col_1. But we need to see any trailing spaces. We can do this by concatenating a character to the end of the string. I am also going to use the function that tells us the length of the string.

Demo 07: Concatenate and display the data

```
select  col_id
, concat(col_1, 'X') as col_1, length(col_1)
, concat(col_2, 'X') as col_2, length(col_2)
from a_testbed.z_tst_strings ;
```

```
+--------+--------+---------------+-------------+---------------+
| col_id | col_1  | length(col_1) | col_2       | length(col_2) |
+--------+--------+---------------+-------------+---------------+
|      1 | zebraX |             5 | zebraX      |             5 |
|      2 | zebraX |             5 | zebra  X    |             7 |
|      3 | zebraX |             5 |   zebraX    |             7 |
|      4 | zebraX |             5 | zebraX      |             5 |
|      5 | zebraX |             5 | zebra     X |            10 |
+--------+--------+---------------+-------------+---------------+
```

Looking at col_1 we don't see any trailing spaces even though this was defined as a char(10) type column. The length is always 5- for the 5 characters in "zebra"

Looking at col_2 we can see that the trailing spaces as well as the leading spaces were preserved and the length reflects this.

To quote the manual:

"When CHAR values are stored, they are right-padded with spaces to the specified length. When CHAR values are retrieved, trailing spaces are removed."   That might not be what you expected but this is a pretty clear statement of intent.
" VARCHAR values are not padded when they are stored. Trailing spaces are retained when values are stored and retrieved, in conformance with standard SQL

(In some earlier versions of MySQL this is not the case; before MySQL 5.0.3, trailing spaces are removed from values when they are stored into a VARCHAR column; this means that the spaces also are absent from retrieved values. This means that if you are working with a variety of MySQL versions you will need to watch out for this. There is a trend in MySQL to come closer to the standards with each new version-but that does mean that you may have to watch code that worked in one version and now works differently.)

The question of whether trailing spaces should be stored in **your** database is a **business decision**. You can have the application that supplies data to your insert statement remove trailing spaces- and maybe remove leading spaces. It is not common to remove internal spaces- so you would not change "`Grevy's Zebra`" to "`Grevy'sZebra`" but you might want the application to collapse contiguous spaces changing "`Grevy 's   Zebra`" to "`Grevy's Zebra`".

There is an option PAD_CHAR_TO_FULL_LENGTH which forces char types to return the full character values without stripping the trailing blanks.

3. Comparing string data and trailing spaces

Now let's go back to our table and run the following query against col_2.

Demo 08:   Testing and trailing spaces

```
select  col_id, col_2,  concat(col_2, 'X')
from a_testbed.z_tst_strings
where col_2 = 'zebra';
+--------+------------+--------------------+
| col_id | col_2      | concat(col_2, 'X') |
+--------+------------+--------------------+
|      1 | zebra      | zebraX             |
|      2 | zebra      | zebra  X           |
|      4 | zebra      | zebraX             |
|      5 | zebra      | zebra     X        |
+--------+------------+--------------------+
```

We get back the rows where col_2 stored the string "zebra' and also where it stored "zebra" with trailing spaces even though they are not the same value.
" All MySQL collations are of type PADSPACE. This means that all CHAR and VARCHAR values in MySQL are compared without regard to any trailing spaces. … This is true for all MySQL versions. "

Leading spaces are significant for comparisons; trailing blanks are not.

Demo 09: Binary testing and trailing spaces

```
select  col_id, col_2,  concat(col_2, 'X')
from a_testbed.z_tst_strings
where col_2 = binary 'zebra';
+--------+-------+--------------------+
| col_id | col_2 | concat(col_2, 'X') |
+--------+-------+--------------------+
|      1 | zebra | zebraX             |
|      4 | zebra | zebraX             |
+--------+-------+--------------------+
```

Demo 10: Binary testing and trailing spaces

```
select  col_id, col_2,  concat(col_2, 'X')
from a_testbed.z_tst_strings
where col_2 = binary 'zebra  ';
+--------+---------+--------------------+
| col_id | col_2   | concat(col_2, 'X') |
+--------+---------+--------------------+
|      2 | zebra   | zebra  X           |
+--------+---------+--------------------+
```

Binary testing is not the same as using the collation and it is sensitive to trailing spaces.

At this point your head is probably spinning. This is not difficult but it is a lot to keep in mind when you work with data that comes into your database from sources with varying attitudes towards whitespace. Often the best solution is to clean the data before it is inserted. You might wish to remove leading and trailing spaces. You might have a policy of case patterns for data (although this is much more difficult for real data (e.e. cummings comes to mind) but maybe you could agree on a case pattern for some of the data. Again- the decision about the significance of whitespace in the data is a business decision. MySQL just has its own rules about how to deal with the data you enter.

4. Strings as numbers

Under some circumstances, MySQL will treat alphanumeric string as numbers and may cast them to zero. This is not always a good idea- particularly when you do not pay close attention to your data.

4.1. Casting string literals to numbers

Let's start with literals; the output probably seems OK to you.

Demo 11:

```
select   '450' + 5, '25.08' * 6;
+-----------+-------------+
| '450' + 5 | '25.08' * 6 |
+-----------+-------------+
|       455 |      150.48 |
+-----------+-------------+
```

Now try some other literals; this does give you warnings on the expression but it does treat these the string as 0 values. It does not say that it is invalid to treat 'ABC' as a number.

Demo 12:

```
select   'ABC' + 5;
+-----------+
| 'ABC' + 5 |
+-----------+
|         5 |
+-----------+
1 row in set, 1 warning (0.00 sec)
show warnings;
+---------+------+-------------------------------------------+
| Level   | Code | Message                                   |
+---------+------+-------------------------------------------+
| Warning | 1292 | Truncated incorrect DOUBLE value: 'ABC'   |
+---------+------+-------------------------------------------+
```

Demo 13: You might have hoped that the result would be 15.25- but the first operand is a string. You would get the same result with 5 + '10.25'

```
select   '$10.25' + 5;
+--------------+
| '$10.25' +5 |
+--------------+
|            5 |
+--------------+
1 row in set, 1 warning (0.00 sec)
show warnings;
+---------+------+-------------------------------------------+
| Level   | Code | Message                                   |
+---------+------+-------------------------------------------+
| Warning | 1292 | Truncated incorrect DOUBLE value: '$10.25' |
+---------+------+-------------------------------------------+
1 row in set (0.00 sec)
```

Demo 14: Adding a number to a null returns a null; there is no warning associated with this.

```
select   null  + 5;
+----------+
| null + 5 |
+----------+
|    NULL  |
+----------+
```

Demo 15: But what if the literal string contains digits? What pattern do you see here?

```
select  '123ABC' + 10, '123ABC' * 1, '123,456' * 1;
+---------------+--------------+---------------+
| '123ABC' + 10 | '123ABC' * 1 | '123,456' * 1 |
+---------------+--------------+---------------+
|           133 |          123 |           123 |
+---------------+--------------+---------------+
1 row in set, 3 warnings (0.00 sec)
```

You might like this feature or not- but you do need to be aware of it.

4.2. Using a string column as a predicate

Another demo of string features you might not have expected and probably should not use.

We are using the expression col_1 in a place where we should have a logical expression. MySQL casts this to a number -0 and then treats 0 as a logical False value.

Demo 16:

```
select  col_id, col_1
from a_testbed.z_tst_case
where col_1;
```

```
Empty set (0.08 sec)
```

When we add 1 to col_1 we get 0 + 1 = 1 which is cast to a logical True value.

```
select  col_id, col_1, col_1 +1
from a_testbed.z_tst_case
where col_1 + 1
;
+--------+-------+-----------+
| col_id | col_1 | col_1 + 1 |
+--------+-------+-----------+
|      1 | zebra |         1 |
|      2 | ZEBRA |         1 |
|      3 | Zebra |         1 |
|      4 | ZEBra |         1 |
+--------+-------+-----------+
4 rows in set, 8 warnings (0.00 sec)

mysql> show warnings;
+---------+------+-------------------------------------------+
| Level   | Code | Message                                   |
+---------+------+-------------------------------------------+
| Warning | 1292 | Truncated incorrect DOUBLE value: 'zebra' |
| Warning | 1292 | Truncated incorrect DOUBLE value: 'zebra' |
| Warning | 1292 | Truncated incorrect DOUBLE value: 'ZEBRA' |
| Warning | 1292 | Truncated incorrect DOUBLE value: 'ZEBRA' |
| Warning | 1292 | Truncated incorrect DOUBLE value: 'Zebra' |
| Warning | 1292 | Truncated incorrect DOUBLE value: 'Zebra' |
| Warning | 1292 | Truncated incorrect DOUBLE value: 'ZEBra' |
| Warning | 1292 | Truncated incorrect DOUBLE value: 'ZEBra' |
+---------+------+-------------------------------------------+
8 rows in set (0.00 sec)

select  *
from z_tst_case
where col_1 + 3
;
+--------+-------+
| col_id | col_1 |
+--------+-------+
|      1 | zebra |
|      2 | ZEBRA |
|      3 | Zebra |
|      4 | ZEBra |
+--------+-------+
```

Without running it- what do you think this querywould return? Then try it.

```
   select  *
   from a_vets.vt_animals
   where an_name;
```

Try this one also

```
   select  *
   from a_vets.vt_animals
   where not an_name;
```

4.3. Using a number as a predicate- Do not do this without a Very Good Reason

This is not a string issue, but it seems relevant here in a discussion of MySQL features. What does the following query do?

```
   select an_id, an_name
   from   a_vets.vt_animals
   where  45;
```

We get all of the rows returned.

In Oracle or in SQL Server, this query does not run- it is an error to use a number as a predicate. But MySQL will execute that query returning all of the rows from the table.

The following query in Mysql returns **no rows** in the result set.

```
   select an_id, an_name
   from   a_vets.vt_animals
   where  0;
```

There is a convention that you see in some languages that equate the number 0 with the logical value False and other numbers with the logical value True. You see this fairly often in MySQL code and the people who do this seem pleased with themselves. I would prefer you write code that is more obvious.

Now let's make this a bit more complex- I see students writing this type of code fairly often and it is almost always logically incorrect.

In the animals table we have three animals with names that start with a digit.

```
   | 17025 | 25             |
   | 17026 | 3P#_26         |
   | 17027 | 3P#_25         |
```

What happens when we run the query shown here? First try to think what the person might have meant. This looks to me like they want rows for animals that have a name.

```
   select an_id, an_name
   from a_vets.vt_animals
   where an_name;
```

But we do not get that; instead we get those three animals and a warning on every other row in the table which has a value for the an_name.

```
   +-------+---------+
   | an_id | an_name |
   +-------+---------+
   | 17025 | 25      |
   | 17026 | 3P#_26  |
   | 17027 | 3P#_25  |
   +-------+---------+
   3 rows in set, 28 warnings (0.00 sec)

   Warning (Code 1292): Truncated incorrect INTEGER value: 'Gutsy'
   Warning (Code 1292): Truncated incorrect INTEGER value: 'Kenny'
   Warning (Code 1292): Truncated incorrect INTEGER value: 'Mr Peanut'
   Warning (Code 1292): Truncated incorrect INTEGER value: 'Gutsy'
   . . . Lots more of these
```

The message is saying that you are somehow treating the an_name attribute as if it were an Integer- and that is not right. This is a Warning that you should never ignore.

So what happened? The people who wrote the MySQL language made several decisions that are more common in scripting languages than in dbms systems.

Decision 1: If you use a string value in a place in a query where MySQL expects a number, MySQL will try to cast that string to a number. This is discussed in Section 4 above. The name '25' is cast to the integer 25; the names ' 3P#_26' and ' 3P#_25' are cast to 3. Names such as 'Gutsy' and 'Kenny' are cast to 0.

```
   Warning (Code 1292): Truncated incorrect INTEGER value: 'Gutsy'
   Warning (Code 1292): Truncated incorrect INTEGER value: 'Kenny'
```

Decision 2: The Where clause expects a test / predicate which has the value True or False or Unknown. But the query is now getting values such as 25, 3, and 0 instead. Many DBMs would reject that as bad syntax. MySQL has an attitude to try to keep going.

If you have a number in a place where MySQL: expects a True /False value, MySQL will use a numeric value as being True or False - the number 0 is treated as False and other numbers (such as 24 and 3) as True.

Now we can put this all together.

Animal 10002 has an_name 'Gutsy'. animal 17026 has an_name '3P#_25'. The row for animal 10002 has the Where clause WHERE 'Gutsy' which becomes WHERE 0 which is evaluated as False and that row is not returned.

The row for animal 17026 has the Where clause WHERE '3P#_25' which becomes WHERE 3 which is evaluated as True and that row is returned.

Hopefully, you will never write a query which uses a Where clause such as Where an_name which is filtering for an_names that start with a digit. If you really need to do that, you use a regular expression- we get to these in another unit.

```
   select an_id, an_name
   from a_vets.vt_animals
   where an_name regexp '^[0-9]';
```

I also see queries such as these which run but are unnecessarily confusing.

```
   select ex_id, srv_id, ex_fee
   from a_vets.vt_exam_details
   where ex_fee;

   select ex_id, srv_id, ex_fee
   from a_vets.vt_exam_details
   where NOT ex_fee;
```

It take little extra coding to write better queries.

```
   select ex_id, srv_id, ex_fee
   from a_vets.vt_exam_details
   where ex_fee <> 0;

   select ex_id, srv_id, ex_fee
   from a_vets.vt_exam_details
   where ex_fee = 0;
```