## Table of Contents



Data can be manipulated with operators, such as the concatenation operator for text and the arithmetic operators for numbers. Another way to manipulate data is with built-in functions which we will get to soon. These manipulations result in new values being displayed in the query output or used in the Where clause or Order By clause. The queries shown here do not change the data in the tables.

MySQL will attempt to do automatic conversion of values from one data type to another. Relying on automatic conversion is risky. You should not assume that your data values are always clean or that the conversions work as you anticipate. With MySQL you will find differences in how these features handle implicit casting depending on the MySQL version and the database settings.

# 1. Data types and literals

There are several basic data types available for data to be stored in a table or to be used in expressions. The basic types are:

- String types
- Numeric types
- Date and time types

For our tables we will use only a few types of data. MySQL supports more data types and you can find information about all of these in the Van Lans book pages 496-510.

## 1.1. Character strings

String types have two general types

- Char(99)
- Varchar (99)

CHAR is used for fixed length strings. We use CHAR (2) to store state abbreviations since they all have a 2 letter. If you are storing data where all of the data has the same number of characters- such as SSN, ISBN13, some product codes, then CHAR is appropriate. If necessary the data will be end-padded with blanks to the stated length when stored.

However with MySQL, any trailing blanks in a char attribute are removed when the values are retrieved. Suppose you have an attribute defined as a char (5). You insert into this attribute the value 'cat'. The value is stored as 'cat ' but if you display this attribute in a Select statement, the value display is 'cat' ; if you display the length of the attribute for the row storing 'cat ', it displays a length of 3. This is not a common behavior across different dbms, so you need to watch out for this.

VARCHAR is used for variable length strings. We could use VARCHAR (25) to store customer last names assuming we won't need to store a name longer than 25 characters.

For both of these - if you are defining a table column with char or varchar and try to insert a value longer than the defined length, you will get an error. Char defaults to a length of 1 if you do not state a length; varchar requires a length.

## 1.1. Numbers

Numeric types are divided into two categories

- Exact numbers
  - Numbers that store integers- like 15, 0, -35
  - Fixed point numbers which store numbers with a set number of digits- such as a number that can have a total of 6 digits and two of those digits are after the decimal. So that type could store a value such as 1234.56 or 0.3 but not a value such as 34.5678
- Approximate numbers
  - Floating point numbers that store values such as 5.876E2

Each dbms has further distinctions between types. There is generally a numeric type that takes one byte of storage and stores numbers that range between 0 and 255 or between -128 and +127 and there is generally a numeric type that takes 8 bytes of storage and stores integers with up to 19 digits. And there are other integer types between these. And just to make life more interesting many dbms have synonyms for the various types.

This is the type of stuff that you can look up in a book or manual when you need to know the range of each type. What you do need to know about numeric types is:

- Integer types are exact representation of numbers.
- Integers are faster for calculations
- Fixed decimal types may take more space and more processing time.
- Floating point numbers are stored as approximation of the value; these vary with the overall size of the values they can store ( up to E308) and the numbers of digits of accuracy. These types might be appropriate for storing a value such as a weight or distance.

### 1.1.1. Integers

We usually use INT which will store number between approximately +-2,000,000, 000.

There are other integer types which have different ranges: from the smallest to the largest

tinyint (-128 to +127) ;smallint; mediumint, int; bigint

MySQL also has something called a data type option- an additional bit of information you can add to a data type when declaring a table. Suppose you want a tinyint column but you do not want any negative values to be stored. You can declare the column as **tinyint unsigned**. This means that negative values cannot be inserted and also the range has shifted. A tinyint range is -128 to +127 the range for a tinyint unsigned is 0 to 255.

You can specify unsigned with any of the numeric types; the range shift occurs with the integer types.

### 1.1.2. Fixed precision

Decimal, numeric this lets you set up a type such as decimal(6,2) which can hold numbers from -9999.99 to +9999.99. The first value is the number of digits and the second the number of digits after the decimal point

## 1.1. Temporal Data

Date and Time data types vary more between the various dbms.

MySQL has a DateTime type which stores a point in time. It also has a Date type which has only the date part and a Time type which stores only a time of day part. MySQL also has a Year type which stores year values as an integer and save some space over the various integer types. The range for the dates is '1000-01-01' to '9999-12-31'.

With some settings, MySQL allows you to store what looks like a partial date:, such as 1976-04-00 and invalid dates such as 2015-02-31. We will not discuss these in this class.

There are other special purpose data types which we will look at over the semester.

### 1.2. Literals

Literals are constants- they have a specific value that does not change.

For string literals, enclose the data value within single quotes: `'San Francisco', 'cat';` if you need to include a quote within your literal, use two single quotes; `'San Francisco''s favorite cat';`

(MySQL allows the use of the double quote for a string delimiter; this is not the case for other dbms and you might want to get in the habit of using the single quote delimiter for literals.)

For numeric literals, do not use delimiters; '97' is not a number- it is a string. Do not use commas or percent symbols etc. You can use a leading + or - symbol and a single decimal point. You can also use E notation.

These are valid numeric literals: `5 -58.89 + 0.002 2.3E5 2.3E-3`

For date literals you can use a delimited form such as '2009-07-17'; this is a string literal which MySQL will interpret as a date if it is used in a place where the value is expected to be a date. For example we have a function month which returns the month of a date value. So if you use `month('2009-07-17')`. MySQL will treat that value as a date. You can also use the ANSI standard date expression: `date '2009-05-14'`

## 2. Select without a From clause

Sometimes we want to just see how an expression works. We can use a select statement with an expression and no From clause. In that case we get a single line of output.

Demo 01: Display literals. Since I did not specify a column alias, MySQL uses the literal as the column header. This is one row of output.

```
select 'Hello world', 2013 ;
+-------------+------+
| Hello world | 2013 |
+-------------+------+
| Hello world | 2013 |
+-------------+------+
```

Demo 02: Display literals with aliases.

```
select 'Hello world' as Greeting, 2013  as Year;
+-------------+------+
| Greeting    | Year |
+-------------+------+
| Hello world | 2013 |
+-------------+------+
```

## 3. Manipulating numeric data

MySQL uses the standard arithmetic operators: +, -, * and /. Arithmetic expressions can be used in the Select clause, the Where clause and the Order By clause.

MySQL follows the standard rules for arithmetic precedence. Multiplication and division are carried out before addition and subtraction. Use parentheses to change the order of evaluation.

With the expression: 5 + 3 * 8 which has no parentheses, the multiplication is done first. 5 + 24 = 29.

With the expression: (5 + 3) * 8 which has parentheses, the operation in parentheses is done first , 8* 8 = 64.

Mysql also supports the % operator for modulo arithmetic and the Div operator which returns integers. Note that the vanLans book says that div divides and rounds off- actually with integers, it divides and truncates.

Demo 03: Display arithmetic expressions. If you do not use a column alias, the expression will be used for the header.

```
select   5 * 3 as Col1
,        5 + 8   as Col2
,        5 + 3 * 8 as Col4
,       (5 + 3 ) * 8 as Col5
,       (5 + 3 )  /  3
;
+------+------+------+------+----------------+
| Col1 | Col2 | Col4 | Col5 | (5 + 3 )  /  3 |
+------+------+------+------+----------------+
|   15 |   13 |   29 |   64 |         2.6667 |
+------+------+------+------+----------------+


select 21 % 5, 21 % 6, 21 % 8;
+--------+--------+--------+
| 21 % 5 | 21 % 6 | 21 % 8 |
+--------+--------+--------+
|      1 |      3 |      5 |
+--------+--------+--------+


select 21 / 5, 21 div 5, 24 / 5, 24 div 5;
+--------+----------+--------+----------+
| 21 / 5 | 21 div 5 | 24 / 5 | 24 div 5 |
+--------+----------+--------+----------+
| 4.2000 |        4 | 4.8000 |        4 |
+--------+----------+--------+----------+
```

If any of the operands in an arithmetic expression is Null, then the calculated value is Null.
Division by 0 returns a null.

For working with calculations, I have created a small test table. You can create test tables easily to try out ideas. I put these tables in the a_testbed database.

Demo 04: Create table, see the demo for the inserts

```
Create table z_tst_calc (
   item_id int primary key
 , quantity int not null
 , price  decimal (6,2)
 );
+---------+----------+--------+
| item_id | quantity | price  |
+---------+----------+--------+
|     101 |        1 | 125.12 |
|     102 |        5 |  30.00 |
|     103 |       10 | 101.05 |
|     104 |        1 |  75.50 |
|     105 |       12 |  33.95 |
+---------+----------+--------+
5 rows in set (0.00 sec)
```

Demo 05: Using a calculated column. Price * quantity gives a value commonly called the extended cost.

```
Select item_id
 , price
 , quantity
 , price * quantity as extendedcost
```

```
From a_testbed.z_tst_calc;
+---------+--------+----------+--------------+
| item_id | price  | quantity | extendedcost |
+---------+--------+----------+--------------+
|     101 | 125.12 |        1 |       125.12 |
|     102 |  30.00 |        5 |       150.00 |
|     103 | 101.05 |       10 |      1010.50 |
|     104 |  75.50 |        1 |        75.50 |
|     105 |  33.95 |       12 |       407.40 |
+---------+--------+----------+--------------+
```

Demo 06: Another calculated column.

```
Select item_id
 , price
 , quantity
 , price * quantity as extendedcost
 , price * quantity * 1.085 as ExtCostWithTax
From a_testbed.z_tst_calc;
+---------+--------+----------+--------------+----------------+
| item_id | price  | quantity | extendedcost | ExtCostWithTax |
+---------+--------+----------+--------------+----------------+
|     101 | 125.12 |        1 |       125.12 |      135.75520 |
|     102 |  30.00 |        5 |       150.00 |      162.75000 |
|     103 | 101.05 |       10 |      1010.50 |     1096.39250 |
|     104 |  75.50 |        1 |        75.50 |       81.91750 |
|     105 |  33.95 |       12 |       407.40 |      442.02900 |
+---------+--------+----------+--------------+----------------+
```

Demo 07: You cannot meaningfully use the column alias in the select to continue the calculations. Suppose you want to add a $5 handling fee per item line.

The following does not run; you get an error message that ExtendedCost is an unknown column

```
Select item_id
 , price
 , quantity
 , price * quantity as extendedcost
 , extendedcost + 5  as ExtCostWithShipping
From a_testbed.z_tst_calc;
```

Demo 08: This "runs" but is not what we want.

I can try quoting the alias. The following does run and produces output but the last column just shows the value 5 for each row.

```
Select item_id
 , price
 , quantity
 , price * quantity as extendedcost
 , "extendedcost" + 5  as ExtCostWithShipping
From a_testbed.z_tst_calc;
+---------+--------+----------+--------------+---------------------+
| item_id | price  | quantity | extendedcost | ExtCostWithShipping |
+---------+--------+----------+--------------+---------------------+
|     101 | 125.12 |        1 |       125.12 |                   5 |
|     102 |  30.00 |        5 |       150.00 |                   5 |
|     103 | 101.05 |       10 |      1010.50 |                   5 |
|     104 |  75.50 |        1 |        75.50 |                   5 |
|     105 |  33.95 |       12 |       407.40 |                   5 |
+---------+--------+----------+--------------+---------------------+
5 rows in set, 5 warnings (0.00 sec)

Warning (Code 1292): Truncated incorrect DOUBLE value: 'extendedcost'
Warning (Code 1292): Truncated incorrect DOUBLE value: 'extendedcost'
Warning (Code 1292): Truncated incorrect DOUBLE value: 'extendedcost'
Warning (Code 1292): Truncated incorrect DOUBLE value: 'extendedcost'
```

```
Warning (Code 1292): Truncated incorrect DOUBLE value: 'extendedcost'
```

You get a warning message for each line. If you did not use the \W option, then you can give the command: `show warnings.` You get the same message for each line in the result set. MySQL did a cast- and cast the string as a value of zero.

With MySQL you need to get into the habit of checking each result set for warnings and reading the warnings. MySQL does more conversions than some other databases and these may not always be appropriate.

Demo 09: We will get to functions soon; this uses the Round function. When you use a function such as Round, do not include a space between the name of the function and the opening parentheses.

```
Select item_id
 , price
 , quantity
 , price * quantity as extendedcost
 , round(price * quantity* 1.085,2) as ExtCostWithTax
From a_testbed.z_tst_calc;
+---------+--------+----------+--------------+----------------+
| item_id | price  | quantity | extendedcost | ExtCostWithTax |
+---------+--------+----------+--------------+----------------+
|     101 | 125.12 |        1 |       125.12 |         135.76 |
|     102 |  30.00 |        5 |       150.00 |         162.75 |
|     103 | 101.05 |       10 |      1010.50 |        1096.39 |
|     104 |  75.50 |        1 |        75.50 |          81.92 |
|     105 |  33.95 |       12 |       407.40 |         442.03 |
+---------+--------+----------+--------------+----------------+
```

Demo 10: Using a calculated value as a sort key.

```
Select item_id
 , price
 , quantity
 , price * quantity as extendedcost
 , round(price * quantity* 1.085,2) as ExtCostWithTax
From a_testbed.z_tst_calc
Order by round(price * quantity* 1.085,2) ;
+---------+--------+----------+--------------+----------------+
| item_id | price  | quantity | extendedcost | ExtCostWithTax |
+---------+--------+----------+--------------+----------------+
|     104 |  75.50 |        1 |        75.50 |          81.92 |
|     101 | 125.12 |        1 |       125.12 |         135.76 |
|     102 |  30.00 |        5 |       150.00 |         162.75 |
|     105 |  33.95 |       12 |       407.40 |         442.03 |
|     103 | 101.05 |       10 |      1010.50 |        1096.39 |
+---------+--------+----------+--------------+----------------+
```

Demo 11: Using a column alias as a sort key can cause problems. If you need to delimit the alias, enclose the alias in back ticks. Other quotes to do not give the desired result.

```
Select item_id
 , price
 , quantity
 , price * quantity as extendedcost
 , round(price * quantity* 1.085,2) as ExtCostWithTax
From a_testbed.z_tst_calc
Order by `ExtCostWithTax` ;
```

Same result as previous query.

Demo 12: We are having a $50.00 off sale. This might not be such a good idea! We have negative prices

```
Select item_id
 , price
 , quantity
```

```
 , price * quantity as extendedcost
 , (price- 50) * quantity as salecost
From a_testbed.z_tst_calc;
+---------+--------+----------+--------------+--------------+
| item_id | price  | quantity | extendedcost |     salecost |
+---------+--------+----------+--------------+--------------+
|     101 | 125.12 |        1 |       125.12 |        75.12 |
|     102 |  30.00 |        5 |       150.00 |      -100.00 |
|     103 | 101.05 |       10 |      1010.50 |       510.50 |
|     104 |  75.50 |        1 |        75.50 |        25.50 |
|     105 |  33.95 |       12 |       407.40 |      -192.60 |
+---------+--------+----------+--------------+--------------+
```

# 4. Concatenation of strings

Concatenating strings means to put the strings together one after the other. You use the concat function to concatenate strings.

Concatenation does not add spaces between the strings being put together. Include spacing by concatenating in a literal space.

Demo 13: Concatenating strings: MySQL propagates nulls in the concat function Some animals do not have a value for the name and the expression returns a null for the concat result.

```
select an_name
, an_type
, concat(an_name,  ' is a ', an_type) as Animal
From a_vets.vt_animals
limit 8;
+-----------+-----------+-------------------------+
| an_name   | an_type   | Animal                  |
+-----------+-----------+-------------------------+
| Gutsy     | cat       | Gutsy is a cat          |
| Kenny     | snake     | Kenny is a snake        |
| NULL      | bird      | NULL                    |
| NULL      | bird      | NULL                    |
| Mr Peanut | bird      | Mr Peanut is a bird     |
| Gutsy     | porcupine | Gutsy is a porcupine    |
| Big Mike  | chelonian | Big Mike is a chelonian |
| George    | chelonian | George is a chelonian   |
+-----------+-----------+-------------------------+
```

Demo 14: You can concatenate values of different data types and MySQL will do type casts.

```
select
  concat('ID: ', an_id, ' ' , an_name,  ' was born on  ', an_dob) as Animal
From a_vets.vt_animals
order by an_name desc
limit 8;
+-------------------------------------------+
| Animal                                    |
+-------------------------------------------+
| ID: 21001 Yoggie was born on  2009-05-22  |
| ID: 21318 Waldrom was born on  2012-06-11 |
| ID: 16003 Ursula was born on  2006-06-06  |
| ID: 15401 Pinkie was born on  1998-03-15  |
| ID: 19845 Pinkie was born on  2009-02-02  |
| ID: 16004 Napper was born on  2006-06-06  |
| ID: 12035 Mr Peanut was born on  1995-02-28 |
| ID: 21317 Manfried was born on  2011-06-11 |
+-------------------------------------------+
```

# 5. Testing calculations with nulls

A null is a placeholder for a non-existent value. Whether or not you should allow nulls in database tables will always be an issue. Some people believe that you should never allow nulls in your tables. Certainly it is easier to work with tables that cannot accept nulls.

However you might have troubles if you always insist on non-null attributes. Do you really want to turn away sales because a customer says they have no first name? Are you positive that you can legally refuse to hire someone because they have no phone number just because your employee table has a not null phone number attribute?

What you need to know for now is how nulls are treated in calculations. We will set up another small test table to look at these calculations. We will use an id column, a nullable column that stores strings, a nullable column that stores integers, and a nullable column that stores floats.

Demo 15:

```
create table a_testbed.z_tst_nulls (
  col_id      int          not null primary key
, col_string  varchar(10)  null
, col_int     int          null
, col_float   float        null
);
```

We need very few rows to test this.

```
insert into a_testbed.z_tst_nulls  values (1, 'abc', 10,   10.567);
insert into a_testbed.z_tst_nulls  values (2, 'abc', null, 20.222);
insert into a_testbed.z_tst_nulls  values (3, null,  30,   null);
insert into a_testbed.z_tst_nulls  values (4, null,  null, null);

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 |
+--------+------------+---------+-----------+
```

Demo 16: Now a few expressions

Use concat to concatenate strings; nulls propagate in string concatenation.
nulls propagate in arithmetic
float arithmetic may produce more digits after the decimal than you expected.

```
select col_id
, concat('XYZ', col_string)  as stringTest
, 25 + col_int    as intTest
, 25 + col_float  as floatTest
from a_testbed.z_tst_nulls;
+--------+------------+---------+------------------+
| col_id | stringTest | intTest | floatTest        |
+--------+------------+---------+------------------+
|      1 | XYZabc     |      35 | 35.5670003890991 |
|      2 | XYZabc     |    NULL | 45.2220001220703 |
|      3 | NULL       |      55 |             NULL |
|      4 | NULL       |    NULL |             NULL |
+--------+------------+---------+------------------+
```

Demo 17: Now add one more row to the table with a string that could be cast as an integer. If the string can be cast to a number, then MySQL will do this. This is not good style but it is commonly done.

With the first two rows the string 'abc' was cast to a zero (and generated a warning) which was added to 3.
With the next two rows , the null string was treated as a null number which propagated in arithmetic
With the last row, the string '12' was cast to the number 12, added to 3 which resulted in 15.

```
insert into a_testbed.z_tst_nulls  values (5, '12',  12, 12.2);
select
  col_string
, col_string + 3 as stringTest
from a_testbed.z_tst_nulls;
+------------+------------+
| col_string | stringTest |
+------------+------------+
| abc        |          3 |
| abc        |          3 |
| NULL       |       NULL |
| NULL       |       NULL |
| 12         |         15 |
+------------+------------+
5 rows in set, 2 warnings (0.00 sec)

+---------+------+----------------------------------------+
| Level   | Code | Message                                |
+---------+------+----------------------------------------+
| Warning | 1292 | Truncated incorrect DOUBLE value: 'abc' |
| Warning | 1292 | Truncated incorrect DOUBLE value: 'abc' |
+---------+------+----------------------------------------+
2 rows in set (0.00 sec)
```

## 5.1. Coalesce

Comparing these two selects, we can see that nulls propagate in arithmetic and also in string concatenation using the concat function.

Since nulls create problems for us, a dbms has a function to substitute another value for the null. This function is coalesce. For now we will use coalesce with two arguments- the first will be a column name and the second argument is a value to use if the column value is null.

Demo 18: Using Coalesce

```
select col_id
, coalesce(col_string, 'DataMissing')  as stringTest
, coalesce(col_int,    'NoData')       as intTest1
, coalesce(col_int,    -20)            as intTest2
, coalesce(col_float,  'NoData')       as floatTest1
, coalesce(col_float,   29.95)         as floatTest2
from a_testbed.z_tst_nulls;
+--------+-------------+----------+----------+------------+--------------------+
| col_id | stringTest  | intTest1 | intTest2 | floatTest1 | floatTest2         |
+--------+-------------+----------+----------+------------+--------------------+
|      1 | abc         | 10       |       10 | 10.567     | 10.567000389099121 |
|      2 | abc         | NoData   |      -20 | 20.222     | 20.222000122070312 |
|      3 | DataMissing | 30       |       30 | NoData     |              29.95 |
|      4 | DataMissing | NoData   |      -20 | NoData     |              29.95 |
|      5 | 12          | 12       |       12 | 12.2       | 12.199999809265137 |
+--------+-------------+----------+----------+------------+--------------------+
```

Note that MySQL has a different attitude than many other dbms in that it will allow you to return a string message from coalesce even when the other argument is numeric. For other dbms, the expressions for column intTest1 and floatTest1 would be invalid.

Part of the reason for doing this is to encourage you to create small tests to find out how your dbms works. It is generally quicker to set up a small test table like this than to page through a book or try an internet search.

One of the problems with internet searches is that many web pages do not tell you which version of the software they are using, which system settings are in place- and that does not even count the web pages that are just wrong. Some web pages don't even seem to say which dbms they are using. You certainly should be able to check some things out on web pages- but you still need to verify anything you find.