Before you actually begin using SQL to interact with MySQL, you should have an understanding of how values must be used in your queries.  Always abide by these rules:

* numeric values should not be quoted.
* numeric values must not contain commas.
* string values (char, varchar, and text column types) must be always quoted.
* date and time values must always be quoted.
* the word NULL must not be quoted.

SQL keywords are case-insensitive.

That means you can type all your SQL constructs lowercase if you wish.

The SELECT statement is probably the most used SQL construct of all.

  The term SELECT is used to return ---rows of records that meet certain criteria.

You may ask MySQL to return the data stored in whichever

---columns you want for a particular table.

You can return the all the columns or select just a few.

To return all the columns, you'll use the select \* command, such as:

            SELECT \* from users;

If you want certain columns returned,

you will need to explicitly state which columns you'll need, such as:

            SELECT user\_id, first\_name, last\_name  
            FROM users;

There are a few benefits to being explicit about which columns are selected.

* performance:  there's no reason to fetch columns that you will not be using.
* order:  you can return columns in an order other than their layout in the table.
* accessibility:  it allows you manipulate the values in those columns using functions.

The problem with the SELECT statement

---is that it will automatically retrieve every record.

To improve the efficiency of your SELECT statement,

there are different conditionals you can use to refine your output.

These conditionals utilize the SQL term

WHERE to set conditions for the values returned.

SELECT expense\_amount   
FROM expenses  
WHERE expense\_amount > 10.00;

SELECT client\_id   
FROM clients  
WHERE client\_name = 'Acme Industries';

This is a basic introduction to the concepts in which you'll be introduced in the chapters for this module.

Please take your time going through each of the examples to gain the confidence of the SQL structures presented.

There are additional chapter notes and videos for your perusal.

|  |
| --- |
| **Column and Data Types** |

There are three basic column types in MySQL:

numeric types,

string or text types,

and date and time types.

**Numeric Types**

Numeric types are used for ---storing numbers.

The types **int** (integer)

and **float** (floating-point number) represent

two subtypes of numeric types:

the exact numeric types

and the approximate numeric types.

Numeric types may be constrained by a display width M and,

floating-point types,

a number of decimal places, D.

These numbers go after the declaration:

           salary decimal(10, 2)

This has a display width of 10 with two digits after the decimal point.

You may choose to use neither parameter, the display width only, or both the display width and the number of decimal places. 

Numeric types may also be followed by the keywords UNSIGNED and/or ZEROFILL.

---The UNSIGNED keyword specifies that the column contains only zero or positive numbers.

---The ZEROFILL keyword means that the number will be displayed with leading zeroes.

**Numeric or Decimal**

These types are the same, and DECIMAL may also be abbreviated to DEC.

These types are used to store exact floating-point values

---and are typically used to store monetary or currency type values.

They have the same range as double-precision floating-point numbers.

**Integer and Variations**

This type is stored as INT.

This is a standard integer,

stored in 4 bytes,

giving a range of 232possible values.

There are also various variations on INT:

A TINYINT is 1 byte (28 possible values).  The keywords BIT and BOOL are synonyms for TINYINT.

A SMALLINT is 2 bytes (216 possible values).

A MEDIUMINT is 3 bytes (224 possible values).

A BIGINT is 8 bytes (264 possible values).

**Float**

This is a single-precision floating-point number.

It can represent a positive number between 1.18x10-38 to 3.40x1038

and a similar range for negative numbers.

**Double**

This is a double-precision floating-point number.

Synonyms for DOUBLE are REAL and DOUBLE PRECISION.

They can represent a positive number between 2.23x10-308 to 1.80x10308

and a similar range of negative numbers.

**String and Text Types**

MySQL supports various string and text types.

The basic types are CHAR, VARCHAR, TEXT, BLOB, ENUM, and SET.

**Char**

CHAR is used to store fixed-length strings.

CHAR is usually followed by a string length:

          CHAR(20)

If you do not specify a length,

you will get CHAR(1).

The maximum length of a CHAR is 255 characters.

When CHARs are stored, they will always be the exact length you specify.

This is achieved by padding the contents of the column with spaces.

These spaces are automatically stripped

when the contents of a CHAR column are retrieved.

Obviously, storing a CHAR takes up more space on disk

than storing an equivalent variable-length string.

The trade-off is that it is faster

to retrieve rows from a table

in which all the ---columns are of fixed widths

(that is, CHAR, numeric, or date).

Often, speed is more important than disk space,

so you may choose to make text fields that are not going to vary a great deal anyway into CHAR as a small optimization.

Both CHAR and VARCHAR types can be preceded with the keyword NATIONAL,

meaning to restrict the contents to the standard character set.

This is the default in MySQL, so you only need to use it for cross-platform compatibility.

CHAR and VARCHAR can both be followed by the keyword BINARY,

--meaning that they should be treated as case sensitive

when evaluating string comparisons.

The default is for strings to be compared in a case insensitive fashion.

**VARCHAR**

VARCHAR obviously stores variable-length strings.

You specify the width in parentheses after the type:

          VARCHAR(10)

The range is 0 – 255.

**TEXT, BLOB, and Variations**

 The TEXT types are

used for storing longer pieces of text

than you fit into a CHAR or VARCHAR.

BLOB stands for Binary Large OBject.

These types are the same

except that BLOBs

are intended to store binary data rather than text.

Comparisons on BLOBs are ---case sensitive,

and on TEXTs, they are ---not case sensitive.

They are both variable in length,

but both come in various sizes:

TINYTEXT or TINYBLOB can hold up to 255 (28 – 1) characters or bytes.

TEXT or BLOB can hold up to 65,535 (216 – 1) characters or bytes (64KB).

MEDIUMTEXT or MEDIUMBLOB can hold up to 16,777,215 (224 – 1) characters or bytes (16MB).

LONGTEXT or LONGBLOB can hold up to 4,294,967,295 (232 – 1) characters or bytes (4GB).

**ENUM**

This type allows you ---to list a set of possible values.

Each row can contain ---one value--- from the enumerated set.

You declare an ENUM as follows:

          gender enum(‘m’, ‘f’)

Enumerated types can also be NULL,

so the possible values of gender are m, f, NULL, or error.

**SET**

 The SET type is similar to ENUM

except that rows may contain ---a set of values--- from the enumerated set.

**DATE and TIME Types**

DATE

 The date type stores a date.

MySQL expects the date in ISO year-month-day order,

avoiding trans-Atlantic arguments.

Dates are displayed as YYYY-MM-DD.

TIME

 This type stores a time,

displayed as HH:MM:SS.

DATETIME

 This is a combination of the previous types.

The format is YYYY-MM-DD HH:MM:SS.

TIMESTAMP

 This is a useful type.

If you do not set this column in a particular row,

---or set it to NULL,

---it will store the time

---that row was inserted

---or last changed.

When you receive a timestamp,

---it will be displayed in the DATETIME format.

YEAR

 This type stores a year.

When you declare a column of this type,

you can declare it as  YEAR(2)

or YEAR(4) to specify the number of digits.

YEAR(4) is the default.

YEAR(2) represents the range 1970 to 2069.

|  |
| --- |
| **Concatenation** |

**Using Concat()** 

If you wanted to combine two or more columns together

so that they display as one column,

you need to use the concat() function. 

**Example 1:** 

*Select all reps displaying the first and last name in one column.*

Select rep\_num, concat (first\_name, “ “, last\_name) as name

By using the concat function,

you can combine several columns

and display them in a sequence beneficial to your query.

In this case, the concatenation put the rep’s first name,

followed by space, followed by the rep’s last name.

The column name was given the name ‘name’ to display as the column.

Another example:

Select rep\_num, concat (first\_name, “, “, last\_name) as name

The only difference here is that the rep’s last name with a comma and a space then the first name displays.

|  |
| --- |
| **Regular Expressions** |

**Using RLIKE**

The RLIKE function can be used to match on the basis of regular expressions. 

A regular expression ---is a pattern that describes the general shape of a string.

There is a special notation

for describing the features

we would like to see in matching strings.   

First, a literal string matches the string.

So, the pattern ‘cat’ matches ‘cat’.

However, it also matches ‘catacomb’

and ‘the cat sat on the mat’.

The pattern cat matches ‘cat’ anywhere inside the target string. 

If you want to match only the word ‘cat’,

then the pattern would need to be ‘^cat$’.

The caret (^) means “anchor to the start of the string;”

in other words, the first thing at the start of a matching string is the word ‘cat’.

The dollar sign ($) means “anchor to the end of the string”;

in other words, the last thing in the string must be the word ‘cat’.

So the pattern ‘^cat$’ can match only the string ‘cat’ and nothing else. 

Regular expressions also support wildcards,

just as LIKE does.

However, the wildcard is different.

There is only one, the dot (.) that will match any single character.

So, ‘.at’ matches ‘cat’, ‘bat’, ‘mat’ and so on. 

You only need a single wildcard character

because you can also specify how often characters (including wildcards)

can appear in a string.

For example,

the special \* character after a character

means that character may appear zero or more times.

So, the pattern ‘n\*’ matches ‘’, ‘nn’, ‘nnn’, and so on.

You can group characters with parentheses,

so ‘(cat)\*’ matches ‘’, ‘cat’, ‘catcat’, ‘catcatcat’, and so on.

You can also use the wildcard,

so ‘.\*’ matches any number of any character

– basically anything. 

Also, the plus sign (+)

means that the character or string

---before it should be repeated one or more times,

and the question mark (?)

means to match zero times or one time.

You can also list a specific range,

so for example, ‘(cat)(2,4)’ matches ‘catcat’, ‘catcatcat’, and ‘catcatcatcat’. 

As well as listing specific characters and strings,

you can list sets of characters.

These appear in square brackets.

For example, the pattern  ‘[a-z]’

matches any single letter

and ‘[a-z]\*’ matches any number of letters. 

Finally, there are a number of character classes,

which are predefined sets.

For example, [[:alnum:]] matches any alphanumeric character. 

**Example 1:**

*Find all customers with the characters ‘ea’ in the customer name.*

Select \* from customer

Where customer\_name rlike ‘ea’

|  |
| --- |
| **Using Compound Conditions** |

Thus far, the conditions that you’ve been testing for are considered

---simple conditions.

It’s now time to move into compound conditions.

You form compound conditions

by connecting two or more simple conditions together

using the AND, OR and NOT operators.

When the AND operator connects simple conditions,

all the simple conditions must be true

---in order for the compound condition to be true.

When the OR operator connects simple conditions,

the compound condition will be true

---whenever any one of the simple conditions is true. 

Preceding a condition by the NOT operator

---reverses the truth of the original condition.

For example, if the original condition is true,

the new condition will be false;

if the original condition is false,

the new one will be true. 

**Example 1:** 

Using the premier\_products database: 

*List the descriptions of all parts that are located in warehouse 3 and for which there are more than 25 units on hand.* 

In this example, you need to retrieve those parts that meet

*both* conditions –

the warehouse number is equal to 3

AND the number of units on hand is greater than 25.

To find the answer, you form a compound condition

using the AND operator as shown below:

select description

from part

where warehouse = ‘3’

and on\_hand >25;

 As you may have noticed the construct above,

each condition appears on a separate line.

Some may prefer to put the conditions

on the same line with parentheses

around each simple condition as shown below:

select description

from part

where warehouse = ‘3’ and on\_hand >25;

These two methods accomplish the same thing.  It’s simply a matter of style. 

**Example 2:** 

*List the descriptions of all parts that are located in warehouse 3 or for which there are more than 25 units on hand.* 

In this example, you need to retrieve those parts

for which the warehouse number is equal to 3,

*or*the number of units on hand is greater than 25,

*or* both.

To do this, you will form a compound expression using the OR operator:

select description

from part

where warehouse = ‘3’

or on\_hand >25;

**Example 3:** 

*List the descriptions of all parts that are not in warehouse 3.* 

In this example, you could use a simple condition and the “not equal to” operator: 

(where warehouse <> ‘3’) 

As an alternative, you could use the EQUAL operator (=) in the condition and precede the entire condition with the NOT operator:

select description

from part

where not(warehouse = ‘3’);

 You don’t need to enclose the condition WAREHOUSE = ‘3’ in parentheses,

but doing so makes the command more readable.

Notice that by phrasing the condition in this form,

you avoid the problem of determining

whether your SQL implementation uses the <> or != not equal to operator.

  FYI – our version of MySQL accepts both: <> and the != operators.  

select description

from part

where (warehouse != ‘3’);

**Example 4:** 

*List the number, name, and balance of all customers with balances greater than or equal to $2,000 and less than or equal to $5,000.* 

You can use the where clause and the AND operator to retrieve the data:

Select customer\_num, customer\_name, balance

From customer

Where balance >= 2000

And balance <=5000;

**Using the BETWEEN operator**

An alternative to the above example is to use the BETWEEN operator:

Select customer\_num, customer\_name, balance

From customer

Where balance between 2000 And 5000;

The BETWEEN operator is not an essential feature of SQL; you have just seen that you can obtain the same result without using it.  Using the BETWEEN operator, however, does make certain SELECT statements simpler to construct. 

**Note:**  The BETWEEN operator is inclusive, meaning that a value equal to either end would be selected.  In the clause, BETWEEN 2000 and 5000, for example, values of either 2,000 or 5,000 would make the condition true.

**Using Computed Columns** 

You can use computed columns in SQL queries.  A computed column is a column that does not exist in the database but can be computed using data in the existing columns.  Computations can involve any arithmetic operators + for addition, - for subtraction, \* for multiplication and / for division. 

**Example 5:** 

*Find the number, name and available credit (the credit limit minus the balance) for each customer.* 

There is no column for available credit in the premier\_product database, but you can compute the available credit from two columns that are present:  CREDIT\_LIMIT and BALANCE.  To compute the available credit, you can use the expression CREDIT\_LIMIT – BALANCE as shown below:

The parentheses around the calculation (CREDIT\_LIMIT – BALANCE) are not essential, but improves readability. 

**NOTE:**Some SQL implementations use special headings for computed columns, such as SUM1 or COUNT2.  You can assign a name to a computed column by following the computation with the word AS and the desired name.   

The following query assigns the name AVAILABLE\_CREDIT to the computation:

**NOTE:**  You can use names containing spaces following the word AS.  You do so by enclosing the name in quotation marks.  For example,  AS “AVAILABLE CREDIT”. 

**Example 6:** 

*Find the number, name, and available credit for each customer with at least $5,000 of available credit.*