MySQL Notes

Basics

1. Viewing all the databases

**SHOW DATABASES;**

1. Creating a new database

**CREATE DATABASE <name>;**

Example:

**CREATE DATABASE hello\_world\_db;**

1. Deleting a database

**DROP DATABASE <name>;**

Example:

**DROP DATABASE hello\_world\_db;**

1. Selecting a database for use

**USE <name>;**

Example:

**USE dog\_walking\_app;**

1. To view which database we are currently using

**SELECT database();**

NOTE: Suppose we select a database, say test. Using the ‘SELECT database;’ command will give us test as the output, i.e. the name of the database we are currently using. If we delete this database, then ‘SELECT database();’ will show NULL. This is because we are currently not using any database.

1. A **database** is an organized collection of [data](https://en.wikipedia.org/wiki/Data_(computing)), generally stored and accessed electronically from a computer system.
2. Creating a new table

**CREATE TABLE <table\_name>(col\_name data\_type, col\_name2 data type2, … );**

1. Show all tables in a database

**SHOW TABLES;**

1. Viewing columns from a table

**SHOW COLUMNS FROM <table\_name>;**

Example:

**SHOW COLUMNS FROM cat;**

1. Describing tables

**DESC <table\_name>;**

1. Deleting tables

**DROP TABLE <table\_name>;**

1. Inserting into a table

**INSERT INTO <table\_name>(col\_1, col\_2, …) VALUES (val\_for\_col\_1, val\_for\_col\_2, …);**

Example:

**INSERT INTO cats(name, age) VALUES(“Jetson”, 7);**

Example 2:

**INSERT INTO cats(age, name) VALUES(5, “Sammy”);**

1. Another way to insert. Similar to the previous command but here we have to specify data according to order in the schema.

**INSERT INTO cats VALUES(“Jetson”, 8);**

Here we cannot use (8, “Jetson”) because the schema specifies the name first.

1. Inserting multiple values into the schema

Example:

**INSERT INTO cats(name, age)**

**VALUES (“Theon”, 1),**

**(“Charlie”, 2),**

**(“Harry”, 5);**

1. If the message displayed after executing a query says “there are warnings”, we can view those warnings using

**SHOW WARNINGS:**

1. If an attribute(column) is specified as allowing null values i.e. NULL = YES, then we could use queries like

*cats(name, age)*

**INSERT INTO cats(name) VALUES(“Josey”);**

**INSERT INTO cats() VALUES();**

1. To prevent columns from allowing null values use the “NOT NULL” when creating tables

**CREATE TABLE cats2**

**(**

**name VARCHAR(50) NOT NULL,**

**age INT NOT NULL**

**);**

1. Now if we do not specify a columns value or use one of the queries as in point 16 above we will get the following error

Field 'age' doesn't have a default value

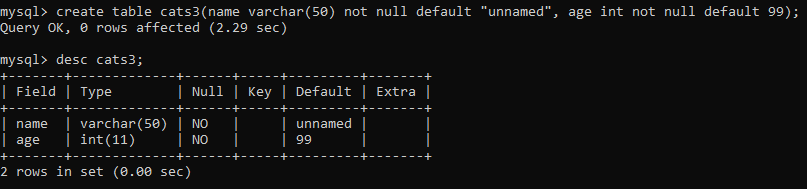
1. To solve the problem above we can use “DEFAULT” keyword to specify default values to the columns

Syntax:

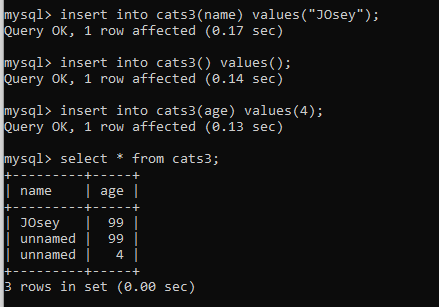
**CREATE TABLE <name>(col1 d\_type1 DEFAULT <value>, col2 d\_type2 DEFAULT <value>);**

Example:

**CREATE TABLE CATS(name VARCHAR(50) NOT NULL DEFAULT “My Cat”, age INT NOT NULL DEFAULT 4);**



1. Following the point above, now we can execute queries like



without any errors or warnings.

1. We use “NOT NULL” because we can manually specify something, i.e. a column value to be NULL.

**INSERT INTO cats(name, age) VALUES(“Montana”, NULL);**

We use “NOT NULL” to prevent such instances, where we explicitly need some value. We add default values so that in case no value is provided, default value can be used instead of a NULL.

1. Until now the tables that we used could allow us to insert rows where all the values are same. This would lead to redundancy. So we use a concept called PRIMARY KEY. A primary key is a column in a relation that uniquely identifies each row in the table. No two rows can have the same value for the primary key.

Specifying the primary key.

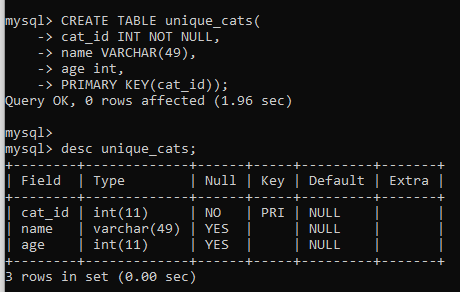
**CREATE TABLE unique\_cats(**

**cat\_id INT NOT NULL,**

**name VARCHAR(49),**

**age int,**

**PRIMARY KEY(cat\_id));**



1. We can use AURO\_INCREMENT to automatically increase the cat\_id value.

**CREATE TABLE unique\_cats2**

**(**

**cat\_id INT NOT NULL AUTO\_INCREMENT,**

**name VARCHAR(45),**

**age INT,**

**PRIMARY KEY(cat\_id)**

**);**

After this we can insert new rows without the cat\_id columns, because it will be auto incremented for each new data that we insert

Example:

**INSERT INTO unique\_cats2(name, age) VALUES(“Mow”, 4);**

1. **SELECT** is used to read data from database.

**SELECT \* FROM cats;**

‘ \* ’ means five me all columns.

To specify specific columns we can use

**SELECT name, breed FROM cats;**

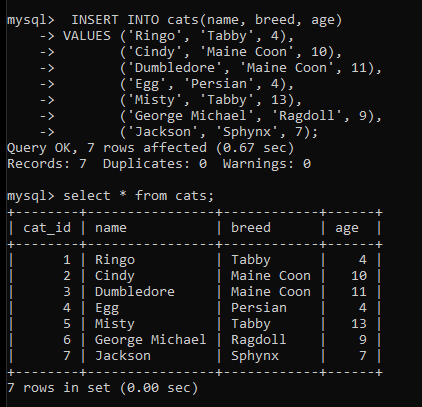
1. WHERE is used to specify particular data that we want from the table. WHERE is also used when we want to update or delete specific data from the database.

Examples:

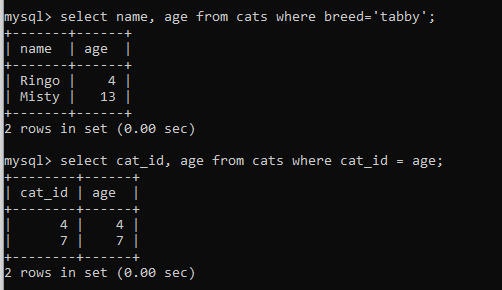
**SELECT \* FROM cats WHERE age=4;**

**SELECT \* FROM cats WHERE name=””;**

I will use the following table for the upcoming queries



1. NOTE: SQL is case insensitive, so Egg, egg, EgG, all mean the same thing.
2. We can specify stings in either ‘ ‘ or “ “
3. Some queries for the above table



1. Using **Aliases**. Aliases are the temporary names given to table or column for the purpose of a particular SQL query. It is used when name of column or table is used other than their original names, but the modified name is only temporary.
   1. Aliases are created to make table or column names more readable.
   2. The renaming is just a temporary change and table name does not change in the original database.
   3. Aliases are useful when table or column names are big or not very readable.
   4. These are preferred when there are more than one table involved in a query.

Syntax:

**SELECT column AS alias\_name FROM table\_name;**

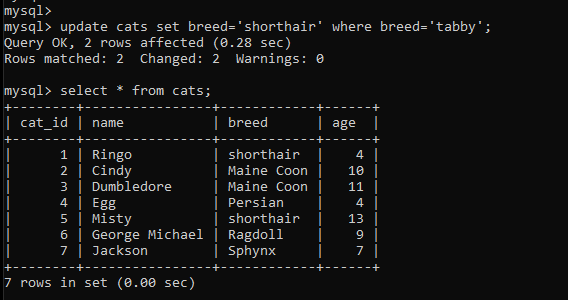
Example:

**SELECT cat\_id AS id from cats;**

1. Updating data.

Example:

**UPDATE cats SET breed=”Shorthair” WHERE breed=”Tabby”;**



Example: Set Misty’s age equal to 14

**UPDATE cats SET age=14 WHERE name=’Misty’;**

1. Deleting data.

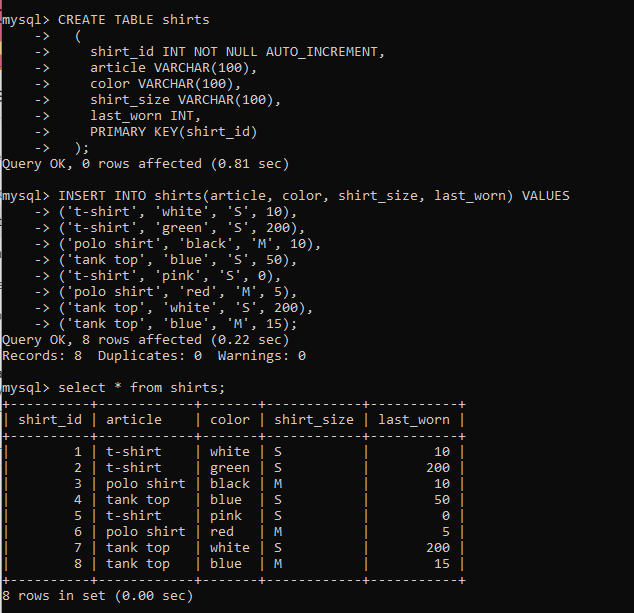
**DELETE FROM cats WHERE name=”egg”;**

Note: It is always a good idea to select the data and see it first before deleting.

1. To delete all the data from the table without deleting the table use

**DELETE FROM cats;**

1. Example: We will use the following table



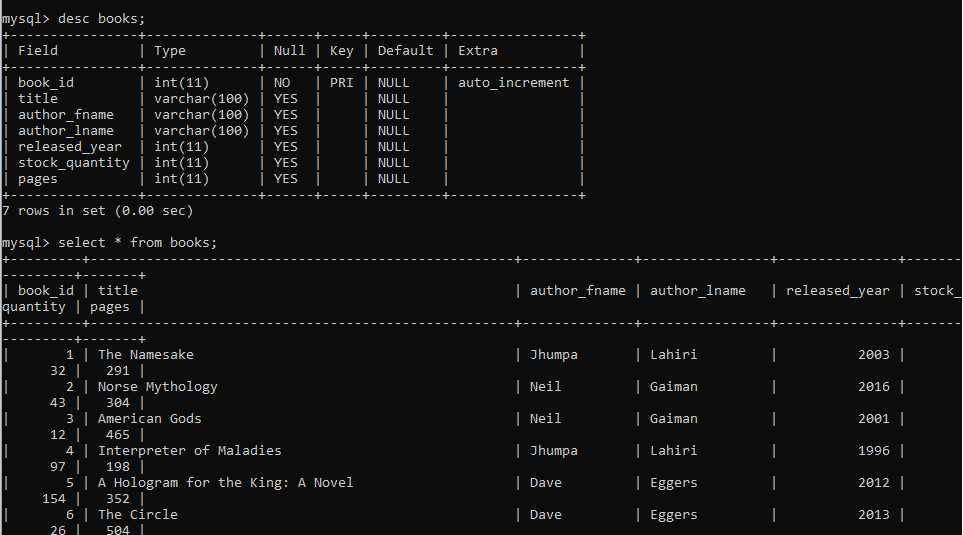
* Update all white shirts. Change size to ‘XS’ and color to ‘off white’.
* **UPDATE shirts SET color=’off white’, shirt\_size=’XS’ WHERE color=’white’;**

String Functions

1. Before going to string functions lets see how to run a .sql file with some sql query written in it. Suppose we have a file myFile.sql with a query to create a table. To run this from the console we have to go the directory which stores these files. Run MySQL from that directory and to run the file type

source myFile.sql

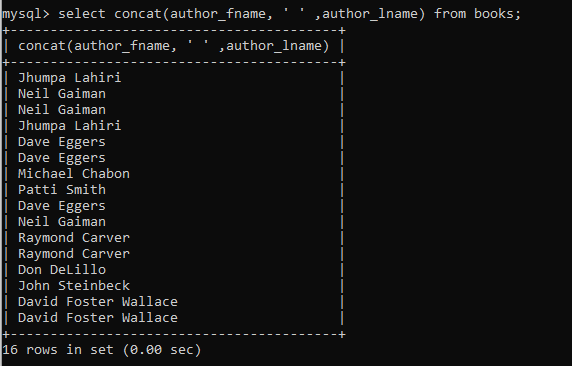
1. We will be using the following table for this section.



1. CONCAT() function: Returns the string that results from concatenating the arguments. May have one or more arguments. If all arguments are nonbinary strings, the result is a nonbinary string. If the arguments include any binary strings, the result is a binary string. A numeric argument is converted to its equivalent nonbinary string form.

It returns null if any argument is null.

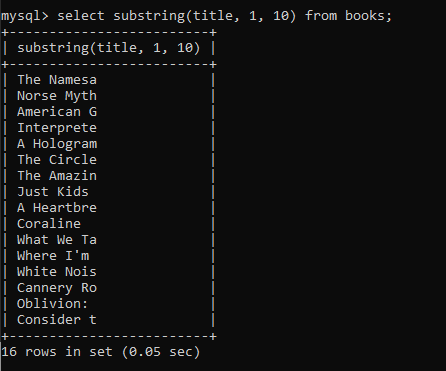
If we want to display the full name of the author we could use:



CONCAT\_WS() stands for Concatenate With Separator and is a special form of CONCAT(). The first argument is the separator for the rest of the arguments. The separator is added between the strings to be concatenated. The separator can be a string, as can the rest of the arguments. If the separator is NULL, the result is NULL.

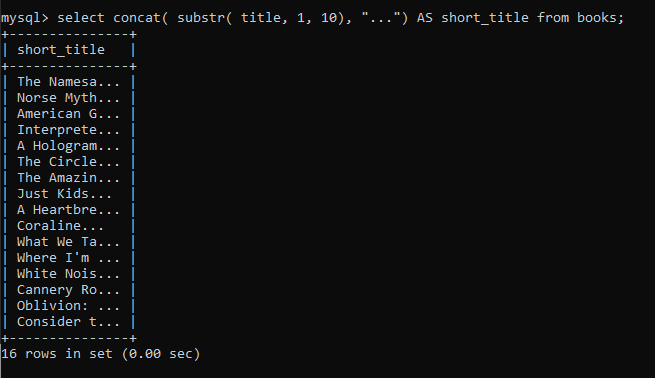
1. SQL does not use zero based indexing like most programming languages. It uses indexing starting at 1.
2. SUBSTRING(). Used to display only a part of a string. It has 3 main uses
   1. SUBSTRING(string, index1, index2), here the output will be string from index1 to index2
   2. SUBSTRING(string, index), here the output will start at index to the end of string
   3. SUBSTRING(string, negative\_index), similar to python negative indexing, here it will print from negative\_index till the end of string.

Example:



1. NOTE: we can use SUBSTR() instead of SUBSTRING()
2. NOTE: we can combine various string functions

Example:



1. REPLACE()

Syntax: REPLACE(str, from\_str, to\_str)

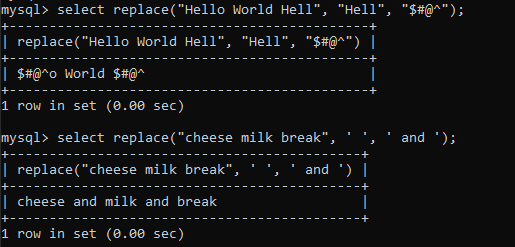
Returns the string str with all occurrences of the string from\_str replaced by the string to\_str. REPLACE() performs a case-sensitive match when searching for from\_str.

Example1:

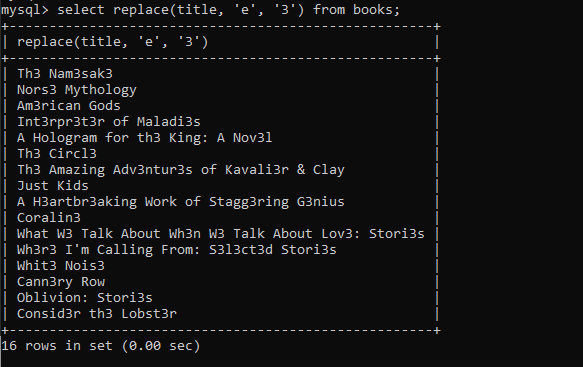
SELECT REPLACE('www.mysql.com', 'w', 'Ww');

-> 'WwWwWw.mysql.com'

Example:

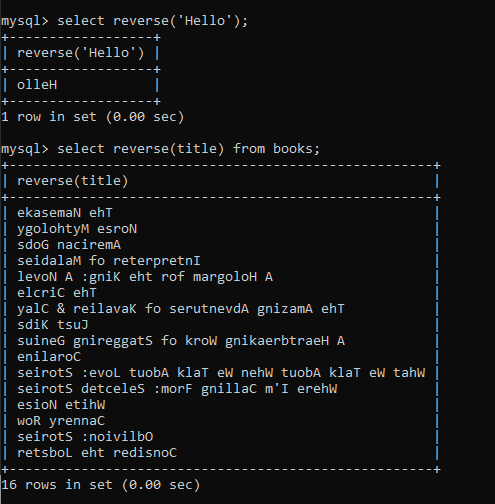


Example 2:



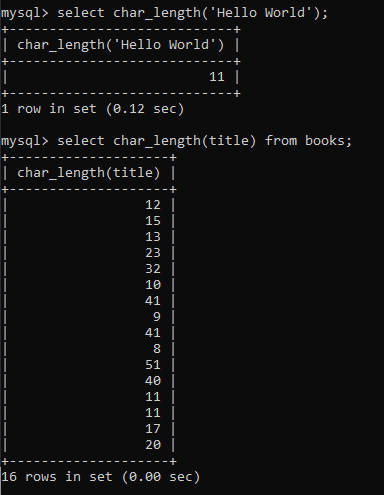
1. REVERSE(): it does what it says it does. No strings attached!

Example:



1. CHAR\_LENGTH(): It returns the number of characters in a string.

Example:



1. Changing a string’s case, UPPER() and LOWER().



Refining Selections

1. Selecting distinct elements form a table. Used in conjunction with select.

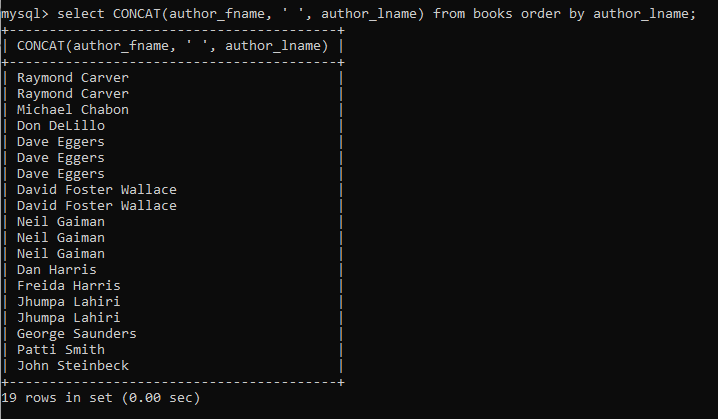
**SELECT DISTINCT author\_lname FROM books;**

If we want to select distinct author from books tables, and suppose some rows have same last name but different first name then we could use the following

**SELECT DISTINCT author\_fname, author\_lname FROM books;**

1. Using ORDER BY to sort our results.

Example: the following query displays first name and last name order by last names, i.e. according to ascending order of last names.



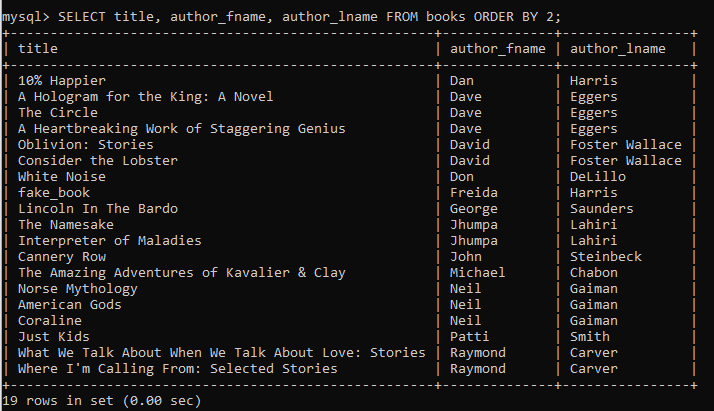
To sort in descending order

**SELECT author\_lname FROM books ORDER BY author\_lname DESC;**

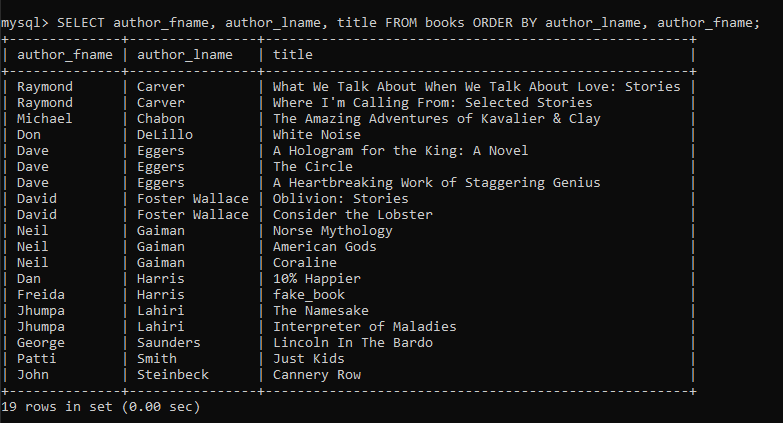
Example 2: Sort the book names sorted by release year



A shortcut. In the following example we will sort by author\_fname. Here the 2 denotes the second argument of SELECT i.e. author\_fname.



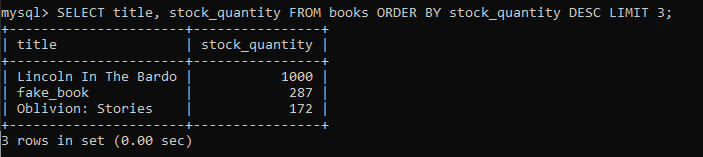
We can also sort using two attributes as comparator. This works by first sorting elements using the first attribute, if two elements have the same value for first attribute it will sort using the second attribute.



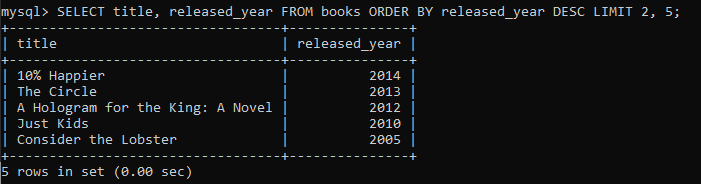
In the above example we first sort by author\_lname and then by author\_fname.

1. LIMIT allows us to specify a number telling us how many results we want to select.

For example, if, we want the top 3 books with highest stock we could use

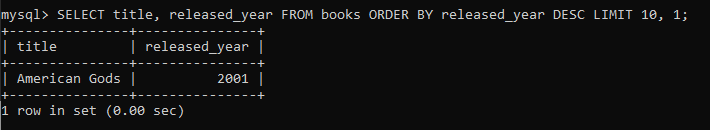


Using LIMIT with 2 arguments. The two comma separated arguments allow us to specify the start and the end row. In the following example we will get 2nd ,3rd, 4th and 5th most recent book by release year.

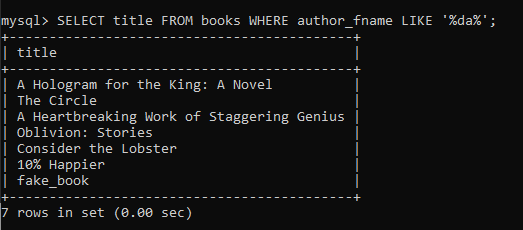


We will not get the most recent book released in 2017.

The following code will give us the 11th book



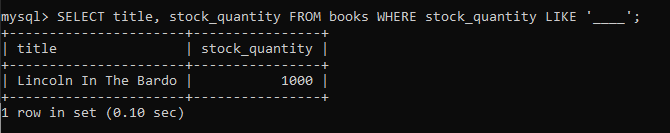
1. Using LIKE for better searching the data. We could use LIKE for situations like, if someone comes to out bookstore and wants to buy a book but he has forgotten the name of the book. All he knows is that the authors name starts with ‘DA’. We use like in such situations. The following example shows how we can fetch all books with author names having ‘da’ in their first names.



We could also use

**SELECT title, author\_fname FROM books WHERE author\_fname LIKE ‘da%’;**

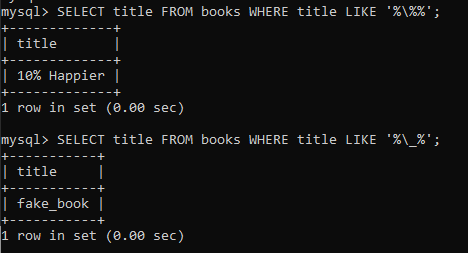
1. Using ‘\_’ and LIKE. Each underscore denotes 1 character. So if we want to select stock\_quantity that is 4 characters long we could use.



1. But what if we want to search for books that have % or \_ sign in their names? We use escape character for it.

Example:

**SELECT title FROM books WHERE title LIKE ‘%\%%’;**



Aggregate Functions

1. COUNT(). It does what it says. Say we want to know how many books are there in the table.

**SELECT COUNT(\*) FROM books;**

**SELECT COUNT(author\_lname) FROM books;**

**SELECT COUNT(DISTINCT author\_lname) FROM books;**

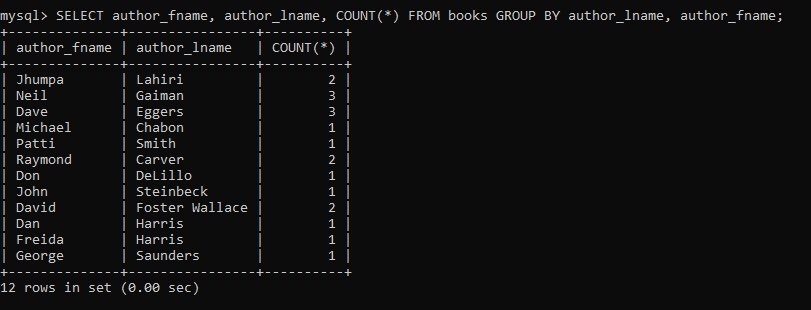
**SELECT COUNT(DISTINCT author\_fname, author\_lname) FROM books;**

Exercise: How many titles contain ‘the’

**SELECT COUNT(\*) FROM books WHERE title LIKE '%the%';**

1. GROUP BY: it summarizes or aggregates identical data into single rows

Example: Find the number of books by each author



Example: group by released\_year



1. MIN() and MAX()

**SELECT MIN(released\_year) FROM books;**

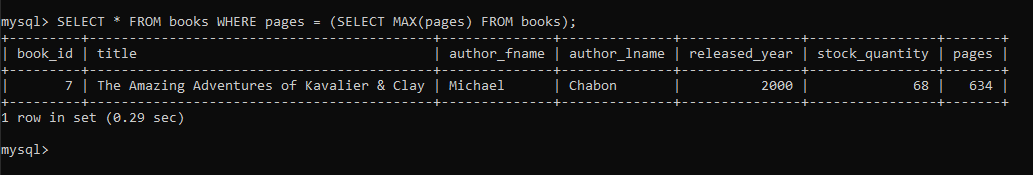
**SELECT MAX(pages) FROM books;**

Now suppose we want to print the title of the book with highest number of pages. One way to do this is

**SELECT MAX(pages), title FROM books;**

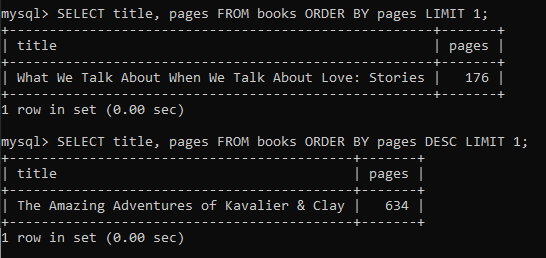
However this will give incorrect result. So we could use subqueries.

1. SUBQUERIES intro. To solve the problem above we will use subqueries



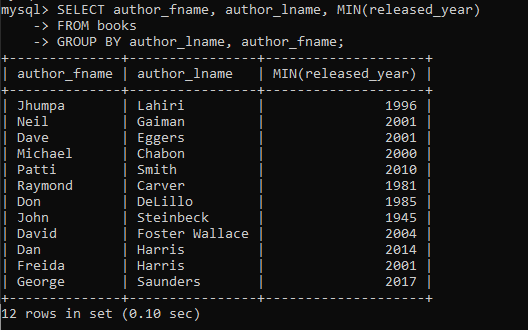
However subqueries are a little slower since we are in essence running two queries.

So we could solve the same problem using ORDER BY

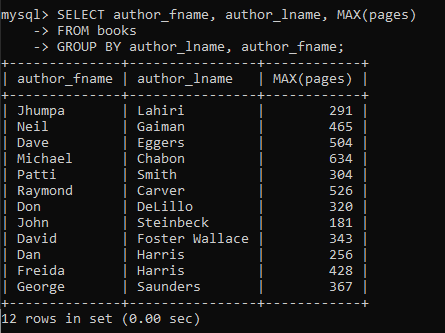


1. Using MIN/MAX with GROUP BY:

Consider the problem: Find the year each author published their first book.



Example: Find the longest page count for each author

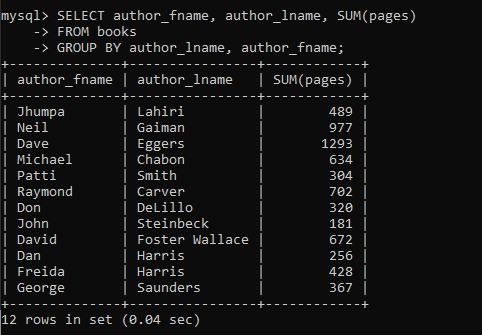


1. SUM()

Example: sum all pages in the entire database

**SELECT SUM(pages) FROM books;**

Example: sum all pages each author has written



1. AVG(): calculates average

Example: Calculate the average released\_year across all books

**SELECT AVG(released\_year) FROM books;**

Example: Calculate the average page count across all books

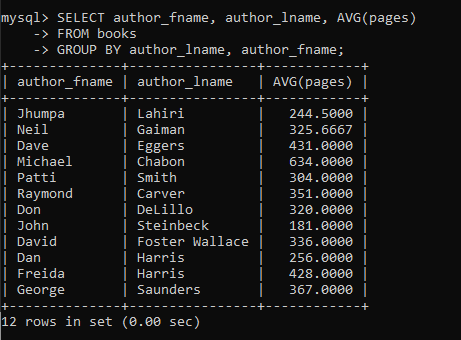
**SELECT AVG(pages) FROM books;**

Using AVG() with group by

Example: Calculate the average stock\_quantity for books released in the same year

**SELECT AVG(stock\_quantity) FROM books GROUP BY released\_year;**

Example: calculate average page count for each author



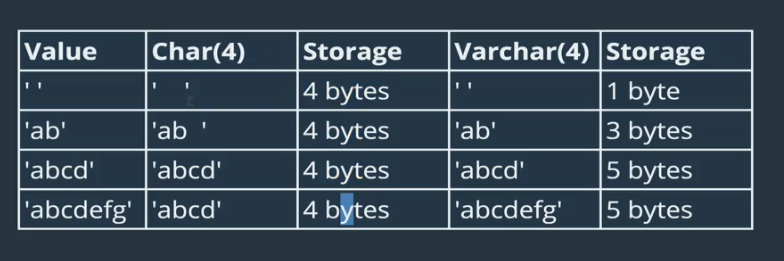
Revisiting Datatypes

1. Key difference between VARCHAR and CHAR: CHAR has a fixed length.

<https://stackoverflow.com/a/15553059/5767939>

CHAR is faster for fixed length text. E.g. State names(NY, CH), sex(M, F)

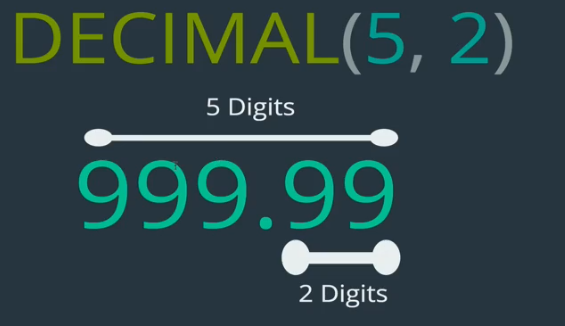
The following table shows the difference between char and varchar.



1. DECIMAL(a, b):

a = total number of digits, before and after the decimal point

b = number of digits after the decimal point



If we try to insert a number that is too large than the constraint, it will store the highest possible decimal number. For e.g. for DECIMAL(5, 2), if we try to insert 3834989, 999.99 will be stored.

Similarly if we specify more than 2 digits for the above example, it will round off, the value, i.e. 298.99999, will become 299.00

1. Difference between FLOAT, DOUBLE, and DECIMAL in MySQL.

DOUBLE is fixed-point data type and calculations are exact.

The FLOAT and DOUBLE data types are floating-point types and calculations are approximate.

Because of problems with floating-point types they will be deprecated in future versions of MySQL.

However floating-point types have several advantages. They can store larger numbers using less space, but it comes at the cost of precision.

1. Always use DECIMAL unless precision is not a problem.
2. DATE: values with a date but no time

Format: YYYY-MM-DD

Example usage: date of birth

1. TIME: stores time but not dates

Format: HH:MM:SS

1. DATETIME: values with date and time

Format: YYYY-MM-DD HH:MM:SS

1. Creating tables with date-time data types

**CREATE TABLE people(name VARCHAR(100), birthdate DATE, birthtime TIME, birthdt DATETIME);**

Inserting values

**INSERT INTO people(name, birthdate, birthtime, birthdt) VALUES(‘Padma’, ‘1989-11-11’, ’10:07:34’, ‘1989-11-11 10:07:34’);**

1. Functions to manipulate data time data types

CURDATE(): gives the current date

CURTIME(): gives the current time

NOW(): gives the current date-time

Using these functions we could use queries like

**INSERT INTO people(name, birthdate, birthtime, birthdt) VALUES(‘Microwave’, CURDATE(), CURTIME(), NOW());**

We could also use these functions to compare time and do date math.

1. <https://dev.mysql.com/doc/refman/5.5/en/date-and-time-functions.html>
2. Some more data time functions

DAY() – extracts day from date, data-time

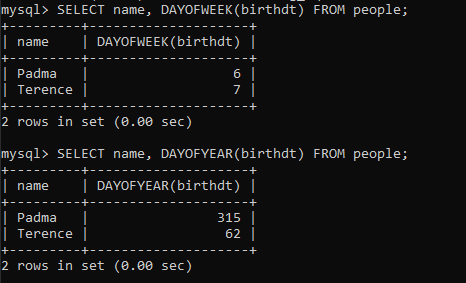
SELECT name, DAY(birthdt) FROM people;

DAYNAME() – extracts and converts day name from data, date-time

SELECT name, DAYNAME(birthdt) FROM people;

DAYOFWEEK()

DAYOFYEAR()



1. More functions

MONTH()

WEEK()

YEAR()

MONTHNAME()

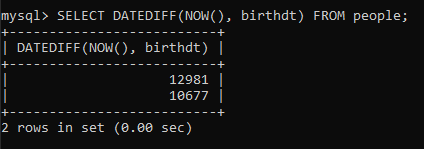
DATE\_FORMAT()

**SELECT DATE\_FORMAT(birthdt, '%Y %M %W') FROM people;**

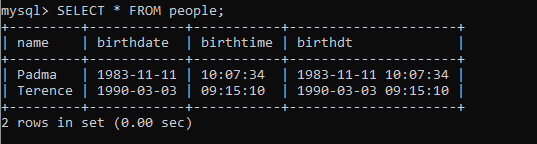
1. Date math. The following functions are available

DATEDIFF(expr1, expr2): returns the number of days between two dates. It takes both DATE and DATETIME as arguments.

Example: The following query returns the number of days since a persons birthday

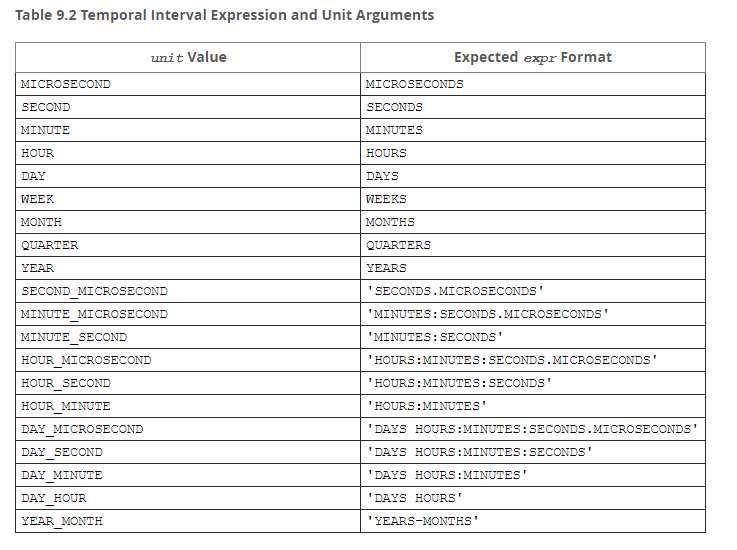


1. Note the following is the table for this entire section

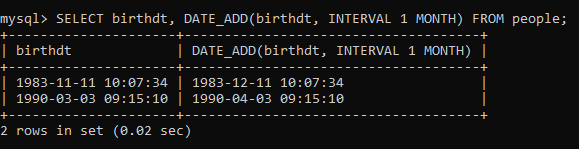


1. DATE\_ADD(): It has a little weird syntax, unlike the ones seen so far

Refer the documentation here: <https://dev.mysql.com/doc/refman/5.7/en/date-and-time-functions.html#function_date-add>



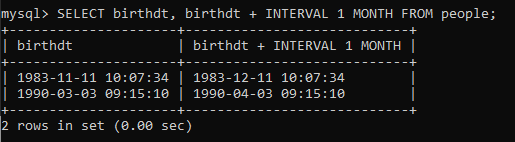
Example: If we want to add 1 month to each birthdate



DATE\_SUB(), works the same way as date add.

1. Another way of adding datetime is to use the +/- .

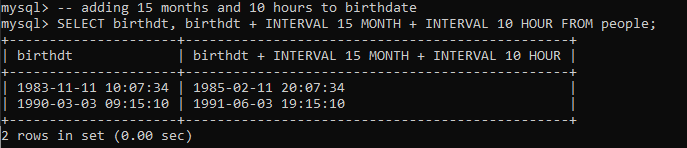
For the problem above we could use something like



Example 2: Subtract 5 months

SELECT birthdt, birthdt – INTERVAL 5 MONTHS FROM people;

Example: if we want to add 15 months and 10 hours, and if we had used DATE\_ADD() we would have to do two separate queries. However using +/- we could do this in a single query



1. Working with timestamps
2. TIMESTAMP: it is also a datatype in MySQL. It stores both date and time just like DATETIME, however there is one major difference between the two.

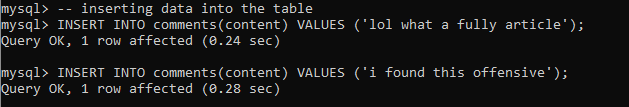
DATETIME stores dates from year 1000 to 9999

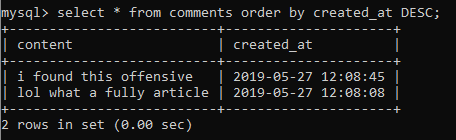
TIMESTAMP stores dates from year 1970 to 2038

We still use TIMESTAMP is because it takes less space.

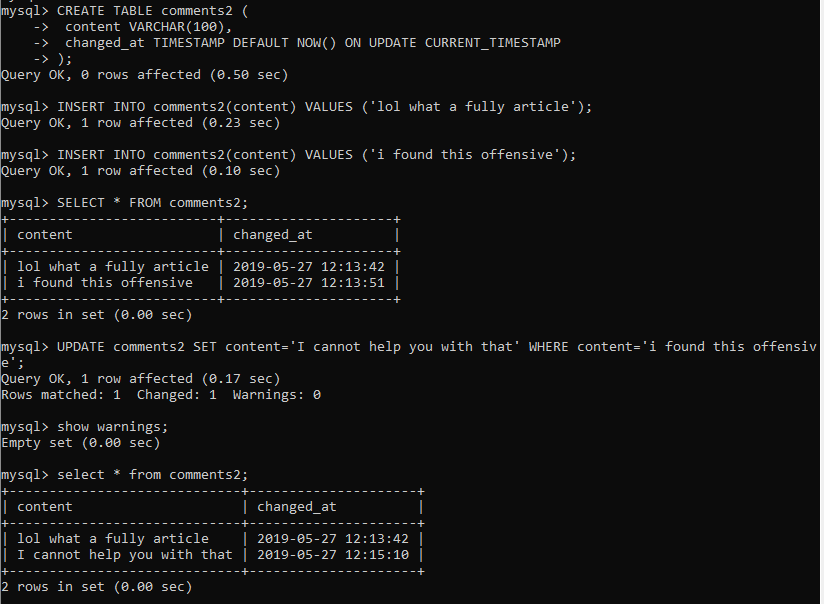
Creating a table using a TIMESTAMP datatype

**CREATE TABLE comments(content VARCHAR(100), created\_at TIMESTAMP DEFAULT NOW());**





1. There’s another important concept with storing comments. They can be edited. So we need to store the most recent version with that timestamp.



NOTE: CURRENT\_TIMESTAMP does the same thing as NOW()

Logical Operators

1. Not equal ( != ): It is similar to the ‘=’ operator that we have been using.

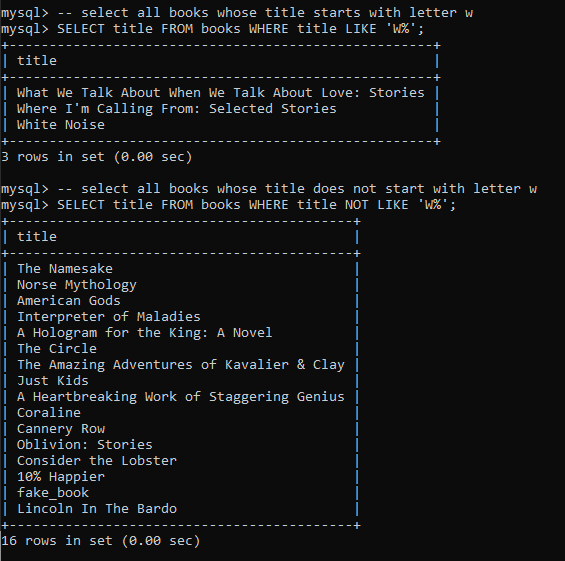
**SELECT \* FROM books WHERE released\_year = 2017;**

The above query selects all books that were released in the year 2017. Now if we want to select those books that were not released in the year 2017, we use the != operator.

**SELECT \* FROM books WHERE released\_year != 2017;**

1. NOT LIKE: it is the opposite of LIKE. Like LIKE it is used to match patterns and select data.

Example of LIKE vs NOT LIKE

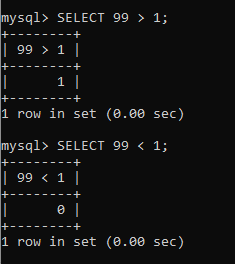


1. >, <, >=, <= operators:

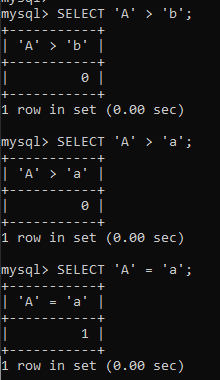
Example: select all books released after the year 2000

**SELECT \* FROM books WHERE released\_year > 2000;**

**SELECT title, stock\_quantity FROM books WHERE stock\_quantity >= 100;**



Since MySQL is case insensitive, so ‘A’ is equal to ‘a’.



1. Logical And (&&) operator:

Example: select books published by Dave Eggers, published after year 2010.

**SELECT title FROM books WHERE author\_fname=’Dave’ AND author\_lname=’Eggers’ AND released\_year > 2010;**

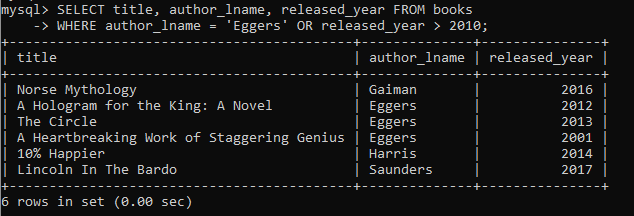
NOTE: we could either use keyword ‘AND’ or the symbol ‘&&’.

We could even write more complicated queries like

**SELECT title, author\_lname, released\_year FROM books, WHERE author\_lname = ‘Eggers’ AND released\_year > 2010 AND title LIKE ‘%novel%’;**

1. Logical or( || ) operator:

Example: select book names where author name is ‘eggers’ or released year > 2010



1. BETWEEN operator: it allows us to select values based of off two values – lower and upper range.

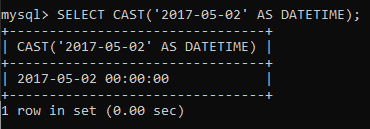
Example: find all books that have page count between 100 and 500 pages

**SELECT title, pages FROM books WHERE pages BETWEEN 100 AND 500;**

Example: find all books that were released between 2004 and 2015

**SELECT title, released\_year FROM books WHERE released\_year BETWEEN 2004 && 2015;**

1. We can also use **NOT BETWEEN**, to select those rows that are not in some range.
2. Using CAST(): it converts one data type to another



NOTE: when we are using BETWEEN with date or time values, it is best to use CAST() to convert the values to the desired data type.

Example: suppose we want to select people with birthday between Jan 1 1980 and Jan 1 2000.

**SELECT name, birthdt FROM people WHERE birthdt BETWEEN ‘1980-01-01’ AND ‘2000-01-01’;**

While the above query will run perfectly without any errors the best practice is to use the following.

**SELECT name, birthdt FROM people WHERE birthdt BETWEEN CAST(‘1980-01-01’ AS DATETIME) AND CAST(‘2000-01-01’ AS DATETIME);**

1. IN operator:

Example: select all books by Lahiri OR Carver OR Smith

One way of doing this is

**SELECT title, author\_lname FROM books WHERE author\_lname = ‘Carver’ OR author\_lname = ‘Lahiri’ OR author\_lname = ‘Smith’;**

A better way to do this is to use IN

**SELECT title, author\_lname FROM books WHERE author\_lname IN (‘Carver’, ‘Lahiri’, ‘Smith’);**

1. There is also a NOT IN

Example: see all books that were not published in even number years.

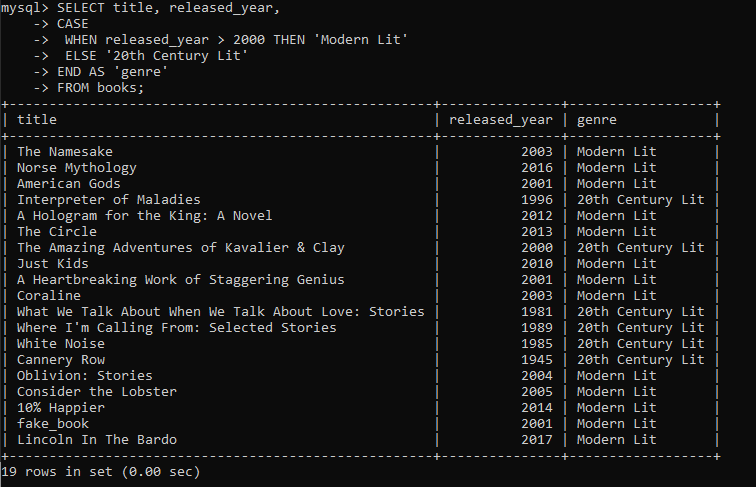
**SELECT \* FROM books WHERE released\_year NOT IN (2000, 2002, 2004, 2006, 2008, 2010, 2012, 2014, 2016, 2018);**

Yet another way to do this is

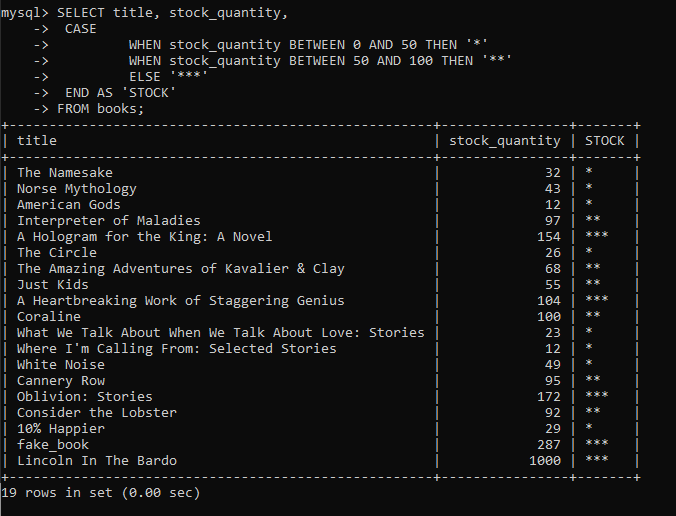
**SELECT \* FROM books WHERE released\_year % 2 != 0;**

1. Using case statements to add logic to queries

Example: suppose we want to select title, released\_year, and genre(not in the table) from books. If released year is greater than 2000 genre should be ‘Modern Lit’ else it should be ‘20th Century Lit’



Another example:



NOTE: CASE is exactly like if-else in most programming languages. So the order of the WHEN part matters.

Joins

1. Relationship basics: there are 4 types of relationships

One to One relationship: e.g. Customer and Customer Details, for each customer, we store the most commonly used info like username, password and corresponding to each customer we have more info about that customer in another table.

Once to Many relationship: e.g. Book and Reviews, one book can have many reviews, but each review belongs to a single book

Many to Many relationship: e.g. Books and Authors, a book can have many author and an author can write many books

Many to One relationship

1. One to Many Relationships is the main focus of this module.
2. Creating foreign keys.

**CREATE TABLE customers(**

**id INT AUTO\_INCREMENT PRIMARY KEY,**

**first\_name VARCHAR(100),**

**last\_name VARCHAR(100),**

**email VARCHAR(100)**

**);**

**CREATE TABLE orders(**

**id INT AUTO\_INCREMENT PRIMARY KEY,**

**order\_date DATE,**

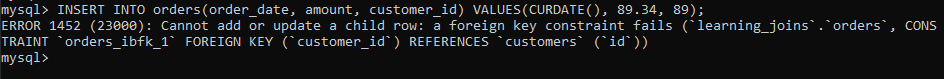
**amount DECIMAL(8,2),**

**customer\_id INT,**

***FOREIGN KEY(customer\_id) REFERENCES customers(id)***

**);**

1. Since customer\_id is a foreign key in orders table. When we try to insert a row whose customer\_id does not match any customers.id we will get the following error.



Since 89 is not a valid customers.id, we get this error.