

# The Complete mySQL bootcamp from Beginner to Expert

## 1. Setting Up Tools & Introduction:

MySQL Database & Tables commands, MySQL Comments, Insert data into a table, Null Values, escape characters & Double Quotes, Default Values, Primary Key & Auto\_Increment

## 2. MySQL CRUD:

Create, Read, Update, Delete, Aliases, CREATE, SELECT, UPDATE, DELETE Queries, CRUD Exercises

## 3. MySQL string functions:

CONCAT(), CONCAT\_WS(), SUBSTR(), REPLACE(), REVERSE(), CHAR\_LENGTH(), LENGTH(), UPPER(), LOWER(), INSERT(), LEFT(), RIGHT(), REPEAT(), TRIM(), String Functions Exercises

## 4. Refining Selections:

DISTINCT, ORDER BY ASC DESC, LIMIT , LIKE & %\_ wildcards, Exercises

## 5. Aggregate Functions:

COUNT, GROUP BY, MIN & MAX, subqueries, Grouping by multiple columns, Min & Max with grouping, SUM, AVG, Exercises

## 6. Revisiting Data Types:

CHAR & VARCHAR, INT, TINYINT, SMALLINT, MEDIUMINT, BIGINT, DECIMAL, FLOAT & DOUBLE, DATE, TIME, DATETIME, CURDATE & CURTIME, NOW(), MONTHNAME(), YEAR(), DAYOFWEEK(), DAYOFYEAR(), HOUR(), MINUTE(), SECOND(), Timestamps, default NOW() & ON UPDATE NOW(), Exercises

## 7. Logical & Comparison Operators:

Comparison operators: !=, NOT LIKE, >, <, <=, >= | Logical operators: AND, OR, BETWEEN, NOT BETWEEN, DATE & TIME comparisons, IN operator , MODULO, CASE, IS NULL, IS NOT NULL, Exercises.

## 8. Constraints & ALTER TABLE:

Unique constraint | Check Constraints | Named Constraints | Multiple Column Constraints | Alter table: Adding Columns, dropping columns, renaming tables, modify Columns, change column, adding constraint, dropping constraint

## 9. One to Many & Joins:

Data Relationships: one-to-one, one-to-many, many-to-many | PRIMARY\_KEY, FOREIGN\_KEY | Joins: Cross Join, Inner Join (JOIN ON), Inner-Join with Group by, Left Join, left-join with Group By & IFNULL(), Right Join | On Delete Cascade | Exercises

# The Complete mySQL bootcamp from Beginner to Expert

## 10. Many to Many & Joins:

many to many Join or Union table with FOREIGN KEY | Using Joins on many to many

## 11. Introducing Views:

views introduction | updatable views | HAVING clause | WITH ROLLUP | MySQL Modes: STRICT\_TRANS\_TABLES

## 12. Window Functions:

Windows Functions introduction | Using OVER() | Partition By clause | ORDER BY with Windows | RANK() | DENSE\_RANK() | ROW\_NUMBER() | NTILE() | FIRST\_VALUE() | LAST\_VALUE() | NTH\_VALUE() | LEAD() | LAG()

## 13. Instagram Database Clone:

Create tables: users, photos, comments, likes, follows & hashtags | Import large SQL data file | Instagram clone database tasks

# MySQL Setting Up & Introduction

# MySQL Basic Database Commands

```
CREATE database <name>;
```

```
CREATE database animal_shelter;
```

```
CREATE database DogApp;
```

```
CREATE database Dog App;
```

Database names cannot contain spaces but underscores are acceptable.

```
DROP database <name>;
```

```
DROP database animal_shelter;
```

Dropping a database will completely remove it, i.e. delete it and is irreversible.

```
SHOW databases;
```

Will show a list of databases on the SQL server.

```
USE database <name>;
```

```
USE database animal_shelter;
```

Using a database means that we enter into that database and all commands that we execute hence will be to that database. We can verify what database we are in: 

```
SELECT database();
```

# Tools

Install mySQL server with a crappy GUI called MySQL workbench.

<https://dev.mysql.com/downloads/file/?id=514517>

Install a less crappy visual GUI.

<https://dbgate.org/database/mysql-client.html>

SQL code beutifier.

<https://codebeautify.org/sqlformatter>

# MySQL Documentation

MySQL String functions docs

<https://dev.mysql.com/doc/refman/8.0/en/string-functions.html>

MySQL aggregate functions docs

<https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html>

MySQL aggregate functions docs

<https://dev.mysql.com/doc/refman/8.0/en/data-types.html>

MySQL Date and time functions docs

<https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html>

MySQL FORMAT\_DATE() function

[https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html#function\\_date-format](https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html#function_date-format)

MySQL Alter Table

<https://dev.mysql.com/doc/refman/8.0/en/alter-table.html>

# MySQL Tables

Databases contain tables of data, with the data type for each column being specified.

## Number Data types

- INT
- SMALLINT
- TINYINT
- MEDIUMINT
- BIGINT
- DECIMAL
- NUMERIC
- FLOAT
- DOUBLE
- BIT

## String Data types

- CHAR
- VARCHAR
- BINARY
- VARBINARY
- BLOB
- TINYBLOB
- MEDIUMBLOB
- LONGBLOB
- TEXT
- TINYTEXT
- MEDIUMTEXT
- LONGTEXT
- ENUM

## Date Data types

- DATE
- DATETIME
- TIMESTAMP
- VARBINARY
- TIME
- YEAR

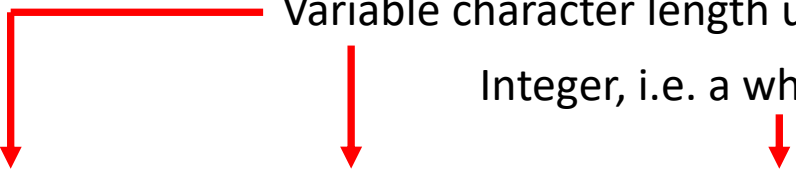
For more information on data types and the differences between them consult the MySQL manual:

<https://dev.mysql.com/doc/refman/8.0/en/data-types.html>

# MySQL Table Commands

```
CREATE TABLE <tablename> (  
  column_name data_type,  
  Column_name data_type );
```

```
CREATE TABLE cats (  
  cat_name VARCHAR(100),  
  cat_breed VARCHAR(100),  
  cat_age INT );
```



Variable character length upto 100 characters VARCHAR(100)

Integer, i.e. a whole number, no decimals.

cat_name	cat_breed	cat_age
John Snow	Russian Blue	6
Syria	Domestic Short Hair	7
Mia	Maine Coon	5

**SHOW TABLES;** Will show a list of tables in the database.

**SHOW COLUMNS FROM table\_name;** Will show a list columns of the table.

**DESC table\_name;** Will describe a table and gives us the same information as show collumns from.



# MySQL Table Commands

```
DROP TABLE table_name;      DROP TABLE dogs;
```

Dropping a table will completely remove it, i.e. delete it and is irreversible.

## MySQL Comments

```
-- this command will show tables  
SHOW TABLES;
```

A double hyphen before a line of SQL code will indicate that it is a comment and not run this line of SQL code.

# MySQL Insert data into a table

```
INSERT INTO <tablename>
( column_name, column_name, column_name, column_name )
VALUES ( row_value, row_value, row_value );
```

```
INSERT INTO cats
(cat_name, cat_breed, cat_age)
VALUES ('John Snow', 'Russian Blue', 7);
```

It is possible to enter multiple rows with values separated by a coma.

```
INSERT INTO cats
(cat_name, cat_breed, cat_age)
VALUES
('John Snow', 'Russian Blue', 7),
('Syria', 'Domestic Short Hair', 7),
('Mia', 'Maine Coon', 6);
```

# MySQL NULL Values

```
INSERT INTO cats () VALUES ();
```

MySQL by default lets us insert null values where null is empty. Not zero because zero is an integer value. But we want cat name, breed and age to be required values.

cat_name	cat_breed	cat_age
John Snow	Russian Blue	6
Syria	Domestic Short Hair	7
Mia	Maine Coon	5
NULL	NULL	NULL

```
CREATE TABLE cats (  
  cat_name VARCHAR(100) NOT NULL,  
  cat_breed VARCHAR(100) NOT NULL,  
  cat_age INT NOT NULL );
```

We have to add the NOT NULL constraint for the columns when we create a table to only permit actual values being added.

```
INSERT INTO cats () VALUES ();
```

**ERROR 1364 (HY000): Field 'cat\_name' doesn't have a default value**

Now it will not let us insert NULL values

# MySQL Escape Characters and double quotes

```
INSERT INTO cats
  (cat_name, cat_breed, cat_age)
VALUES
  ('John's cat', 'Russian Blue', 7);
```

We wrap strings in single quote marks but if a string value has an apostrophe in it it will see this as the end of the string and throw an error. We can also wrap the string in double quotes but some MySQL programs will not accept this so it is best to always use single quotes.

```
INSERT INTO cats
  (cat_name, cat_breed, cat_age)
VALUES
  ('John\'s cat', 'Russian Blue', 7);
```

To escape the apostrophe i.e. not have it included in the string we need to put a backslash in front of it.

```
INSERT INTO cats
  (cat_name, cat_breed, cat_age)
VALUES
  ('John"the cat"', 'Russian Blue', 7);
```

We can include double quotes as part of a string that is contained within single quotes without any issues. The single quotes mark the beginning and end of the string. What is in between is the value of the string.

# MySQL Default Values

```
CREATE TABLE cats (  
    cat_name VARCHAR(100) DEFAULT 'Unknown',  
    cat_breed VARCHAR(100) DEFAULT 'Domestic Short Hair',  
    cat_age INT DEFAULT 0  
);
```

```
INSERT INTO cats  
    (cat_name, cat_breed, cat_age)  
VALUES  
    ('John Snow', 'Russian Blue', 7),  
    ('Syria', 'Domestic Short Hair', 7),  
    ('Mia', 'Maine Coon', 6)  
;
```

```
INSERT INTO cats () VALUES ();
```

The animal\_shelter takes in animals without collar or chip with the most common breed is domestic short hair .

We can set a DEFAULT name is unknown. DEFAULT breed is Domestic short hair.

Also we can add a default value for age where the age of a cat is unknown, I this case I have chosen to signify 0 as the default age because it must be an integer and that is the most logical dummy integer.

Default Values	cat_name	cat_breed	cat_age
	John Snow	Russian Blue	6
	Syria	Domestic Short Hair	7
	Mia	Maine Coon	5
	Unknown	Domestic Short Hair	0

# MySQL Primary Key and Auto-Increment

cat_name	cat_breed	cat_age
John Snow	Russian Blue	6
Syria	Domestic Short Hair	7
Mia	Maine Coon	5
Unknown	Domestic Short Hair	0
Unknown	Domestic Short Hair	0

The animal shelter has two cats without collar or chip so both rows contain identical data. This is an issue because how do we distinguish between them to perform an operation?

```
CREATE TABLE cats (  
    cat_ID INT NOT NULL PRIMARY KEY AUTO_INCREMENT,  
    cat_name VARCHAR(100) DEFAULT 'Unknown',  
    cat_breed VARCHAR(100) DEFAULT 'Domestic Short Hair',  
    cat_age INT DEFAULT 0  
);
```

It is normal in MySQL that the first column of each table is an ID field that is obligatory, i.e. NOT NULL, and that is an INTEGER and that is auto filled in incrementing the number by one with each new row entry. This automatically avoids duplication of the primary key. It is technically redundant to specify NOT NULL and PRIMARY KEY because a primary key can NEVER be NULL.

```

INSERT INTO cats
    (cat_name, cat_breed, cat_age)
VALUES
    ('John Snow', 'Russian Blue', 7),
    ('Syria', 'Domestic Short Hair', 7),
    ('Mia', 'Main Coon', 6)
;

```

```

INSERT INTO cats () VALUES ();

INSERT INTO cats () VALUES ();

```

Cat_ID	cat_name	cat_breed	cat_age
1	John Snow	Russian Blue	6
2	Syria	Domestic Short Hair	7
3	Mia	Maine Coon	5
4	Unknown	Domestic Short Hair	0
5	Unknown	Domestic Short Hair	0

Now we have a row of cats where each row is identified by a unique primary key number.

```

mysql> DESC cats;
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default          | Extra          |
+-----+-----+-----+-----+-----+-----+
| cat_ID     | int           | NO   | PRI | NULL             | auto_increment |
| cat_name   | varchar(100)  | YES  |     | Unknown          |                |
| cat_breed  | varchar(100)  | YES  |     | Domestic Short Hair |                |
| cat_age    | int           | YES  |     | 0                |                |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

```

The animal\_shelter has an employees table.

```
CREATE TABLE employees (  
    employee_ID INT NOT NULL PRIMARY KEY AUTO_INCREMENT,  
    employee_lastName VARCHAR(100) NOT NULL,  
    employee_firstName VARCHAR(100) NOT NULL,  
    employee_middleName VARCHAR(100),  
    employee_age INT NOT NULL,  
    employee_status VARCHAR(100) DEFAULT 'Employed'  
);
```

```
mysql> DESC employees;
```

Field	Type	Null	Key	Default	Extra
employee_ID	int	NO	PRI	NULL	auto_increment
employee_lastName	varchar(100)	NO		NULL	
employee_firstName	varchar(100)	NO		NULL	
employee_middleName	varchar(100)	YES		NULL	
employee_age	int	NO		NULL	
employee_status	varchar(100)	YES		Employed	

```
6 rows in set (0.00 sec)
```

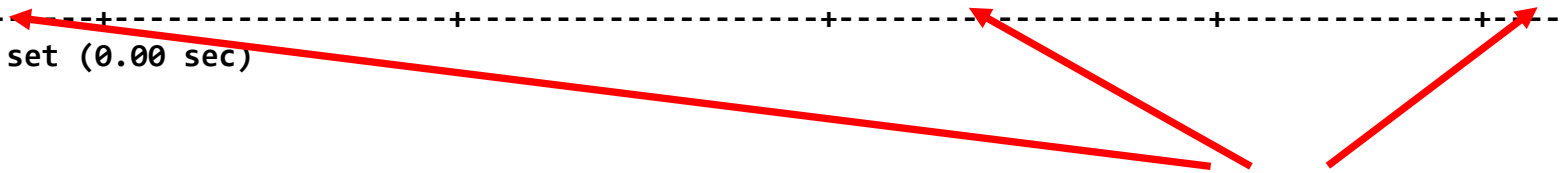


```
INSERT INTO employees
  (employee_firstName, employee_lastName, employee_age)
VALUES
  ('Catman', 'Chris', 40)
;
```

```
mysql> SELECT * FROM employees;
```

employee_ID	employee_lastName	employee_firstName	employee_middleName	employee_age	employee_status
1	Chris	Catman	NULL	40	Employed

1 row in set (0.00 sec)



Note that we only have to specify the not null values, the rest are **automatically created**.

# MySQL CRUD:

Create

Read

Update

Delete

# MySQL CRUD: Create, Read, Update, Delete

```
INSERT INTO cats (cat_name, cat_breed, cat_age)
VALUES ('Ringo', 'Tabby', 4),
      ('Cindy', 'Maine Coon', 10),
      ('Dumbledore', 'Maine Coon', 11),
      ('Egg', 'Persian', 4),
      ('Misty', 'Tabby', 13),
      ('George Michael', 'Ragdoll', 9),
      ('Jackson', 'Sphynx', 7);
```

We can create new row using  
INSERT INTO.

```
mysql> select * from cats;
```

cat_ID	cat_name	cat_breed	cat_age
1	John Snow	Russian Blue	7
2	Syria	Domestic Short Hair	7
3	Mia	Maine Coon	6
4	Unknown	Domestic Short Hair	0
5	Unknown	Domestic Short Hair	0
6	Ringo	Tabby	4
7	Cindy	Maine Coon	10
8	Dumbledore	Maine Coon	11
9	Egg	Persian	4
10	Misty	Tabby	13
11	George Michael	Ragdoll	9
12	Jackson	Sphynx	7

```
12 rows in set (0.00 sec)
```

```
mysql> select cat_name, cat_age from cats;
```

cat_name	cat_age
John Snow	7
Syria	7
Mia	6
Unknown	0
Unknown	0
Ringo	4
Cindy	10
Dumbledore	11
Egg	4
Misty	13
George Michael	9
Jackson	7

12 rows in set (0.00 sec)

```
SELECT * FROM cats;
```

We can select all columns and rows from a table.

```
SELECT cat_name FROM cats;
```

```
SELECT cat_name, cat_age FROM cats;
```

We can select a specific column or columns from a table.



```
SELECT * FROM cats WHERE cat_age = 4;
```

We can use the where clause to filter our select query.

```
mysql> select * FROM cats WHERE cat_age = 4;
```

cat_ID	cat_name	cat_breed	cat_age
6	Ringo	Tabby	4
9	Egg	Persian	4

2 rows in set (0.00 sec)

```
mysql> SELECT cat_name, cat_age FROM cats WHERE cat_age = 4;
```

cat_name	cat_age
Ringo	4
Egg	4

2 rows in set (0.00 sec)

```
SELECT cat_name, cat_age FROM cats WHERE cat_age = 4;
```

```
mysql> SELECT cat_name, cat_breed FROM cats WHERE cat_breed = 'Tabby';
```

```
+-----+-----+
| cat_name | cat_breed |
+-----+-----+
| Ringo    | Tabby     |
| Misty    | Tabby     |
+-----+-----+
2 rows in set (0.00 sec)
```

When we want to match a string we use single quotes.

## MySQL READ: Aliases

```
mysql> SELECT cat_ID as 'reference', cat_name, cat_breed FROM cats WHERE cat_breed = 'Tabby';
```

```
+-----+-----+-----+
| reference | cat_name | cat_breed |
+-----+-----+-----+
|          6 | Ringo    | Tabby     |
|         10 | Misty    | Tabby     |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

We can use an alias as a temporary column name for the output of the query if we might want something more human understandable.

# MySQL UPDATE Queries

```
UPDATE cats
SET cat_breed = 'Domestic Short Hair'
WHERE cat_breed = 'Tabby';
```

**NOTE:** if we make the mistake of not including the where clause then all rows will be updated which could damage the data in a large SQL table.

**NOTE:** It is advisable to run a select query to verify the rows and data before running the update query to avoid inadvertently changing data you do not want to.

```
mysql> SELECT * from cats;
```

cat_ID	cat_name	cat_breed	cat_age
1	John Snow	Russian Blue	7
2	Syria	Domestic Short Hair	7
3	Mia	Maine Coon	6
4	Unknown	Domestic Short Hair	0
5	Unknown	Domestic Short Hair	0
6	Ringo	Domestic Short Hair	4
7	Cindy	Maine Coon	10
8	Dumbledore	Maine Coon	11
9	Egg	Persian	4
10	Misty	Domestic Short Hair	13
11	George Michael	Ragdoll	9
12	Jackson	Sphynx	7

```
12 rows in set (0.00 sec)
```

To update we have to specify three parts:

- 1) The table
- 2) The column we would like to update and the new value
- 3) The condition using where clause.

```
UPDATE cats
SET cat_age = 12
WHERE cat_ID = 7 || cat_id = 8;
```

In this example I want to update the ages of Cindy and dumbledore to 12 because they are litter mates but I cannot use where breed = Maine Coon so I have to be more specific.

# MySQL DELETE Queries

```
mysql> SELECT * from cats;
```

cat_ID	cat_name	cat_breed	cat_age
1	John Snow	Russian Blue	7
2	Syria	Domestic Short Hair	7
3	Mia	Maine Coon	6
4	Unknown	Domestic Short Hair	0
5	Unknown	Domestic Short Hair	0
6	Ringo	Domestic Short Hair	4
7	Cindy	Maine Coon	12
8	Dumbledore	Maine Coon	12
9	Egg	Persian	4
10	Misty	Domestic Short Hair	13
11	George Michael	Ragdoll	9
12	Jackson	Sphynx	7

```
12 rows in set (0.00 sec)
```

```
DELETE from cats  
WHERE cat_name = 'Egg';
```

In this example I want to delete the cat called Egg. Without the WHERE clause all rows would be delete.

```
DELETE from cats  
WHERE cat_name = 'Egg';
```

In this example I want to delete the cats with age 4. Without the WHERE clause all rows would be delete.

# MySQL CRUD exercises

```
CREATE TABLE shirts (  
    shirt_id INT PRIMARY KEY  
    AUTO_INCREMENT,  
    article VARCHAR(50),  
    color VARCHAR(20),  
    shirt_size VARCHAR(2),  
    last_worn INT  
);  
  
INSERT INTO shirts  
    (article, color, shirt_size,  
    last_worn)  
VALUES  
    ('t-shirt', 'white', 'S', 10),  
    ('t-shirt', 'green', 'S', 200),  
    ('polo shirt', 'black', 'M', 10),  
    ('tank top', 'blue', 'S', 50),  
    ('t-shirt', 'pink', 'S', 10),  
    ('polo shirt', 'red', 'S', 0),  
    ('tank top', 'white', 'S', 200),  
    ('tank top', 'blue', 'M', 15)  
;
```

```
INSERT INTO shirts  
(article, color, shirt_size, last_worn) VALUES  
('polo shirt', 'purple', 'M', 50);
```

```
SELECT article, color FROM shirts;
```

```
SELECT shirt_id FROM shirts WHERE shirt_size = 'M';
```

```
SELECT article, color, shirt_size, last_worn FROM  
shirts WHERE shirt_size = 'M';
```

```
SELECT article, color, shirt_size, last_worn  
FROM shirts WHERE shirt_size = 'M';
```

```
UPDATE shirts SET shirt_size = 'M' WHERE  
article = 'polo shirt';
```

```
UPDATE shirts SET last_worn = 0 WHERE  
last_worn = 15;
```

```
UPDATE shirts SET shirt_size = 'XL', color =  
'off white' WHERE color = 'white';
```

```
DELETE from shirts WHERE last_worn = 200;
```

```
DELETE from shirts WHERE article = 'tank  
top';
```

```
DELETE FROM shirts;
```

```
DROP TABLE shirts;
```



# MySQL String Functions

# MySQL Creating the Test Database

```
CREATE DATABASE book_shop;  
USE book_shop;  
SHOW tables;  
Empty set (0.01 sec)
```

We can create a new database and start using it. It is an empty database with no tables.

**source**

**C:/users/EL\_EL/Desktop/book\_data.sql**  
**Query OK, 0 rows affected (0.01 sec)**

**Query OK, 16 rows affected (0.01 sec)**  
**Records: 16 Duplicates: 0 Warnings: 0**

Rather than copy out all the SQL code and paste it into the command line we can specify an absolute file path to an SQL file and load it directly from the file using the source command.

```
mysql> SELECT * FROM books;
```

book_id	title	author_fname	author_lname	released_year	stock_quantity	pages
1	The Namesake	Jhumpa	Lahiri	2003	32	291
2	Norse Mythology	Neil	Gaiman	2016	43	304
3	American Gods	Neil	Gaiman	2001	12	465
4	Interpreter of Maladies	Jhumpa	Lahiri	1996	97	198
5	A Hologram for the King: A Novel	Dave	Eggers	2012	154	352
6	The Circle	Dave	Eggers	2013	26	504
7	The Amazing Adventures of Kavalier & Clay	Michael	Chabon	2000	68	634
8	Just Kids	Patti	Smith	2010	55	304
9	A Heartbreaking Work of Staggering Genius	Dave	Eggers	2001	104	437
10	Coraline	Neil	Gaiman	2003	100	208
11	What We Talk About When We Talk About Love: Stories	Raymond	Carver	1981	23	176
12	Where I'm Calling From: Selected Stories	Raymond	Carver	1989	12	526
13	White Noise	Don	DeLillo	1985	49	320
14	Cannery Row	John	Steinbeck	1945	95	181
15	Oblivion: Stories	David	Foster Wallace	2004	172	329
16	Consider the Lobster	David	Foster Wallace	2005	92	343

```
16 rows in set (0.00 sec)
```

# MySQL String functions - CONCAT

```
SELECT CONCAT(author_fname, ' ', author_lname) FROM books;
```

```
+-----+
| CONCAT(author_fname, ' ', author_lname) |
+-----+
| Jhumpa Lahiri                          |
| Neil Gaiman                            |
| Neil Gaiman                            |
| Jhumpa Lahiri                          |
| Dave Eggers                            |
| Dave Eggers                            |
| Michael Chabon                         |
| Patti Smith                            |
| Dave Eggers                            |
| Neil Gaiman                            |
| Raymond Carver                         |
| Raymond Carver                         |
| Don DeLillo                            |
| John Steinbeck                         |
| David Foster Wallace                   |
| David Foster Wallace                   |
+-----+
16 rows in set (0.00 sec)
```

CONCAT will allow us to select data from a table and combine two or more columns into one column output.

We can tidy this up a bit more by using AS to rename the output column.

```
+-----+
| author_name                            |
+-----+
| Jhumpa Lahiri                          |
| Neil Gaiman                            |
| Neil Gaiman                            |
| Jhumpa Lahiri                          |
| Dave Eggers                            |
| Dave Eggers                            |
| Michael Chabon                         |
| Patti Smith                            |
| Dave Eggers                            |
| Neil Gaiman                            |
| Raymond Carver                         |
| Raymond Carver                         |
| Don DeLillo                            |
| John Steinbeck                         |
| David Foster Wallace                   |
| David Foster Wallace                   |
+-----+
16 rows in set (0.00 sec)
```

```
SELECT CONCAT(author_fname, ' ', author_lname) AS author_name FROM books;
```

# MySQL String functions – CONCAT\_WS (with Seperator)

CONCAT\_WS will concatenate columns but with a pre-defined string between each column.

```
SELECT CONCAT_WS('-',author_fname, author_lname) AS author FROM books;
```

```
+-----+
| author |
+-----+
| Jhumpa-Lahiri |
| Neil-Gaiman |
| Neil-Gaiman |
| Jhumpa-Lahiri |
| Dave-Eggers |
| Dave-Eggers |
| Michael-Chabon |
| Patti-Smith |
| Dave-Eggers |
| Neil-Gaiman |
| Raymond-Carver |
| Raymond-Carver |
| Don-DeLillo |
| John-Steinbeck |
| David-Foster Wallace |
| David-Foster Wallace |
+-----+
16 rows in set (0.00 sec)
```

Note how the first parameter of the CONCAT\_WS is the string that we want to use as the separator.

This could be useful to build a list of URLs of author names.

# MySQL String functions – SUBSTR(Sub-string)

```
SELECT SUBSTR('HEllo World', 1, 4);
```

```
+-----+
| SUBSTR('HEllo World', 1, 4) |
+-----+
| HEll                          |
+-----+
```

```
SELECT SUBSTR('HEllo World', 7);
```

```
+-----+
| SUBSTR('HEllo World', 7) |
+-----+
| World                    |
+-----+
```

```
SELECT SUBSTR('HEllo World', -1);
```

```
+-----+
| SUBSTR('HEllo World', -1) |
+-----+
| d                          |
+-----+
```

```
SELECT SUBSTR('HEllo World', -5, 3);
```

```
+-----+
| SUBSTR('HEllo World', -5, 3) |
+-----+
| Wor                          |
+-----+
```

SUBSTR will take a sub section of a string where the first parameter is the string, the second is the starting point and the third is the number of characters.

i.e. take from position 1 four characters from 'Hello World'.

If I only specify a starting position then it will include everything from that position till the end of the string.

I can also start counting from the end of the string using negative numbers, i.e. show me the last character of the string.

If I want 3 characters starting from the fifth till last character:

```
SELECT SUBSTR(title, 1, 15) FROM books;
```

A real world example of this might be to abbreviate the book titles to 15 characters.

initial	author_lname
J	Lahiri
N	Gaiman
N	Gaiman
J	Lahiri
D	Eggers
D	Eggers
M	Chabon
P	Smith
D	Eggers
N	Gaiman
R	Carver
R	Carver
D	DeLillo
J	Steinbeck
D	Foster Wallace
D	Foster Wallace

```
+-----+
| SUBSTR(title, 1, 15) |
+-----+
| The Namesake        |
| Norse Mythology     |
| American Gods       |
| Interpreter of       |
| A Hologram for      |
| The Circle          |
| The Amazing Adv     |
| Just Kids           |
| A Heartbreaking     |
| Coraline            |
| What We Talk About |
| Where I'm Callin'   |
| White Noise         |
| Cannery Row         |
| Oblivion: Stori     |
| Consider the Lo     |
+-----+
16 rows in set (0.00 sec)
```

Or to get authors first name initial and last name

```
SELECT SUBSTR(author_fname, 1, 1) AS initial, author_lname FROM books;
```

# MySQL String functions – Using CONCAT with SUBSTR

```
SELECT CONCAT(SUBSTR(title, 1, 10), '...') AS short_title FROM BOOKS;
```

```
+-----+
| short_title |
+-----+
| The Namesa... |
| Norse Myth... |
| American G... |
| Interprete... |
| A Hologram... |
| The Circle... |
| The Amazin... |
| Just Kids...  |
| A Heartbre... |
| Coraline...   |
| What We Ta... |
| Where I'm ... |
| White Nois... |
| Cannery Ro... |
| Oblivion: ... |
| Consider t... |
+-----+
16 rows in set (0.00 sec)
```

In this example we want to concatenate the first 10 characters of the book title with ....

To do this we nest the SUBSTR() function within the CONCAT() function. The nested function will be run first then the outer function.

We can also use concatenate and substring to make firstname initial dot author lastname.

```
+-----+
| author |
+-----+
| J.Lahiri |
| N.Gaiman |
| N.Gaiman |
| J.Lahiri |
| D.Eggers |
| D.Eggers |
| M.Chabon |
| P.Smith  |
| D.Eggers |
| N.Gaiman |
| R.Carver |
| R.Carver |
| D.DeLillo |
| J.Steinbeck |
| D.Foster Wallace |
| D.Foster Wallace |
+-----+
16 rows in set (0.00 sec)
```

```
SELECT CONCAT(SUBSTR(author_fname,1,1), '.',author_lname)
AS author FROM books;
```

```
SELECT CONCAT(SUBSTR(author_fname,1,1), '.',SUBSTR(author_lname,1,1)) AS
author_initials FROM books;
```

```
+-----+
| author_initials |
+-----+
| J.L
| N.G
| N.G
| J.L
| D.E
| D.E
| M.C
| P.S
| D.E
| N.G
| R.C
| R.C
| D.D
| J.S
| D.F
| D.F
+-----+
```

16 rows in set (0.00 sec)

Or just get the authors initials.

Or just get authors initials and  
Short title in two columns.

```
+-----+-----+
| author_init | short_title |
+-----+-----+
| J.L
| N.G
| N.G
| J.L
| D.E
| D.E
| M.C
| P.S
| D.E
| N.G
| R.C
| R.C
| D.D
| J.S
| D.F
| D.F
+-----+-----+
```

16 rows in set (0.00 sec)

```
SELECT CONCAT(
    SUBSTR(author_fname,1,1), '.',SUBSTR(author_lname,1,1))AS author_init,
    CONCAT(SUBSTR(title, 1, 10), '...') AS short_title
FROM books;
```



# MySQL String functions – REPLACE

Replace does not replace the data in the SQL table but is a function to format the output of the data.

```
SELECT REPLACE(title, ' ', '-') AS url_title FROM BOOKS;
```

```
+-----+
| url_title
+-----+
| The-Namesake
| Norse-Mythology
| American-Gods
| Interpreter-of-Maladies
| A-Hologram-for-the-King:-A-Novel
| The-Circle
| The-Amazing-Adventures-of-Kavalier-&-Clay
| Just-Kids
| A-Heartbreaking-Work-of-Staggering-Genius
| Coraline
| What-We-Talk-About-When-We-Talk-About-Love:-Stories
| Where-I'm-Calling-From:-Selected-Stories
| White-Noise
| Cannery-Row
| Oblivion:-Stories
| Consider-the-Lobster
+-----+
16 rows in set (0.00 sec)
```

Replace takes three parameters:

- 1 – what we want to operate on
- 2 – What we want to replace
- 3 – what we want to replace it with.

Replace is case sensitive.

# MySQL String functions – REVERSE

```
SELECT REVERSE('Hello World');
```

```
+-----+
| REVERSE('Hello World') |
+-----+
| dlroW olleH            |
+-----+
1 row in set (0.00 sec)
```

```
SELECT REVERSE(author_fname) FROM books;
```

```
+-----+
| REVERSE(author_fname) |
+-----+
| apmuhJ                |
| lieN                  |
| lieN                  |
| apmuhJ                |
| evaD                  |
| evaD                  |
| leahciM               |
| ittaP                 |
| evaD                  |
| lieN                  |
| dnomyaR               |
| dnomyaR               |
| noD                   |
| nhoJ                  |
| divaD                 |
| divaD                 |
+-----+
16 rows in set (0.00 sec)
```

# MySQL String functions – CHAR\_LENGTH

```
SELECT CHAR_LENGTH(title) AS total_characters, title FROM books;
```

total_characters	title
12	The Namesake
15	Norse Mythology
13	American Gods
23	Interpreter of Maladies
32	A Hologram for the King: A Novel
10	The Circle
41	The Amazing Adventures of Kavalier & Clay
9	Just Kids
41	A Heartbreaking Work of Staggering Genius
8	Coraline
51	What We Talk About When We Talk About Love: Stories
40	Where I'm Calling From: Selected Stories
11	White Noise
11	Cannery Row
17	Oblivion: Stories
20	Consider the Lobster

CHAR-LENGTH gives the total number of characters in a string.

# MySQL String functions –LENGTH

```
SELECT LENGTH(title) AS bytes, title FROM books;
```

bytes	title
12	The Namesake
15	Norse Mythology
13	American Gods
23	Interpreter of Maladies
32	A Hologram for the King: A Novel
10	The Circle
41	The Amazing Adventures of Kavalier & Clay
9	Just Kids
41	A Heartbreaking Work of Staggering Genius
8	Coraline
51	What We Talk About When We Talk About Love: Stories
40	Where I'm Calling From: Selected Stories
11	White Noise
11	Cannery Row
17	Oblivion: Stories
20	Consider the Lobster

LENGTH gives the size of a string in bytes. For ASCII it is one byte for character.

# MySQL String functions – UPPER & LOWER

```
SELECT UPPER(title) FROM BOOKS;
```

UPPER(title)
THE NAMESAKE
NORSE MYTHOLOGY
AMERICAN GODS
INTERPRETER OF MALADIES
A HOLOGRAM FOR THE KING: A NOVEL
THE CIRCLE
THE AMAZING ADVENTURES OF KAVALIER & CLAY
JUST KIDS
A HEARTBREAKING WORK OF STAGGERING GENIUS
CORALINE
WHAT WE TALK ABOUT WHEN WE TALK ABOUT LOVE: STORIES
WHERE I'M CALLING FROM: SELECTED STORIES
WHITE NOISE
CANNERY ROW
OBLIVION: STORIES
CONSIDER THE LOBSTER

UPPER & LOWER change the case of a string.

```

SELECT CONCAT(
    'www.thebookstore.com/',
    (REPLACE(CONCAT_WS('/',
        LOWER(REPLACE(CONCAT_WS('-', author_fname, author_lname), ' ', '-')),
        LOWER(REPLACE(title, ' ', '-'))), ':', ''))
    AS book_url
FROM books;

```

book_url
www.thebookstore.com/jhumpa-lahiri/the-namesake
www.thebookstore.com/neil-gaiman/norse-mythology
www.thebookstore.com/neil-gaiman/american-gods
www.thebookstore.com/jhumpa-lahiri/interpreter-of-maladies
www.thebookstore.com/dave-eggers/a-hologram-for-the-king-a-novel
www.thebookstore.com/dave-eggers/the-circle
www.thebookstore.com/michael-chabon/the-amazing-adventures-of-kavalier-&-clay
www.thebookstore.com/patti-smith/just-kids
www.thebookstore.com/dave-eggers/a-heartbreaking-work-of-staggering-genius
www.thebookstore.com/neil-gaiman/coraline
www.thebookstore.com/raymond-carver/what-we-talk-about-when-we-talk-about-love-stories
www.thebookstore.com/raymond-carver/where-i'm-calling-from-selected-stories
www.thebookstore.com/don-delillo/white-noise
www.thebookstore.com/john-steinbeck/cannery-row
www.thebookstore.com/david-foster-wallace/oblivion-stories
www.thebookstore.com/david-foster-wallace/consider-the-lobster

Using a combination of CONCAT\_WS, LOWER, and REPLACE we can build something like a book URL.

# MySQL String functions – INSERT

```
SELECT INSERT('Quadratic', 3, 4, 'What');
```

```
+-----+
| INSERT('Quadratic', 3, 4, 'What') |
+-----+
| QuWhattic                          |
+-----+
```

INSERT a string into another string and takes four parameters.

- 1 – The string which we want to insert into
- 2 – The start position
- 3 – The number of characters we want to overwrite
- 4 – The string to be inserted

# MySQL String functions – LEFT & RIGHT

```
SELECT LEFT('foobarbar', 5);
```

```
+-----+
| LEFT('foobarbar', 5) |
+-----+
| fooba                |
+-----+
```

```
SELECT RIGHT('foobarbar', 5);
```

```
+-----+
| RIGHT('foobarbar', 5) |
+-----+
| arbar                 |
+-----+
```

Select the left most or right most characters where the number is the number of characters

# MySQL String functions – REPEAT

```
SELECT REPEAT('MySQL', 3);
```

```
+-----+
| REPEAT('MySQL', 3) |
+-----+
| MySQLMySQLMySQL    |
+-----+
```

Repeat will repeat a string without spaces a specified number of times.

# MySQL String functions – TRIM

```
SELECT TRIM('  bar ');
```

```
+-----+
| TRIM('  bar  ') |
+-----+
| bar              |
+-----+
```

Trim can be used to remove whitespaces or any leading or trailing characters.

```
SELECT TRIM(LEADING '.' FROM '...bar...');
```

```
+-----+
| TRIM(LEADING '.' FROM '...bar...') |
+-----+
| bar...                               |
+-----+
```

```
SELECT TRIM(TRAILING '.' FROM '...bar...');
```

```
+-----+
| TRIM(TRAILING '.' FROM '...bar...') |
+-----+
| ...bar                               |
+-----+
```

```
SELECT TRIM(BOTH '.' FROM '...bar...');
```

```
+-----+
| TRIM(BOTH '.' FROM '...bar...') |
+-----+
| bar                               |
+-----+
```



# MySQL String functions – EXERCISES

```
SELECT
    author_fname AS forward,
    REVERSE(author_fname) AS backwards
FROM books;
```

```
SELECT REPLACE(title, ' ', '->') AS title FROM books;
```

title
The->Namesake
Norse->Mythology
American->Gods
Interpreter->of->Maladies
A->Hologram->for->the->King:->A->Novel
The->Circle
The->Amazing->Adventures->of->Kavalier->&->Clay
Just->Kids
A->Heartbreaking->Work->of->Staggering->Genius
Coraline
What->We->Talk->About->When->We->Talk->About->Love:->Stories
Where->I'm->Calling->From:->Selected->Stories
White->Noise
Cannery->Row
Oblivion:->Stories
Consider->the->Lobster

forward	backwards
Jhumpa	apmuhJ
Neil	lieN
Neil	lieN
Jhumpa	apmuhJ
Dave	evaD
Dave	evaD
Michael	leahciM
Patti	ittatP
Dave	evaD
Neil	lieN
Raymond	dnomyaR
Raymond	dnomyaR
Don	noD
John	nhoJ
David	divaD
David	divaD

```
SELECT CONCAT(
    title,' was released in
    ',released_year
) AS 'blurb'
FROM books;
```

```
+-----+
| blurb                                     |
+-----+
| The Namesake was released in 2003        |
| Norse Mythology was released in 2016     |
| American Gods was released in 2001       |
| Interpreter of Maladies was released in 1996 |
| A Hologram for the King: A Novel was released in 2012 |
| The Circle was released in 2013          |
| The Amazing Adventures of Kavalier & Clay was released in 2000 |
| Just Kids was released in 2010           |
| A Heartbreaking Work of Staggering Genius was released in 2001 |
| Coraline was released in 2003            |
| What We Talk About When We Talk About Love: Stories was released in 1981 |
| Where I'm Calling From: Selected Stories was released in 1989 |
| White Noise was released in 1985         |
| Cannery Row was released in 1945         |
| Oblivion: Stories was released in 2004   |
| Consider the Lobster was released in 2005 |
+-----+
```

```
SELECT CONCAT(
    UPPER(author_fname),
    ' ',
    UPPER(author_lname))
AS 'full name in caps'
FROM books;
```

```
+-----+
| full name in caps                       |
+-----+
| JHUMPA LAHIRI                           |
| NEIL GAIMAN                             |
| NEIL GAIMAN                             |
| JHUMPA LAHIRI                           |
| DAVE EGGERS                             |
| DAVE EGGERS                             |
| MICHAEL CHABON                          |
| PATTI SMITH                             |
| DAVE EGGERS                             |
| NEIL GAIMAN                             |
| RAYMOND CARVER                          |
| RAYMOND CARVER                          |
| DON DELILLO                             |
| JOHN STEINBECK                          |
| DAVID FOSTER WALLACE                     |
| DAVID FOSTER WALLACE                     |
+-----+
```

```
SELECT
    title,
    CHAR_LENGTH(title)
    AS character_count
FROM books;
```

title	character_count
The Namesake	12
Norse Mythology	15
American Gods	13
Interpreter of Maladies	23
A Hologram for the King: A Novel	32
The Circle	10
The Amazing Adventures of Kavalier & Clay	41
Just Kids	9
A Heartbreaking Work of Staggering Genius	41
Coraline	8
What We Talk About When We Talk About Love: Stories	51
Where I'm Calling From: Selected Stories	40
White Noise	11
Cannery Row	11
Oblivion: Stories	17
Consider the Lobster	20

```

SELECT CONCAT(
    SUBSTR(title, 1, 10), '...') AS short_title,
    CONCAT(author_lname,',',author_fname) AS author,
    CONCAT(stock_quantity,' in stock') AS quantity
FROM books;

```

short_title	author	quantity
The Namesa...	Lahiri,Jhumpa	32 in stock
Norse Myth...	Gaiman,Neil	43 in stock
American G...	Gaiman,Neil	12 in stock
Interprete...	Lahiri,Jhumpa	97 in stock
A Hologram...	Eggers,Dave	154 in stock
The Circle...	Eggers,Dave	26 in stock
The Amazin...	Chabon,Michael	68 in stock
Just Kids...	Smith,Patti	55 in stock
A Heartbre...	Eggers,Dave	104 in stock
Coraline...	Gaiman,Neil	100 in stock
What We Ta...	Carver,Raymond	23 in stock
Where I'm ...	Carver,Raymond	12 in stock
White Nois...	DeLillo,Don	49 in stock
Cannery Ro...	Steinbeck,John	95 in stock
Oblivion: ...	Foster Wallace,David	172 in stock
Consider t...	Foster Wallace,David	92 in stock

# Refining Selections

# MySQL - Refining Select – Add new books to the table

```
INSERT INTO books
```

```
(title, author_fname, author_lname, released_year, stock_quantity, pages)
```

```
VALUES
```

```
('10% Happier', 'Dan', 'Harris', 2014, 29, 256),
```

```
('fake_book', 'Freida', 'Harris', 2001, 287, 428),
```

```
('Lincoln In The Bardo', 'George', 'Saunders', 2017, 1000, 367);
```

book_id	title	author_fname	author_lname	released_year	stock_quantity	pages
1	The Namesake	Jhumpa	Lahiri	2003	32	291
2	Norse Mythology	Neil	Gaiman	2016	43	304
3	American Gods	Neil	Gaiman	2001	12	465
4	Interpreter of Maladies	Jhumpa	Lahiri	1996	97	198
5	A Hologram for the King: A Novel	Dave	Eggers	2012	154	352
6	The Circle	Dave	Eggers	2013	26	504
7	The Amazing Adventures of Kavalier & Clay	Michael	Chabon	2000	68	634
8	Just Kids	Patti	Smith	2010	55	304
9	A Heartbreaking Work of Staggering Genius	Dave	Eggers	2001	104	437
10	Coraline	Neil	Gaiman	2003	100	208
11	What We Talk About When We Talk About Love: Stories	Raymond	Carver	1981	23	176
12	Where I'm Calling From: Selected Stories	Raymond	Carver	1989	12	526
13	White Noise	Don	DeLillo	1985	49	320
14	Cannery Row	John	Steinbeck	1945	95	181
15	Oblivion: Stories	David	Foster Wallace	2004	172	329
16	Consider the Lobster	David	Foster Wallace	2005	92	343
17	10% Happier	Dan	Harris	2014	29	256
18	fake_book	Freida	Harris	2001	287	428
19	Lincoln In The Bardo	George	Saunders	2017	1000	367

# MySQL – Refining Select – DISTINCT Select

```
SELECT DISTINCT author_lname FROM books;
```

The DISTINCT keyword before the column name will only show one occurrence of that column name in the table. I.e. We have authors that have more than one book or authors that have the same surname.

author
Jhumpa Lahiri
Neil Gaiman
Dave Eggers
Michael Chabon
Patti Smith
Raymond Carver
Don DeLillo
John Steinbeck
David Foster Wallace
Dan Harris
Freida Harris
George Saunders

author_lname
Lahiri
Gaiman
Eggers
Chabon
Smith
Carver
DeLillo
Steinbeck
Foster Wallace
Harris
Saunders

```
SELECT DISTINCT CONCAT(author_fname, ' ', author_lname)  
AS author FROM books;
```

But we are losing authors that have the same last name. If we want a list of one author per row we can use the DISTINCT keyword on the concatenation of first name and last name.

```
SELECT DISTINCT author_fname, author_lname  
FROM books;
```

The logic of this SQL is that author first name and author last name both must be DISTINCT.

author_fname	author_lname
Jhumpa	Lahiri
Neil	Gaiman
Dave	Eggers
Michael	Chabon
Patti	Smith
Raymond	Carver
Don	DeLillo
John	Steinbeck
David	Foster Wallace
Dan	Harris
Freida	Harris
George	Saunders

# MySQL – Refining Select – ORDER BY

```
SELECT book_id, author_fname, author_lname FROM books
ORDER BY author_fname;
```

book_id	author_fname	author_lname
17	Dan	Harris
5	Dave	Eggers
6	Dave	Eggers
9	Dave	Eggers
15	David	Foster Wallace
16	David	Foster Wallace
13	Don	DeLillo
18	Freida	Harris
19	George	Saunders
1	Jhumpa	Lahiri
4	Jhumpa	Lahiri
14	John	Steinbeck
7	Michael	Chabon
2	Neil	Gaiman
3	Neil	Gaiman
10	Neil	Gaiman
8	Patti	Smith
11	Raymond	Carver
12	Raymond	Carver

Order by will sort the rows in order by the column specified. It sorts in ascending order automatically i.e A to Z or numerically Zero to highest number.

We can also change the order to descending, i.e. highest value to lowest value.

```
SELECT book_id, author_fname, author_lname
FROM books ORDER BY author_fname DESC;
```

book_id	author_fname	author_lname
17	Dan	Harris
5	Dave	Eggers
6	Dave	Eggers
9	Dave	Eggers
15	David	Foster Wallace
16	David	Foster Wallace
13	Don	DeLillo
18	Freida	Harris
19	George	Saunders
1	Jhumpa	Lahiri
4	Jhumpa	Lahiri
14	John	Steinbeck
7	Michael	Chabon
2	Neil	Gaiman
3	Neil	Gaiman
10	Neil	Gaiman
8	Patti	Smith
11	Raymond	Carver
12	Raymond	Carver



```
SELECT title, pages FROM BOOKS  
ORDER BY released_year;
```

ORDER BY will also work if we want to order by a column that we are not selecting.

book_id	author_fname	author_lname	pages
17	Dan	Harris	256
5	Dave	Eggers	352
6	Dave	Eggers	504
9	Dave	Eggers	437
15	David	Foster Wallace	329
16	David	Foster Wallace	343
13	Don	DeLillo	320
18	Freida	Harris	428
19	George	Saunders	367
1	Jhumpa	Lahiri	291
4	Jhumpa	Lahiri	198
14	John	Steinbeck	181
7	Michael	Chabon	634
2	Neil	Gaiman	304
3	Neil	Gaiman	465
10	Neil	Gaiman	208
8	Patti	Smith	304
11	Raymond	Carver	176
12	Raymond	Carver	526

title	pages
Cannery Row	181
What We Talk About When We Talk About Love: Stories	176
White Noise	320
Where I'm Calling From: Selected Stories	526
Interpreter of Maladies	198
The Amazing Adventures of Kavalier & Clay	634
American Gods	465
A Heartbreaking Work of Staggering Genius	437
fake_book	428
The Namesake	291
Coraline	208
Oblivion: Stories	329
Consider the Lobster	343
Just Kids	304
A Hologram for the King: A Novel	352
The Circle	504
10% Happier	256
Norse Mythology	304
Lincoln In The Bardo	367

```
SELECT book_id, author_fname,  
author_lname, pages FROM books  
ORDER BY 2;
```

We can order by the first, second, third etc column we are selecting by using ORDER BY <column number>.

```
SELECT author_lname, released_year, title FROM books
ORDER BY author_lname;
```

author_lname	released_year	title
Carver	1981	What We Talk About When We Talk About Love: Stories
Carver	1989	Where I'm Calling From: Selected Stories
Chabon	2000	The Amazing Adventures of Kavalier & Clay
DeLillo	1985	White Noise
Eggers	2012	A Hologram for the King: A Novel
Eggers	2013	The Circle
Eggers	2001	A Heartbreaking Work of Staggering Genius
Foster Wallace	2004	Oblivion: Stories
Foster Wallace	2005	Consider the Lobster
Gaiman	2016	Norse Mythology
Gaiman	2001	American Gods
Gaiman	2003	Coraline
Harris	2014	10% Happier
Harris	2001	fake_book
Lahiri	2003	The Namesake
Lahiri	1996	Interpreter of Maladies
Saunders	2017	Lincoln In The Bardo
Smith	2010	Just Kids
Steinbeck	1945	Cannery Row

But what if we want an additional sort to order the books by release year for each author?

```
SELECT author_lname, released_year, title FROM books
ORDER BY author_lname, released_year;
```

author_lname	released_year	title
Carver	1981	What We Talk About When We Talk About Love: Stories
Carver	1989	Where I'm Calling From: Selected Stories
Chabon	2000	The Amazing Adventures of Kavalier & Clay
DeLillo	1985	White Noise
Eggers	2001	A Heartbreaking Work of Staggering Genius
Eggers	2012	A Hologram for the King: A Novel
Eggers	2013	The Circle
Foster Wallace	2004	Oblivion: Stories
Foster Wallace	2005	Consider the Lobster
Gaiman	2001	American Gods
Gaiman	2003	Coraline
Gaiman	2016	Norse Mythology
Harris	2001	fake_book
Harris	2014	10% Happier
Lahiri	1996	Interpreter of Maladies
Lahiri	2003	The Namesake
Saunders	2017	Lincoln In The Bardo
Smith	2010	Just Kids
Steinbeck	1945	Cannery Row

Now the first sort (ORDER BY) is performed on the author last name then a second sort is performed by the Released year. The syntax to do this is:  
`ORDER BY <first column sort>, <second column sort>`

```
SELECT author_lname, released_year, title FROM books
ORDER BY author_lname, released_year DESC;
```

author_lname	released_year	title
Carver	1989	Where I'm Calling From: Selected Stories
Carver	1981	What We Talk About When We Talk About Love: Stories
Chabon	2000	The Amazing Adventures of Kavalier & Clay
DeLillo	1985	White Noise
Eggers	2013	The Circle
Eggers	2012	A Hologram for the King: A Novel
Eggers	2001	A Heartbreaking Work of Staggering Genius
Foster Wallace	2005	Consider the Lobster
Foster Wallace	2004	Oblivion: Stories
Gaiman	2016	Norse Mythology
Gaiman	2003	Coraline
Gaiman	2001	American Gods
Harris	2014	10% Happier
Harris	2001	fake_book
Lahiri	2003	The Namesake
Lahiri	1996	Interpreter of Maladies
Saunders	2017	Lincoln In The Bardo
Smith	2010	Just Kids
Steinbeck	1945	Cannery Row

We can also change the direction of the sort, Now author last name is ascending i.e. A to Z. Release year is descending, i.e. most recent year first.

```
SELECT CONCAT(author_fname, ' ', author_lname) AS author,
released_year, title from books ORDER BY author, released_year DESC;
```

author	released_year	title
Dan Harris	2014	10% Happier
Dave Eggers	2013	The Circle
Dave Eggers	2012	A Hologram for the King: A Novel
Dave Eggers	2001	A Heartbreaking Work of Staggering Genius
David Foster Wallace	2005	Consider the Lobster
David Foster Wallace	2004	Oblivion: Stories
Don DeLillo	1985	White Noise
Freida Harris	2001	fake_book
George Saunders	2017	Lincoln In The Bardo
Jhumpa Lahiri	2003	The Namesake
Jhumpa Lahiri	1996	Interpreter of Maladies
John Steinbeck	1945	Cannery Row
Michael Chabon	2000	The Amazing Adventures of Kavalier & Clay
Neil Gaiman	2016	Norse Mythology
Neil Gaiman	2003	Coraline
Neil Gaiman	2001	American Gods
Patti Smith	2010	Just Kids
Raymond Carver	1989	Where I'm Calling From: Selected Stories
Raymond Carver	1981	What We Talk About When We Talk About Love: Stories

We can also order by an alias.

# MySQL – Refining Select – LIMIT

```
SELECT book_id, title, released_year from BOOKS LIMIT 5;
```

book_id	title	released_year
1	The Namesake	2003
2	Norse Mythology	2016
3	American Gods	2001
4	Interpreter of Maladies	1996
5	A Hologram for the King: A Novel	2012

LIMIT will limit the number of rows returned by the value specified. In this case we are not ORDERing the rows so it will limit to the first five book\_id's

```
SELECT book_id, title, released_year from BOOKS  
ORDER BY released_year LIMIT 5;
```

book_id	title	released_year
14	Cannery Row	1945
11	What We Talk About When We Talk About Love: Stories	1981
13	White Noise	1985
12	Where I'm Calling From: Selected Stories	1989
4	Interpreter of Maladies	1996

Now we are first ORDERing by release\_year then LIMITing the rows to the first five.

```
SELECT book_id, title, released_year from BOOKS  
ORDER BY released_year DESC LIMIT 5;
```

book_id	title	released_year
19	Lincoln In The Bardo	2017
2	Norse Mythology	2016
17	10% Happier	2014
6	The Circle	2013
5	A Hologram for the King: A Novel	2012

I can also get the five most recently released books by ORDERing by release\_year DESCending then LIMITing to 5.

```
SELECT book_id, title, released_year from BOOKS  
ORDER BY released_year DESC LIMIT 1,5;
```

book_id	title	released_year
2	Norse Mythology	2016
17	10% Happier	2014
6	The Circle	2013
5	A Hologram for the King: A Novel	2012
8	Just Kids	2010

I can select where to start and end the limit, i.e. start at row 1 and go for 5 rows.

This is usefull for pagination i.e. show rows 1 to 25 on the first page, rows 26 to 50 on the second page, 51 to 75 on the third etc..

# MySQL – LIKE and Wildcards

```
SELECT title, author_fname, author_lname FROM books
WHERE author_fname='David';
```

title	author_fname	author_lname
Oblivion: Stories	David	Foster Wallace
Consider the Lobster	David	Foster Wallace

We want to find all books by author whos first name is David or Dave. We could run two queries for each.

```
SELECT title, author_fname, author_lname FROM books
WHERE author_fname LIKE '%Dav%';
```

title	author_fname	author_lname
A Hologram for the King: A Novel	Dave	Eggers
The Circle	Dave	Eggers
A Heartbreaking Work of Staggering Genius	Dave	Eggers
Oblivion: Stories	David	Foster Wallace
Consider the Lobster	David	Foster Wallace

The LIKE keyword means find all rows where author's first name contains the characters 'Dav' and the % means any characters. So %Dav% means any characters then Dav then any characters. The % is called a wildcard.



```
SELECT title, author_fname, author_lname FROM books
WHERE author_fname LIKE '%Da%';
```

title	author_fname	author_lname
A Hologram for the King: A Novel	Dave	Eggers
The Circle	Dave	Eggers
A Heartbreaking Work of Staggering Genius	Dave	Eggers
Oblivion: Stories	David	Foster Wallace
Consider the Lobster	David	Foster Wallace
10% Happier	Dan	Harris
fake_book	Freida	Harris

Note that %Da% will find all rows with a da somewhere in author\_fname and is not case specific.

```
SELECT title, author_fname, author_lname FROM books
WHERE author_fname LIKE 'da%';
```

Removing the wildcard before da and only having it after means that it will find rows starting with da and any characters after.

title	author_fname	author_lname
A Hologram for the King: A Novel	Dave	Eggers
The Circle	Dave	Eggers
A Heartbreaking Work of Staggering Genius	Dave	Eggers
Oblivion: Stories	David	Foster Wallace
Consider the Lobster	David	Foster Wallace
10% Happier	Dan	Harris

```
SELECT title, author_fname, author_lname
FROM books WHERE author_fname LIKE '%da';
```

title	author_fname	author_lname
fake_book	Freida	Harris

A wildcard at the beginning only will find author\_lname rows with a da at the end.

```
SELECT title, author_fname, author_lname FROM books
WHERE title LIKE ':%:';
```

title	author_fname	author_lname
A Hologram for the King: A Novel	Dave	Eggers
What We Talk About When We Talk About Love: Stories	Raymond	Carver
Where I'm Calling From: Selected Stories	Raymond	Carver
Oblivion: Stories	David	Foster Wallace

We can find all books that have a colon in the title.

```
SELECT title, author_fname, author_lname FROM books
WHERE author_fname LIKE '____';
```

title	author_fname	author_lname
Norse Mythology	Neil	Gaiman
American Gods	Neil	Gaiman
A Hologram for the King: A Novel	Dave	Eggers
The Circle	Dave	Eggers
A Heartbreaking Work of Staggering Genius	Dave	Eggers
Coraline	Neil	Gaiman
Cannery Row	John	Steinbeck

An underscore means exactly one character so in this example we are using four underscores to find author's whose first name is exactly four characters long.

```
SELECT title, author_fname, author_lname FROM books
WHERE title LIKE '%\%%';
```

title	author_fname	author_lname
10% Happier	Dan	Harris

What if we want to search for something that has a percent sign in the string? We can use an escape character.

```
SELECT title, author_fname, author_lname FROM books
WHERE title LIKE '%\_%';
```

title	author_fname	author_lname
fake_book	Freida	Harris

What if we want to search for something that has an underscore sign in the string? We can use an escape character.

# MySQL – Refining Select – Exercises

```
SELECT title, author_fname, author_lname FROM books
WHERE title LIKE '%stories%';
```

title	author_fname	author_lname
What We Talk About When We Talk About Love: Stories	Raymond	Carver
Where I'm Calling From: Selected Stories	Raymond	Carver
Oblivion: Stories	David	Foster Wallace

Select all books that contain stories.

```
SELECT title, pages FROM books ORDER BY pages DESC LIMIT 1;
```

title	pages
The Amazing Adventures of Kavalier & Clay	634

Find the longest book

```
SELECT CONCAT(SUBSTR(title, 1, 20), ' - ', released_year) AS summary
FROM books ORDER BY released_year DESC LIMIT 3;
```

summary
Lincoln In The Bardo - 2017
Norse Mythology - 2016
10% Happier - 2014

Print a summary of the three most recently released books

```
SELECT CONCAT(SUBSTR(title, 1, 20), ' - ',released_year) AS summary
FROM books ORDER BY released_year DESC LIMIT 3;
```

summary
Lincoln In The Bardo - 2017
Norse Mythology - 2016
10% Happier - 2014

Print a summary of the three most recently released books

```
SELECT title, author_lname FROM books WHERE author_lname LIKE '% %';
```

title	author_lname
Oblivion: Stories	Foster Wallace
Consider the Lobster	Foster Wallace

Find all books where the author's last name contains a space

```
SELECT title, author_lname FROM books WHERE author_lname LIKE '% %';
```

title	author_lname
Oblivion: Stories	Foster Wallace
Consider the Lobster	Foster Wallace

Find all books where the author's last name contains a space

```
SELECT title, released_year, stock_quantity FROM books ORDER BY
stock_quantity LIMIT 3;
```

title	released_year	stock_quantity
Where I'm Calling From: Selected Stories	1989	12
American Gods	2001	12
What We Talk About When We Talk About Love: Stories	1981	23

Find the three books with the lowest stock quantity.

```
SELECT title, author_lname FROM books ORDER BY author_lname, title;
```

title	author_lname
What We Talk About When We Talk About Love: Stories	Carver
Where I'm Calling From: Selected Stories	Carver
The Amazing Adventures of Kavalier & Clay	Chabon
White Noise	DeLillo
A Heartbreaking Work of Staggering Genius	Eggers
A Hologram for the King: A Novel	Eggers
The Circle	Eggers
Consider the Lobster	Foster Wallace
Oblivion: Stories	Foster Wallace
American Gods	Gaiman
Coraline	Gaiman
Norse Mythology	Gaiman
10% Happier	Harris
fake_book	Harris
Interpreter of Maladies	Lahiri
The Namesake	Lahiri
Lincoln In The Bardo	Saunders
Just Kids	Smith
Cannery Row	Steinbeck

Sort the books by author last name then by book title

```
SELECT UPPER(CONCAT('my favorite author is ',author_fname,'
',author_lname,'!')) AS yell FROM books ORDER BY author_lname;
```

```
+-----+
| yell                                     |
+-----+
| MY FAVORITE AUTHOR IS RAYMOND CARVER!   |
| MY FAVORITE AUTHOR IS RAYMOND CARVER!   |
| MY FAVORITE AUTHOR IS MICHAEL CHABON!   |
| MY FAVORITE AUTHOR IS DON DELILLO!     |
| MY FAVORITE AUTHOR IS DAVE EGGERS!     |
| MY FAVORITE AUTHOR IS DAVE EGGERS!     |
| MY FAVORITE AUTHOR IS DAVE EGGERS!     |
| MY FAVORITE AUTHOR IS DAVID FOSTER WALLACE! |
| MY FAVORITE AUTHOR IS DAVID FOSTER WALLACE! |
| MY FAVORITE AUTHOR IS NEIL GAIMAN!      |
| MY FAVORITE AUTHOR IS NEIL GAIMAN!      |
| MY FAVORITE AUTHOR IS NEIL GAIMAN!      |
| MY FAVORITE AUTHOR IS DAN HARRIS!       |
| MY FAVORITE AUTHOR IS FREIDA HARRIS!    |
| MY FAVORITE AUTHOR IS JHUMPA LAHIRI!    |
| MY FAVORITE AUTHOR IS JHUMPA LAHIRI!    |
| MY FAVORITE AUTHOR IS GEORGE SAUNDERS!  |
| MY FAVORITE AUTHOR IS PATTI SMITH!      |
| MY FAVORITE AUTHOR IS JOHN STEINBECK!   |
+-----+
```

Print out the following phrase for each book author and sort by last name.

# Aggregate Functions



# MySQL – Aggregate Functions – Count()

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

COUNT(*)
19

This query will count the number of rows in our table.

```
SELECT COUNT(author_fname) FROM books;
```

COUNT(author_fname)
19

This query will count everytime an author\_fname is present in that column.

```
SELECT COUNT(DISTINCT author_fname) FROM books;
```

This query will count distinct author\_fnames i.e. count once each time a unique value is present.

COUNT(DISTINCT author_fname)
12

```
SELECT COUNT(DISTINCT released_year) FROM books;
```

This query will count distinct released\_year i.e. count once each time a unique value is present.

COUNT(DISTINCT released_year)
16

```
SELECT COUNT(author_lname) FROM books;
```

COUNT(author_lname)
11

There are 11 distinct author last names in the table

```
SELECT title FROM books WHERE title LIKE '%the%';
```

title
The Namesake
A Hologram for the King: A Novel
The Circle
The Amazing Adventures of Kavalier & Clay
Consider the Lobster
Lincoln In The Bardo

We want to know the number of books that have 'the' in the title.

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

COUNT(*)
6

We need to count all rows from books where title is like 'the'.

# MySQL – Aggregate Functions – group by

“Group by aggregates or summarises identical data into single rows.”

```
SELECT author_lname FROM books GROUP BY author_lname;
```

If we group by author\_lname then we get one row per author.

title	author_lname
What We Talk About When We Talk About Love: Stories	Carver
Where I'm Calling From: Selected Stories	Carver
The Amazing Adventures of Kavalier & Clay	Chabon
White Noise	DeLillo
A Hologram for the King: A Novel	Eggers
The Circle	Eggers
A Heartbreaking Work of Staggering Genius	Eggers
Oblivion: Stories	Foster Wallace
Consider the Lobster	Foster Wallace
Norse Mythology	Gaiman
American Gods	Gaiman
Coraline	Gaiman
10% Happier	Harris
fake_book	Harris
The Namesake	Lahiri
Interpreter of Maladies	Lahiri
Lincoln In The Bardo	Saunders
Just Kids	Smith
Cannery Row	Steinbeck

author_lname
Lahiri
Gaiman
Eggers
Chabon
Smith
Carver
DeLillo
Steinbeck
Foster Wallace
Harris
Saunders

What is actually happening behind the scenes is that rows with the same author\_lname are being grouped.

```
SELECT author_lname, COUNT(*) FROM books GROUP BY author_lname;
```

author_lname	COUNT(*)
Lahiri	2
Gaiman	3
Eggers	3
Chabon	1
Smith	1
Carver	2
DeLillo	1
Steinbeck	1
Foster Wallace	2
Harris	2
Saunders	1

We can count the number of books (or rows) for each author\_lname.

```
SELECT author_lname, COUNT(*) AS books_written  
FROM books GROUP BY author_lname ORDER BY  
books_written DESC;
```

We can use an alias to rename the count to books written and order by this.

author_lname	books_written
Gaiman	3
Eggers	3
Lahiri	2
Carver	2
Foster Wallace	2
Harris	2
Chabon	1
Smith	1
DeLillo	1
Steinbeck	1
Saunders	1

# MySQL – Aggregate Functions – Min & Max

```
SELECT MIN(released_year) FROM books;
```

MIN(released_year)
1945

Min & Max can be used to find the lowest or highest values in a column.

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

MAX(pages)
634

Find the book with the largest number of pages

```
SELECT MIN(author_lname) FROM books;
```

MIN(author_lname)
Carver

IT does work alphabetically, finding the lowest letter in the column.

```
SELECT MIN(author_lname), MAX(author_lname) FROM books;
```

MIN(author_lname)	MAX(author_lname)
Carver	Steinbeck

It can be used to select multiple columns.

# MySQL – Aggregate Functions – subqueries

```
SELECT title FROM books WHERE pages = (SELECT MAX(pages) FROM books);
```

```
+-----+
| title |
+-----+
| The Amazing Adventures of Kavalier & Clay |
+-----+
```

I want to select the title of the book with the maximum number of pages. To do this I first select titles from books then I need the where clause to match it to a **sub query in parenthesis**, In this case select max pages from books. The sub query will run first.

```
INSERT INTO books (title, pages) VALUES ('my life in words', 634);
```

```
SELECT title FROM books WHERE pages = (SELECT MAX(pages) FROM books);
```

```
+-----+
| title |
+-----+
| The Amazing Adventures of Kavalier & Clay |
| my life in words |
+-----+
```

If I insert another book with the same number of pages and use the above query it will now give two book titles with the same maximum number of pages.

```
SELECT title, pages FROM books ORDER BY pages DESC LIMIT 1;
```

```
+-----+-----+
| title | pages |
+-----+-----+
| The Amazing Adventures of Kavalier & Clay | 634 |
+-----+-----+
```

This is better than using limit 1 because it would only find one book even though there are two with the same maximum number of pages.

```
SELECT released_year, title FROM books WHERE released_year =  
(SELECT MIN(released_year) FROM books);
```

released_year	title
1945	Cannery Row

Another example of a sub query would be to find title and released year of the book with the lowest released year.

# MySQL – Aggregate Functions – Grouping by multiple columns

```
SELECT author_fname, author_lname FROM books ORDER BY author_lname;
```

author_fname	author_lname
Raymond	Carver
Raymond	Carver
Michael	Chabon
Don	DeLillo
Dave	Eggers
Dave	Eggers
Dave	Eggers
David	Foster Wallace
David	Foster Wallace
Neil	Gaiman
Neil	Gaiman
Neil	Gaiman
Dan	Harris
Freida	Harris
Jhumpa	Lahiri
Jhumpa	Lahiri
George	Saunders
Patti	Smith
John	Steinbeck

There are two authors with the same last name so we would get an innacurate count.

```
SELECT author_lname, COUNT(*) from books  
GROUP BY author_lname;
```

We can see that harris has two rows when grouping by last name. one row for Dan Harris and One row for Freida Harris.

author_lname	COUNT(*)
Lahiri	2
Gaiman	3
Eggers	3
Chabon	1
Smith	1
Carver	2
DeLillo	1
Steinbeck	1
Foster Wallace	2
Harris	2
Saunders	1



```
SELECT author_fname, author_lname, COUNT(*) from books
GROUP BY author_lname, author_fname;
```

author_fname	author_lname	COUNT(*)
Jhumpa	Lahiri	2
Neil	Gaiman	3
Dave	Eggers	3
Michael	Chabon	1
Patti	Smith	1
Raymond	Carver	2
Don	DeLillo	1
John	Steinbeck	1
David	Foster Wallace	2
Dan	Harris	1
Freida	Harris	1
George	Saunders	1

If the rows are grouped by author\_fname and author\_lname then Harris shows up twice with one book for each author.

```
SELECT CONCAT(author_fname, ' ', author_lname)
AS author, COUNT(*) FROM books GROUP BY
author;
```

We could also concatenate first name and last name as an alias then count the grouping of this alias.

author	COUNT(*)
Jhumpa Lahiri	2
Neil Gaiman	3
Dave Eggers	3
Michael Chabon	1
Patti Smith	1
Raymond Carver	2
Don DeLillo	1
John Steinbeck	1
David Foster Wallace	2
Dan Harris	1
Freida Harris	1
George Saunders	1

# MySQL – Aggregate Functions – Min & Max with GROUP BY

```
SELECT author_fname, author_lname, MIN(released_year)
FROM books GROUP BY author_fname, author_lname;
```

author_fname	author_lname	MIN(released_year)
Jhumpa	Lahiri	1996
Neil	Gaiman	2001
Dave	Eggers	2001
Michael	Chabon	2000
Patti	Smith	2010
Raymond	Carver	1981
Don	DeLillo	1985
John	Steinbeck	1945
David	Foster Wallace	2004
Dan	Harris	2014
Freida	Harris	2001
George	Saunders	2017

Find the year each author published their first book.

```

SELECT
    CONCAT(author_fname, ' ', author_lname) AS author,
    COUNT(*) AS books_written,
    MIN(released_year) AS first_book,
    MAX(released_year) AS last_book,
    MAX(pages) AS logest_pages_count FROM books
FROM books GROUP BY author ORDER BY books_written;

```

author	books_written	first_book	last_book	logest_pages_count
Michael Chabon	1	2000	2000	634
Patti Smith	1	2010	2010	304
Don DeLillo	1	1985	1985	320
John Steinbeck	1	1945	1945	181
Dan Harris	1	2014	2014	256
Freida Harris	1	2001	2001	428
George Saunders	1	2017	2017	367
Jhumpa Lahiri	2	1996	2003	291
Raymond Carver	2	1981	1989	526
David Foster Wallace	2	2004	2005	343
Neil Gaiman	3	2001	2016	465
Dave Eggers	3	2001	2013	504

Using these techniques we can build sophisticated queries like an author summary table.

# MySQL – Aggregate Functions – SUM

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

SUM(pages)
6623

Sum will add up all the values of a column for example.

```
SELECT CONCAT(author_fname, ' ', author_lname) AS  
author, SUM(pages) FROM BOOKS GROUP BY author;
```

We can combine sum with group by to find the total number of pages written per author.

author	SUM(pages)
Jhumpa Lahiri	489
Neil Gaiman	977
Dave Eggers	1293
Michael Chabon	634
Patti Smith	304
Raymond Carver	702
Don DeLillo	320
John Steinbeck	181
David Foster Wallace	672
Dan Harris	256
Freida Harris	428
George Saunders	367

```
SELECT SUM(author_lname) FROM books;
```

SUM(author_lname)
0

Sum only works on numeric data types.

# MySQL – Aggregate Functions – AVG (average)

```
SELECT AVG(released_year) FROM books;
```

```
+-----+
| AVG(pages) |
+-----+
| 348.5789 |
+-----+
```

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

```
+-----+
| AVG(released_year) |
+-----+
| 1999.7895 |
+-----+
```

AVG will find the average value

```
SELECT AVG(stock_quantity) FROM books;
```

```
+-----+
| AVG(stock_quantity) |
+-----+
| 128.9474 |
+-----+
```

```
+-----+-----+
| released_year | AVG(stock_quantity) |
+-----+-----+
| 2003 | 66.0000 |
| 2016 | 43.0000 |
| 2001 | 134.3333 |
| 1996 | 97.0000 |
| 2012 | 154.0000 |
| 2013 | 26.0000 |
| 2000 | 68.0000 |
| 2010 | 55.0000 |
| 1981 | 23.0000 |
| 1989 | 12.0000 |
| 1985 | 49.0000 |
| 1945 | 95.0000 |
| 2004 | 172.0000 |
| 2005 | 92.0000 |
| 2014 | 29.0000 |
| 2017 | 1000.0000 |
+-----+-----+
```

```
SELECT AVG(stock_quantity) FROM books
GROUP BY released_year;
```

Calculate the average stock quantity for books released in the same year

```

SELECT
    released_year AS year,
    AVG(stock_quantity),
    count(*) AS books_released_in_year
FROM books GROUP BY released_year ORDER BY year;

```

year	AVG(stock_quantity)	books_released_in_year
1945	95.0000	1
1981	23.0000	1
1985	49.0000	1
1989	12.0000	1
1996	97.0000	1
2000	68.0000	1
2001	134.3333	3
2003	66.0000	2
2004	172.0000	1
2005	92.0000	1
2010	55.0000	1
2012	154.0000	1
2013	26.0000	1
2014	29.0000	1
2016	43.0000	1
2017	1000.0000	1

Here we have a table that shows the average stock quantity for released year ordered by year.

# MySQL – Aggregate Functions – EXERCISES

```
SELECT SUM(stock_quantity) AS total_number_of_books from books;
```

total_number_of_books
2450

Find the total number of books in the book\_shop

```
SELECT released_year AS year, COUNT(*) AS  
books_released FROM books GROUP BY year ORDER BY year;
```

Find number of books released per each year.

author	avg_rel_year
John Steinbeck	1945.0000
Raymond Carver	1985.0000
Don DeLillo	1985.0000
Jhumpa Lahiri	1999.5000
Michael Chabon	2000.0000
Freida Harris	2001.0000
David Foster Wallace	2004.5000
Neil Gaiman	2006.6667
Dave Eggers	2008.6667
Patti Smith	2010.0000
Dan Harris	2014.0000
George Saunders	2017.0000

Find average release  
year per author

```
SELECT  
    CONCAT(author_fname, ' ', author_lname) as author,  
    AVG(released_year) AS avg_rel_year  
FROM books GROUP BY author ORDER BY avg_rel_year;
```

year	books_released
1945	1
1981	1
1985	1
1989	1
1996	1
2000	1
2001	3
2003	2
2004	1
2005	1
2010	1
2012	1
2013	1
2014	1
2016	1
2017	1

```
SELECT CONCAT(author_fname, ' ', author_lname) AS author, pages FROM books WHERE
pages = (SELECT MAX(pages) FROM books);
```

author	pages
Michael Chabon	634

Find the author that has written longest book.

```
SELECT
    released_year AS year,
    COUNT(*) AS '# books',
    avg(pages) AS avg_pages
FROM books GROUP BY released_year ORDER BY year;
```

Build table of year, number of books released and average pages per book in that year.

year	# books	avg_pages
1945	1	181.0000
1981	1	176.0000
1985	1	320.0000
1989	1	526.0000
1996	1	198.0000
2000	1	634.0000
2001	3	443.3333
2003	2	249.5000
2004	1	329.0000
2005	1	343.0000
2010	1	304.0000
2012	1	352.0000
2013	1	504.0000
2014	1	256.0000
2016	1	304.0000
2017	1	367.0000



# Revisiting Data Types

# MySQL – Data Types – CHAR & VARCHAR

```
DESC friends;
```

Field	Type	Null	Key	Default	Extra
name	varchar(10)	YES		NULL	

A table called friends with one column that is VARCHAR(10) i.e. variable length characters upto a maximum of 10.

```
SELECT * FROM friends;
```

name
Tom
Juan Pablo
James

The friends table has 3 rows with each row containing varchar of 10 or less characters.

```
DESC states;
```

Field	Type	Null	Key	Default	Extra
abbr	char(2)	YES		NULL	

A table called states that has one column that stores a fixed length string of two characters. CHAR(2).

```
SELECT * FROM states;
```

abbr
CO
CA
NY

The states table has 3 rows with each row containing CHAR of 2.

**CHAR** is used to store character strings where the **length is fixed**, from **zero to 255 characters**. I.e. abbreviation codes for states, IATA airport codes, country codes, car license plates, postal codes, Y & N flags etc.

**VARCHAR** is used for **variable length** data such as names, addresses, favorite food etc...

```
SELECT * FROM states;
```

abbr
CO
CA
NY
X

When a value is added to a CHAR column that is under the specified length, MySQL will store it right-padding of whitespace to fill the length to the value defined. When the Data is retrieved MySQL will remove the whitespace. This happens behind the scenes and is transparent to the user.

In this example the CHAR length is 2 so MySQL is adding one whitespace padding to the right of the value X to make it upto 2 characters. CHAR rows will occupy the same amount of memory because they are fixed length.

As each character uses 1 byte (assuming ASCII) then using CHAR is more efficient because the columns will always be of fixed byte size. So if data is plus or minus 1 character always the same length then it makes a lot more sense to use CHAR and not VARCHAR.

# MySQL – Data Types – INT, TINYINT, BIGINT

Type	Storage (Bytes)	Minimum Value Signed	Minimum Value Unsigned	Maximum Value Signed	Maximum Value Unsigned
TINYINT	1	-128	0	127	255
SMALLINT	2	-32768	0	32767	65535
MEDIUMINT	3	-8388608	0	8388607	16777215
INT	4	-2147483648	0	2147483647	4294967295
BIGINT	8	$-2^{63}$	0	$2^{63}-1$	$2^{64}-1$

The type of integer (whole number) determines the amount of space needed in bytes to store data.

For example, to optimise the database, a value such as the number of children that a parent has would be tiny INT. Another example would be the number of cars, computers, TVs etc in a household.

Signed means that we permit a negative value because the integer could have a minus sign in front of it. It is good practice to specify if an integer column would be only positive numbers, i.e. unsigned because this permits a larger number using the bytes available.

```
CREATE TABLE parent (children TINYINT UNSIGNED);
```

```
DESC parent;
```

```
+-----+-----+-----+-----+-----+-----+
| Field | Type           | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| children | tinyint unsigned | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
```

# MySQL – Data Types – DECIMAL

Total number of digits

**DECIMAL (5,2)**

How many of those numbers come after the decimal

This is a maximum size, we can have upto three numbers before the decimal, or 2 or 1.

So the permitted range for decimal (5,2) would be 0.00 to 999.99.

We could not store 1000.99.

Decimal is good for storing currency values.

**DESC products;**

price
5.25
999.99
0.00

**SELECT \* FROM products;**

Field	Type	Null	Key	Default	Extra
price	decimal(5,2)	YES		NULL	

A table called products with one column that contains price as a DECIMAL(5,2)

Another Example of data that works well as a decimal is coordinates of GPS location. Degrees, minutes and seconds can be converted into decimal format and stored as DECIMAL(9,6), i.e. 6 digits after the decimal and upto three digits before the decimal.

**41°38'55.0"N 2°02'41.0"E => 41.648606, 2.044734**

```
INSERT INTO products (price) VALUES (1000.2);
ERROR 1264 (22003): Out of range value for column 'price' at row 1
```

1000.2 although 5 digits long does not follow the format of maximum three digits and two decimals

```
INSERT INTO products (price) VALUES (99.022);
Query OK, 1 row affected, 1 warning (0.00 sec)
```

**99.022** although 5 digits long does not follow the format of maximum three digits and two decimals and yet it has entered the value without throwing an error.

MySQL will automatically truncate the decimal part of it exceeds the digit length by rounding up or down. It will enter the data less precise.

```
SHOW WARNINGS;
```

Level	Code	Message
Note	1265	Data truncated for column 'price' at row 1

```
SELECT * FROM products;
```

price
5.25
999.99
0.00
99.02



# MySQL – Data Types – FLOAT & DOUBLE

Decimal is a very precise way to store a number. Float and Double are less precise but take up less bytes in the SQL table. A float will take up 4 bytes and a double 8 bytes. Float starts to get imprecise around 7 digits and Double around 15 digits.

**DESC numbers;**

Field	Type	Null	Key	Default	Extra
float_col	float	YES		NULL	
double_col	double	YES		NULL	

A table called numbers with a column of FLOAT data type and a column of DOUBLE data type.

```
INSERT INTO numbers (float_col, double_col) VALUES (1.123, 1.123);
INSERT INTO numbers (float_col, double_col) VALUES (1.1234567, 1.1234567);
INSERT INTO numbers (float_col, double_col) VALUES (1.12345678, 1.12345678);
```

**SELECT \* FROM numbers;**

float_col	double_col
1.123	1.123
1.12346	1.1234567
1.12346	1.12345678

At 7 digits after the decimal the float has lost precision and rounded.



```

INSERT INTO numbers (float_col, double_col) VALUES (1.1234567890123, 1.1234567890123);
INSERT INTO numbers (float_col, double_col) VALUES (1.123456789012345, 1.123456789012345);
INSERT INTO numbers (float_col, double_col) VALUES (1.1234567890123456, 1.1234567890123456);
INSERT INTO numbers (float_col, double_col) VALUES (1.12345678901234567, 1.12345678901234567);
INSERT INTO numbers (float_col, double_col) VALUES (1.123456789012345678, 1.123456789012345678);
INSERT INTO numbers (float_col, double_col) VALUES (1.1234567890123456789, 1.1234567890123456789);

```

```
SELECT * FROM numbers;
```

float_col	double_col
1.123	1.123
1.12346	1.1234567
1.12346	1.12345678
1.12346	1.1234567890123
1.12346	1.123456789012345
1.12346	1.1234567890123457
1.12346	1.1234567890123457
1.12346	1.1234567890123457
1.12346	1.1234567890123457

At 15 digits after the decimal the double has lost precision and rounded.

The data type needed for non integer numbers depends on the level of precision required versus the speed of any calculations and size of storage.



# MySQL – Data Types – DATES & TIMES

## MySQL DATE

**YYYY-MM-DD**

MySQL has a specific way of storing dates. It is 4 digits for year then dash then 2 digits for month then a dash and 2 digits for day.

## MySQL TIME

**HH:MM:SS**

Time also has a very specific format also. It can be a time of the day such as 13:12:17 or an interval of time.

## MySQL DATE TIME

**YYYY-MM-DD HH:MM:SS**

DATE TIME also has a very specific format and can be used to store the date and time that a user signed up or the data and time for a calendar appointment etc.

```
CREATE TABLE people (
    name VARCHAR(100),
    birth_year DATE,
    birth_Time TIME,
    birth_date_time DATETIME
);
```

Field	Type	Null	Key	Default	Extra
name	varchar(100)	YES		NULL	
birth_year	date	YES		NULL	
birth_Time	time	YES		NULL	
birth_date_time	datetime	YES		NULL	

```
INSERT INTO people (
    name, birth_year, birth_Time, birth_date_time
) VALUES (
    'Elton', '2000-12-25', '11:00:00', '2000-12-25 11:00:00'
);
```

```
SELECT * FROM PEOPLE;
```

name	birth_year	birth_Time	birth_date_time
Elton	2000-12-25	11:00:00	2000-12-25 11:00:00

```
INSERT INTO people (name, birth_year, birth_Time, birth_date_time) VALUES ('Lulu', '1985-04-11', '09:00:10', '1985-04-11 09:00:10');
INSERT INTO people (name, birth_year, birth_Time, birth_date_time) VALUES ('Juan', '2020-08-15', '23:59:00', '2020-08-15 23:59:00');
```

```
SELECT * FROM PEOPLE;
```

name	birth_year	birth_Time	birth_date_time
Elton	2000-12-25	11:00:00	2000-12-25 11:00:00
Lulu	1985-04-11	09:00:10	1985-04-11 09:00:10
Juan	2020-08-15	23:59:00	2020-08-15 23:59:00

# MySQL – Data Types – CURDATE & CURTIME, NOW()

```
SELECT CURRENT_TIME();
```

```
SELECT CURTIME();
```

```
+-----+
| CURTIME() |
+-----+
| 18:16:32 |
+-----+
```

```
SELECT CURRENT_DATE();
```

```
SELECT CURDATE();
```

```
+-----+
| CURDATE() |
+-----+
| 2022-11-08 |
+-----+
```

```
SELECT CURRENT_TIMESTAMP();
```

```
SELECT NOW();
```

```
+-----+
| NOW() |
+-----+
| 2022-11-08 18:18:11 |
+-----+
```

These are inbuilt MySQL functions to calculate the current Date, current time and current date time. CURRENT\_ can be shortened to CUR. NOW() is short for CURRENT\_TIMESTAMP();

```
INSERT INTO people (
    name, birth_year, birth_Time, birth_date_time
) VALUES (
    'Hazel', CURDATE(), CURTIME(), NOW()
);
```

```
SELECT * FROM PEOPLE;
```

```
+-----+-----+-----+-----+
| name | birth_year | birth_Time | birth_date_time |
+-----+-----+-----+-----+
| Elton | 2000-12-25 | 11:00:00 | 2000-12-25 11:00:00 |
| Lulu | 1985-04-11 | 09:00:10 | 1985-04-11 09:00:10 |
| Juan | 2020-08-15 | 23:59:00 | 2020-08-15 23:59:00 |
| Hazel | 2022-11-08 | 18:26:05 | 2022-11-08 18:26:05 |
+-----+-----+-----+-----+
```

The current functions  
are used to auto fill  
dates and times.



# MySQL – Data Types – DateTime functions

```
SELECT birth_year, DAY(birth_year) FROM people;
```

birth_year	DAY(birth_year)
2000-12-25	25
1985-04-11	11
2020-08-15	15
2022-11-08	8

DAY function will extract the day as a value from 1 to 31 from DATE or DATETIME

```
SELECT birth_year, DAYOFWEEK(birth_year) FROM people;
```

birth_year	DAYOFWEEK(birth_year)
2000-12-25	2
1985-04-11	5
2020-08-15	7
2022-11-08	3

DAYOFWEEK function will return the day as a value from 1 to 7 from DATE or DATETIME where

1 => Sunday,  
2 => Monday,  
3 => Tuesday ....  
7 => Saturday.

```
SELECT birth_year, DAYOFYEAR(birth_year) FROM people;
```

birth_year	DAYOFYEAR(birth_year)
2000-12-25	360
1985-04-11	101
2020-08-15	228
2022-11-08	312

DAYOFYEAR function will extract the day as a value from 1 to 266 representing the day of the year where 1 is the 1<sup>st</sup> of January, 2 is the 2<sup>nd</sup> of January etc.

```
SELECT birth_year, MONTHNAME(birth_year) FROM people;
```

birth_year	MONTHNAME(birth_year)
2000-12-25	December
1985-04-11	April
2020-08-15	August
2022-11-08	November

MONTHNAME function will extract the name of the month as a string from DATE or DATETIME

```
SELECT birth_time, MONTHNAME(birth_time) FROM people;
```

birth_time	MONTHNAME(birth_time)
11:00:00	November
09:00:10	November
23:59:00	November
18:26:05	November

If we try and extract month name from a TIME then it will return the current month as a string because it assumes that the times are from today.

```
SELECT birth_date_time, YEAR(birth_date_time), MONTHNAME(birth_date_time) FROM people;
```

birth_date_time	YEAR(birth_date_time)	MONTHNAME(birth_date_time)
2000-12-25 11:00:00	2000	December
1985-04-11 09:00:10	1985	April
2020-08-15 23:59:00	2020	August
2022-11-08 18:26:05	2022	November

All of these functions will also work with date time.

# MySQL – Data Types – Time functions

```
SELECT birth_Time, HOUR(birth_Time) FROM people;
```

birth_Time	HOUR(birth_Time)
11:00:00	11
09:00:10	9
23:59:00	23
18:26:05	18

HOUR function will extract the hour as a value from TIME or DATETIME

```
SELECT birth_Time, MINUTE(birth_Time) FROM people;
```

birth_Time	MINUTE(birth_Time)
11:00:00	0
09:00:10	0
23:59:00	59
18:26:05	26

MINUTE function will extract the minutes as a value TIME or DATETIME. Note how it is MINUTE singular and not minutes.

```
SELECT birth_Time, SECOND(birth_Time) FROM people;
```

birth_Time	SECOND(birth_Time)
11:00:00	0
09:00:10	10
23:59:00	0
18:26:05	5

SECOND function will extract the seconds as a value TIME or DATETIME.

```
SELECT birth_date_time, DATE(birth_date_time), TIME(birth_date_time) FROM people;
```

birth_date_time	DATE(birth_date_time)	TIME(birth_date_time)
2000-12-25 11:00:00	2000-12-25	11:00:00
1985-04-11 09:00:10	1985-04-11	09:00:10
2020-08-15 23:59:00	2020-08-15	23:59:00
2022-11-08 18:26:05	2022-11-08	18:26:05

It is also possible to extract the date or time from DATETIME

# MySQL – Data Types – Formatting Dates

```
SELECT birth_year,  
       CONCAT(MONTHNAME(birth_year), ' ', DAY(birth_year), ' ', YEAR(birth_year))  
       AS Date_of_Birth  
FROM people;
```

birth_year	Date_of_Birth
2000-12-25	December 25 2000
1985-04-11	April 11 1985
2020-08-15	August 15 2020
2022-11-08	November 8 2022

It is possible to format the date using the existing inbuilt functions but this is a little dirty and long winded.

```
SELECT birth_year, DATEFORMAT(birth_year, 'a% %D %M %Y') FROM people;
```

birth_year	DATE_FORMAT(birth_year, 'a% %D %M %Y')
2000-12-25	Mon 25th December 2000
1985-04-11	Thu 11th April 1985
2020-08-15	Sat 15th August 2020
2022-11-08	Tue 8th November 2022

DATE\_FORMAT function takes two parameters, the DATE or DATETIME we want to format then a special string that specifies how we want the date. This string is a series of letters followed by percent sign. Each letter represents a specific format for a component of the Date time.

[https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html#function\\_date-format](https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html#function_date-format)



```
SELECT birth_date_time, TIME_FORMAT(birth_date_time, '%H:%i') FROM people;
```

birth_date_time	TIME_FORMAT(birth_date_time, '%H:%i')
2000-12-25 11:00:00	11:00
1985-04-11 09:00:10	09:00
2020-08-15 23:59:00	23:59
2022-11-08 18:26:05	18:26

TIME\_FORMAT function takes two parameters, the TIME or DATETIME we want to format then a special string that specifies how we want the time. This string is a series of letters followed by percent sign. Each letter represents a specific format for a component of the time.

```
SELECT birth_date_time, TIME_FORMAT(birth_date_time, '%r') FROM people;
```

birth_date_time	TIME_FORMAT(birth_date_time, '%r')
2000-12-25 11:00:00	11:00:00 AM
1985-04-11 09:00:10	09:00:10 AM
2020-08-15 23:59:00	11:59:00 PM
2022-11-08 18:26:05	06:26:05 PM

We can also change from a 24h clock to a 12h clock with AM or PM.

# MySQL – Data Types – Date Math

```
SELECT DATEDIFF(CURDATE(), birth_year) FROM PEOPLE;
```

```
+-----+
| DATEDIFF(CURDATE(), birth_year) |
+-----+
|                                |
|                                |
|                                |
|                                |
|                                |
|                                |
|                                |
|                                |
|                                |
|                                |
|                                |
+-----+
```

DATEDIFF will return the difference  
in days between two dates.

```
SELECT DATE_ADD(CURDATE(), INTERVAL 1 YEAR);
```

```
+-----+
| DATE_ADD(CURDATE(), INTERVAL 1 YEAR) |
+-----+
| 2023-11-08                           |
+-----+
```

DATE\_ADD will add a specific amount to a component  
of the date, i.e. add 1 year to current date.

```
SELECT DATE_ADD(CURDATE(), INTERVAL 5 MONTH);
```

```
+-----+
| DATE_ADD(CURDATE(), INTERVAL 5 MONTH) |
+-----+
| 2023-04-08                             |
+-----+
```

DATE\_ADD will add a specific amount to a component  
of the date, i.e. add 5 months to current date.

```
SELECT DATE_ADD(CURDATE(), INTERVAL 5 MONTH);
```

```
+-----+
| DATE_ADD(CURDATE(), INTERVAL 5 MONTH) |
+-----+
| 2023-04-08                             |
+-----+
```

DATE\_ADD will add a specific amount to a component of the date, i.e. add 5 months to current date.

```
SELECT DATE_SUB(CURDATE(), INTERVAL 5 MONTH);
```

```
+-----+
| DATE_SUB(CURDATE(), INTERVAL 5 MONTH) |
+-----+
| 2022-06-08                             |
+-----+
```

DATE\_SUB will subtract a specific amount from a component of the date, i.e. add 5 months to current date.

```
SELECT birth_year, DATE_ADD(birth_year, INTERVAL 18 YEAR) AS date_18th_birthday
FROM people;
```

```
+-----+-----+
| birth_year | date_18th_birthday |
+-----+-----+
| 2000-12-25 | 2018-12-25         |
| 1985-04-11 | 2003-04-11         |
| 2020-08-15 | 2038-08-15         |
| 2022-11-08 | 2040-11-08         |
+-----+-----+
```

DATE\_ADD can be used to calculate, for example, the 18<sup>th</sup> birthday from date of birth.

```
SELECT birth_time, TIMEDIFF(birth_time, CURTIME()) FROM people;
```

birth_time	TIMEDIFF(birth_time, CURTIME())
11:00:00	-09:48:13
09:00:10	-11:48:03
23:59:00	03:10:47
18:26:05	-02:22:08

TIMEDIFF also works.

```
SELECT NOW() - INTERVAL 18 YEAR;
```

NOW() - INTERVAL 18 YEAR
2004-11-08 20:56:30

We can also use this syntax to add or subtract from dates.

```
SELECT NOW() - INTERVAL 18 YEAR;
```

NOW() - INTERVAL 18 YEAR
2004-11-08 20:56:30

We can also use this syntax to add or subtract from dates.

```
SELECT name, birth_year, birth_year + INTERVAL 21 YEAR AS date_21st_birthday
FROM people;
```

name	birth_year	date_21st_birthday
Elton	2000-12-25	2021-12-25
Lulu	1985-04-11	2006-04-11
Juan	2020-08-15	2041-08-15
Hazel	2022-11-08	2043-11-08

Date calculations can also be done using this syntax.  
Note that it is DAY, MONTH, YEAR singular not plural.

```
SELECT name, birth_year, YEAR(birth_year + INTERVAL 21 YEAR) AS turns_21_in
FROM people;
```

name	birth_year	turns_21_in
Elton	2000-12-25	2021
Lulu	1985-04-11	2006
Juan	2020-08-15	2041
Hazel	2022-11-08	2043

Another example is to only extract the year from a date calculation.

# MySQL – Data Types – Timestamps

**Timestamp** like datetime stores data in format YYYY-MM-DD HH:MM:SS however it uses a much smaller byte size because the timestamp has a **range of '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC**.

The **DATETIME** type is used for values that contain both date and time parts. MySQL retrieves and displays DATETIME values in 'YYYY-MM-DD hh:mm:ss' format. **The supported range is '1000-01-01 00:00:00' to '9999-12-31 23:59:59'**.

Timestamp can be used when the datetime only falls in this range for all rows.

<https://dev.mysql.com/doc/refman/8.0/en/datetime.html>

# MySQL – Data Types – Default & on update timestamps

```
CREATE TABLE captions (  
    img_id INT PRIMARY KEY,  
    caption_text VARCHAR(150),  
    created_at DATETIME default NOW()  
);
```

We can create columns that autofill with the current datetime or timestamp when a row is inserted.

Field	Type	Null	Key	Default	Extra
img_id	int	NO	PRI	NULL	
caption_text	varchar(150)	YES		NULL	
created_at	datetime	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

```
INSERT INTO captions (  
    img_id, caption_text  
) VALUES (  
    1, 'Sunset views from the top of the castle'  
);
```

```
SELECT * FROM captions;
```

img_id	caption_text	created_at
1	Sunset views from the top of the castle	2022-11-08 21:34:37

Note how the current date time is autofilled when a new row is created.

```
CREATE TABLE captions (
  img_id INT PRIMARY KEY,
  caption_text VARCHAR(150),
  created_at DATETIME default NOW()
);
```

We can create columns that autofill with the current datetime or timestamp when a row is inserted.

Field	Type	Null	Key	Default	Extra
img_id	int	NO	PRI	NULL	
caption_text	varchar(150)	YES		NULL	
created_at	datetime	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

For any TIMESTAMP or DATETIME column in a table, you can assign the current timestamp as the default value, the auto-update value, or both:

An auto-initialized column is set to the current timestamp for inserted rows that specify no value for the column.

**An auto-updated column is automatically updated to the current timestamp when the value of any other column in the row is changed from its current value.**

<https://dev.mysql.com/doc/refman/5.6/en/timestamp-initialization.html>



```
CREATE TABLE captions2 (
    img_id INT PRIMARY KEY,
    caption_text VARCHAR(150),
    created_at DATETIME default NOW(),
    updated_At DATETIME ON UPDATE NOW()
);
```

We can create a datetime or timestamp column that automatically updates when any other column in the table changes.

Field	Type	Null	Key	Default	Extra
img_id	int	NO	PRI	NULL	
caption_text	varchar(150)	YES		NULL	
created_at	datetime	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED
updated_At	datetime	YES		NULL	on update CURRENT_TIMESTAMP

```
INSERT INTO captions2 (
    img_id, caption_text
) VALUES (
    1, 'Sunset views from the top of the castle'
);
```

Note that the updated\_at column has a value of NUL because we did not specify a default value and this row has not been updated yet.

```
SELECT * FROM captions;
```

img_id	caption_text	created_at	updated_At
1	Sunset views from the top of the castle	2022-11-08 21:46:50	NULL

```
UPDATE captions2 SET caption_text='Sunset views from castle' WHERE img_id = 1;
```

When the caption\_text column is updated the updated\_At column value changes to datetime of now().

```
SELECT * FROM captions;
```

img_id	caption_text	created_at	updated_At
1	Sunset views from castle	2022-11-08 21:46:50	2022-11-08 22:01:07

# MySQL – Data Types – Exercises

```
CREATE TABLE inventory (  
    item_name VARCHAR(100),  
    price DECIMAL (7,2) or FLOAT or DOUBLE,  
    quantity TINYINT  
);
```

Price is always less than 1million dolars

```
SELECT  
DAYOFWEEK(NOW()),  
DATE_FORMAT(NOW(), '%W'),  
SELECT DATE_FORMAT(NOW(), '%m/%d/%y')  
DATE_FORMAT(NOW(), '%M %D at %k:%i');
```

DAYOFWEEK(NOW())	DATE_FORMAT(NOW(), '%W')	DATE_FORMAT(NOW(), '%m/%d/%Y')	DATE_FORMAT(NOW(), '%M %D at %H:%i')
3	Tuesday	11/08/2022	November 8th at 22:39

```
CREATE TABLE tweets (  
    tweet_content VARCHAR(180),  
    username VARCHAR(20),  
    created_at DATETIME default NOW()  
);
```

Field	Type	Null	Key	Default	Extra
tweet_content	varchar(180)	YES		NULL	
username	varchar(20)	YES		NULL	
created_at	datetime	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

# Comparison & Logical Operators

# MySQL – Comparison Operators – Not Equal

```
SELECT * FROM books WHERE released_year != 2017;
```

book_id	title	author_fname	author_lname	released_year	stock_quantity	pages
1	The Namesake	Jhumpa	Lahiri	2003	32	291
2	Norse Mythology	Neil	Gaiman	2016	43	304
3	American Gods	Neil	Gaiman	2001	12	465
4	Interpreter of Maladies	Jhumpa	Lahiri	1996	97	198
5	A Hologram for the King: A Novel	Dave	Eggers	2012	154	352
6	The Circle	Dave	Eggers	2013	26	504
7	The Amazing Adventures of Kavalier & Clay	Michael	Chabon	2000	68	634
8	Just Kids	Patti	Smith	2010	55	304
9	A Heartbreaking Work of Staggering Genius	Dave	Eggers	2001	104	437
10	Coraline	Neil	Gaiman	2003	100	208
11	What We Talk About When We Talk About Love: Stories	Raymond	Carver	1981	23	176
12	Where I'm Calling From: Selected Stories	Raymond	Carver	1989	12	526
13	White Noise	Don	DeLillo	1985	49	320
14	Cannery Row	John	Steinbeck	1945	95	181
15	Oblivion: Stories	David	Foster Wallace	2004	172	329
16	Consider the Lobster	David	Foster Wallace	2005	92	343
17	10% Happier	Dan	Harris	2014	29	256
18	fake_book	Freida	Harris	2001	287	428

18 rows in set (0.06 sec)

**!=** means not equal. So this query will select books where the released year is not 2017. WE can see that it returned 18 rows of books when the table contains a count of 19 books, meaning one book was written in 2017.

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

COUNT(*)
19

```
SELECT title, author_lname FROM books WHERE author_lname != 'Gaiman';
```

title	author_lname
The Namesake	Lahiri
Interpreter of Maladies	Lahiri
A Hologram for the King: A Novel	Eggers
The Circle	Eggers
The Amazing Adventures of Kavalier & Clay	Chabon
Just Kids	Smith
A Heartbreaking Work of Staggering Genius	Eggers
What We Talk About When We Talk About Love: Stories	Carver
Where I'm Calling From: Selected Stories	Carver
White Noise	DeLillo
Cannery Row	Steinbeck
Oblivion: Stories	Foster Wallace
Consider the Lobster	Foster Wallace
10% Happier	Harris
fake_book	Harris
Lincoln In The Bardo	Saunders

!= author last name is not Gaiman.

# MySQL – Comparison Operators – Not Like

```
SELECT title FROM books WHERE title LIKE '% %';
```

title
The Namesake
Norse Mythology
American Gods
Interpreter of Maladies
A Hologram for the King: A Novel
The Circle
The Amazing Adventures of Kavalier & Clay
Just Kids
A Heartbreaking Work of Staggering Genius
What We Talk About When We Talk About Love: Stories
Where I'm Calling From: Selected Stories
White Noise
Cannery Row
Oblivion: Stories
Consider the Lobster
10% Happier
Lincoln In The Bardo

'% %' Matches a string of space. i.e.  
find all books with a space in the title.

```
SELECT title FROM books WHERE title NOT LIKE '% %';
```

title
Coraline
fake_book

We use the word NOT with LIKE and ! With equals.

This query matches all book titles without a space in the title.



```
SELECT title, author_fname, author_lname FROM books WHERE author_fname LIKE 'da%';
```

title	author_fname	author_lname
A Hologram for the King: A Novel	Dave	Eggers
The Circle	Dave	Eggers
A Heartbreaking Work of Staggering Genius	Dave	Eggers
Oblivion: Stories	David	Foster Wallace
Consider the Lobster	David	Foster Wallace
10% Happier	Dan	Harris

find all authors who's first name starts with da.

```
SELECT title, author_fname, author_lname FROM books WHERE author_fname NOT LIKE 'da%';
```

title	author_fname	author_lname
The Namesake	Jhumpa	Lahiri
Norse Mythology	Neil	Gaiman
American Gods	Neil	Gaiman
Interpreter of Maladies	Jhumpa	Lahiri
The Amazing Adventures of Kavalier & Clay	Michael	Chabon
Just Kids	Patti	Smith
Coraline	Neil	Gaiman
What We Talk About When We Talk About Love: Stories	Raymond	Carver
Where I'm Calling From: Selected Stories	Raymond	Carver
White Noise	Don	DeLillo
Cannery Row	John	Steinbeck
fake_book	Freida	Harris
Lincoln In The Bardo	George	Saunders

The inversion of this would be to add NOT before the LIKE to find all authors who's names do not begin with da.

```
SELECT title FROM books WHERE title NOT LIKE '%e%';
```

title
Just Kids

All books without the letter e in the title

# MySQL – Comparison Operators – Greater Than

```
SELECT * FROM books WHERE released_year > 2000;
```

find all books where release year **is greater than** 2000.

book_id	title	author_fname	author_lname	released_year	stock_quantity	pages
1	The Namesake	Jhumpa	Lahiri	2003	32	291
2	Norse Mythology	Neil	Gaiman	2016	43	304
3	American Gods	Neil	Gaiman	2001	12	465
5	A Hologram for the King: A Novel	Dave	Eggers	2012	154	352
6	The Circle	Dave	Eggers	2013	26	504
8	Just Kids	Patti	Smith	2010	55	304
9	A Heartbreaking Work of Staggering Genius	Dave	Eggers	2001	104	437
10	Coraline	Neil	Gaiman	2003	100	208
15	Oblivion: Stories	David	Foster Wallace	2004	172	329
16	Consider the Lobster	David	Foster Wallace	2005	92	343
17	10% Happier	Dan	Harris	2014	29	256
18	fake_book	Freida	Harris	2001	287	428
19	Lincoln In The Bardo	George	Saunders	2017	1000	367

```
SELECT * FROM books WHERE pages > 500;
```

find all books with more than 500 pages.

book_id	title	author_fname	author_lname	released_year	stock_quantity	pages
6	The Circle	Dave	Eggers	2013	26	504
7	The Amazing Adventures of Kavalier & Clay	Michael	Chabon	2000	68	634
12	Where I'm Calling From: Selected Stories	Raymond	Carver	1989	12	526

```
SELECT 80 > 40;
```

```
SELECT 80 > 90;
```

```
+-----+
| 80 > 40 |
+-----+
|      1 |
+-----+
```

```
+-----+
| 80 > 90 |
+-----+
|      0 |
+-----+
```

When we evaluate if one value is greater than another MySQL will return a Boolean. I.e. 1 for true and zero for false.

```
SELECT * FROM books WHERE pages > 500;
```

book_id	title	author_fname	author_lname	released_year	stock_quantity	pages
6	The Circle	Dave	Eggers	2013	26	504
7	The Amazing Adventures of Kavalier & Clay	Michael	Chabon	2000	68	634
12	Where I'm Calling From: Selected Stories	Raymond	Carver	1989	12	526

When MySQL is evaluating if pages is greater than 500, it is in the background finding columns that have a Boolean 1 result to this evaluation.

```
SELECT SELECT 1 > NULL;
```

1 > NULL
NULL

When we try and compare something to nothing we get NULL so if a book in the table does not have a pages value specified then it will always evaluate to NULL.

# MySQL – Comparison Operators – Less Than or equal to

```
SELECT title, released_year FROM books WHERE released_year < 2000;
```

title	released_year
Interpreter of Maladies	1996
What We Talk About When We Talk About Love: Stories	1981
Where I'm Calling From: Selected Stories	1989
White Noise	1985
Cannery Row	1945

find all books where release year **is less than** 2000.

```
SELECT title, released_year FROM books WHERE released_year <= 2000;
```

title	released_year
Interpreter of Maladies	1996
The Amazing Adventures of Kavalier & Clay	2000
What We Talk About When We Talk About Love: Stories	1981
Where I'm Calling From: Selected Stories	1989
White Noise	1985
Cannery Row	1945

find all books where release year  
**is less than or equal to** 2000.  
Note that we also get books from  
2000.

```
SELECT title, pages, released_year FROM books WHERE pages <= 200;
```

title	pages	released_year
Interpreter of Maladies	198	1996
What We Talk About When We Talk About Love: Stories	176	1981
Cannery Row	181	1945

find all books where pages **is less than or equal to** 200.

```
SELECT title, released_year FROM books WHERE released_year >= 2010;
```

title	released_year
Norse Mythology	2016
A Hologram for the King: A Novel	2012
The Circle	2013
Just Kids	2010
10% Happier	2014
Lincoln In The Bardo	2017

find all books where release year  
**is grater than or equal to** 2000.

```
SELECT title, released_year FROM books WHERE released_year <= 1985;
```

title	released_year
What We Talk About When We Talk About Love: Stories	1981
White Noise	1985
Cannery Row	1945

find all books where release year  
**is less than or equal to** 1985.

# MySQL – Logical Operators – And

```
SELECT title, author_lname, released_year FROM books WHERE author_lname = 'Eggers'  
AND released_year >= 2010;
```

title	author_lname	released_year
A Hologram for the King: A Novel	Eggers	2012
The Circle	Eggers	2013

find all books where author **last name is 'Eggers'**  
**AND** release year **is grater than or equal to** 2010.

```
SELECT title, author_lname, released_year FROM books WHERE author_lname = 'Eggers'  
AND released_year >= 2010 AND title LIKE '%novel%';
```

title	author_lname	released_year
A Hologram for the King: A Novel	Eggers	2012

Multiple AND's can be chained to together so  
that the query results meet many conditions.

When we use AND each condition has to evaluate to be true for the row to be included in the results.

```
SELECT title, pages FROM books WHERE CHAR_LENGTH(title) > 30 AND pages > 500;
```

title	pages
The Amazing Adventures of Kavalier & Clay	634
Where I'm Calling From: Selected Stories	526

Select books where title has greater than 30  
characters **AND** greater than 500 pages.

# MySQL – Logical Operators – Or

```
SELECT title, author_lname, released_year FROM books WHERE author_lname = 'Eggers'
OR released_year > 2010;
```

title	author_lname	released_year
Norse Mythology	Gaiman	2016
A Hologram for the King: A Novel	Eggers	2012
The Circle	Eggers	2013
A Heartbreaking Work of Staggering Genius	Eggers	2001
10% Happier	Harris	2014
Lincoln In The Bardo	Saunders	2017

find all books where author **last name is 'Eggers'** OR release year **is grater than** 2010.

Either left or right condition can be true in an OR condition.

```
SELECT title, pages FROM books WHERE pages < 200 OR title LIKE '%Stories%';
```

title	pages
Interpreter of Maladies	198
What We Talk About When We Talk About Love: Stories	176
Where I'm Calling From: Selected Stories	526
Cannery Row	181
Oblivion: Stories	329

This query will find books that are short with less than 200 pages **OR** that are collection of stories, i.e. contain the word stories in the title.

# MySQL – Logical Operators – Between

```
SELECT title, released_year FROM books WHERE released_year >= 2004
AND released_year <= 2015;
```

title	released_year
A Hologram for the King: A Novel	2012
The Circle	2013
Just Kids	2010
Oblivion: Stories	2004
Consider the Lobster	2005
10% Happier	2014

This query uses AND to find all books released between 2004 and 2015.

```
SELECT title, released_year FROM books WHERE released_year BETWEEN 2004 AND 2015;
```

title	released_year
A Hologram for the King: A Novel	2012
The Circle	2013
Just Kids	2010
Oblivion: Stories	2004
Consider the Lobster	2005
10% Happier	2014

We can achieve the same using the between keyword which is a lot cleaner than more than or equal to AND less than or equal to.

```
SELECT title, released_year FROM books WHERE released_year BETWEEN 2004 AND 2014;
```

title	released_year
A Hologram for the King: A Novel	2012
The Circle	2013
Just Kids	2010
Oblivion: Stories	2004
Consider the Lobster	2005
10% Happier	2014

Note that between includes rows that match the lower and upper ranges.



```
SELECT title, pages FROM books WHERE pages BETWEEN 200 AND 300;
```

title	pages
The Namesake	291
Coraline	208
10% Happier	256

This query uses between to find books with a range of pages.

```
SELECT title, pages FROM books WHERE pages NOT BETWEEN 200 AND 300;
```

title	pages
Norse Mythology	304
American Gods	465
Interpreter of Maladies	198
A Hologram for the King: A Novel	352
The Circle	504
The Amazing Adventures of Kavalier & Clay	634
Just Kids	304
A Heartbreaking Work of Staggering Genius	437
What We Talk About When We Talk About Love: Stories	176
Where I'm Calling From: Selected Stories	526
White Noise	320
Cannery Row	181
Oblivion: Stories	329
Consider the Lobster	343
fake_book	428
Lincoln In The Bardo	367

This query uses between combined with NOT to find books with pages that are not in the specified between range.

# MySQL – Comparing Dates

```
SELECT * FROM people WHERE birth_year < '2015-01-01';
```

name	birth_year	birth_Time	birth_date_time
Elton	2000-12-25	11:00:00	2000-12-25 11:00:00
Lulu	1985-04-11	09:00:10	1985-04-11 09:00:10

Comparison operators can be used to compare dates when we specify the date in the correct format like a string.

```
SELECT * FROM people WHERE YEAR(birth_year) < 2005;
```

name	birth_year	birth_Time	birth_date_time
Elton	2000-12-25	11:00:00	2000-12-25 11:00:00
Lulu	1985-04-11	09:00:10	1985-04-11 09:00:10

A better way would be to extract the year from birth\_year then use a comparison operator. This is a lot more specific than using a date string.

```
SELECT * FROM people WHERE birth_time > '12:00:00';
```

name	birth_year	birth_Time	birth_date_time
Juan	2020-08-15	23:59:00	2020-08-15 23:59:00
Hazel	2022-11-08	18:26:05	2022-11-08 18:26:05

Time can also be compared using the correctly formatted time string.

```
SELECT * FROM people WHERE HOUR(birth_time) > 12;
```

name	birth_year	birth_Time	birth_date_time
Juan	2020-08-15	23:59:00	2020-08-15 23:59:00
Hazel	2022-11-08	18:26:05	2022-11-08 18:26:05

Again it is cleaner to extract the part of time we want to query then run the operator.

```
SELECT CAST('9:3:5' AS TIME);
```

MySQL can convert a string into a specific date type, for example it can convert a colon sperated string into TIME.

```
+-----+
| CAST('9:3:5' AS TIME) |
+-----+
| 09:03:05               |
+-----+
```

```
SELECT * FROM people WHERE birth_time BETWEEN '12:00:00' AND '19:00:00';
```

```
+-----+-----+-----+-----+
| name  | birth_year | birth_Time | birth_date_time |
+-----+-----+-----+-----+
| Hazel | 2022-11-08 | 18:26:05   | 2022-11-08 18:26:05 |
+-----+-----+-----+-----+
```

We can use a string for time and compare using between. This works because SQL has figured out that the string is time format.

```
SELECT * FROM people WHERE birth_time BETWEEN CAST('12:00:00' AS TIME)
AND CAST('19:00:00' AS TIME);
```

```
+-----+-----+-----+-----+
| name  | birth_year | birth_Time | birth_date_time |
+-----+-----+-----+-----+
| Hazel | 2022-11-08 | 18:26:05   | 2022-11-08 18:26:05 |
+-----+-----+-----+-----+
```

But the correct way would be to cast the string as a specific datatype such as TIME.

```
SELECT * FROM people WHERE HOUR(birth_time) BETWEEN 12 AND 19;
```

```
+-----+-----+-----+-----+
| name  | birth_year | birth_Time | birth_date_time |
+-----+-----+-----+-----+
| Hazel | 2022-11-08 | 18:26:05   | 2022-11-08 18:26:05 |
+-----+-----+-----+-----+
```

Better yet is to extract the part of the TIME we want to use then use between with values, not strings.

# MySQL – The IN Operator

```
SELECT title, author_lname FROM books WHERE author_lname = 'Carver' OR  
author_lname = 'Lahiri' OR author_lname = 'Smith';
```

title	author_lname
The Namesake	Lahiri
Interpreter of Maladies	Lahiri
Just Kids	Smith
What We Talk About When We Talk About Love: Stories	Carver
Where I'm Calling From: Selected Stories	Carver

We can chain together several author surnames with OR statements to find their books but there is a simpler way.

```
SELECT title, author_lname FROM books WHERE author_lname  
IN ('Carver', 'Lahiri', 'Smith');
```

title	author_lname
The Namesake	Lahiri
Interpreter of Maladies	Lahiri
Just Kids	Smith
What We Talk About When We Talk About Love: Stories	Carver
Where I'm Calling From: Selected Stories	Carver

Using IN we can specify a series of values to match.

```
SELECT title, author_lname FROM books WHERE author_lname
NOT IN ('Carver', 'Lahiri', 'Smith');
```

title	author_lname
Norse Mythology	Gaiman
American Gods	Gaiman
A Hologram for the King: A Novel	Eggers
The Circle	Eggers
The Amazing Adventures of Kavalier & Clay	Chabon
A Heartbreaking Work of Staggering Genius	Eggers
Coraline	Gaiman
White Noise	DeLillo
Cannery Row	Steinbeck
Oblivion: Stories	Foster Wallace
Consider the Lobster	Foster Wallace
10% Happier	Harris
fake_book	Harris
Lincoln In The Bardo	Saunders

IN can be negated with NOT.

```
SELECT title, released_year FROM BOOKS WHERE released_year NOT IN ('2000', '2004', '2006', '2008', '2010', '2012', '2014', '2016') ORDER BY released_year;
```

title	released_year
Cannery Row	1945
What We Talk About When We Talk About Love: Stories	1981
White Noise	1985
Where I'm Calling From: Selected Stories	1989
Interpreter of Maladies	1996
American Gods	2001
A Heartbreaking Work of Staggering Genius	2001
fake_book	2001
The Namesake	2003
Coraline	2003
Consider the Lobster	2005
A Hologram for the King: A Novel	2012
The Circle	2013
10% Happier	2014
Norse Mythology	2016
Lincoln In The Bardo	2017

We can use IN to exclude books from even years after 2000.

The query can be refined to exclude old books, i.e. those written before 2000.

```
SELECT
    title, released_year
FROM BOOKS WHERE
    released_year >= 2000
AND
    released_year NOT IN ('2000',
        '2004', '2006', '2008', '2010',
        '2012', '2014', '2016'
    ) ORDER BY released_year;
```

title	released_year
American Gods	2001
A Heartbreaking Work of Staggering Genius	2001
fake_book	2001
The Namesake	2003
Coraline	2003
Consider the Lobster	2005
A Hologram for the King: A Novel	2012
The Circle	2013
Lincoln In The Bardo	2017

# MySQL – Modulo

```
SELECT 10 / 4;
```

```
+-----+
| 10 / 4 |
+-----+
| 2.5000 |
+-----+
```

```
SELECT 10 % 4;
```

```
+-----+
| 10 % 4 |
+-----+
|      2 |
+-----+
```

```
SELECT 17 % 6;
```

```
+-----+
| 17 % 6 |
+-----+
|      5 |
+-----+
```

The modulo denoted by the % sign is the integer left over when dividing 10 by 4 or 17 by 6.

```
SELECT 8 % 4;
```

```
+-----+
| 8 % 4 |
+-----+
|      0 |
+-----+
```

```
SELECT 2008 % 2;
```

```
+-----+
| 2008 % 2 |
+-----+
|          0 |
+-----+
```

```
SELECT 2001 % 2;
```

```
+-----+
| 2001 % 2 |
+-----+
|          1 |
+-----+
```

Modulo can be used to find odd and even numbers because an odd number will always have a remainder of 1 and an even number will always have a remainder of 0 when divided by 2.

```
SELECT
    title, released_year
FROM BOOKS
WHERE
    released_year >= 2000 AND
    released_year % 2 != 1
ORDER BY released_year;
```

```
+-----+-----+
| title                                | released_year |
+-----+-----+
| The Amazing Adventures of Kavalier & Clay | 2000 |
| Oblivion: Stories                     | 2004 |
| Just Kids                             | 2010 |
| A Hologram for the King: A Novel      | 2012 |
| 10% Happier                           | 2014 |
| Norse Mythology                        | 2016 |
+-----+-----+
```

Books released after 2000 and not in ODD years using MODULO.

# MySQL – The CASE Operator

```
SELECT title, released_year,  
       CASE  
         WHEN released_year >= 2000 THEN 'Modern Lit'  
         ELSE '20th Century Lit'  
       END AS GENERE  
FROM books;
```

When the release year is greater than 2000 then the corresponding value should be modern lit.

Else the value is 20<sup>th</sup> century lit.

title	released_year	GENERE
The Namesake	2003	Modern Lit
Norse Mythology	2016	Modern Lit
American Gods	2001	Modern Lit
Interpreter of Maladies	1996	20th Century Lit
A Hologram for the King: A Novel	2012	Modern Lit
The Circle	2013	Modern Lit
The Amazing Adventures of Kavalier & Clay	2000	Modern Lit
Just Kids	2010	Modern Lit
A Heartbreaking Work of Staggering Genius	2001	Modern Lit
Coraline	2003	Modern Lit
What We Talk About When We Talk About Love: Stories	1981	20th Century Lit
Where I'm Calling From: Selected Stories	1989	20th Century Lit
White Noise	1985	20th Century Lit
Cannery Row	1945	20th Century Lit
Oblivion: Stories	2004	Modern Lit
Consider the Lobster	2005	Modern Lit
10% Happier	2014	Modern Lit
fake_book	2001	Modern Lit
Lincoln In The Bardo	2017	Modern Lit

Case statements open with CASE and close with END.

They contain the WHEN and THEN clause and the ELSE clause.



```

SELECT title, stock_quantity,
       CASE
           WHEN stock_quantity between 0 AND 40 THEN '*'
           WHEN stock_quantity between 41 AND 70 THEN '**'
           WHEN stock_quantity between 71 AND 100 THEN '***'
           WHEN stock_quantity between 101 AND 140 THEN '****'
           ELSE '*****'
       END AS STOCK
FROM books;

```

MySQL will evaluate each of the WHEN statements in sequence till it finds a match which can be slow when running this on large SQL tables.

title	stock_quantity	STOCK
The Namesake	32	*
Norse Mythology	43	**
American Gods	12	*
Interpreter of Maladies	97	***
A Hologram for the King: A Novel	154	*****
The Circle	26	*
The Amazing Adventures of Kavalier & Clay	68	**
Just Kids	55	**
A Heartbreaking Work of Staggering Genius	104	****
Coraline	100	***
What We Talk About When We Talk About Love: Stories	23	*
Where I'm Calling From: Selected Stories	12	*
White Noise	49	**
Cannery Row	95	***
Oblivion: Stories	172	*****
Consider the Lobster	92	***
10% Happier	29	*
fake_book	287	*****
Lincoln In The Bardo	1000	*****

```

SELECT title, stock_quantity,
       CASE
           WHEN stock_quantity <= 40 THEN '*'
           WHEN stock_quantity <= 70 THEN '**'
           WHEN stock_quantity <= 100 THEN '***'
           WHEN stock_quantity <= 140 THEN '****'
           ELSE '*****'
       END AS STOCK
FROM books;

```

Because MySQL evaluates each when statement in sequence to find the one that matches then jumps to the END when it finds a match, only one case statement can be true so we can use this to shorten the between clause to a less than or equal to statement.

title	stock_quantity	STOCK
The Namesake	32	*
Norse Mythology	43	**
American Gods	12	*
Interpreter of Maladies	97	***
A Hologram for the King: A Novel	154	*****
The Circle	26	*
The Amazing Adventures of Kavalier & Clay	68	**
Just Kids	55	**
A Heartbreaking Work of Staggering Genius	104	****
Coraline	100	***
What We Talk About When We Talk About Love: Stories	23	*
Where I'm Calling From: Selected Stories	12	*
White Noise	49	**
Cannery Row	95	***
Oblivion: Stories	172	*****
Consider the Lobster	92	***
10% Happier	29	*
fake_book	287	*****
Lincoln In The Bardo	1000	*****

A case statement can have an unlimited number of WHEN THEN clauses but only one of them will ever be true and return a value from the THEN.

# MySQL – The IS NULL Operator

```
INSERT INTO books (title, author_fname, released_year, stock_quantity)
VALUES (null_book, John, 2000, 200);
```

book_id	title	author_fname	author_lname	released_year	stock_quantity	pages
1	The Namesake	Jhumpa	Lahiri	2003	32	291
2	Norse Mythology	Neil	Gaiman	2016	43	304
3	American Gods	Neil	Gaiman	2001	12	465
4	Interpreter of Maladies	Jhumpa	Lahiri	1996	97	198
5	A Hologram for the King: A Novel	Dave	Eggers	2012	154	352
6	The Circle	Dave	Eggers	2013	26	504
7	The Amazing Adventures of Kavalier & Clay	Michael	Chabon	2000	68	634
8	Just Kids	Patti	Smith	2010	55	304
9	A Heartbreaking Work of Staggering Genius	Dave	Eggers	2001	104	437
10	Coraline	Neil	Gaiman	2003	100	208
11	What We Talk About When We Talk About Love: Stories	Raymond	Carver	1981	23	176
12	Where I'm Calling From: Selected Stories	Raymond	Carver	1989	12	526
13	White Noise	Don	DeLillo	1985	49	320
14	Cannery Row	John	Steinbeck	1945	95	181
15	Oblivion: Stories	David	Foster Wallace	2004	172	329
16	Consider the Lobster	David	Foster Wallace	2005	92	343
17	10% Happier	Dan	Harris	2014	29	256
18	fake_book	Freida	Harris	2001	287	428
19	Lincoln In The Bardo	George	Saunders	2017	1000	367
22	null_book	John	NULL	2000	200	NULL

```
SELECT * FROM books WHERE author_lname = NULL;
```

Empty set (0.00 sec)

```
SELECT * FROM books WHERE author_lname IS NULL;
```

book_id	title	author_fname	author_lname	released_year	stock_quantity	pages
22	null_book	John	NULL	2000	200	NULL

Cannot query for values equal to Null. We have to use the IS NULL operator.

```
SELECT * FROM books WHERE author_lname IS NOT NULL;
```

Null operator can be negated with  
IS NOT NULL

book_id	title	author_fname	author_lname	released_year	stock_quantity	pages
1	The Namesake	Jhumpa	Lahiri	2003	32	291
2	Norse Mythology	Neil	Gaiman	2016	43	304
3	American Gods	Neil	Gaiman	2001	12	465
4	Interpreter of Maladies	Jhumpa	Lahiri	1996	97	198
5	A Hologram for the King: A Novel	Dave	Eggers	2012	154	352
6	The Circle	Dave	Eggers	2013	26	504
7	The Amazing Adventures of Kavalier & Clay	Michael	Chabon	2000	68	634
8	Just Kids	Patti	Smith	2010	55	304
9	A Heartbreaking Work of Staggering Genius	Dave	Eggers	2001	104	437
10	Coraline	Neil	Gaiman	2003	100	208
11	What We Talk About When We Talk About Love: Stories	Raymond	Carver	1981	23	176
12	Where I'm Calling From: Selected Stories	Raymond	Carver	1989	12	526
13	White Noise	Don	DeLillo	1985	49	320
14	Cannery Row	John	Steinbeck	1945	95	181
15	Oblivion: Stories	David	Foster Wallace	2004	172	329
16	Consider the Lobster	David	Foster Wallace	2005	92	343
17	10% Happier	Dan	Harris	2014	29	256
18	fake_book	Freida	Harris	2001	287	428
19	Lincoln In The Bardo	George	Saunders	2017	1000	367

```
DELETE FROM books WHERE author_lname IS NULL;
```

Query OK, 1 row affected (0.00 sec)

```
SELECT * FROM books WHERE author_lname IS NULL;
```

Empty set (0.00 sec)

# MySQL – Comparison & Logical Operators Exercises

```
SELECT title, released_year FROM books WHERE released_year < 1980;
```

title	released_year
Cannery Row	1945

```
SELECT title, author_lname FROM books WHERE author_lname IN ('Eggers', 'Chabon');
```

title	author_lname
A Hologram for the King: A Novel	Eggers
The Circle	Eggers
The Amazing Adventures of Kavalier & Clay	Chabon
A Heartbreaking Work of Staggering Genius	Eggers

```
SELECT title, author_lname, released_year FROM books WHERE author_lname = 'Lahiri'  
AND released_year > 2000;
```

title	author_lname	released_year
The Namesake	Lahiri	2003

```
SELECT title, pages FROM books WHERE pages BETWEEN 100 AND 200;
```

title	pages
Interpreter of Maladies	198
What We Talk About When We Talk About Love: Stories	176
Cannery Row	181

```
SELECT title, author_lname FROM books WHERE author_lname LIKE 'c%' OR 's%';
```

title	author_lname
The Amazing Adventures of Kavalier & Clay	Chabon
What We Talk About When We Talk About Love: Stories	Carver
Where I'm Calling From: Selected Stories	Carver

```
SELECT title, author_lname FROM books WHERE author_lname LIKE 'c%' OR  
author_lname LIKE 's%';
```

title	author_lname
The Amazing Adventures of Kavalier & Clay	Chabon
Just Kids	Smith
What We Talk About When We Talk About Love: Stories	Carver
Where I'm Calling From: Selected Stories	Carver
Cannery Row	Steinbeck
Lincoln In The Bardo	Saunders

```
SELECT title, author_lname FROM books WHERE SUBSTR(author_lname, 1, 1)  
IN ('C', 'S');
```

title	author_lname
The Amazing Adventures of Kavalier & Clay	Chabon
Just Kids	Smith
What We Talk About When We Talk About Love: Stories	Carver
Where I'm Calling From: Selected Stories	Carver
Cannery Row	Steinbeck
Lincoln In The Bardo	Saunders

```

SELECT title, author_lname,
       CASE
         WHEN title LIKE '%stories%' THEN 'Short Stories'
         WHEN title = 'Just Kids' OR title = 'A Heartbreaking Work of Staggering Genius' THEN 'Memoir'
         ELSE 'Novel'
       END AS TYPE
FROM books;

```

title	author_lname	TYPE
The Namesake	Lahiri	Novel
Norse Mythology	Gaiman	Novel
American Gods	Gaiman	Novel
Interpreter of Maladies	Lahiri	Novel
A Hologram for the King: A Novel	Eggers	Novel
The Circle	Eggers	Novel
The Amazing Adventures of Kavalier & Clay	Chabon	Novel
Just Kids	Smith	Memoir
A Heartbreaking Work of Staggering Genius	Eggers	Memoir
Coraline	Gaiman	Novel
What We Talk About When We Talk About Love: Stories	Carver	Short Stories
Where I'm Calling From: Selected Stories	Carver	Short Stories
White Noise	DeLillo	Novel
Cannery Row	Steinbeck	Novel
Oblivion: Stories	Foster Wallace	Short Stories
Consider the Lobster	Foster Wallace	Novel
10% Happier	Harris	Novel
fake_book	Harris	Novel
Lincoln In The Bardo	Saunders	Novel

```

SELECT author_fname, author_lname,
       CASE
           WHEN COUNT(*) = 1 THEN '1 book'
           ELSE CONCAT(COUNT(*), ' books')
       END AS count
FROM books
WHERE author_lname IS NOT NULL
GROUP BY author_fname, author_lname;

```

author_fname	author_lname	count
Jhumpa	Lahiri	2 books
Neil	Gaiman	3 books
Dave	Eggers	3 books
Michael	Chabon	1 book
Patti	Smith	1 book
Raymond	Carver	2 books
Don	DeLillo	1 book
John	Steinbeck	1 book
David	Foster Wallace	2 books
Dan	Harris	1 book
Freida	Harris	1 book
George	Saunders	1 book



# Constraints & ALTER TABLE

# MySQL – Constraints - Unique

```
CREATE TABLE companies (  
    supplier_id INT AUTO_INCREMENT,  
    name VARCHAR(255) NOT NULL,  
    phone VARCHAR(15) NOT NULL UNIQUE,  
    address VARCHAR(255) NOT NULL,  
    PRIMARY KEY (supplier_id)  
);
```

Table columns can have many types of constraints that restrict the data we can put in, i.e. auto\_increment or NOT NULL or PRIMARY KEY.

Another type of constraint is UNIQUE which means that like PRIMARY KEY, there cannot be another row with the same data.

```
DESC companies;
```

Field	Type	Null	Key	Default	Extra
supplier_id	int	NO	PRI	NULL	auto_increment
name	varchar(255)	NO		NULL	
phone	varchar(15)	NO	UNI	NULL	
address	varchar(255)	NO		NULL	

```
INSERT INTO companies (name, phone, address)  
VALUES ('Pet Store', '34 222 222 222', 'dog street, large city, somewhere');  
Query OK, 1 row affected (0.01 sec)
```

```
INSERT INTO companies (name, phone, address)  
VALUES ('cat Store', '34 222 222 222', 'cat drive, village, nowhere');
```

**ERROR 1062 (23000): Duplicate entry '34 222 222 222' for key 'companies.phone'**

# MySQL – Constraints – Check Constraints

```
CREATE TABLE nightclub_members (  
    name VARCHAR(50),  
    age INT CHECK (age > 18)  
);
```

Check constraints are custom defined constraints. i.e. check if age is greater than 18.

When we insert a row of data the column with the check has to pass the test and if the result is 1 then the row is inserted if the result is 0 then the whole row is considered invalid

Field	Type	Null	Key	Default	Extra
name	varchar(50)	YES		NULL	
age	int	YES		NULL	

```
DESC nightclub_members;
```

```
INSERT INTO nightclub_members (name, age) VALUES ('John', 19);  
Query OK, 1 row affected (0.00 sec)
```

```
INSERT INTO nightclub_members (name, age) VALUES ('Joe', 17);  
ERROR 3819 (HY000): Check constraint 'nightclub_members_chk_1'  
is violated.
```

Another example would be to use a check of age INT CHECK (age > 0) so that negative values for age cannot be entered.

```
CREATE TABLE palindromes (  
    word VARCHAR(100) CHECK(REVERSE(word) = word)  
);
```

The CHECK constrain is not limited to numbers but can use any of the previously learn built in functions and operators.

```
mysql> insert INTO palindromes (word) VALUES ('racecar');  
Query OK, 1 row affected (0.00 sec)
```

```
mysql> insert INTO palindromes (word) VALUES ('hannah');  
Query OK, 1 row affected (0.00 sec)
```

```
mysql> insert INTO palindromes (word) VALUES ('pear');  
ERROR 3819 (HY000): Check constraint 'palindromes_chk_1' is violated.
```

```
CREATE TABLE nightclub_members2 (  
    name VARCHAR(50),  
    age INT, CONSTRAINT age_over_18 CHECK (age > 18)  
);
```

When we create a CHECK constraint MySQL automatically assigns a name to it, i.e. chk\_1 for the first constraint, chk\_2 for the second etc... But we can also define a short name.

```
INSERT INTO nightclub_members2 (name, age) VALUES ('Joe', 17);  
ERROR 3819 (HY000): Check constraint 'age_over_18' is violated.
```

# MySQL – Constraints – Multiple column Constraints

```
CREATE TABLE companies2 (  
    name VARCHAR(255) NOT NULL,  
    address VARCHAR(255) NOT NULL,  
    CONSTRAINT name_address UNIQUE (name, address)  
);
```

Name and address have unique defined as a named constraint. What this means is that the combination of name and address must be unique.

```
INSERT INTO companies2 (name, address) Values ('auto repair', '123 Spruce Street');  
Query OK, 1 row affected (0.00 sec)
```

```
INSERT INTO companies2 (name, address) Values ('auto repair', '123 Spruce Street');  
ERROR 1062 (23000): Duplicate entry 'auto repair-123 Spruce Street' for key  
'companies2.name_address'
```

```
INSERT INTO companies2 (name, address) Values ('luigio pizzas', '123 Spruce  
Street');  
Query OK, 1 row affected (0.00 sec)
```

Note how it will not let us create a new row when the combination of columns of name and address are both not unique. But we can input multiple different names to the same address. I.e. many businesses in the same building.

```
CREATE TABLE houses (  
    purchase_price INT NOT NULL,  
    sale_price INT NOT NULL,  
    CONSTRAINT sale_more_than_purchase CHECK (sale_price >= purchase_price)  
);
```

This table is for someone that buys and sells houses. They are not allowed to enter a row where the sale price is less than the purchase price.

```
INSERT INTO houses (purchase_price, sale_price) VALUES (100, 200);  
Query OK, 1 row affected (0.00 sec)
```

```
INSERT INTO houses (purchase_price, sale_price) VALUES (500, 200);  
ERROR 3819 (HY000): Check constraint 'sale_more_than_purchase' is violated.
```

# MySQL – Alter table – Adding Columns

```
ALTER TABLE companies ADD COLUMN city VARCHAR(25);
```

Alter table is a versatile command used to make many different types of changes to a table. In this example we are ADDing a COLUMN

```
SELECT * FROM companies;
```

Because we did not specify it, the default values for existing rows in the table for the new column is NULL.

supplier_id	name	phone	address	city
1	Pet Store	34 222 222 222	dog street, large city, somewhere	NULL
3	cat Store	34 333 333 333	cat drive, village, nowhere	NULL

```
ALTER TABLE companies ADD COLUMN employee_count INT NOT NULL;
```

If I set an interger column to not null when adding a column, the **default integer value is zero**. This can be changed by setting a default value.

```
SELECT * FROM companies;
```

supplier_id	name	phone	address	city	employee_count
1	Pet Store	34 222 222 222	123 dog street	NULL	0
3	cat Store	34 333 333 333	3 Cat Drive	NULL	0

```
ALTER TABLE companies ADD COLUMN employee_count INT NOT NULL DEFAULT 1;
```

# MySQL – Alter table – Drop Columns

```
ALTER TABLE companies DROP COLUMN phone;
```

Drop column will delete a column from a table and remove all the data from that column.

```
SELECT * FROM companies;
```

supplier_id	name	address	city	employee_count
1	Pet Store	123 dog street	NULL	0
3	cat Store	3 Cat Drive	NULL	0

# MySQL – Alter table – Change name

```
ALTER TABLE RENAME companies TO suppliers;
```

A table can be renamed using ALTER TABLE or RENAME syntax.

```
RENAME TABLE companies TO suppliers;
```

```
SELECT * FROM companies;
```

```
ERROR 1146 (42S02): Table 'friends.companies' doesn't exist
```

```
SELECT * FROM suppliers;
```

supplier_id	name	address	city	employee_count
1	Pet Store	123 dog street	NULL	0
3	cat Store	3 Cat Drive	NULL	0



# MySQL – Alter table – rename Columns

```
ALTER TABLE companies RENAME COLUMN name TO company_name;
```

```
SELECT * FROM companies;
```

supplier_id	company_name	address	city	employee_count
1	Pet Store	123 dog street	NULL	0
3	cat Store	3 Cat Drive	NULL	0

ALTER TABLE command can be used to rename a column where the column to be renamed is specified.

# MySQL – Alter table – Modify Columns

```
DESC companies;
```

Field	Type	Null	Key	Default	Extra
supplier_id	int	NO	PRI	NULL	auto_increment
company_name	varchar(255)	NO		NULL	
address	varchar(255)	NO		NULL	
city	varchar(25)	YES		NULL	
employee_count	int	NO		NULL	

A column can be modified to change parameters such as the datatype and length and default value.

```
ALTER TABLE companies MODIFY company_name VARCHAR(100) DEFAULT 'unknown';
```

```
DESC companies;
```

Field	Type	Null	Key	Default	Extra
supplier_id	int	NO	PRI	NULL	auto_increment
company_name	varchar(100)	YES		unknown	
address	varchar(255)	NO		NULL	
city	varchar(25)	YES		NULL	
employee_count	int	NO		NULL	

However, this can be dangerous. For example if company\_name has rows with company names of more than 100 variable characters in length and we modified this to VARCHAR(100) then all the rows with data greater than 100 characters would be truncated.

```
INSERT INTO companies (address, employee_count) VALUES ('26 street', 20);
```

```
SELECT * FROM companies;
```

supplier_id	company_name	address	city	employee_count
1	Pet Store	123 dog street	NULL	0
3	cat Store	3 Cat Drive	NULL	0
4	unknown	26 street	NULL	20

Now we can insert a new row without specifying company name because it has a default of unknown set.

```
ALTER TABLE companies CHANGE address company_address VARCHAR(100) DEFAULT 'unknown';
```

supplier_id	company_name	company_address	city	employee_count
1	Pet Store	123 dog street	NULL	0
3	cat Store	3 Cat Drive	NULL	0
4	unknown	26 street	NULL	20

```
SELECT * FROM companies;
```

```
DESC companies;
```

Field	Type	Null	Key	Default	Extra
supplier_id	int	NO	PRI	NULL	auto_increment
company_name	varchar(100)	YES		unknown	
company_address	varchar(100)	YES		unknown	
city	varchar(25)	YES		NULL	
employee_count	int	NO		NULL	

It is possible to rename a column and change one or more parameters using CHANGE in place of MODIFY where the old column name is specified then the new column name then the parameters.

# MySQL – Alter table – Modify Constraints

```
ALTER TABLE houses ADD CONSTRAINT positive_purchase_price  
CHECK (purchase_price >= 0);
```

I can add a constraint to a column.

```
INSERT INTO houses (purchase_price, sale_price) VALUES (-200, 500);  
ERROR 3819 (HY000): Check constraint 'positive_purchase_price' is violated.
```

```
ALTER TABLE houses DROP CONSTRAINT positive_purchase_price;
```

I can drop a constraint to a column as long as I have the name of the constraint.

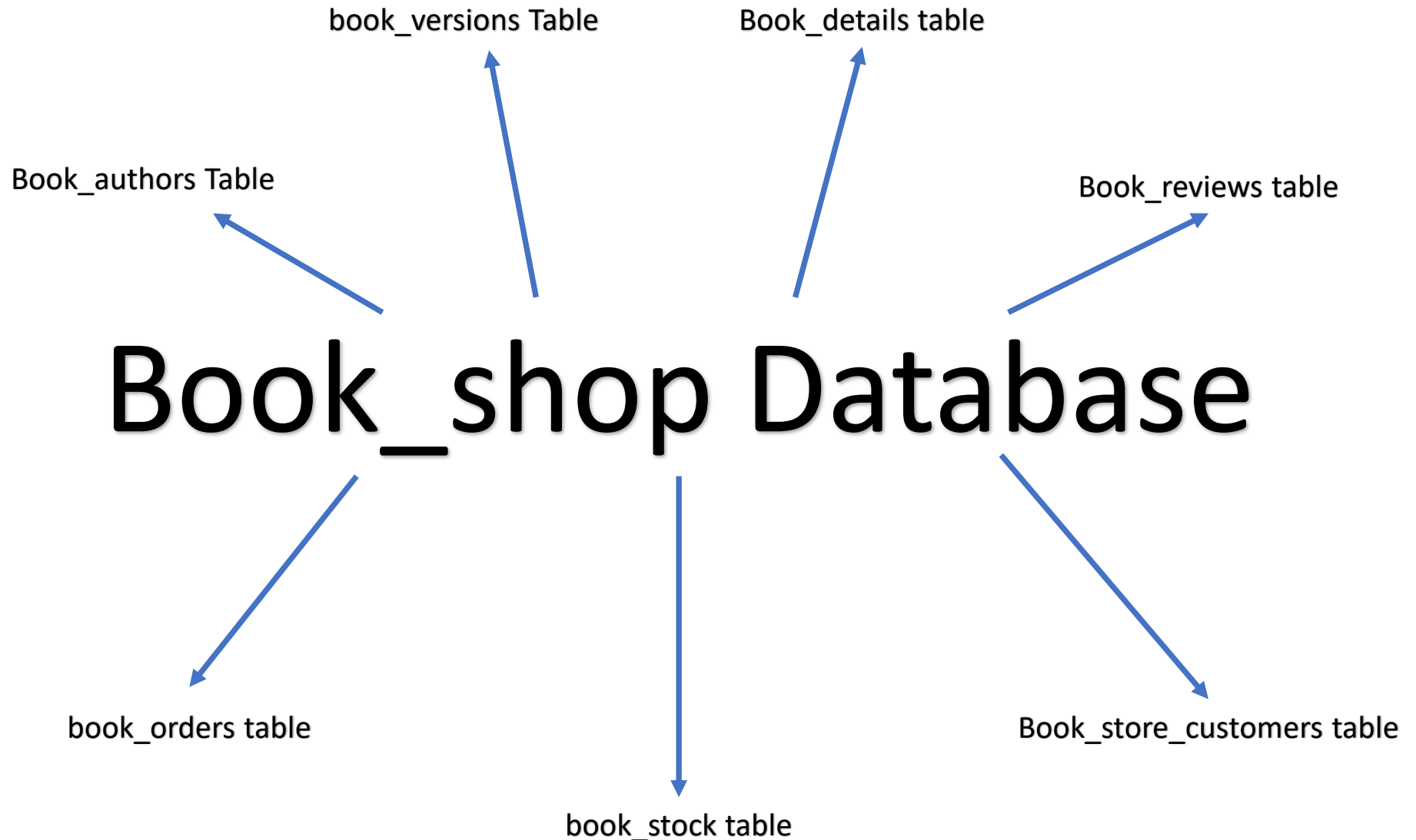
```
INSERT INTO houses (purchase_price, sale_price) VALUES (-200, 500);  
Query OK, 1 row affected (0.00 sec)
```

```
SELECT * FROM houses;
```

	purchase_price	sale_price
	100	200
	-200	500

# One to Many & Joins

# MySQL – Data Relationships



# MySQL – Data Relationships – One to One Relationship

Customer Credentials Table

custCred_username	CustCred_email	custCred_hashedPassword

Here is a table with three columns of the most used customer data.

Customer Details Table

custDet_address	CustDet_dob	custDet_fname	custDet_lname

More data about each customer is stored in the Customer details table

This is a one to one relationship because there can only be one row in the customer details table for each row in the customer credentials table.

# MySQL – Data Relationships – One to Many Relationship

Customer Credentials Table

custCred_username	CustCred_email	custCred_hashedPassword

Here is a table with three columns of the most used customer data.

Customer Orders Table

order_id	order_date	order_amount	Order_fulfilled_date

Here is an orders table

This is a one to Many relationship because there can be multiple orders for each customer but each order can only have one customer.



# MySQL – Data Relationships – One to Many Relationship

Books Table

book_id	book_ISDN	book_publish_date	book_language

Book\_reviews Table

review_id	review_date	reviewer	review_text	Review_rating

This is a one to Many relationship because there can be multiple reviews for each book but each review is only for one specific book.

# MySQL – Data Relationships – Many to Many Relationship

**Books Table**

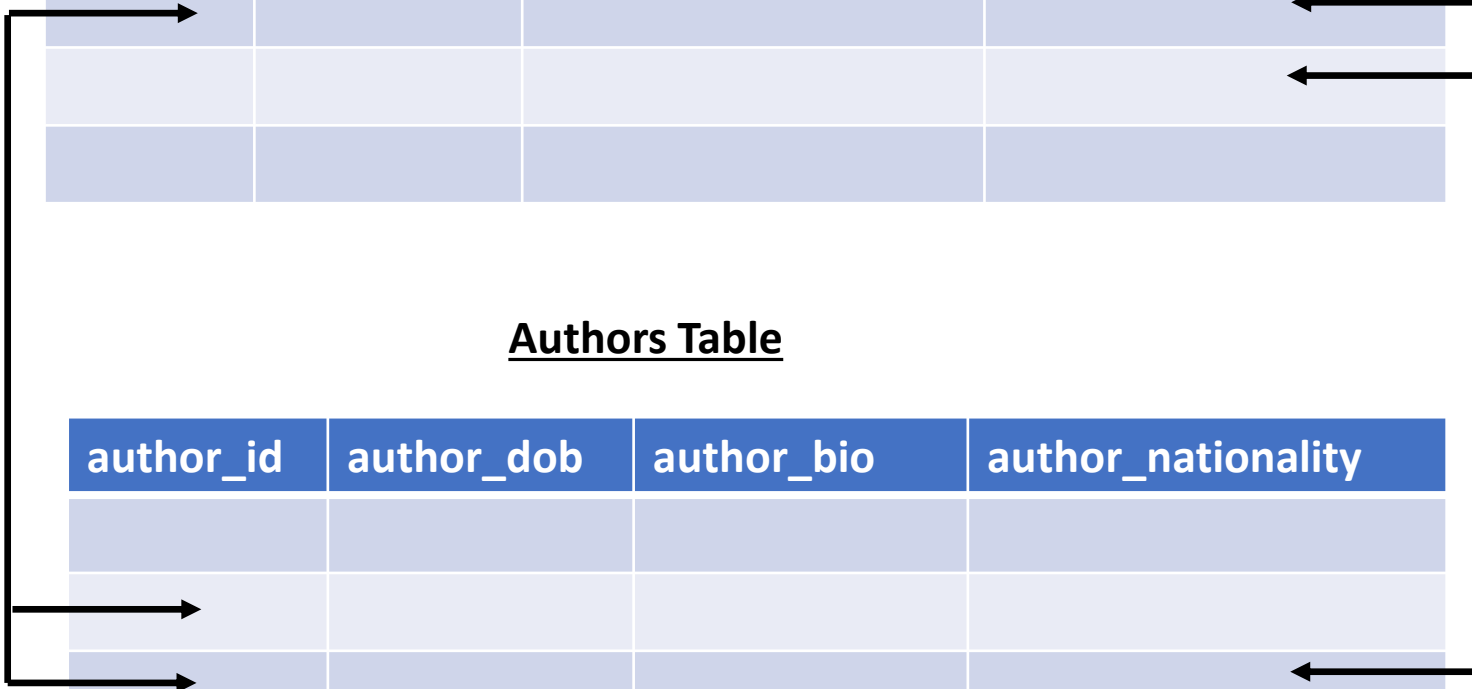
book_id	book_ISDN	book_publish_date	book_language

A book can  
have more  
than one  
author

**Authors Table**

author_id	author_dob	author_bio	author_nationality

An author  
can write  
many books



# MySQL – Data Relationships – One to Many Relationship

Customers Table

customer_id
first_name
last_name
email

Orders Table

order_id
order_date
amount
customer_id

Join

Each order has a customer\_id which is the same as a value in the customer\_id of the customer's table

In the customers table the customer\_id is the **PRIMARY\_KEY**.

In the orders table the Order\_id is the **PRIMARY\_KEY**

In the Orders table the customer\_id is a **FOREIGN\_KEY** because it refers to the primary key of another table.

CUSTOMERS

customer_id	first_name	last_name	email
1	Boy	George	george@gmail.com
2	George	Michael	gm@gmail.com
3	David	Bowie	david@gmail.com
4	Blue	Steele	blue@gmail.com

ORDERS

order_id	order_date	amount	customer_id
1	'2016/02/10'	99.99	1
2	'2017/11/11'	35.50	1
3	'2014/12/12'	800.67	2
4	'2015/01/03'	12.50	2

# MySQL – Data Relationships – Foreign Key

```
CREATE TABLE customers (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    first_name VARCHAR(50),  
    last_name VARCHAR(50),  
    email VARCHAR(50)  
);
```

```
CREATE TABLE orders (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    order_date DATE,  
    amount DECIMAL(8,2),  
    customer_id INT  
);
```

```
INSERT INTO customers (first_name, last_name, email)  
VALUES ('Boy', 'George', 'george@gmail.com'),  
      ('George', 'Michael', 'gm@gmail.com'),  
      ('David', 'Bowie', 'david@gmail.com'),  
      ('Blue', 'Steele', 'blue@gmail.com'),  
      ('Bette', 'Davis', 'bette@gmail.com');
```

```
INSERT INTO orders (order_date, amount, customer_id)  
VALUES ('2016-02-10', '99.99', 1),  
      ('2017-11-11', '35.50', 1),  
      ('2014-12-12', '800.67', 2),  
      ('2015-01-03', '12.50', 2),  
      ('1999-04-11', '450.25', 5);
```

```
SELECT * FROM customers;
```

id	first_name	last_name	email
1	Boy	George	george@gmail.com
2	George	Michael	gm@gmail.com
3	David	Bowie	david@gmail.com
4	Blue	Steele	blue@gmail.com
5	Bette	Davis	bette@gmail.com

```
SELECT * FROM orders;
```

id	order_date	amount	customer_id
1	2016-02-10	99.99	1
2	2017-11-11	35.50	1
3	2014-12-12	800.67	2
4	2015-01-03	12.50	2
5	1999-04-11	450.25	5

Now we have two simple table  
with customers and orders.

```
INSERT INTO orders (order_date, amount, customer_id)
VALUES ('2022-11-11', 50.68, 975);
Query OK, 1 row affected (0.01 sec)
```

We can insert a new order and give it a fake customer\_id and the query runs without any issues.

There is no connection between the customers and orders table.

```
CREATE TABLE orders (
    id INT PRIMARY KEY AUTO_INCREMENT,
    order_date DATE,
    amount DECIMAL(8,2),
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES customers(id)
);
```

To define this connection we have to set the foreign key to a column and then specify what column it references in the customers table.

```
INSERT INTO orders (order_date, amount, customer_id) VALUES ('2022-11-11', 50.68,
975);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails
(`relationships`.`orders`, CONSTRAINT `orders_ibfk_1` FOREIGN KEY (`customer_id`)
REFERENCES `customers` (`id`))
```

```
INSERT INTO orders (order_date, amount, customer_id) VALUES ('2022-11-11', 50.68, 3);
Query OK, 1 row affected (0.01 sec)
```

Now when we try to insert a new order with a fake customer\_id it fails because the fake customer\_id is not in the customers table. However it works when the customer\_id is genuine.

# MySQL – Data Relationships – Cross Joins

```
SELECT * FROM orders WHERE customer_id =  
(SELECT id FROM customers WHERE last_name = 'George');
```

id	order_date	amount	customer_id
1	2016-02-10	99.99	1
2	2017-11-11	35.50	1

```
SELECT * FROM customers, orders;
```

I can run a query that outputs both tables in one but it will give me a duplicate of the same order for each one of the customers.

This query is not really useful because it does not connect specific orders with specific customers.

This is a cross-join because it is cross multiplying both tables together

To find all orders for a particular customer where the customer\_id is not known I could use a sub-query.

id	first_name	last_name	email	id	order_date	amount	customer_id
5	Bette	Davis	bette@gmail.com	1	2016-02-10	99.99	1
4	Blue	Steele	blue@gmail.com	1	2016-02-10	99.99	1
3	David	Bowie	david@gmail.com	1	2016-02-10	99.99	1
2	George	Michael	gm@gmail.com	1	2016-02-10	99.99	1
1	Boy	George	george@gmail.com	1	2016-02-10	99.99	1
5	Bette	Davis	bette@gmail.com	2	2017-11-11	35.50	1
4	Blue	Steele	blue@gmail.com	2	2017-11-11	35.50	1
3	David	Bowie	david@gmail.com	2	2017-11-11	35.50	1
2	George	Michael	gm@gmail.com	2	2017-11-11	35.50	1
1	Boy	George	george@gmail.com	2	2017-11-11	35.50	1
5	Bette	Davis	bette@gmail.com	3	2014-12-12	800.67	2
4	Blue	Steele	blue@gmail.com	3	2014-12-12	800.67	2
3	David	Bowie	david@gmail.com	3	2014-12-12	800.67	2
2	George	Michael	gm@gmail.com	3	2014-12-12	800.67	2
1	Boy	George	george@gmail.com	3	2014-12-12	800.67	2
5	Bette	Davis	bette@gmail.com	4	2015-01-03	12.50	2
4	Blue	Steele	blue@gmail.com	4	2015-01-03	12.50	2
3	David	Bowie	david@gmail.com	4	2015-01-03	12.50	2
2	George	Michael	gm@gmail.com	4	2015-01-03	12.50	2
1	Boy	George	george@gmail.com	4	2015-01-03	12.50	2
5	Bette	Davis	bette@gmail.com	5	1999-04-11	450.25	5
4	Blue	Steele	blue@gmail.com	5	1999-04-11	450.25	5
3	David	Bowie	david@gmail.com	5	1999-04-11	450.25	5
2	George	Michael	gm@gmail.com	5	1999-04-11	450.25	5
1	Boy	George	george@gmail.com	5	1999-04-11	450.25	5
5	Bette	Davis	bette@gmail.com	7	2022-11-11	50.68	3
4	Blue	Steele	blue@gmail.com	7	2022-11-11	50.68	3
3	David	Bowie	david@gmail.com	7	2022-11-11	50.68	3
2	George	Michael	gm@gmail.com	7	2022-11-11	50.68	3
1	Boy	George	george@gmail.com	7	2022-11-11	50.68	3

# MySQL – Data Relationships – Inner Join

```
SELECT * FROM customers JOIN orders ON customers.id = orders.customer_id;
```

id	first_name	last_name	email	id	order_date	amount	customer_id
1	Boy	George	george@gmail.com	1	2016-02-10	99.99	1
1	Boy	George	george@gmail.com	2	2017-11-11	35.50	1
2	George	Michael	gm@gmail.com	3	2014-12-12	800.67	2
2	George	Michael	gm@gmail.com	4	2015-01-03	12.50	2
3	David	Bowie	david@gmail.com	7	2022-11-11	50.68	3
5	Bette	Davis	bette@gmail.com	5	1999-04-11	450.25	5

Select all rows from customers and orders where the join condition is met.

In the orders table the customer\_id column is joined to the id column from the customers table.

We can select all from the customers table where we JOIN ON customer.id is equal to orders.customer\_id. Note how we denote table then column with a dot separating them.

I can also slim down the query to only show the columns I want.

```
SELECT
    CONCAT(first_name, last_name) AS customer_name,
    order_date,
    amount
FROM customers
JOIN orders ON customers.id = orders.customer_id;
```

customer_name	order_date	amount
BoyGeorge	2016-02-10	99.99
BoyGeorge	2017-11-11	35.50
GeorgeMichael	2014-12-12	800.67
GeorgeMichael	2015-01-03	12.50
DavidBowie	2022-11-11	50.68
BetteDavis	1999-04-11	450.25

# MySQL – Data Relationships – Inner Join with Group By

```
SELECT * FROM orders JOIN customers ON customers.id = orders.customer_id;
```

id	order_date	amount	customer_id	id	first_name	last_name	email
1	2016-02-10	99.99	1	1	Boy	George	george@gmail.com
2	2017-11-11	35.50	1	1	Boy	George	george@gmail.com
3	2014-12-12	800.67	2	2	George	Michael	gm@gmail.com
4	2015-01-03	12.50	2	2	George	Michael	gm@gmail.com
7	2022-11-11	50.68	3	3	David	Bowie	david@gmail.com
5	1999-04-11	450.25	5	5	Bette	Davis	bette@gmail.com

The join can be done in reverse where orders is joined with customers.

```
SELECT
    first_name, last_name, SUM(amount) AS total
FROM customers
JOIN orders ON orders.customer_id = customers.id
GROUP BY first_name, last_name
ORDER BY total ASC;
```

first_name	last_name	total
Boy	George	135.49
Bette	Davis	450.25
George	Michael	813.17

We can make a total of amounts from the orders table per customer by using group by on the customers table.



# MySQL – Data Relationships – Left Join

```
SELECT first_name, last_name, order_date, amount FROM customers LEFT JOIN orders ON  
orders.customer_id = customers.id;
```

first_name	last_name	order_date	amount
Boy	George	2016-02-10	99.99
Boy	George	2017-11-11	35.50
George	Michael	2014-12-12	800.67
George	Michael	2015-01-03	12.50
David	Bowie	NULL	NULL
Blue	Steele	NULL	NULL
Bette	Davis	1999-04-11	450.25

With a left join we are selecting all rows from the left side which is the customers table and then matching records from the right side, Orders table. Note that where there is not match, the right side is NULL.

Note that this differs from an inner join because we are matching from left and right with an inner and only displaying where they overlap.

This can be usefull in certain scenarios such as identifying customers that have not placed orders.  
i.e. NULLs

```
SELECT order_date, amount, first_name, last_name FROM orders LEFT JOIN customers ON  
customers.id = orders.customer_id;
```

order_date	amount	first_name	last_name
2016-02-10	99.99	Boy	George
2017-11-11	35.50	Boy	George
2014-12-12	800.67	George	Michael
2015-01-03	12.50	George	Michael
1999-04-11	450.25	Bette	Davis

If the left Join is switched around then we get the same result as an inner join. This is because an order row can only exist if a customer exists.

# MySQL – Data Relationships – Left Join with Group By

```
SELECT
    first_name, last_name, SUM(amount) AS total
FROM customers
LEFT JOIN orders ON orders.customer_id = customers.id
GROUP BY first_name, last_name
ORDER BY total;
```

Now when we do a group by we can sum the orders per customer as before but we also have the customers that have not made an order showing up as NULL.

first_name	last_name	total
David	Bowie	NULL
Blue	Steele	NULL
Boy	George	135.49
Bette	Davis	450.25
George	Michael	813.17

We can replace the NULL with something more relevant such as zero using the IFNULL() function

first_name	last_name	total
David	Bowie	0.00
Blue	Steele	0.00
Boy	George	135.49
Bette	Davis	450.25
George	Michael	813.17

```
SELECT
    first_name, last_name, IFNULL(SUM(amount), 0) AS total
FROM customers
LEFT JOIN orders ON orders.customer_id = customers.id
GROUP BY first_name, last_name
ORDER BY total;
```

Where IFNULL(*what we want to check, the new value*)

# MySQL – Data Relationships – Right Join

```
INSERT INTO orders (order_date, amount) VALUES (CURDATE(), 100);  
Query OK, 1 row affected (0.01 sec)
```

id	order_date	amount	customer_id
1	2016-02-10	99.99	1
2	2017-11-11	35.50	1
3	2014-12-12	800.67	2
4	2015-01-03	12.50	2
5	1999-04-11	450.25	5
8	2022-11-11	100.00	NULL

We created the orders table with a constraint of a foreign key joining customer.id and orders.customer\_id so we cannot enter in a row to orders where the customer\_id is not in the customers table so why did the above row insertion work?

```
DESC orders;
```

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
order_date	date	YES		NULL	
amount	decimal(8,2)	YES		NULL	
customer_id	int	YES	MUL	NULL	

Looking at the Schema, customer\_id permits NULL as a value. If this was a real world scenario then this should be set to NOT NULL to prevent dummy customer\_id's in the orders table.

This is, however, usefull to demonstrate a right join.

```
SELECT first_name, last_name, order_date, amount FROM customers
RIGHT JOIN orders ON customers.id = customer_id;
```

first_name	last_name	order_date	amount
Boy	George	2016-02-10	99.99
Boy	George	2017-11-11	35.50
George	Michael	2014-12-12	800.67
George	Michael	2015-01-03	12.50
Bette	Davis	1999-04-11	450.25
NULL	NULL	2022-11-11	100.00

With a right join we are selecting all records on the right side, orders table and matching records on the left side, Customers table.

## MySQL – Data Relationships – On delete Cascade

Customers Table

customer_id
first_name
last_name
email

Orders Table

order_id
order_date
amount
customer_id

Join

We can have customers with out orders but not orders without customers.

But what happens when we delete a customer with orders?

```
DELETE FROM customers where last_name = "george";  
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key  
constraint fails (`relationships`.`orders`, CONSTRAINT `orders_ibfk_1`  
FOREIGN KEY (`customer_id`) REFERENCES `customers` (`id`))
```

Customer Boy George has two orders in the orders table. When we try and delete Boy George it fails because of a foreign key constraint. This is because the orders table is defined to not allow rows where the customer\_id is not equal to an id from the customer table. This would equally fail if we tried to drop the whole customers table because of the link from orders table.

There are multiple options to solve this such as updating the customer id to something like NULL or unknow or deleted upon deletion of customer. However there is an option to delete any orders related to corresponding customer upon deletion of customer.

```
CREATE TABLE orders (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    order_date DATE,  
    amount DECIMAL(8,2),  
    customer_id INT,  
    FOREIGN KEY (customer_id) REFERENCES customers(id) ON DELETE CASCADE  
);
```

Add the option of **ON DELETE CASCADE** to the foreign key of the orders table.

```
DELETE FROM customers where last_name = "george";  
Query OK, 1 row affected (0.01 sec)
```

Now when we delete a customer the deletion cascades down to the orders table and deletes corresponding orders rows as well.

```
SELECT * FROM customers;
```

id	first_name	last_name	email
2	George	Michael	gm@gmail.com
3	David	Bowie	david@gmail.com
4	Blue	Steele	blue@gmail.com
5	Bette	Davis	bette@gmail.com

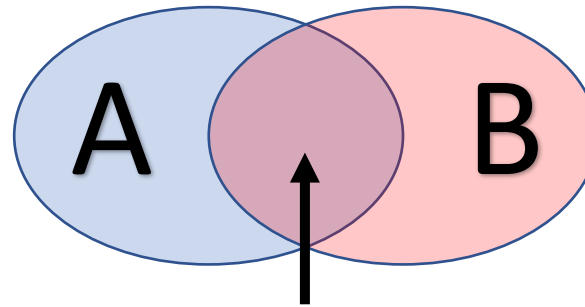
```
SELECT * FROM orders;
```

id	order_date	amount	customer_id
3	2014-12-12	800.67	2
4	2015-01-03	12.50	2
5	1999-04-11	450.25	5

This is not always a desirable thing. We may want a record of orders, even if the customer leaves but in certain situations it is useful.

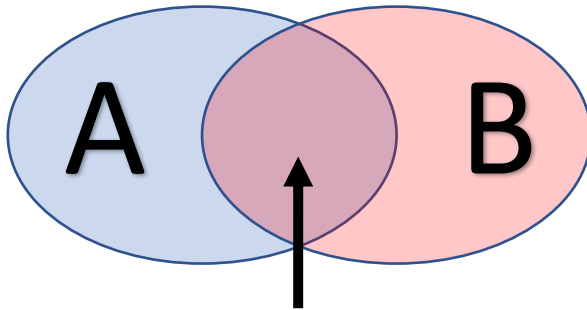
# MySQL – Data Relationships – Joins summary

## Inner Join



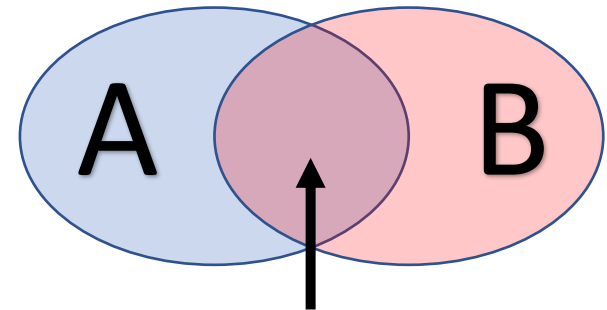
Select all records from A & B  
where the Join condition is met.

## Left Join



Select Everything from A, along  
with any matching records in B.

## Right Join



Select Everything from B, along  
with any matching records in A.

# MySQL – Data Relationships – Joins Exercises

```
CREATE TABLE students (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    first_name VARCHAR(50)  
);  
  
CREATE TABLE papers (  
    title VARCHAR(100),  
    grade TINYINT(3),  
    student_id INT,  
    FOREIGN KEY (student_id) REFERENCES students(id)  
);  
  
INSERT INTO students (first_name) VALUES  
('Caleb'), ('Samantha'), ('Raj'), ('Carlos'), ('Lisa');  
  
INSERT INTO papers (student_id, title, grade ) VALUES  
(1, 'My First Book Report', 60),  
(1, 'My Second Book Report', 75),  
(2, 'Russian Lit Through The Ages', 94),  
(2, 'De Montaigne and The Art of The Essay', 98),  
(4, 'Borges and Magical Realism', 89);
```



```
SELECT first_name, title, grade FROM students JOIN papers ON students.id
= papers.student_id ORDER BY grade DESC;
```

first_name	title	grade
Samantha	De Montaigne and The Art of The Essay	98
Samantha	Russian Lit Through The Ages	94
Carlos	Borges and Magical Realism	89
Caleb	My Second Book Report	75
Caleb	My First Book Report	60

```
SELECT first_name, IFNULL(title, 'Missing', IFNULL(grade, 0)) FROM students
LEFT JOIN papers ON students.id = papers.student_id;
```

first_name	title	grade
Caleb	My First Book Report	60
Caleb	My Second Book Report	75
Samantha	Russian Lit Through The Ages	94
Samantha	De Montaigne and The Art of The Essay	98
Raj	NULL	NULL
Carlos	Borges and Magical Realism	89
Lisa	NULL	NULL

```
SELECT first_name, IFNULL(title, 'Missing'), IFNULL(grade, 0) FROM students
LEFT JOIN papers ON students.id = papers.student_id;
```

first_name	IFNULL(title, 'Missing')	IFNULL(grade, 0)
Caleb	My First Book Report	60
Caleb	My Second Book Report	75
Samantha	Russian Lit Through The Ages	94
Samantha	De Montaigne and The Art of The Essay	98
Raj	Missing	0
Carlos	Borges and Magical Realism	89
Lisa	Missing	0

```
SELECT
    first_name, AVG(IFNULL(grade, 0)) AS average
FROM students
LEFT JOIN papers ON students.id = papers.student_id
GROUP BY first_name
ORDER BY average DESC;
```

first_name	average
Samantha	96.0000
Carlos	89.0000
Caleb	67.5000
Raj	0.0000
Lisa	0.0000

```

SELECT
    first_name, IFNULL(AVG(grade), 0) AS average,
    CASE
        WHEN IFNULL(AVG(grade), 0) >= 75 THEN 'passing'
        ELSE 'failing'
    END AS passing_status
FROM students
LEFT JOIN papers ON students.id = papers.student_id
GROUP BY first_name
ORDER BY average DESC;

```

first_name	average	passing_status
Samantha	96.0000	passing
Carlos	89.0000	passing
Caleb	67.5000	failing
Raj	0.0000	failing
Lisa	0.0000	failing

# Many to Many & Joins

# MySQL – Data Relationships – Many to Many Overview

Books  $\leftrightarrow$  Authors  
Blog Posts  $\leftrightarrow$  Tags  
Students  $\leftrightarrow$  Classes

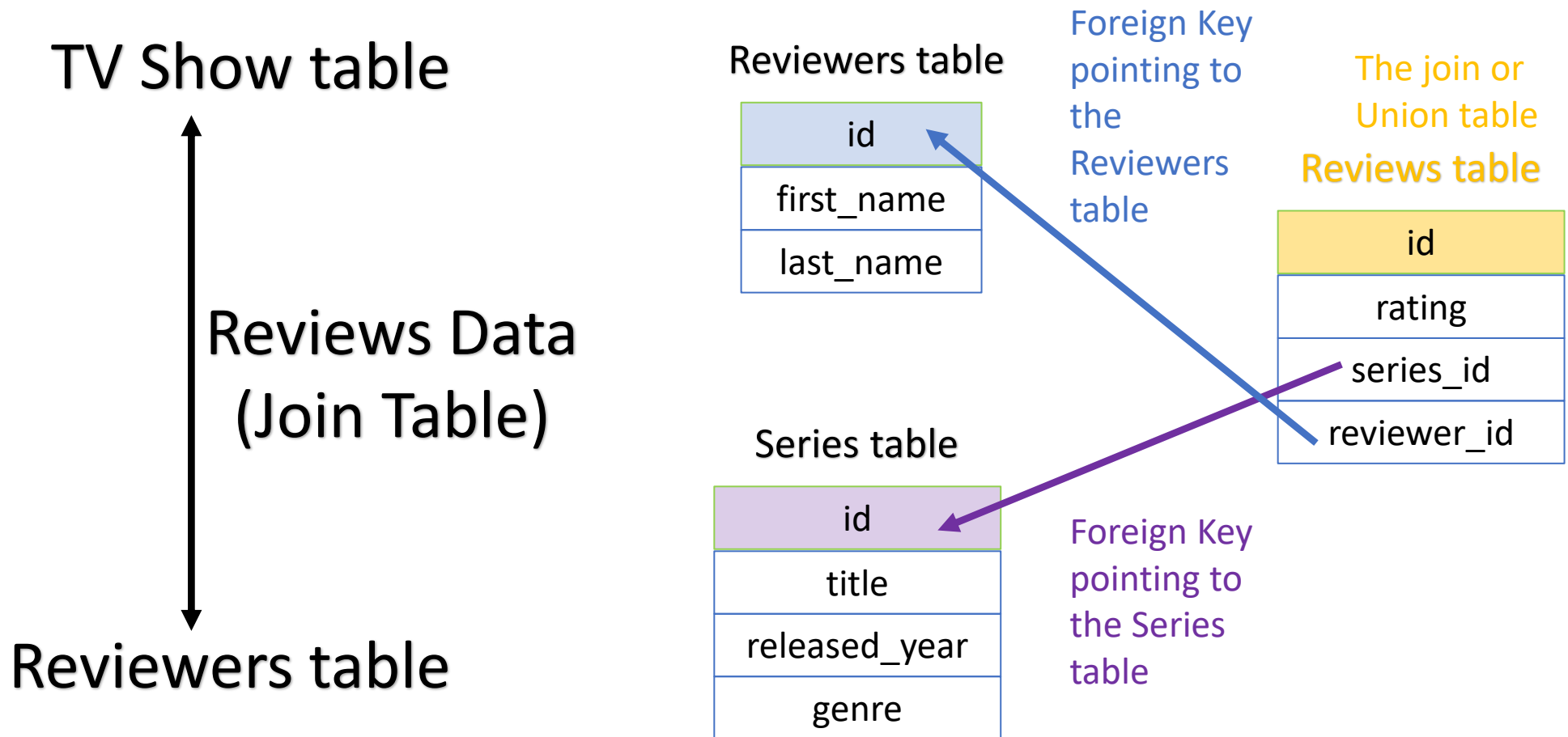
A book can have multiple authors and an Author can have multiple books.

Blog posts can have multiple tags and Tags can apply to multiple posts.

Students can have multiple classes and Classes can have multiple students.

# TV Show ↔ Reviewers

A TV show can have multiple reviewers and Reviewers can rate multiple TV shows.



# MySQL – Data Relationships – Many to Many tables

```
CREATE TABLE reviewers (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    first_name VARCHAR(50) NOT NULL,  
    last_name VARCHAR(50) NOT NULL  
);
```

```
CREATE TABLE series (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    title VARCHAR(100),  
    released_year YEAR,  
    genre VARCHAR(100)  
);
```

```
CREATE TABLE reviews (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    rating DECIMAL(2 , 1 ),  
    series_id INT,  
    reviewer_id INT,  
    FOREIGN KEY (series_id)  
        REFERENCES series (id),  
    FOREIGN KEY (reviewer_id)  
        REFERENCES reviewers (id)  
);
```

```
INSERT INTO series (title, released_year,  
genre) VALUES  
    ('Archer', 2009, 'Animation'),  
    ('Arrested Development', 2003, 'Comedy'),  
    ("Bob's Burgers", 2011, 'Animation'),  
    ('Bojack Horseman', 2014, 'Animation'),  
    ("Breaking Bad", 2008, 'Drama'),  
    ('Curb Your Enthusiasm', 2000, 'Comedy'),  
    ("Fargo", 2014, 'Drama'),  
    ('Freaks and Geeks', 1999, 'Comedy'),  
    ('General Hospital', 1963, 'Drama'),  
    ('Halt and Catch Fire', 2014, 'Drama'),  
    ('Malcolm In The Middle', 2000, 'Comedy'),  
    ('Pushing Daisies', 2007, 'Comedy'),  
    ('Seinfeld', 1989, 'Comedy'),  
    ('Stranger Things', 2016, 'Drama');
```

```
INSERT INTO reviewers (first_name, last_name)
VALUES
    ('Thomas', 'Stoneman'),
    ('Wyatt', 'Skaggs'),
    ('Kimbra', 'Masters'),
    ('Domingo', 'Cortes'),
    ('Colt', 'Steele'),
    ('Pinkie', 'Petit'),
    ('Marlon', 'Crafford');
```

```
INSERT INTO reviews(series_id, reviewer_id, rating) VALUES
    (1,1,8.0),(1,2,7.5),(1,3,8.5),(1,4,7.7),(1,5,8.9),
    (2,1,8.1),(2,4,6.0),(2,3,8.0),(2,6,8.4),(2,5,9.9),
    (3,1,7.0),(3,6,7.5),(3,4,8.0),(3,3,7.1),(3,5,8.0),
    (4,1,7.5),(4,3,7.8),(4,4,8.3),(4,2,7.6),(4,5,8.5),
    (5,1,9.5),(5,3,9.0),(5,4,9.1),(5,2,9.3),(5,5,9.9),
    (6,2,6.5),(6,3,7.8),(6,4,8.8),(6,2,8.4),(6,5,9.1),
    (7,2,9.1),(7,5,9.7),
    (8,4,8.5),(8,2,7.8),(8,6,8.8),(8,5,9.3),
    (9,2,5.5),(9,3,6.8),(9,4,5.8),(9,6,4.3),(9,5,4.5),
    (10,5,9.9),
    (13,3,8.0),(13,4,7.2),
    (14,2,8.5),(14,3,8.9),(14,4,8.9);
```



```
SELECT title, rating FROM series JOIN reviews ON series.id = reviews.series_id;
```

title	rating		
		Curb Your Enthusiasm	6.5
		Curb Your Enthusiasm	7.8
Archer	8.0	Curb Your Enthusiasm	8.8
Archer	7.5	Curb Your Enthusiasm	8.4
Archer	8.5	Curb Your Enthusiasm	9.1
Archer	7.7	Fargo	9.1
Archer	8.9	Fargo	9.7
Arrested Development	8.1	Freaks and Geeks	8.5
Arrested Development	6.0	Freaks and Geeks	7.8
Arrested Development	8.0	Freaks and Geeks	8.8
Arrested Development	8.4	Freaks and Geeks	9.3
Arrested Development	9.9	General Hospital	5.5
Bob's Burgers	7.0	General Hospital	6.8
Bob's Burgers	7.5	General Hospital	5.8
Bob's Burgers	8.0	General Hospital	4.3
Bob's Burgers	7.1	General Hospital	4.5
Bob's Burgers	8.0	Halt and Catch Fire	9.9
Bojack Horseman	7.5	Seinfeld	8.0
Bojack Horseman	7.8	Seinfeld	7.2
Bojack Horseman	8.3	Stranger Things	8.5
Bojack Horseman	7.6	Stranger Things	8.9
Bojack Horseman	8.5	Stranger Things	8.9
Breaking Bad	9.5		
Breaking Bad	9.0		
Breaking Bad	9.1		
Breaking Bad	9.3		
Breaking Bad	9.9		

Using an inner join we can get all the ratings for each tv show.

```
SELECT
    title, ROUND(AVG(rating), 2) AS avg_rating
FROM series
JOIN reviews ON series.id = reviews.series_id
GROUP BY title
ORDER BY avg_rating DESC;
```

Using inner join and group by we can get a list of average ratings for each tv show. Note the ROUND function where the 2<sup>nd</sup> parameter is the number of decimal places.

title	avg_rating
Halt and Catch Fire	9.90000
Fargo	9.40000
Breaking Bad	9.36000
Stranger Things	8.76667
Freaks and Geeks	8.60000
Archer	8.12000
Curb Your Enthusiasm	8.12000
Arrested Development	8.08000
Bojack Horseman	7.94000
Seinfeld	7.60000
Bob's Burgers	7.52000
General Hospital	5.38000

```
SELECT title AS unreviewed_series
FROM series
LEFT JOIN reviews ON series.id = reviews.series_id
WHERE rating IS NULL;
```

unreviewed_series
Malcolm In The Middle
Pushing Daisies

A left join can be used to find all series that do not have a review.

```
SELECT title AS unreviewed_series
FROM reviews
RIGHT JOIN series ON series.id = reviews.series_id
WHERE rating IS NULL;
```

unreviewed_series
Malcolm In The Middle
Pushing Daisies

This can also be achieved with a right join

```
SELECT first_name, last_name, rating FROM reviewers JOIN reviews ON
reviews.reviewer_id = reviewers.id;
```

first_name	last_name	rating	Kimbra	Masters	8.0	Domingo	Cortes	8.9
Thomas	Stoneman	8.0	Kimbra	Masters	7.1	Colt	Steele	8.9
Thomas	Stoneman	8.1	Kimbra	Masters	7.8	Colt	Steele	9.9
Thomas	Stoneman	7.0	Kimbra	Masters	9.0	Colt	Steele	8.0
Thomas	Stoneman	7.5	Kimbra	Masters	7.8	Colt	Steele	8.5
Thomas	Stoneman	9.5	Kimbra	Masters	6.8	Colt	Steele	9.9
Wyatt	Skaggs	7.5	Kimbra	Masters	8.0	Colt	Steele	9.1
Wyatt	Skaggs	7.6	Kimbra	Masters	8.9	Colt	Steele	9.7
Wyatt	Skaggs	9.3	Domingo	Cortes	7.7	Colt	Steele	9.3
Wyatt	Skaggs	6.5	Domingo	Cortes	6.0	Colt	Steele	4.5
Wyatt	Skaggs	8.4	Domingo	Cortes	8.0	Colt	Steele	9.9
Wyatt	Skaggs	9.1	Domingo	Cortes	8.3	Pinkie	Petit	8.4
Wyatt	Skaggs	7.8	Domingo	Cortes	9.1	Pinkie	Petit	7.5
Wyatt	Skaggs	5.5	Domingo	Cortes	8.8	Pinkie	Petit	8.8
Wyatt	Skaggs	8.5	Domingo	Cortes	8.5	Pinkie	Petit	4.3
Kimbra	Masters	8.5	Domingo	Cortes	5.8			
					7.2			

Using inner join and we can get a list reviews by first name and last name.

```
SELECT
    genre, ROUND(AVG(rating), 2) AS avg_rating
FROM series
JOIN reviews ON series.id = reviews.series_id
GROUP BY genre
ORDER BY avg_rating DESC;
```

genre	avg_rating
Comedy	8.16
Drama	8.04
Animation	7.86

Here we use an inner join to group genre by average rating rounded to two decimal places.

```

SELECT
    first_name,
    last_name,
    COUNT(rating) AS num_ratings,
    IFNULL(ROUND(AVG(rating), 2), 0) AS avg_rating,
    IFNULL(MIN(rating), 0) as min_rating,
    IFNULL(MAX(rating), 0) AS max_rating,
    CASE
        WHEN COUNT(rating) > 0 THEN 'ACTIVE'
        ELSE 'INACTIVE'
    END AS reviewer_status
FROM reviewers
LEFT JOIN reviews ON reviewers.id = reviews.reviewer_id
GROUP BY first_name, last_name;

```

Using a left join and some inbuilt functions we can get a ratings summary table per reviewer name.

We can use case for the NULL values to set reviewer status to active or inactive. i.e. no reviews is an inactive reviewer.

first_name	last_name	num_ratings	avg_rating	min_rating	max_rating	reviewer_status
Thomas	Stoneman	5	8.02	7.0	9.5	ACTIVE
Wyatt	Skaggs	9	7.80	5.5	9.3	ACTIVE
Kimbra	Masters	9	7.99	6.8	9.0	ACTIVE
Domingo	Cortes	10	7.83	5.8	9.1	ACTIVE
Colt	Steele	10	8.77	4.5	9.9	ACTIVE
Pinkie	Petit	4	7.25	4.3	8.8	ACTIVE
Marlon	Crafford	0	0.00	0.0	0.0	INACTIVE

```

SELECT
    first_name,
    last_name,
    COUNT(rating) AS num_ratings,
    IFNULL(ROUND(AVG(rating), 2), 0) AS avg_rating,
    IFNULL(MIN(rating), 0) AS min_rating,
    IFNULL(MAX(rating), 0) AS max_rating,
    IF (COUNT(rating) > 0, 'ACTIVE', 'INACTIVE') AS reviewer_status

FROM reviewers
LEFT JOIN reviews ON reviewers.id = reviews.reviewer_id
GROUP BY first_name, last_name;

```

first_name	last_name	num_ratings	avg_rating	min_rating	max_rating	reviewer_status
Thomas	Stoneman	5	8.02	7.0	9.5	ACTIVE
Wyatt	Skaggs	9	7.80	5.5	9.3	ACTIVE
Kimbra	Masters	9	7.99	6.8	9.0	ACTIVE
Domingo	Cortes	10	7.83	5.8	9.1	ACTIVE
Colt	Steele	10	8.77	4.5	9.9	ACTIVE
Pinkie	Petit	4	7.25	4.3	8.8	ACTIVE
Marlon	Crafford	0	0.00	0.0	0.0	INACTIVE

Note that where the condition has only two possibilities, i.e inactive and active then an **IF statement** can be used rather than a case which is better suited for multiple conditions.

```

SELECT
    first_name,
    last_name,
    COUNT(rating) AS num_ratings,
    IFNULL(ROUND(AVG(rating), 2), 0) AS avg_rating,
    IFNULL(MIN(rating), 0) as min_rating,
    IFNULL(MAX(rating), 0) AS max_rating,
    CASE
        WHEN COUNT(rating) >= 10 THEN 'POWERUSER'
        WHEN COUNT(rating) > 0 THEN 'ACTIVE'
        ELSE 'INACTIVE'
    END AS reviewer_status
FROM reviewers
LEFT JOIN reviews ON reviewers.id = reviews.reviewer_id
GROUP BY first_name, last_name;

```

An example of using case with three conditions.

first_name	last_name	num_ratings	avg_rating	min_rating	max_rating	reviewer_status
Thomas	Stoneman	5	8.02	7.0	9.5	ACTIVE
Wyatt	Skaggs	9	7.80	5.5	9.3	ACTIVE
Kimbra	Masters	9	7.99	6.8	9.0	ACTIVE
Domingo	Cortes	10	7.83	5.8	9.1	POWERUSER
Colt	Steele	10	8.77	4.5	9.9	POWERUSER
Pinkie	Petit	4	7.25	4.3	8.8	ACTIVE
Marlon	Crafford	0	0.00	0.0	0.0	INACTIVE

```

SELECT title, rating, CONCAT(first_name, ' ', last_name) AS reviewer
FROM reviews
JOIN series ON reviews.series_id = series.id
JOIN reviewers ON reviews.reviewer_id = reviewers.id

```

title	rating	reviewer			
Archer	8.0	Thomas Stoneman	Archer	7.7	Domingo Cortes
Arrested Development	8.1	Thomas Stoneman	Arrested Development	6.0	Domingo Cortes
Bob's Burgers	7.0	Thomas Stoneman	Bob's Burgers	8.0	Domingo Cortes
Bojack Horseman	7.5	Thomas Stoneman	Bojack Horseman	8.3	Domingo Cortes
Breaking Bad	9.5	Thomas Stoneman	Breaking Bad	9.1	Domingo Cortes
Archer	7.5	Wyatt Skaggs	Curb Your Enthusiasm	8.8	Domingo Cortes
Bojack Horseman	7.6	Wyatt Skaggs	Freaks and Geeks	8.5	Domingo Cortes
Breaking Bad	9.3	Wyatt Skaggs	General Hospital	5.8	Domingo Cortes
Curb Your Enthusiasm	6.5	Wyatt Skaggs	Seinfeld	7.2	Domingo Cortes
Curb Your Enthusiasm	8.4	Wyatt Skaggs	Stranger Things	8.9	Domingo Cortes
Fargo	9.1	Wyatt Skaggs	Archer	8.9	Colt Steele
Freaks and Geeks	7.8	Wyatt Skaggs	Arrested Development	9.9	Colt Steele
General Hospital	5.5	Wyatt Skaggs	Bob's Burgers	8.0	Colt Steele
Stranger Things	8.5	Wyatt Skaggs	Bojack Horseman	8.5	Colt Steele
Archer	8.5	Kimbra Masters	Breaking Bad	9.9	Colt Steele
Arrested Development	8.0	Kimbra Masters	Curb Your Enthusiasm	9.1	Colt Steele
Bob's Burgers	7.1	Kimbra Masters	Fargo	9.7	Colt Steele
Bojack Horseman	7.8	Kimbra Masters	Freaks and Geeks	9.3	Colt Steele
Breaking Bad	9.0	Kimbra Masters	General Hospital	4.5	Colt Steele
Curb Your Enthusiasm	7.8	Kimbra Masters	Halt and Catch Fire	9.9	Colt Steele
General Hospital	6.8	Kimbra Masters	Arrested Development	8.4	Pinkie Petit
Seinfeld	8.0	Kimbra Masters	Bob's Burgers	7.5	Pinkie Petit
Stranger Things	8.9	Kimbra Masters	Freaks and Geeks	8.8	Pinkie Petit
			General Hospital	4.3	Pinkie Petit

Here we are combining data from all three tables using two inner joins.

# Introducing Views



# MySQL – Views

```
SELECT
    title, released_year, genre, rating, CONCAT(first_name, ' ', last_name) AS reviewer
FROM reviews
JOIN series ON reviews.series_id = series.id
JOIN reviewers ON reviews.reviewer_id = reviewers.id;
```

title	released_year	genre	rating	reviewer
Archer	2009	Animation	8.0	Thomas Stoneman
Arrested Development	2003	Comedy	8.1	Thomas Stoneman
Bob's Burgers	2011	Animation	7.0	Thomas Stoneman
Bojack Horseman	2014	Animation	7.5	Thomas Stoneman
Breaking Bad	2008	Drama	9.5	Thomas Stoneman
Archer	2009	Animation	7.5	Wyatt Skaggs
Bojack Horseman	2014	Animation	7.6	Wyatt Skaggs
. . . . .				
. . . . .				
. . . . .				
Bob's Burgers	2011	Animation	7.5	Pinkie Petit
Freaks and Geeks	1999	Comedy	8.8	Pinkie Petit
General Hospital	1963	Drama	4.3	Pinkie Petit

47 rows in set (0.00 sec)

The above SQL snippet gives us a long table of 47 rows which might be the starting point for many other queries involving groupings or orderby or CASE etc...

To not repeat code this SQL code snippet can be saved as a virtual table called a **VIEW**.

```
CREATE VIEW full_reviews AS
```

```
SELECT
```

```
    title, released_year, genre, rating, CONCAT(first_name, ' ', last_name) AS reviewer
-> FROM reviews
-> JOIN series ON reviews.series_id = series.id
-> JOIN reviewers ON reviews.reviewer_id = reviewers.id;
```

```
SHOW TABLES;
```

```
+-----+
| Tables_in_tv_db |
+-----+
| full_reviews    |
| reviewers       |
| reviews        |
| series          |
+-----+
```

We can create a view giving it a name, i.e. full\_review then using AS to specify the SQL query. The view will show up as a table even though it is a virtual table and can be queried as if it was a normal table and used for other queries like groupings.

```
SELECT * FROM full_reviews;
```

```
+-----+-----+-----+-----+-----+
| title                | released_year | genre    | rating | reviewer          |
+-----+-----+-----+-----+-----+
| Archer               | 2009          | Animation | 8.0     | Thomas Stoneman  |
| Arrested Development | 2003          | Comedy   | 8.1     | Thomas Stoneman  |
| Bob's Burgers        | 2011          | Animation | 7.0     | Thomas Stoneman  |
| . . .               |               |          |         |                   |
| . . .               |               |          |         |                   |
| . . .               |               |          |         |                   |
| Bob's Burgers        | 2011          | Animation | 7.5     | Pinkie Petit     |
| Freaks and Geeks     | 1999          | Comedy   | 8.8     | Pinkie Petit     |
| General Hospital     | 1963          | Drama    | 4.3     | Pinkie Petit     |
+-----+-----+-----+-----+-----+
47 rows in set (0.00 sec)
```

```
SELECT
    genre, AVG(rating)
FROM full_reviews
GROUP BY genre;
```

```
+-----+-----+
| genre    | AVG(rating) |
+-----+-----+
| Animation | 7.86000     |
| Comedy   | 8.16250     |
| Drama    | 8.04375     |
+-----+-----+
```

# MySQL – Updatable Views

```
SELECT *  
FROM full_reviews;
```

Because a VIEW is not a real table there are limitations on what we can do to the data in it.

title	released_year	genre	rating	reviewer
Archer	2009	Animation	8.0	Thomas Stoneman
Arrested Development	2003	Comedy	8.1	Thomas Stoneman
Bob's Burgers	2011	Animation	7.0	Thomas Stoneman
. . . .				
. . . .				
. . . .				
Bob's Burgers	2011	Animation	7.5	Pinkie Petit
Freaks and Geeks	1999	Comedy	8.8	Pinkie Petit
General Hospital	1963	Drama	4.3	Pinkie Petit

47 rows in set (0.00 sec)

```
DELETE FROM full_reviews WHERE released_uear = 2010;  
ERROR 1395 (HY000): Can not delete from join view 'tv_db.full_reviews'
```

<https://dev.mysql.com/doc/refman/8.0/en/view-updatability.html>

A view is not updatable if it contains any of the following:

Aggregate functions or window functions (SUM(), MIN(), MAX(), COUNT(), and so forth) | DISTINCT | GROUP BY | HAVING | UNION or UNION ALL | Subquery in the select list (Nondependent subqueries in the select list fail for INSERT, but are okay for UPDATE, DELETE. For dependent subqueries in the select list, no data change statements are permitted.) | Certain joins (see additional join discussion later in this section) | Reference to nonupdatable view in the FROM clause | Subquery in the WHERE clause that refers to a table in the FROM clause | Refers only to literal values (in this case, there is no underlying table to update) | ALGORITHM = TEMPTABLE (use of a temporary table always makes a view nonupdatable) | Multiple references to any column of a base table (fails for INSERT, okay for UPDATE, DELETE)

```
CREATE VIEW sorted_series AS SELECT * FROM series ORDER BY released_year;
```

```
SELECT * FROM sorted_series;
```

id	title	released_year	genre
9	General Hospital	1963	Drama
13	Seinfeld	1989	Comedy
8	Freaks and Geeks	1999	Comedy
6	Curb Your Enthusiasm	2000	Comedy
11	Malcolm In The Middle	2000	Comedy
2	Arrested Development	2003	Comedy
12	Pushing Daisies	2007	Comedy
5	Breaking Bad	2008	Drama
1	Archer	2009	Animation
3	Bob's Burgers	2011	Animation
4	Bojack Horseman	2014	Animation
7	Fargo	2014	Drama
10	Halt and Catch Fire	2014	Drama
14	Stranger Things	2016	Drama

It is possible to manipulate data in some views. In this example we can insert into our sorted\_series view.

```
INSERT INTO sorted_series (title, released_year, genre)  
VALUES ('The great', 2020, 'Comedy');
```

```
SELECT * FROM sorted_series;
```

Notice that although we inserted into the view the underlying table was also updated with the new row.

```
SELECT * FROM series;
```

id	title	released_year	genre
9	General Hospital	1963	Drama
13	Seinfeld	1989	Comedy
8	Freaks and Geeks	1999	Comedy
6	Curb Your Enthusiasm	2000	Comedy
11	Malcolm In The Middle	2000	Comedy
2	Arrested Development	2003	Comedy
12	Pushing Daisies	2007	Comedy
5	Breaking Bad	2008	Drama
1	Archer	2009	Animation
3	Bob's Burgers	2011	Animation
4	Bojack Horseman	2014	Animation
7	Fargo	2014	Drama
10	Halt and Catch Fire	2014	Drama
14	Stranger Things	2016	Drama
15	The great	2020	Comedy

id	title	released_year	genre
1	Archer	2009	Animation
2	Arrested Development	2003	Comedy
3	Bob's Burgers	2011	Animation
4	Bojack Horseman	2014	Animation
5	Breaking Bad	2008	Drama
6	Curb Your Enthusiasm	2000	Comedy
7	Fargo	2014	Drama
8	Freaks and Geeks	1999	Comedy
9	General Hospital	1963	Drama
10	Halt and Catch Fire	2014	Drama
11	Malcolm In The Middle	2000	Comedy
12	Pushing Daisies	2007	Comedy
13	Seinfeld	1989	Comedy
14	Stranger Things	2016	Drama
15	The great	2020	Comedy

```
DELETE FROM sorted_series  
WHERE id=15;
```

We can delete from view and the underlying table will also change.

# MySQL – Replacing/Altering Views

```
CREATE VIEW sorted_series AS SELECT * FROM series ORDER BY released_year;
```

id	title	released_year	genre
9	General Hospital	1963	Drama
13	Seinfeld	1989	Comedy
8	Freaks and Geeks	1999	Comedy
.	.	.	.
.	.	.	.
.	.	.	.
7	Fargo	2014	Drama
10	Halt and Catch Fire	2014	Drama
14	Stranger Things	2016	Drama

14 rows in set (0.00 sec)

We have thi already existing view that we want to modify to now sort released\_year DESC.

If we use CREATE and try an overwrite this view it will not work givin an error that view already exists just like it was a table.

```
CREATE OR REPLACE VIEW sorted_series  
AS SELECT * FROM series  
ORDER BY released_year DESC;
```

```
ALTER VIEW sorted_series  
AS SELECT * FROM series  
ORDER BY released_year DESC;
```

If we use **CREATE OR REPLACE** then it will overwrite the existing view with the new one.

id	title	released_year	genre
14	Stranger Things	2016	Drama
4	Bojack Horseman	2014	Animation
7	Fargo	2014	Drama
.	.	.	.
.	.	.	.
.	.	.	.
8	Freaks and Geeks	1999	Comedy
13	Seinfeld	1989	Comedy
9	General Hospital	1963	Drama

14 rows in set (0.00 sec)

**ALTER VIEW** tdoes the same thing.

# MySQL – Having

```
SELECT title, AVG(rating)
FROM full_reviews
GROUP BY title;
```

We can apply a grouping  
to the full\_reviews view.

title	AVG(rating)
Archer	8.12000
Arrested Development	8.08000
Bob's Burgers	7.52000
. . . .	
. . . .	
Stranger Things	8.76667
Seinfeld	7.60000
Halt and Catch Fire	9.90000

12 rows in set (0.00 sec)

```
SELECT
    title, AVG(rating), COUNT(rating) AS review_count
FROM full_reviews
GROUP BY title HAVING COUNT(rating) > 1;
```

title	AVG(rating)	review_count
Archer	8.12000	5
Arrested Development	8.08000	5
Bob's Burgers	7.52000	5
Bojack Horseman	7.94000	5
Breaking Bad	9.36000	5
Curb Your Enthusiasm	8.12000	5
Fargo	9.40000	2
Freaks and Geeks	8.60000	4
General Hospital	5.38000	5
Stranger Things	8.76667	3
Seinfeld	7.60000	2

11 rows in set (0.01 sec)

But we want to only include titles that  
have a more than 1 review. **HAVING** is  
applied to the grouping and works like  
a where clause.

# MySQL – WITH ROLLUP

We can apply a grouping to the full\_reviews view.

```
SELECT title, AVG(rating)
FROM full_reviews
GROUP BY title;
```

```
+-----+-----+
| title                | AVG(rating) |
+-----+-----+
| Archer                | 8.12000     |
| Arrested Development | 8.08000     |
| Bob's Burgers        | 7.52000     |
| . . .                |             |
| . . .                |             |
| Stranger Things       | 8.76667     |
| Seinfeld              | 7.60000     |
| Halt and Catch Fire  | 9.90000     |
+-----+-----+
12 rows in set (0.00 sec)
```

```
SELECT title, AVG(rating) FROM full_reviews GROUP BY title WITH ROLLUP;
```

```
+-----+-----+
| title                | AVG(rating) |
+-----+-----+
| Archer                | 8.12000     |
| Arrested Development | 8.08000     |
| Bob's Burgers        | 7.52000     |
| Bojack Horseman       | 7.94000     |
| Breaking Bad          | 9.36000     |
| Curb Your Enthusiasm | 8.12000     |
| Fargo                 | 9.40000     |
| Freaks and Geeks      | 8.60000     |
| General Hospital      | 5.38000     |
| Halt and Catch Fire   | 9.90000     |
| Seinfeld              | 7.60000     |
| Stranger Things       | 8.76667     |
| NULL                  | 8.02553     |
+-----+-----+
13 rows in set (0.00 sec)
```

Notice that if we do grouping **WITH ROLLUP** we get an extra row at the bottom.

This row does not have a title because it is an average of all the AVG(rating) columns. Or an average of an average.



```
SELECT title, COUNT(rating) FROM full_reviews GROUP BY title WITH ROLLUP;
```

title	COUNT(rating)
Archer	5
Arrested Development	5
Bob's Burgers	5
Bojack Horseman	5
Breaking Bad	5
Curb Your Enthusiasm	5
Fargo	2
Freaks and Geeks	4
General Hospital	5
Halt and Catch Fire	1
Seinfeld	2
Stranger Things	3
NULL	47

13 rows in set (0.00 sec)

IF we do COUNT with grouping and have **WITH ROLLUP** we get an extra row at the bottom.

This rollup row is now a count of all the counts, i.e. a total of the number of ratings.

```
SELECT released_year, AVG(rating) FROM full_reviews
GROUP BY released_year WITH ROLLUP;
```

We can do rollup on a rating per released\_year grouping.

released_year	AVG(rating)
1963	5.38000
1989	7.60000
1999	8.60000
2000	8.12000
2003	8.08000
2008	9.36000
2009	8.12000
2011	7.52000
2014	8.55000
2016	8.76667
NULL	8.02553

11 rows in set (0.01 sec)

```
SELECT released_year, genre, AVG(rating) FROM full_reviews
GROUP BY released_year, genre;
```

released_year	genre	AVG(rating)
2009	Animation	8.12000
2003	Comedy	8.08000
2011	Animation	7.52000
...	...	...
1963	Drama	5.38000
2016	Drama	8.76667
1989	Comedy	7.60000

11 rows in set (0.00 sec)

Here we are grouping by released\_year then genre. I.e. in 2009 animation series had an average rating of 8.12.

```
SELECT released_year, genre, AVG(rating)
FROM full_reviews
GROUP BY released_year, genre WITH ROLLUP;
```

In 1963 Drama series had an average rating of 5.38.

In 1963 the average rating for all genres is 5.38 (ROLLUP of genres)

If we look at year 2014 we get an average for animation and Drama then an average of all genre's for that year.

At the bottom is a rollup average of all genres for all years.

released_year	genre	AVG(rating)
1963	Drama	5.38000
1963	NULL	5.38000
1989	Comedy	7.60000
1989	NULL	7.60000
1999	Comedy	8.60000
1999	NULL	8.60000
2000	Comedy	8.12000
2000	NULL	8.12000
2003	Comedy	8.08000
2003	NULL	8.08000
2008	Drama	9.36000
2008	NULL	9.36000
2009	Animation	8.12000
2009	NULL	8.12000
2011	Animation	7.52000
2011	NULL	7.52000
2014	Animation	7.94000
2014	Drama	9.56667
2014	NULL	8.55000
2016	Drama	8.76667
2016	NULL	8.76667
NULL	NULL	8.02553

22 rows in set (0.00 sec)

# MySQL – MySQL Modes

<https://dev.mysql.com/doc/refman/8.0/en/sql-mode.html>

MySQL modes are settings that can be enabled and disabled to make our SQL perform in different ways. MySQL has GLOBAL modes and SESSION modes.

```
SELECT @@GLOBAL.sql_mode;
```

```
+-----+
| @@GLOBAL.sql_mode |
+-----+
| STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION |
+-----+
```

```
SET GLOBAL sql_mode = 'modes';
```

Setting GLOBAL modes changes the whole SQL engine and are saved and will be as set on restart of the SQL server.

```
SELECT @@SESSION.sql_mode;
```

```
+-----+
| @@SESSION.sql_mode |
+-----+
| STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION |
+-----+
```

```
SET SESSION sql_mode = 'modes';
```

Session settings are only applied at the current session and upon restarting the SQL server will revert back to the global settings.

In general it is best not to change the modes unless there is a specific reason to do so.

STRICT\_TRANS\_TABLES is one of the most important SQL modes. This concerns what happens when you try and do an invalid operation in a SQL query such as trying to insert a string into an integer column?

```
INSERT INTO reviews (rating) VALUES ('hi');
```

```
ERROR 1366 (HY000): Incorrect decimal value: 'hi' for column 'rating' at row 1
```

```
SET SESSION sql_mode = '';
```

Now if I set the SESSION SQL mode to empty string, I will remove all the default settings including strict transactional tables mode.

```
INSERT INTO reviews (rating) VALUES ('hi');
```

```
Query OK, 1 row affected, 1 warning (0.01 sec)
```

It now lets me insert a string into a decimal column.

```
SELECT * FROM reviews;
```

This time it gave a warning not an error. Errors do not execute but warnings do. Note how MySQL converted the string into a decimal of value zero before inserting it into the table.

STRICT\_TRANS\_TABLES is a very important setting and should not really be changed.

id	rating	series_id	reviewer_id
1	8.0	1	1
2	7.5	1	2
3	8.5	1	3
...			
46	8.9	14	3
47	8.9	14	4
48	0.0	NULL	NULL

48 rows in set (0.00 sec)

# Window Functions

# MySQL – MySQL Window Functions

<https://dev.mysql.com/doc/refman/8.0/en/window-functions.html>

emp_no	Department	salary
8	sales	59000.00
12	sales	60000.00
20	customer service	56000.00
21	customer service	55000.00

```
SELECT * FROM employees;
```

Starting with a simple table of four rows we can use group by to reduce down the data into single rows by grouping one or more column, in this case the department row. We get two groups of employees, one for sales and one for customer service.

department	AVG(salary)
sales	59500.000000
customer service	55500.000000

```
SELECT department, AVG(salary) FROM employees GROUP BY salary;
```

Windows functions are similar to group by in that they perform aggregate operations on a group of rows but **PRODUCE A RESULT FOR EACH ROW**.

```
SELECT emp_no, department, salary, AVG(salary) OVER(PARTITION BY department) AS dept_avg  
FROM employees;
```

emp_no	department	salary	dept_avg
20	customer service	56000.00	55500.000000
21	customer service	55000.00	55500.000000
8	sales	59000.00	59500.000000
12	sales	60000.00	59500.000000

We have three columns of our original data plus the calculation of the average salary partitioned by department. Note the end column where every row has the average department salary. Windows functions allow us to look at aggregate information alongside the original rows.

# MySQL – MySQL Window Functions – Test Data

```
CREATE TABLE employees (  
    emp_no INT PRIMARY KEY AUTO_INCREMENT,  
    department VARCHAR(20),  
    salary INT);
```

```
INSERT INTO employees (department, salary)  
VALUES  
( 'engineering', 80000),  
( 'engineering', 69000),  
( 'engineering', 70000),  
( 'engineering', 103000),  
( 'engineering', 67000),  
( 'engineering', 89000),  
( 'engineering', 91000),  
( 'sales', 59000),  
( 'sales', 70000),  
( 'sales', 159000),  
( 'sales', 72000),  
( 'sales', 60000),  
( 'sales', 61000),  
( 'sales', 61000),  
( 'customer service', 38000),  
( 'customer service', 45000),  
( 'customer service', 61000),  
( 'customer service', 40000),  
( 'customer service', 31000),  
( 'customer service', 56000),  
( 'customer service', 55000);
```

```
SELECT * FROM employees;
```

emp_no	department	salary
1	engineering	80000
2	engineering	69000
3	engineering	70000
4	engineering	103000
5	engineering	67000
6	engineering	89000
7	engineering	91000
8	sales	59000
9	sales	70000
10	sales	159000
11	sales	72000
12	sales	60000
13	sales	61000
14	sales	61000
15	customer service	38000
16	customer service	45000
17	customer service	61000
18	customer service	40000
19	customer service	31000
20	customer service	56000
21	customer service	55000

# MySQL – MySQL Window Functions – using OVER()

```
SELECT department, AVG(salary)
FROM employees
GROUP BY department;
```

department	AVG(salary)
engineering	81285.7143
sales	77428.5714
customer service	46571.4286

```
SELECT department, AVG(salary)
FROM employees;
```

```
+-----+
| department | AVG(salary) |
+-----+
| engineering | 68428.5714 |
+-----+
```

Above we have two examples of aggregate functions. We can group by columns such as department or get an aggregate for the whole table.

But if we alter the syntax a little to include the OVER()  
function after the aggregate function .....

```
SELECT department, AVG(salary) OVER() FROM employees;
```

[illegible]

We get the same number, 68428.5714 one time for each row. Window functions work by taking our dataset and forming it into one or more windows and performs the functions without collapsing the dataset.



```
SELECT emp_no, department, salary, AVG(salary) OVER() FROM employees;
```

emp_no	department	salary	AVG(salary) OVER()
1	engineering	80000	68428.5714
2	engineering	69000	68428.5714
3	engineering	70000	68428.5714
4	engineering	103000	68428.5714
5	engineering	67000	68428.5714
6	engineering	89000	68428.5714
7	engineering	91000	68428.5714
8	sales	59000	68428.5714
9	sales	70000	68428.5714
10	sales	159000	68428.5714
11	sales	72000	68428.5714
12	sales	60000	68428.5714
13	sales	61000	68428.5714
14	sales	61000	68428.5714
15	customer service	38000	68428.5714
16	customer service	45000	68428.5714
17	customer service	61000	68428.5714
18	customer service	40000	68428.5714
19	customer service	31000	68428.5714
20	customer service	56000	68428.5714
21	customer service	55000	68428.5714

If we run a generic SELECT on all columns then average the salary column with OVER() we get an additional column of average salary across the whole dataset added into each row as the last column.

The dataset is not collapsed to show this average like it would be with group by.

This could be usefull if we want to compare every employees salary to the dataset averages, minimums, maximums etc in one SQL function without needing other queries to calculate the aggregates.

```
SELECT emp_no, department, salary, MIN(salary) OVER(), MAX(salary) OVER() FROM employees;
```

emp_no	department	salary	MIN(salary) OVER()	MAX(salary) OVER()
1	engineering	80000	31000	159000
2	engineering	69000	31000	159000
3	engineering	70000	31000	159000
4	engineering	103000	31000	159000
5	engineering	67000	31000	159000
6	engineering	89000	31000	159000
7	engineering	91000	31000	159000
8	sales	59000	31000	159000
9	sales	70000	31000	159000
10	sales	159000	31000	159000
11	sales	72000	31000	159000
12	sales	60000	31000	159000
13	sales	61000	31000	159000
14	sales	61000	31000	159000
15	customer service	38000	31000	159000
16	customer service	45000	31000	159000
17	customer service	61000	31000	159000
18	customer service	40000	31000	159000
19	customer service	31000	31000	159000
20	customer service	56000	31000	159000
21	customer service	55000	31000	159000

We could also see a table of employees that also shows the minimum and maximum salaries for the whole dataset using the OVER() function after each aggregate function.

# MySQL – MySQL Window Functions – Partition By clause

```
SELECT emp_no, department, salary,  
       AVG(salary) OVER(PARTITION BY department) AS dept_avg,  
       MIN(salary) OVER(PARTITION BY department) AS dept_min,  
       MAX(salary) OVER(PARTITION BY department) AS dept_max  
FROM employees;
```

emp_no	department	salary	dept_avg	dept_min	dept_max
15	customer service	38000	46571.4286	31000	61000
16	customer service	45000	46571.4286	31000	61000
17	customer service	61000	46571.4286	31000	61000
18	customer service	40000	46571.4286	31000	61000
19	customer service	31000	46571.4286	31000	61000
20	customer service	56000	46571.4286	31000	61000
21	customer service	55000	46571.4286	31000	61000
1	engineering	80000	81285.7143	67000	103000
2	engineering	69000	81285.7143	67000	103000
3	engineering	70000	81285.7143	67000	103000
4	engineering	103000	81285.7143	67000	103000
5	engineering	67000	81285.7143	67000	103000
6	engineering	89000	81285.7143	67000	103000
7	engineering	91000	81285.7143	67000	103000
8	sales	59000	77428.5714	59000	159000
9	sales	70000	77428.5714	59000	159000
10	sales	159000	77428.5714	59000	159000
11	sales	72000	77428.5714	59000	159000
12	sales	60000	77428.5714	59000	159000
13	sales	61000	77428.5714	59000	159000
14	sales	61000	77428.5714	59000	159000

In the previous example we only had one window which was the whole dataset.

Here we are dividing or partitioning the dataset into three windows by department then performing our aggregate functions on the data in each window.

Windowing is like grouping because we are performing aggregates on a group of rows BUT the rows are not collapsed.

```

SELECT emp_no, department, salary,
       AVG(salary) OVER(PARTITION BY department) AS dept_avg,
       MIN(salary) OVER(PARTITION BY department) AS dept_min,
       MAX(salary) OVER(PARTITION BY department) AS dept_max,
       AVG(salary) OVER() AS company_avg
FROM employees;

```

Here we are windowing the dataset by department and also have a window for the whole dataset to show company average of salary.

emp_no	department	salary	dept_avg	dept_min	dept_max	company_avg
15	customer service	38000	46571.4286	31000	61000	68428.5714
16	customer service	45000	46571.4286	31000	61000	68428.5714
17	customer service	61000	46571.4286	31000	61000	68428.5714
18	customer service	40000	46571.4286	31000	61000	68428.5714
19	customer service	31000	46571.4286	31000	61000	68428.5714
20	customer service	56000	46571.4286	31000	61000	68428.5714
21	customer service	55000	46571.4286	31000	61000	68428.5714
1	engineering	80000	81285.7143	67000	103000	68428.5714
2	engineering	69000	81285.7143	67000	103000	68428.5714
3	engineering	70000	81285.7143	67000	103000	68428.5714
4	engineering	103000	81285.7143	67000	103000	68428.5714
5	engineering	67000	81285.7143	67000	103000	68428.5714
6	engineering	89000	81285.7143	67000	103000	68428.5714
7	engineering	91000	81285.7143	67000	103000	68428.5714
8	sales	59000	77428.5714	59000	159000	68428.5714
9	sales	70000	77428.5714	59000	159000	68428.5714
10	sales	159000	77428.5714	59000	159000	68428.5714
11	sales	72000	77428.5714	59000	159000	68428.5714
12	sales	60000	77428.5714	59000	159000	68428.5714
13	sales	61000	77428.5714	59000	159000	68428.5714
14	sales	61000	77428.5714	59000	159000	68428.5714

```
SELECT emp_no, department, salary, COUNT(*)
OVER(PARTITION BY department) AS dept_count
FROM employees;
```

emp_no	department	salary	dept_count
15	customer service	38000	7
16	customer service	45000	7
17	customer service	61000	7
18	customer service	40000	7
19	customer service	31000	7
20	customer service	56000	7
21	customer service	55000	7
1	engineering	80000	7
2	engineering	69000	7
3	engineering	70000	7
4	engineering	103000	7
5	engineering	67000	7
6	engineering	89000	7
7	engineering	91000	7
8	sales	59000	7
9	sales	70000	7
10	sales	159000	7
11	sales	72000	7
12	sales	60000	7
13	sales	61000	7
14	sales	61000	7

We can also count the employees per department.

```
SELECT emp_no, department, salary, SUM(salary)
OVER(PARTITION BY department) AS dept_payroll,
SUM(salary) OVER() as total_payroll
FROM employees;
```

emp_no	department	salary	dept_payroll	total_payroll
15	customer service	38000	326000	1437000
16	customer service	45000	326000	1437000
17	customer service	61000	326000	1437000
18	customer service	40000	326000	1437000
19	customer service	31000	326000	1437000
20	customer service	56000	326000	1437000
21	customer service	55000	326000	1437000
1	engineering	80000	569000	1437000
2	engineering	69000	569000	1437000
3	engineering	70000	569000	1437000
4	engineering	103000	569000	1437000
5	engineering	67000	569000	1437000
6	engineering	89000	569000	1437000
7	engineering	91000	569000	1437000
8	sales	59000	542000	1437000
9	sales	70000	542000	1437000
10	sales	159000	542000	1437000
11	sales	72000	542000	1437000
12	sales	60000	542000	1437000
13	sales	61000	542000	1437000
14	sales	61000	542000	1437000

We can also sum the employees salary per department to get department total payroll and then a total for company payroll.

# MySQL – MySQL Window Functions – Order By

```
SELECT emp_no, department, salary,  
       SUM(salary) OVER(PARTITION BY department ORDER BY salary) AS rolling_dept_salary,  
       SUM(salary) OVER(PARTITION BY department) AS total_dept_salary  
FROM employees;
```

emp_no	department	salary	rolling_dept_salary	total_dept_salary
19	customer service	31000	31000	326000
15	customer service	38000	69000	326000
18	customer service	40000	109000	326000
16	customer service	45000	154000	326000
21	customer service	55000	209000	326000
20	customer service	56000	265000	326000
17	customer service	61000	326000	326000
5	engineering	67000	67000	569000
2	engineering	69000	136000	569000
3	engineering	70000	206000	569000
1	engineering	80000	286000	569000
6	engineering	89000	375000	569000
7	engineering	91000	466000	569000
4	engineering	103000	569000	569000
8	sales	59000	59000	542000
12	sales	60000	119000	542000
13	sales	61000	241000	542000
14	sales	61000	241000	542000
9	sales	70000	311000	542000
11	sales	72000	383000	542000
10	sales	159000	542000	542000

We still have three different windows by department.

But if we add ORDER BY is included in the partition then the sum aggregate performs differently.

It does a rolling sum and is ascending by default. We could add a DESC clause after ORDER BY and get a descending rolling sum.

```
SELECT emp_no, department, salary,
       MIN(salary) OVER(PARTITION BY department ORDER BY salary DESC) AS rolling_min
FROM employees;
```

emp_no	department	salary	rolling_min
17	customer service	61000	61000
20	customer service	56000	56000
21	customer service	55000	55000
16	customer service	45000	45000
18	customer service	40000	40000
15	customer service	38000	38000
19	customer service	31000	31000
4	engineering	103000	103000
7	engineering	91000	91000
6	engineering	89000	89000
1	engineering	80000	80000
3	engineering	70000	70000
2	engineering	69000	69000
5	engineering	67000	67000
10	sales	159000	159000
11	sales	72000	72000
9	sales	70000	70000
13	sales	61000	61000
14	sales	61000	61000
12	sales	60000	60000
8	sales	59000	59000

We still have three different windows by department.

But if we add ORDER BY is included in the partition then the min aggregate performs differently.

It does a rolling min and if we set it to descending then the rolling min will countdown to the lowest salary in the window.

# MySQL – MySQL Window Functions – Rank()

```
SELECT emp_no, department, salary,  
       RANK() OVER(ORDER BY salary DESC) AS overall_salary_rank  
FROM employees;
```

emp_no	department	salary	overall_salary_rank
10	sales	159000	1
4	engineering	103000	2
7	engineering	91000	3
6	engineering	89000	4
1	engineering	80000	5
11	sales	72000	6
3	engineering	70000	7
9	sales	70000	7
2	engineering	69000	9
5	engineering	67000	10
13	sales	61000	11
14	sales	61000	11
17	customer service	61000	11
12	sales	60000	14
8	sales	59000	15
20	customer service	56000	16
21	customer service	55000	17
16	customer service	45000	18
18	customer service	40000	19
15	customer service	38000	20
19	customer service	31000	21

We can rank the salaries. We need to specify ORDER BY in the over clause otherwise MySQL does not know what to rank.

But rank is not the same as row number. Note how when two employees have the same **salary the rank number is the same**.

Not also that it does not continue from rank 8 but **skips to rank 9**.

There are three people that hold joint rank 11 So it skips two rank numbers and resumes at 14.



```
SELECT emp_no, department, salary,
       RANK() OVER(PARTITION BY department ORDER BY SALARY DESC) AS dept_salary_rank,
       RANK() OVER(ORDER BY salary DESC) AS overall_salary_rank
FROM employees;
```

emp_no	department	salary	dept_salary_rank	overall_salary_rank
10	sales	159000	1	1
4	engineering	103000	1	2
7	engineering	91000	2	3
6	engineering	89000	3	4
1	engineering	80000	4	5
11	sales	72000	2	6
3	engineering	70000	5	7
9	sales	70000	3	7
2	engineering	69000	6	9
5	engineering	67000	7	10
17	customer service	61000	1	11
13	sales	61000	4	11
14	sales	61000	4	11
12	sales	60000	6	14
8	sales	59000	7	15
20	customer service	56000	2	16
21	customer service	55000	3	17
16	customer service	45000	4	18
18	customer service	40000	5	19
15	customer service	38000	6	20
19	customer service	31000	7	21

Employee number **10** is the highest paid in sales department and overall in the company.

Employee number **4** is the highest paid in engineering and second highest paid within the company.

```
SELECT emp_no, department, salary,
       RANK() OVER(PARTITION BY department ORDER BY SALARY DESC) AS dept_salary_rank,
       RANK() OVER(ORDER BY salary DESC) AS overall_salary_rank
FROM employees ORDER BY department;
```

emp_no	department	salary	dept_salary_rank	overall_salary_rank
17	customer service	61000	1	11
20	customer service	56000	2	16
21	customer service	55000	3	17
16	customer service	45000	4	18
18	customer service	40000	5	19
15	customer service	38000	6	20
19	customer service	31000	7	21
4	engineering	103000	1	2
7	engineering	91000	2	3
6	engineering	89000	3	4
1	engineering	80000	4	5
3	engineering	70000	5	7
2	engineering	69000	6	9
5	engineering	67000	7	10
10	sales	159000	1	1
11	sales	72000	2	6
9	sales	70000	3	7
13	sales	61000	4	11
14	sales	61000	4	11
12	sales	60000	6	14
8	sales	59000	7	15

This query can be tidied up a bit by having a general order by department.

Now we can see the highest to lowest earners in each department with their overall salary rank in the company.

Rank is a function that only works with windows but it is not always necessary to partition.

# MySQL – MySQL - Window – Dense Rank & Row Number

```
SELECT emp_no, department, salary,  
       ROW_NUMBER() OVER(PARTITION BY department ORDER BY salary DESC) as dept_row_number,  
       RANK() OVER(PARTITION BY department ORDER BY SALARY DESC) AS dept_salary_rank,  
       RANK() OVER(ORDER BY salary DESC) AS overall_salary_rank  
FROM employees ORDER BY department;
```

emp_no	department	salary	dept_row_number	dept_salary_rank	overall_salary_rank
17	customer service	61000	1	1	11
20	customer service	56000	2	2	16
21	customer service	55000	3	3	17
16	customer service	45000	4	4	18
18	customer service	40000	5	5	19
15	customer service	38000	6	6	20
19	customer service	31000	7	7	21
4	engineering	103000	1	1	2
7	engineering	91000	2	2	3
6	engineering	89000	3	3	4
1	engineering	80000	4	4	5
3	engineering	70000	5	5	7
2	engineering	69000	6	6	9
5	engineering	67000	7	7	10
10	sales	159000	1	1	1
11	sales	72000	2	2	6
9	sales	70000	3	3	7
13	sales	61000	4	4	11
14	sales	61000	5	4	11
12	sales	60000	6	6	14
8	sales	59000	7	7	15

```

SELECT emp_no, department, salary,
       ROW_NUMBER() OVER(PARTITION BY department ORDER BY salary DESC) as dept_row_number,
       RANK() OVER(PARTITION BY department ORDER BY SALARY DESC) AS dept_rank,
       DENSE_RANK() OVER(ORDER BY salary DESC) AS overall_dense_rank
FROM employees ORDER BY department;

```

emp_no	department	salary	dept_row_number	dept_rank	overall_dense_rank
17	customer service	61000	1	1	10
20	customer service	56000	2	2	13
21	customer service	55000	3	3	14
16	customer service	45000	4	4	15
18	customer service	40000	5	5	16
15	customer service	38000	6	6	17
19	customer service	31000	7	7	18
4	engineering	103000	1	1	2
7	engineering	91000	2	2	3
6	engineering	89000	3	3	4
1	engineering	80000	4	4	5
3	engineering	70000	5	5	7
2	engineering	69000	6	6	8
5	engineering	67000	7	7	9
10	sales	159000	1	1	1
11	sales	72000	2	2	6
9	sales	70000	3	3	7
13	sales	61000	4	4	10
14	sales	61000	5	4	10
12	sales	60000	6	6	11
8	sales	59000	7	7	12

Dense rank will not skip numbers when there is a tie.

For example there are two employees that have the seventh highest salaries. The next rank number is 8.

Same applies for 10 where there are three employees that tie. The next rank number is 11.

# MySQL – MySQL Window Functions – NTILE()

```
SELECT emp_no, department, salary,  
       NTILE(4) OVER(ORDER BY salary DESC) AS salary_quartile  
FROM employees ORDER BY salary_quartile;
```

emp_no	department	salary	salary_quartile
10	sales	159000	1
4	engineering	103000	1
7	engineering	91000	1
6	engineering	89000	1
1	engineering	80000	1
11	sales	72000	1
3	engineering	70000	2
9	sales	70000	2
2	engineering	69000	2
5	engineering	67000	2
13	sales	61000	2
14	sales	61000	3
17	customer service	61000	3
12	sales	60000	3
8	sales	59000	3
20	customer service	56000	3
21	customer service	55000	4
16	customer service	45000	4
18	customer service	40000	4
15	customer service	38000	4
19	customer service	31000	4

NTILE will divide the dataset into a number of equal windows where we specify the number of windows in the brackets.

We then use the OVER function to specify the order.

In this example we are dividing salaries into four groups with the top earners in the first group and the bottom earners in the 4th group.

```

SELECT emp_no, department, salary,
       NTILE(4) OVER(PARTITION BY department ORDER BY salary DESC) AS dept_salary_quartile,
       NTILE(4) OVER(ORDER BY salary DESC) AS salary_quartile
FROM employees;

```

emp_no	department	salary	dept_salary_quartile	salary_quartile
10	sales	159000	1	1
4	engineering	103000	1	1
7	engineering	91000	1	1
11	sales	72000	1	1
17	customer service	61000	1	2
20	customer service	56000	1	3
6	engineering	89000	2	1
1	engineering	80000	2	1
9	sales	70000	2	2
13	sales	61000	2	3
21	customer service	55000	2	4
16	customer service	45000	2	4
3	engineering	70000	3	2
2	engineering	69000	3	2
14	sales	61000	3	3
12	sales	60000	3	3
18	customer service	40000	3	4
15	customer service	38000	3	4
5	engineering	67000	4	2
8	sales	59000	4	3
19	customer service	31000	4	4

We can have two NTILEs to break up a window into four buckets of salary then overall salary.

For example, employee number 5 is in the second quartile overall within the company for salary but within the department is in the fourth quartile, i.e. one of the lowest paid engineers.

# MySQL – MySQL Window Functions – First Value

```
SELECT emp_no, department, salary,  
       FIRST_VALUE(emp_no) OVER (ORDER BY salary DESC)  
FROM employees;
```

Finds the first value of a window.

emp_no	department	salary	FIRST_VALUE(emp_no) OVER (ORDER BY salary DESC)
10	sales	159000	10
4	engineering	103000	10
7	engineering	91000	10
6	engineering	89000	10
1	engineering	80000	10
11	sales	72000	10
3	engineering	70000	10
9	sales	70000	10
2	engineering	69000	10
5	engineering	67000	10
13	sales	61000	10
14	sales	61000	10
17	customer service	61000	10
12	sales	60000	10
8	sales	59000	10
20	customer service	56000	10
21	customer service	55000	10
16	customer service	45000	10
18	customer service	40000	10
15	customer service	38000	10
19	customer service	31000	10

In this case there is only one window and we are ordering by salary from highest to lowest so the highest paid will be the first value.

In this case employee 10.

```

SELECT emp_no, department, salary,
       FIRST_VALUE(emp_no) OVER(PARTITION BY department ORDER BY salary DESC) AS highest_paid_dept_employee,
       FIRST_VALUE(emp_no) OVER (ORDER BY salary DESC) AS highest_paid_employee_overall
FROM employees ORDER BY department;

```

emp_no	department	salary	highest_paid_dept_employee	highest_paid_employee_overall
17	customer service	61000	17	10
20	customer service	56000	17	10
21	customer service	55000	17	10
16	customer service	45000	17	10
18	customer service	40000	17	10
15	customer service	38000	17	10
19	customer service	31000	17	10
4	engineering	103000	4	10
7	engineering	91000	4	10
6	engineering	89000	4	10
1	engineering	80000	4	10
3	engineering	70000	4	10
2	engineering	69000	4	10
5	engineering	67000	4	10
10	sales	159000	10	10
11	sales	72000	10	10
9	sales	70000	10	10
13	sales	61000	10	10
14	sales	61000	10	10
12	sales	60000	10	10
8	sales	59000	10	10

The highest paid employee in the company is employee number **10**, and the highest paid in sales.

The highest paid employee in the customer service department is employee number **17**.

The highest paid employee in Engineering is employee number **4**.



```
SELECT emp_no, department, salary,
       LAST_VALUE(emp_no) OVER(PARTITION BY department ORDER BY salary DESC) AS lowest_paid_dept_employee,
       FIRST_VALUE(emp_no) OVER(ORDER BY salary ASC) AS lowest_paid_employee_overall
FROM employees ORDER BY department;
```

emp_no	department	salary	lowest_paid_dept_employee	lowest_paid_employee_overall
19	customer service	31000	19	19
15	customer service	38000	15	19
18	customer service	40000	18	19
16	customer service	45000	16	19
21	customer service	55000	21	19
20	customer service	56000	20	19
17	customer service	61000	17	19
5	engineering	67000	5	19
2	engineering	69000	2	19
3	engineering	70000	3	19
1	engineering	80000	1	19
6	engineering	89000	6	19
7	engineering	91000	7	19
4	engineering	103000	4	19
8	sales	59000	8	19
12	sales	60000	12	19
13	sales	61000	14	19
14	sales	61000	14	19
9	sales	70000	9	19
11	sales	72000	11	19
10	sales	159000	10	19

The lowest paid employee can also be found per department and whole company using LAST\_VALUE. There is also an NTH\_VALUE where we could find the 2<sup>nd</sup>, 3<sup>rd</sup>, etc

# MySQL – MySQL Window Functions – LEAD() & LAG()

```
SELECT emp_no, department, salary,  
       LAG(salary) OVER(ORDER BY salary DESC)  
FROM employees;
```

Lag will evaluate an expression on the previous row.

emp_no	department	salary	LAG(salary) OVER(ORDER BY salary DESC)
10	sales	159000	NULL
4	engineering	103000	159000
7	engineering	91000	103000
6	engineering	89000	91000
1	engineering	80000	89000
11	sales	72000	80000
3	engineering	70000	72000
9	sales	70000	70000
2	engineering	69000	70000
5	engineering	67000	69000
13	sales	61000	67000
14	sales	61000	61000
17	customer service	61000	61000
12	sales	60000	61000
8	sales	59000	60000
20	customer service	56000	59000
21	customer service	55000	56000
16	customer service	45000	55000
18	customer service	40000	45000
15	customer service	38000	40000
19	customer service	31000	38000

In this case we are getting the value of salary from the previous row.

Note how the first row is NULL. This is because it does not have a previous row.

```
SELECT emp_no, department, salary,
       salary - LAG(salary) OVER(ORDER BY salary DESC) AS salary_diff
FROM employees;
```

emp_no	department	salary	salary_diff
10	sales	159000	NULL
4	engineering	103000	-56000
7	engineering	91000	-12000
6	engineering	89000	-2000
1	engineering	80000	-9000
11	sales	72000	-8000
3	engineering	70000	-2000
9	sales	70000	0
2	engineering	69000	-1000
5	engineering	67000	-2000
13	sales	61000	-6000
14	sales	61000	0
17	customer service	61000	0
12	sales	60000	-1000
8	sales	59000	-1000
20	customer service	56000	-3000
21	customer service	55000	-1000
16	customer service	45000	-10000
18	customer service	40000	-5000
15	customer service	38000	-2000
19	customer service	31000	-7000

We could use this to calculate salary difference. i.e. this row's salary minus the previous row's salary.

Note how as before the first row is NULL.

The difference between the highest paid salary and the second highest paid salary is **-56000**.

```
SELECT emp_no, department, salary,
       salary - LEAD(salary) OVER(ORDER BY salary DESC) AS salary_diff
FROM employees;
```

emp_no	department	salary	salary_diff
10	sales	159000	56000
4	engineering	103000	12000
7	engineering	91000	2000
6	engineering	89000	9000
1	engineering	80000	8000
11	sales	72000	2000
3	engineering	70000	0
9	sales	70000	1000
2	engineering	69000	2000
5	engineering	67000	6000
13	sales	61000	0
14	sales	61000	0
17	customer service	61000	1000
12	sales	60000	1000
8	sales	59000	3000
20	customer service	56000	1000
21	customer service	55000	10000
16	customer service	45000	5000
18	customer service	40000	2000
15	customer service	38000	7000
19	customer service	31000	NULL

Lead is the inverse of lag so it will take the value of the following row. This can be used again to calculate salary difference.

The difference between the highest paid and the second highest paid is **56000**.

Note how this time the lowest paid has a value of **NULL** because there is no following row.

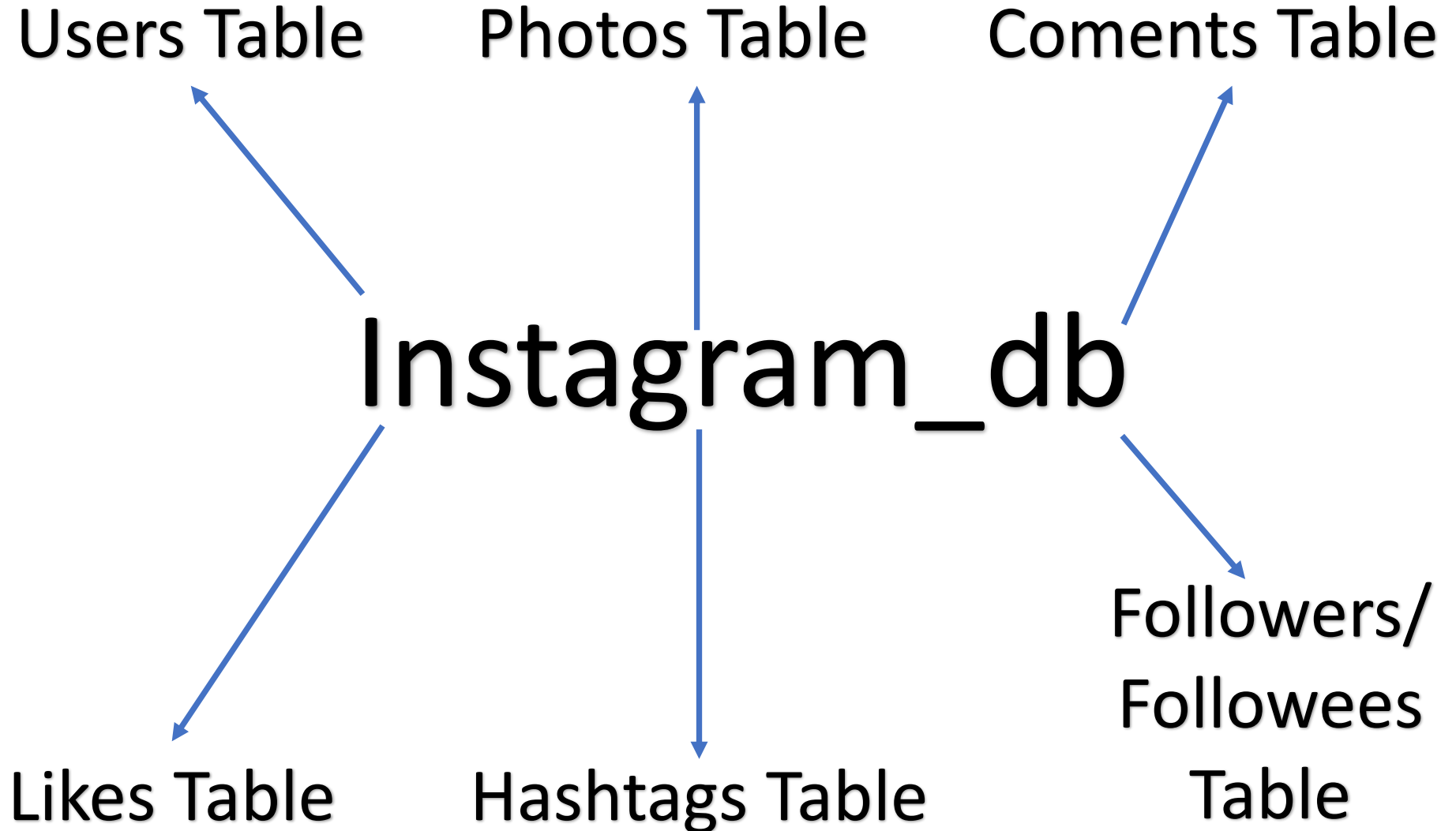
```
SELECT emp_no, department, salary,
       salary - LAG(salary) OVER(PARTITION BY department ORDER BY salary DESC) AS salary_diff
FROM employees;
```

emp_no	department	salary	salary_diff
17	customer service	61000	NULL
20	customer service	56000	-5000
21	customer service	55000	-1000
16	customer service	45000	-10000
18	customer service	40000	-5000
15	customer service	38000	-2000
19	customer service	31000	-7000
4	engineering	103000	NULL
7	engineering	91000	-12000
6	engineering	89000	-2000
1	engineering	80000	-9000
3	engineering	70000	-10000
2	engineering	69000	-1000
5	engineering	67000	-2000
10	sales	159000	NULL
11	sales	72000	-87000
9	sales	70000	-2000
13	sales	61000	-9000
14	sales	61000	0
12	sales	60000	-1000
8	sales	59000	-1000

We can also partition by department. And note how that within the departments the salary differences are not that large Except in the sales team where the difference between the highest paid and second highest paid is 87000. We can deduce that maybe the sales roles are commission based!

# Instagram Database Clone

# MySQL – Instagram Database Clone



```
CREATE DATABASE instagram_clone_db;
```

```
SHOW DATABASES;
```

```
USE instagram_clone_db;
```

Database
animal_shelter
book_shop
friends
information_schema
instagram_clone_db
mysql
performance_schema
relationships
sakila
shirts_db
sys
tv_db
windows_db
world

# MySQL – Create Users Table

```
CREATE TABLE users (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    username VARCHAR(255) UNIQUE NOT NULL,  
    created_at TIMESTAMP default now()  
);
```

```
DECS users;
```

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
username	varchar(255)	NO	UNI	NULL	
created_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

```
INSERT INTO users (username) VALUES ('BlueTheCat'),('CharlieBrown'),('ColtSteele');
```

```
SELECT * FROM users;
```

id	username	created_at
1	BlueTheCat	2022-11-14 15:15:25
2	CharlieBrown	2022-11-14 15:15:25
3	ColtSteele	2022-11-14 15:15:25



# MySQL – Create Photos Table

```
CREATE TABLE photos (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    image_url VARCHAR(255) NOT NULL,  
    user_id INT NOT NULL,  
    created_at TIMESTAMP default now(),  
    FOREIGN KEY(user_id) REFERENCES  
users(id)  
);
```

**DECS users;**

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
image_url	varchar(255)	NO		NULL	
user_id	int	NO	MUL	NULL	
created_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

```
INSERT INTO Photos (image_url, user_id) VALUES ('/random1', 1),('/random2',2),('/random3', 2);
```

**SELECT \* FROM photos;**

id	image_url	user_id	created_at
1	/random1	1	2022-11-14 15:32:21
2	/random2	2	2022-11-14 15:32:21
3	/random3	2	2022-11-14 15:32:21

image_url	username
/random1	BlueTheCat
/random2	CharlieBrown
/random3	CharlieBrown

```
SELECT photos.image_url, users.username  
FROM PHOTOS  
JOIN users ON photos.user_id = users.id;
```

# MySQL – Create Comments Table

```
CREATE TABLE comments (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    comment_text VARCHAR(255) NOT NULL,  
    user_id INT NOT NULL,  
    photo_id INT NOT NULL,  
    created_at TIMESTAMP default now(),  
    FOREIGN KEY(user_id) REFERENCES users(id),  
    FOREIGN KEY(photo_id) REFERENCES  
photos(id)  
);
```

DECS comments;

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
comment_text	varchar(255)	NO		NULL	
user_id	int	NO	MUL	NULL	
photo_id	int	NO	MUL	NULL	
created_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

```
INSERT INTO comments (comment_text, user_id, photo_id)  
VALUES ('Meow!', 1, 2),('Amazing shot!',3,2),('I love it',2,1);
```

SELECT \* FROM comments;

id	comment_text	user_id	photo_id	created_at
1	Meow!	1	2	2022-11-14 15:50:16
2	Amazing shot!	3	2	2022-11-14 15:50:16
3	I love it	2	1	2022-11-14 15:50:16

# MySQL – Create Likes Table

If we add a primary key set to the `user_id` AND `photo_id` then it will not allow duplicate inserts of likes by the same user to the same photo. Each user can only like an image once.

```
CREATE TABLE likes (  
    user_id INT NOT NULL,  
    photo_id INT NOT NULL,  
    created_at TIMESTAMP default now(),  
    FOREIGN KEY(user_id) REFERENCES users(id),  
    FOREIGN KEY(photo_id) REFERENCES photos(id),  
    PRIMARY KEY(user_id, photo_id)  
);
```

DECS likes;

Field	Type	Null	Key	Default	Extra
user_id	int	NO	PRI	NULL	
photo_id	int	NO	PRI	NULL	
created_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

```
INSERT INTO likes (user_id, photo_id) VALUES (1, 1),(2,1),(1,2),(1,3),(3,2);
```

SELECT \* FROM likes;

user_id	photo_id	created_at
1	1	2022-11-14 16:34:46
1	2	2022-11-14 16:34:46
1	3	2022-11-14 16:34:46
2	1	2022-11-14 16:34:46
3	2	2022-11-14 16:34:46

```
INSERT INTO likes (user_id, photo_id) VALUES (1,1);  
ERROR 1062 (23000): Duplicate entry '1-1' for key 'likes.PRIMARY'
```

The primary key constraint prevents us from inserting a duplicate like for an image.

# MySQL – Create Follows Table

```
CREATE TABLE follows (  
    follower_id INT NOT NULL,  
    followee_id INT NOT NULL,  
    created_at TIMESTAMP default now(),  
    FOREIGN KEY(follower_id) REFERENCES users(id),  
    FOREIGN KEY(followee_id) REFERENCES users(id),  
    PRIMARY KEY(follower_id, followee_id)  
);
```

DECS follows;

Field	Type	Null	Key	Default	Extra
follower_id	int	NO	PRI	NULL	
followee_id	int	NO	PRI	NULL	
created_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

```
INSERT INTO follows (follower_id, followee_id) VALUES (1,2),(1,3),(3,1),(2,3);
```

SELECT \* FROM follows;

follower_id	followee_id	created_at
1	2	2022-11-14 16:49:17
1	3	2022-11-14 16:49:17
2	3	2022-11-14 16:49:17
3	1	2022-11-14 16:49:17

```
INSERT INTO follows (follower_id, followee_id) VALUES (1,2);  
ERROR 1062 (23000): Duplicate entry '1-2' for key 'follows.PRIMARY'
```

The primary key constraint prevents us from inserting a duplicate followers

# MySQL – Create Tags Tables

```
SHOW TABLES;
```

```
+-----+
| Tables_in_instagram_clone_db |
+-----+
| comments                      |
| follows                      |
| likes                        |
| photo_tags                   |
| photos                      |
| tags                         |
| users                       |
+-----+
```

```
CREATE TABLE tags (
    id INT PRIMARY KEY AUTO_INCREMENT,
    tag_name VARCHAR(255) UNIQUE,
    created_at TIMESTAMP default now()
);
```

```
CREATE TABLE photo_tags (
    photo_id INT NOT NULL,
    tag_id INT NOT NULL,
    FOREIGN KEY(photo_id) REFERENCES photos(id),
    FOREIGN KEY(tag_id) REFERENCES tags(id),
    PRIMARY KEY(photo_id, tag_id)
);
```

## MySQL – Import SQL data file.

```
source C:/users/EL_EL/Desktop/ig_clone_data.sql;
```

```
SHOW TABLES;
```

```
+-----+
| Tables_in_ig_clone |
+-----+
| comments            |
| follows             |
| likes               |
| photo_tags          |
| photos              |
| tags                |
| users               |
+-----+
```

# MySQL – Instagram clone tasks: Find the 5 oldest users

```
SELECT * FROM USERS ORDER BY created_at limit 5;
```

id	username	created_at
80	Darby_Herzog	2016-05-06 00:14:21
67	Emilio_Bernier52	2016-05-06 13:04:30
63	Elenor88	2016-05-08 01:30:41
95	Nicole71	2016-05-09 17:30:22
38	Jordyn.Jacobson2	2016-05-14 07:56:26

# MySQL – Instagram clone tasks: What day of the week is most popular for new users to sign up?

```
SELECT  
    DAYNAME(created_at) AS day, count(*) as total  
FROM users  
GROUP BY day  
ORDER BY total DESC;
```

Thursday and Sunday are the most popular days of the week for new users to sign up.

day	total
Thursday	16
Sunday	16
Friday	15
Tuesday	14
Monday	14
Wednesday	13
Saturday	12

# MySQL – Instagram clone tasks: Find the users that have never posted a photo

```
SELECT
    username, IFNULL(user_id, 0) AS images_posted
FROM users LEFT JOIN photos ON users.id = photos.user_id
WHERE photos.id IS NULL;
```

These users have not yet posted any photos and could be target for reminder notices to post an image.

username	images_posted
Aniya_Hackett	0
Bartholome.Bernhard	0
Bethany20	0
Darby_Herzog	0
David.Osinski47	0
Duane60	0
Esmeralda.Mraz57	0
Esther.Zulauf61	0
Franco_Keebler64	0
Hulda.Macejkovic	0
Jaclyn81	0
Janelle.Nikolaus81	0
Jessyca_West	0
Julien_Schmidt	0
Kasandra_Homenick	0
Leslie67	0
Linnea59	0
Maxwell.Halvorson	0
Mckenna17	0
Mike.Auer39	0
Morgan.Kassulke	0
Nia_Haag	0
Ollie_Ledner37	0
Pearl7	0
Rocio33	0
Tierra.Trantow	0

# MySQL – Instagram clone tasks: Find the user with the image that has the most likes.

```
SELECT
    username, photos.id, photos.image_url, likes.user_id, count(*) as total
FROM photos
JOIN likes ON likes.photo_id = photos.id
JOIN users ON photos.user_id = users.id
GROUP BY photos.id
ORDER BY total DESC
LIMIT 1;
```

username	id	image_url	user_id	total
Zack_Kemmer93	145	https://jarret.name	3	48

# MySQL – Instagram clone tasks: How many times does our average user post?

```
SELECT (SELECT COUNT(*) FROM photos) / (SELECT COUNT(*) FROM users);
```

(SELECT COUNT(*) FROM photos) / (SELECT COUNT(*) FROM users)
2.5700



# MySQL – Instagram clone tasks: What are the top 5 most used hashtags?

```
SELECT
    tag_name, COUNT(*) as count
FROM tags
JOIN photo_tags ON tags.id = photo_tags.tag_id
GROUP BY tag_name
ORDER BY count
DESC LIMIT 5;
```

tag_name	count
smile	59
beach	42
party	39
fun	38
concert	24

# MySQL – Instagram clone tasks: Find users that have liked every single photo, i.e. bots.

```
SELECT
    username, COUNT(*) AS num_likes
FROM users
JOIN likes ON users.id = likes.user_id
GROUP BY likes.user_id
HAVING num_likes = (SELECT COUNT(*) FROM photos);
```

username	num_likes
Aniya_Hackett	257
Bethany20	257
Duane60	257
Jaclyn81	257
Janelle.Nikolaus81	257
Julien_Schmidt	257
Leslie67	257
Maxwell.Halvorson	257
Mckenna17	257
Mike.Auer39	257
Nia_Haag	257
Ollie_Ledner37	257
Rocio33	257

# MySQL – Additional Resources

## **Beginner to intermediate SQL**

<https://www.educba.com/data-science/data-science-tutorials/mysql-tutorial/>

<https://www.w3resource.com/mysql/mysql-tutorials.php>

<https://www.geeksforgeeks.org/sql-tutorial/?ref=gcse>

<https://www.mysqltutorial.org/>

<https://www.w3schools.com/sql/>

<https://www.geekengine.com/database/sample/>

<https://www.khanacademy.org/computing/computer-programming/sql/>

## **More advanced SQL**

<https://www.hackerrank.com/domains/sql>

<https://leetcode.com/study-plan/sql/>

<https://www.tutorialspoint.com/mysql/index.htm>

[https://sqlzoo.net/wiki/SQL\\_Tutorial](https://sqlzoo.net/wiki/SQL_Tutorial)

## **YouTube Creators (advanced)**

<https://www.youtube.com/c/techTFQ>

<https://www.youtube.com/c/LearnatKnowstar>

<https://www.youtube.com/channel/UCfGTc8zyBjCGg-Ilc4oAxEg/videos>

# Course Content

Tools: mySQL server, mySQL workbench, dbGate | mySQL database commands: SHOW databases, CREATE database, DROP database, USE database, SELECT database() | mySQL datatypes for numbers, strings and dates | Table commands: CREATE TABLE, SHOW TABLES, SHOW COLUMNS FROM, DROP TABLE | mySQL comments | INSERT INTO table single or multiple rows | MySQL Escape characters and double quotes | Default values | primary key and auto\_increment | CRUD: Create, Read, Update and Delete | select query with where clause | aliases | MySQL Update queries | MySQL Delete queries | source command | String functions: CONCAT, CONCAT\_WS, SUBSTR, Combining (nesting) string functions, REPLACE(), | INSERT(), LEFT(), RIGHT(), REPEAT(), TRIM() | Refining selections: DISTINCT, ORDER BY ASC, DESC, LIMIT, LIKE & % or \_ Wildcards | Aggregate functions: COUNT, GROUP BY, MIN & MAX, subqueries, Grouping by multiple columns, Min & Max with grouping, SUM, AVG | Data types: CHAR & VARCHAR, INT, TINYINT, SMALLINT, MEDIUMINT, BIGINT, DECIMAL, FLOAT & DOUBLE, DATE, TIME, DATETIME, CURDATE & CURTIME, NOW(), MONTHNAME(), YEAR(), DAYOFWEEK(), DAYOFYEAR(), HOUR(), MINUTE(), SECOND(), Timestamps, default NOW() & ON UPDATE NOW() | Comparison operators: !=, NOT LIKE, >, <, <=, >= | Logical operators: AND, OR, BETWEEN, NOT BETWEEN, DATE & TIME comparisons, IN operator, MODULO, CASE, IS NULL, IS NOT NULL, | Unique constraint | Check Constraints | Named Constraints | Multiple Column Constraints | Alter table: Adding Columns, dropping columns, renaming tables, modify Columns, | Data Relationships: one-to-one, one-to-many, many-to-many | PRIMARY\_KEY, FOREIGN\_KEY | Joins: Cross Join, Inner Join (JOIN ON), Inner-Join with Group by, Left Join, left-join with Group By & IFNULL(), Right Join | On Delete Cascade | many to many Join or Union table with FOREIGN KEY | Using Joins on many to many |

views introduction | updatable views | HAVING clause | WITH ROLLUP | MySQL Modes: STRICT\_TRANS\_TABLES | Windows Functions introduction | Using OVER() | Partition By clause | ORDER BY with Windows | RANK() | DENSE\_RANK() | ROW\_NUMBER() | NTILE() | FIRST\_VALUE() | LAST\_VALUE() | NTH\_VALUE() | LEAD() | LAG() | Create tables: users, photos, comments, likes, follows & hashtags | Import large SQL data file | Instagram clone database tasks