shawkyebrahim2514

```
DESC <TabelName>
```

- Show all table's columns with columns' properties

```
Show Tables
```

- Show all tables in current scheme (Database)

```
DROP Table <TableName>
```

- Delete the given table name

```
INSERT INTO <TableName> (<Columns name separated with commas>) VALUES (<Values to be attached to each corresponding columns separated with commas>)
```

```
CREATE TABLE cats (
    Name VARCHAR(100) NOT NULL,
    Age INT NOT NULL
);
```

- Create name and age columns with **not null** value (value must be assigned)

```
CREATE TABLE cats (
    Name VARCHAR(100) DEFAULT 'unnamed',
    Age INT DEFAULT 99
);
```

- This code has a **default** name and age if we don't provide them when inserting a new row
- Their values **cannot be NULL** because there are a default values for them aleardy!

```
CREATE TABLE cats (
    CatID INT NOT NULL,
    Name VARCHAR(100) DEFAULT 'unnamed',
    Age INT DEFAULT 99,
    PRIMARY KEY(CatID)
);
```

- Make the CatID unique identifier primary key (to distinguish each row by it)
- Can put 'PRIMARY KEY' directly after the column you want it to be the primary key

```
CREATE TABLE cats (
        CatID INT NOT NULL AUTO_INCREMENT,
        Name VARCHAR(100) DEFAULT 'unnamed',
        Age INT DEFAULT 99,
        PRIMARY KEY(CatID)
);
```

- Make the CatID unique identifier primary key (to distinguish each row by it)
- It will be **increased each time** a new row added

Exercise on employees

```
CREATE TABLE Employees(
   ID INT AUTO_INCREMENT ,
   FirstName VARCHAR(15) NOT NULL ,
   MiddleName VARCHAR(15) ,
   Lastname VARCHAR(15) NOT NULL ,
   Age INT NOT NULL ,
   CurrentState varchar(30) NOT NULL DEFAULT 'Employed',
   PRIMARY KEY (ID)
);

INSERT INTO employees(firstname, middlename, lastname, age)
VALUES ('Shawky', 'Ebrahim', 'Ahmed', 19),
       ('Wafaa', 'Ebrahim', 'Ahmed', 19),
       ('Basmala', 'Gad', 'Ahmed', 19),
       ('Sara', 'Waled', 'Ahmed', 19),
       ('Belal', 'Ahmed', 'Eid', 19);
```

READ commands

```
SELECT * FROM cats;
```

- (*) means show all columns in cats table
- Can replace (*) with any column name (or columns separated with commas)

```
SELECT * FROM cats WHERE age = 4
```

- Can use **WHERE** to specified the output
- If we use **WHERE** with a text, it is **insensitive case**

```
SELECT cat_id AS id, name FROM cats;
```

- **AS** means **alias** name
- It gives the corresponding column another name to use it with the new name
- To make the name separated by space, put it in **quotes** 'first second'

UPDATE commands

```
UPDATE <TableName> SET <ColumnName> = <GivenValue> WHERE <Condition>
```

- **Update** row(s) from the table and **set** any column(s) with value
- The row(s) will be chosen corresponding to the WHERE condition

DELETE commands

```
DELTE FROM cats WHERE name = 'Egg';
```

- **Delete** all rows which name is Egg

Exercise on shirts

```
CREATE TABLE shirts(
    shirt_id INT not null auto_increment primary key ,
    article varchar(30) not null ,
    color varchar(30) not null ,
    shirt_size varchar(5) not null ,
    last_worn INT default 0
);


INSERT INTO shirts (article, color, shirt_size, last_worn) values
    ('t-shirt', 'white', 'S', 10),
    ('t-shirt', 'green', 'S', 200),
    ('polo shirt', 'black', 'M', 100),
    ('tank top', 'blue', 'S', 50),
    ('t-shirt', 'pink', 'S', 0),
    ('polo shirt', 'red', 'M', 5),
    ('tank top', 'white', 'S', 200),
    ('tank top', 'blue', 'M', 15);
```

Exercise on books

```
create table books(
    book_id INT not null auto_increment primary key ,
    title varchar(100),
    author_fname varchar(100),
    author_lname varchar(100),
    released_year INT,
    stock_quantity int,
    pages int
);

INSERT into books(title, author_fname, author_lname, released_year, stock_quantity, pages) VALUES
    ('The Namesake', 'Jhumpa', 'Lahiri', 2003, 32, 291),
    ('Norse Mythology', 'Neil', 'Gaiman', 2016, 34, 304),
    ('American Gods', 'Neil', 'Gaiman', 2001, 12, 465),
    ('Interpreter of Maladies', 'Jhumpa', 'Lahiri', 1996, 97, 198),
    ('A Hologram for the King: A Novel', 'Dave', 'Eggers', 2012, 154, 352),
    ('The Circle', 'Dave', 'Eggers', 2013, 26, 504),
    ('The Amazing Adventures of Kavalier & Clay', 'Michael', 'Chabon', 2000, 68, 634),
    ('Just Kids', 'Patti', 'Smith', 2010, 55, 304),
    ('A Heartbreaking Work of Staggering Genius', 'Dave', 'Eggers', 2001, 104, 437),
    ('Coraline', 'Neil', 'Gaiman', 2003, 100, 208),
    ('What We Talk About When We Talk About Love: Stories', 'Raymond', 'Carver', 1981, 23, 176),
    ('Where I\'m Calling From: Selected Stories', 'Raymond', 'Carver', 1989, 12, 526),
    ('White Noise', 'Don', 'Delillo', 1985, 49, 320),
    ('Cannery Row', 'John', 'Steinbeck', 1945, 95, 181),
    ('Oblivion: Stories', 'David', 'Foster Wallace', 2004, 172, 329),
    ('Consider the Lobster', 'David', 'Foster Wallace', 2005, 92, 343);
```

MySQL string functions

MySQL Functions (w3schools.com)

SELECT CONCAT (column or text, column or text, ….) FROM <TableName>

- **Combine** data for cleaner output
- We can but anything in the **CONCAT** parenthesis (text or column)
- We can use **CONCAT** inside another **CONCAT**

SELECT CONCAT_WS(' - ', title, CONCAT_WS(' ', author_fname, author_lname)) AS 'Book Description' from books

- **CONCAT** with separator
- **CONCAT_WS**(separator, expression1, expression2, expression3,...)

SUBSTRING (string, start, length)

- Work with parts of strings
- If the length doesn't be provided, the length would be the number of characters
- If the length greater than the string size, string will be printed until the end
- We can start from end (**with negative position**)

REPLACE(original string, The substring to be replaced, The new replacement substring)

- Replace a string with another one

CHAR_LENGTH(string)

- Counts characters in a string

Continue practicing with books

```
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);
```

```
SELECT DISTINCT author_lname FRIM books;
```

- Show all **unique** first author name
- If there are many duplicated names, **one** will be shown

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

- Sorting results (**ASCENDING** By Default)
- Print rows that contains only author_lname column, but sorted by author_lname

```
SELECT author_lname FROM books ORDER BY author_lname DESC;
```

- Will be sorted with **descending** order

```
SELECT <Columns> FROM <TableName> ORDERED BY <Number>
```

- This will print rows that contains the given columns but **ordered** with column of <Number> **index** from the result columns

```
SELECT title, author_fname, author_lname FROM books ORDER BY 2
```

- Will be sorted according to the author_fname values (of index 2)

You can use:

```
SELECT * FROM <TableName> ORDER BY <Number>
```

The order of columns here is the order in the table's columns

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

- Sort the rows according to the author_fname
- But if there are many rows with the same author_fname, then they will be sorted according to the author_lname

```
SELECT <Columns> FROM <TableName> LIMIT <Number>
SELECT * FROM books LIMIT 5;
```

- Show the **first 5** rows from the table

```
SELECT <Columns> FROM <tableName> LIMIT <StartFromNumber>, <NumberOfLinitedRows>
```

- Show <NumberOfLimitedRows> rows that exist **after** the row of number <StartFrom-Number>

```
WHERE <Column> LIKE <Expression>
```

- This will match any row that the its <Column> value contain the given expression <Expression> can be:
- **('%da%')**, This will match any value that has the substring 'da' (whether at the first, end, or any position)
- **('da%')**, This will match any value that has the prefix 'da' at the first
- **('%da')**, This will match any value that has the suffix 'da' at the end
- The percent sign (**%**) represents zero, one, or multiple characters
- The underscore sign (_) represents one, single character
- More syntax here: MySQL LIKE Operator (w3schools.com)

```
SELECT COUNT(*) FROM <TableName>
```

- Return the number of rows in the given table
- Can count anything given in it, such as the distinct column value

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

- We can filter the result as we want

```
SELECT COUNT(DISTINCT author_lname) FROM books WHERE author_lname LIKE '%da%'
```

- This will return the number of rows that has distinct author_lname and the value of it must be contain the substring 'da'

```
SELECT author_lname, count(author_lname) FROM books GROUP BY author_lname
```

GROUB BY <Column>

- Summarizes or aggregates identical data into single rows
- It groups every specific <Name> with all rows that contains this <Name> in one group
- This is useful when we want to know the frequency of existing of each value in a column

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

- The above code print only one row and one column that contain the min/max value

```
SELECT * FROM books WHERE released_year = (SELECT MIN(released_year) FROM books);
```

- Get all rows that contain this **MIN** value
- We must put the right hand side code in parenthesis

To find the first year each author published his first book

```
select author_fname, author_lname, min(released_year) from books group by author_fname, author_lname
```

- a new table will be created for each unique full author name
- The new created table contains all rows that belongs to this full author name
- Then the result is the full author name + the minimum year in his created table

```
SELECT SUM(<Column>) FROM <TableName>
```

- Print the sum of all values in the given column

Data types

CHAR(fixedSize)

This type has a fixed number of bytes in memory

- If the assigned text is smaller than the **fixedSize**, the remaining characters will be **spaces**
- If the assigned text is greater than the **fixedSize**, it will **truncate** the assigned text and only store the **first fixedSize characters** (from left)

VARCHAR(size)

- The size is **depend** on the length of the assigned text to it, but has a limit of maximum size

DECIMAL(totalNumberOfDigits, lengthOfFractionalPart)

- if we want to store (3.64), then the type would be **DECIMAL(3,2)**

DATE

- Store the Date in form **(YYYY-MM-DD)**

TIME

- Store the time in form **(hh:mm:ss)**

DATATIME

- Store data and time in the same column in the firm **(YYYY-MM-DD hh:mm:ss)**

Some date and time functions

CURDATE()

- Return the current data

CURTIME()

- Return the current time

NOW()

- Return the current data and the time

Add and Subtract two or mote dates

```
DATEDIFF(firstDate, secondDate)
```

or

```
firstDate - secondDate
```

- Returns **firstDate – secondDate** as a value in days from the firstdate to the second-Date

```
select now() + interval 50 day;
select now() + interval 50 month ;
select now() + interval 50 year;
```

- Can add any date or time to a certain date or time by using **INTERVAL**

DATETIME

- The supported range is '**1000-01-01 00:00:00**' to '**9999-12-31 23:59:59**'

TIMESTAMP

- The supported range is '**1970-01-01 00:00:01**' to '**2038-01-19 03:14:07**'
- **CURRECT_TIMESTAMP** is equal to **NOW()**

```
create table time(
    name varchar(20),
    timePosted timestamp default current_timestamp on update current_timestamp
)
```

- This code will create a table that has a column to know the time where each name created and this time will be updated automatically when you update the correspond-ing name

```
WHERE <Column> NOT LIKE <Expression>
```

- **NOT LIKE** is opposite to **LIKE**
- It will select all rows except the rows that contain the given the given expression

'A' == 'a'

- Because the mysql deals with the characters in **insensitive** way

```
WHERE <Column> BETWEEN < LowerRange > AND <UpperRange>
```

- Match any value that is in **range** [LowerRange, UpperTange]

```
WHERE <Column> NOT BETWEEN < LowerRange > AND <UpperRange>
```

- This is opposite to **BETWEEN**, match any value that doesn't in the given range

```
CAST(<Something> AS <DataType>)
```

- When using **BETWEEN** with date or time values, use **CAST()** to explicitly convert the values to the desired datatype

```
WHERE <Column> IN (set of values)
```

- Match the column's values if exist in the given set
- The values in the set are separated with commas (,)

```
WHERE <Column> NOT IN (set of values)
```

- This is opposite to the **IN**
- Match the columns' values if don't exist in the given set

CASE STATEMENTS

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

- This code will create a column for **title, released_year and a column** that has 'Modern Lit' value if the released_year >= 2000, otherwise, the value will be '20th Century Lit'
- The '**CASE**' must end with '**END**'
- '**WHEN**' must follow with '**THEN**' after the condition

```
select title, stock_quantity, case
                        when stock_quantity between 0 and 50 then '*'
                        when stock_quantity between 51 and 100 then '**'
                        else '***'
                    end AS STOCK
from books
```

Exercise with customers

```
create table customers(
    id int auto_increment primary key ,
    first_name varchar(50),
    last_name varchar(50),
    email varchar(50)
);

create table orders(
    id int auto_increment primary key ,
    order_date datetime,
    amount decimal(8,2),
    customer_id int,
    foreign key (customer_id) references customers(id) ON DELETE CASCADE
);

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) ;
```

- The orders table has a primary key and foreign key
- The foreign key is to the customer_id column and references to the **id** column in the customers table
- (**ON DELETE CASCADE**) means that when we delete any customer, the associated orders' row(s) will also be deleted automatically
- We cannot make the foreign key refers to an **ID** that doesn't exist

```
select first_name, last_name, order_date, amount from customers
join orders on customers.id = orders.customer_id
```

- This code will join the customers table with the order table in a new table with **customer id** as a key between them

Customers table			Orders table		
Customer ID	First Name	Last Name	Order ID	Amount	Customer ID
1	Boy	George	1	99.99	1
2	George	Michael	2	35.50	1
3	David	Bowie	3	800.67	2
4	Blue	Steele	4	12.50	2
5	Bette	Davis	5	450.25	5

If we apply inner join (intersection) as (customers JOIN orders)

- a new table will be created that contains rows such that in these rows, there is an associated key (existing customer to existing order with customer id as a key)

If we apply left join as (customers LEFT JOIN orders)

- Then a new table will be created that contains **all customers' rows** whatever the customer has an order attached with or doesn't have
- The customers who have an order attached with them, will be joined with attached order
- The customers who don't have an order attached with them, the order columns will be **null**

If we apply right join as (customers RIGHT JOIN orders)

- Then a new table will be created that contains **all orders' rows** whatever the order has a customer attached with or doesn't have
- The orders which have a customer attached with them, will be joined with attached customer
- The orders which don't have a customer attached with them, the customer columns will be **null** (This case is unreal, because each order must have a customer)

```
IFNULL(<Column>, <Value>)
```

<Column>: The column to check if it contains null
<Value>: Is the value that the column will be replaced if the column's value is NULL

- If the column`s value is **null**, the result value will be the given **<Value>**
- otherwise, the result will be the column`s not null value

Students and papers exercise

```
create table students(
    id int auto_increment primary key,
    first_name varchar(50)
);

create table papers(
    id int auto_increment primary key ,
    title varchar(50),
    grade int,
    student_id int,
    foreign key (student_id) references students(id) ON DELETE CASCADE
);

insert into students(first_name)
values ('caleb'), ('Samantha'), ('Raj'), ('Carlos'), ('Lisa');

insert into papers (title, grade, student_id)
values ('My First Book Report', 60, 1),
       ('My Second Book Report', 75, 1),
       ('Russian Lit Through The Ages', 94, 2),
       ('De Montaigne and The Art of The Essay', 98, 2),
       ('Borges and Magical Realism', 89, 4);

select * from papers;

select first_name, title, grade from students
join papers on students.id = papers.student_id
order by grade desc;

select first_name, title, grade from students
left join papers on students.id = papers.student_id;

select first_name,
      ifnull(title, 'MISSING') AS title,
      ifnull(grade, 0) AS grade
from students
left join papers on students.id = papers.student_id;

select first_name,
      ifnull(avg(grade), 0) as grade
from students
left join papers on students.id = papers.student_id
group by students.id
order by grade desc;

select first_name,
      ifnull(avg(grade), 0) as grade,
      case
         when grade >= 75 then 'PASSING'
          else 'FAILING'
       end as passing_status
from students
left join papers on students.id = papers.student_id
group by students.id
order by grade desc;
```

Reviews exercise

```
create table reviewers (
    id int auto_increment primary key ,
    first_name varchar(50),
    last_name varchar(50)
);

create table series (
    id int auto_increment primary key ,
    title varchar(100),
    released_year year(4),
    genre varchar(50)
);

create table reviews (
    id int auto_increment primary key ,
    rating decimal(2,1) ,
    series_id int ,
    reviewer_id int ,
    foreign key (series_id) references series(id) on delete cascade ,
    foreign key (reviewer_id) references reviewers(id) on delete cascade
);

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);
```

```
ROUND(<Column>, <Number>)
```

- This will round the given column's value, we can round the number to a specific length from the fraction part by using <Number>
- The <Number> is the length of the fraction part (Default = 0)

Exercise

```
select first_name, last_name, if(rating is null, 0, count(*)) as count,
      ifnull(min(rating), 0) as 'min', ifnull(max(rating), 0) as 'max',
      round(ifnull(avg(rating), 0), 2) as 'avg',
      if(rating is null, 'INACTIVE', 'ACTIVE') as status
from reviews
right join reviewers on reviews.reviewer_id = reviewers.id
group by reviewers.id;

select title as 'series title', rating 'rating of series', concat_ws(' ', first_name, last_name) as 'from reviewer'
from reviews
join reviewers on reviews.reviewer_id = reviewers.id
join series on reviews.series_id = series.id
order by title;
```

Clone Instagram (Basic) - *Scheme*

```
create table users(
    id int auto_increment primary key ,
    -- username should be unique for each user
    username varchar(50) unique not null,
    created_at timestamp default now() on update now()
);

create table photos(
    id int auto_increment primary key ,
    image_url varchar(255) not null,
    user_id int not null,
    created_at timestamp default now() on update now(),
    foreign key (user_id) references users(id) on delete cascade
);

create table comments(
    id int auto_increment primary key ,
    comment_text varchar(255) not null ,
    user_id int not null ,
    photo_id int not null ,
    created_at timestamp default now() on update now() ,
    foreign key (user_id) references users(id) on delete cascade ,
    foreign key (photo_id) references photos(id) on delete cascade
);

create table likes (
    user_id int not null ,
    photo_id int not null ,
    created_at timestamp default now() on update now(),
    foreign key (user_id) references users(id) on delete cascade ,
    foreign key (photo_id) references photos(id) on delete cascade ,
    -- make the like unique for each user in any photo
    primary key (user_id, photo_id)
);

create table follows (
    follower_id int not null ,
    followee_id int not null ,
    created_at timestamp default now() on update now() ,
    foreign key (followee_id) references users(id) on delete cascade ,
    foreign key (follower_id) references users(id) on delete cascade ,
    primary key (follower_id, followee_id)
);

create table tags (
    id int auto_increment primary key ,
    tag_name varchar(50) ,
    created_at timestamp default now()
);

create table photo_tag (
    photo_id int not null ,
    tag_id int not null ,
    foreign key (photo_id) references photos(id) on delete cascade ,
    foreign key  (tag_id) references tags(id) ,
    primary key (photo_id, tag_id)
);
```

Find the 5 oldest users

```
select * from users
order by created_at
limit 5;
```

What day of the week do most users register on?

```
select dayname(created_at) as dayOfWeek, count(*) as total
from users group by dayOfWeek order by total desc limit 1;
```

Created by Chatgpt

```
SELECT dayname(created_at) as dayOfWeek, count(*) as total
FROM users
GROUP BY dayname(created_at)
HAVING total = (SELECT max(total_users)
                FROM (SELECT count(*) as total_users FROM users
                GROUP BY dayname(created_at)) as sub);
```

Find the users who have never posted a photo (inactive users)

```
select users.id, users.username as 'inactive users' from users
left join photos on users.id = photos.user_id
where photos.id is null;
```

Most popular photo(s) and user(s) who created it

```
select photos.id, photos.image_url, count(*) as total, users.username from photos
join likes on likes.photo_id = photos.id
join users on photos.user_id = users.id
group by photo_id
having total = (select max(total_likes)
            from (select count(*) as total_likes from likes
                group by photo_id) as sub);
```

How many times does the average user post?

```
select (count(*) / (select count(*) from users)) as average from photos;
```

What are the top 5 most commonly used Hashtags

```
select tags.id, tags.tag_name, count(*) as total from photo_tag
join tags on photo_tag.tag_id = tags.id
group by photo_tag.tag_id
order by total desc
limit 5;
```

Find users who have liked every single photo on the site

```
select users.id, users.username from likes
join users on likes.user_id = users.id
group by likes.user_id
having count(*) = (select count(*) from photos);
```