DBMS

"" Database Management Systems""

"" Each one is a Manager,
And will execute commands to store in… or bring data from…,
MySQL, PostgreSQL, SQLite & MongoDB!!!
And all are Cursors""

```sql
Comments in SQL with -- or /* */

Create database
CREATE DATABASE myDB;

Use database
USE myDB;

DROP Database
DROP DATABASE myDB;

Make the data read only, change to 1 if you want to be able to make changes in the database
ALTER DATABASE myDB READ ONLY = 1;
                    Error as you cannot, change ONLY to 0
                    DROP DATABASE myDB;
                    ALTER DATABASE myDB READ ONLY = 0;


Create table with columns
CREATE TABLE employess (
             employee_id INT,
             first_name VARCHAR(50),
             last_name VARCHAR(50),
             hourly_pay DECIMAL(5,2),
             hire_date DATE
             );


CREATE TABLE public.chitter_user
(
    username text,
    user_id serial NOT NULL,
    encrypted_password text,
    email text,
    date_joined timestamp without time zone,
    PRIMARY KEY (user_id)
)
WITH (
    OIDS = FALSE
);

ALTER TABLE public.chitter_user
    OWNER to postgres;


CREATE TABLE public.post
(
    post_id serial NOT NULL,
    user_id integer,
    post_text text,
    posted_on timestamp without time zone DEFAULT current_timestamp,
    PRIMARY KEY (post_id),
    CONSTRAINT user_id_constraint FOREIGN KEY (user_id)
        REFERENCES public.chitter_user (user_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE
)
WITH (
    OIDS = FALSE
);

ALTER TABLE public.post
```

```
    OWNER to postgres;


CREATE TABLE public.follower
(
    user_id integer NOT NULL,
    follower_id integer NOT NULL,
    PRIMARY KEY (user_id, follower_id),
    CONSTRAINT user_id_constraint FOREIGN KEY (user_id)
        REFERENCES public.chitter_user (user_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE,
    CONSTRAINT follower_id_constraint FOREIGN KEY (follower_id)
        REFERENCES public.chitter_user (user_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE
)
WITH (
    OIDS = FALSE
);

ALTER TABLE public.follower
    OWNER to postgres;


CREATE TABLE public.chitter_user
(
    username text,
    user_id serial NOT NULL,
    encrypted_password text,
    email text,
    date_joined timestamp without time zone,
    PRIMARY KEY (user_id)
)
WITH (
    OIDS = FALSE
);

ALTER TABLE public.chitter_user
    OWNER to postgres;


CREATE TABLE public.post
(
    post_id serial NOT NULL,
    user_id integer,
    post_text text,
    posted_on timestamp without time zone DEFAULT current_timestamp,
    PRIMARY KEY (post_id),
    CONSTRAINT user_id_constraint FOREIGN KEY (user_id)
        REFERENCES public.chitter_user (user_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE
)
WITH (
    OIDS = FALSE
);

ALTER TABLE public.post
    OWNER to postgres;


CREATE TABLE public.follower
(
    user_id integer NOT NULL,
    follower_id integer NOT NULL,
    PRIMARY KEY (user_id, follower_id),
    CONSTRAINT user_id_constraint FOREIGN KEY (user_id)
        REFERENCES public.chitter_user (user_id) MATCH SIMPLE
        ON UPDATE CASCADE
```

```
        ON DELETE CASCADE,
    CONSTRAINT follower_id_constraint FOREIGN KEY (follower_id)
        REFERENCES public.chitter_user (user_id) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE CASCADE
)
WITH (
    OIDS = FALSE
);

ALTER TABLE public.follower
    OWNER to postgres;


CREATE TABLE public.secret_user
(
    user_id integer NOT NULL,
    first_name character varying,
    last_name character varying,
    code_name character varying,
    country character varying,
    organization character varying,
    salary integer,
    knows_kong_fu boolean,
    PRIMARY KEY (user_id)
)


CREATE TABLE martian
(
martian_id SERIAL PRIMARY KEY,
first_name CHARACTER VARYING(25),
last_name CHARACTER VARYING(25),
base_id INTEGER,
super_id INTEGER
);

CREATE TABLE base
(
base_id SERIAL PRIMARY KEY,
base_name CHARACTER VARYING(30),
founded DATE
);


CREATE TABLE visitor
(
visitor_id SERIAL PRIMARY KEY,
host_id INTEGER,
first_name CHARACTER VARYING(25),
last_name CHARACTER VARYING(25)
);


CREATE TABLE inventory
(
base_id INTEGER,
supply_id INTEGER,
quantity INTEGER
);


CREATE TABLE supply
(
supply_id SERIAL PRIMARY KEY,
name CHARACTER VARYING(30),
description TEXT,
quantity INTEGER
);
```

```
CREATE TABLE martian_confidential
(
martian_id SERIAL PRIMARY KEY,
first_name CHARACTER VARYING(25),
last_name CHARACTER VARYING(25),
base_id INTEGER,
super_id INTEGER,
salary INTEGER,
dna_id CHARACTER VARYING(30)
);


Select table from database
SELECT * FROM mydb.employess;

Change name of table
RENAME TABLE employess TO employees;

Add column to table
ALTER TABLE employees ADD phone_number VARCHAR(15);

Rename column
ALTER TABLE employees RENAME COLUMN phone_number TO email;

Modify column
ALTER TABLE employees MODIFY COLUMN email VARCHAR(100);

Use AFTER keyword or FIRST
ALTER TABLE employees MODIFY email VARCHAR(100) AFTER last_name;

Drop column
ALTER TABLE employees DROP COLUMN email;


Insert single row into a Table
INSERT INTO employees VALUES (1, "Eugene", "Krabs", 25.50, "2023-01-02");

Check how much data we have in chitter_user
SELECT COUNT (*) FROM chitter_user;
INSERT INTO chitter_user(user_id, username, encrypted_password, email, date_joined) VALUES(DEFAULT, 'firstuser','fdhfdfdf87df', 'fakemail@fakedomain.fake','2019-02-25');
Check if data was added
SELECT * FROM chitter_user;
INSERT INTO chitter_user (username, encrypted_password) VALUES('sendouser','d754453jhj');
SELECT * FROM chitter_user;
INSERT INTO post (user_id, post_text) VALUES (1, 'Hello World'),(1, 'Hello Solar System!');
SELECT * FROM post;
Check what you have sent through python
SELECT * FROM post LIMIT 20; --
SELECT COUNT(*) FROM song;
SELECT MIN(year_released), MAX(year_released) FROM song;
0 means that there are song without release year
SELECT DISTINCT year_released FROM song ORDER BY year_released;
so many are there without year_released
SELECT COUNT(*) FROM song WHERE year_released = 0;
SELECT * FROM song;

Insert multiple rows into a Table
INSERT INTO employees
       VALUES (2, "Squidward","Tentacles",15.00,"2023-01-03"),
              (3, "Spongebob", "Sqarepants", 12.50, "2023-01-04"),
              (4, "Patrick","Star", 12.50, "2023-01-05"),
              (5, "Sandy", "Cheeks", 17.25, "2023-01-06" );
INSERT INTO employees (employee_id, first_name, last_name) VALUES (7, "Sqheldon","Plankton");

INSERT INTO martian
(martian_id, first_name, last_name, base_id, super_id)
VALUES
(DEFAULT, 'Ray', 'Bradbury', 1, NULL),
(DEFAULT, 'John', 'Black', 4, 10),
```

```
(DEFAULT, 'Samuel', 'Hinkston', 4, 2),
(DEFAULT, 'Jeff', 'Spender', 1, 9),
(DEFAULT, 'Sam', 'Parkhill', 2, 12),
(DEFAULT, 'Elma', 'Parkhill', 3, 8),
(DEFAULT, 'Melissa', 'Lewis', 1, 1),
(DEFAULT, 'Mark', 'Watney', 3, NULL),
(DEFAULT, 'Beth', 'Johanssen', 1, 1),
(DEFAULT, 'Chris', 'Beck', 4, NULL),
(DEFAULT, 'Nathaniel', 'York', 4, 2),
(DEFAULT, 'Elon', 'Musk', 2, NULL),
(DEFAULT, 'John', 'Carter', NULL, 8);

INSERT INTO martian_confidential
(martian_id, first_name, last_name, base_id, super_id, salary, dna_id)
VALUES
(DEFAULT, 'Ray', 'Bradbury', 1, NULL, 155900, 'gctaggaatgtagaatctcctgttg'),
(DEFAULT, 'John', 'Black', 4, 10, 120100, 'cagttaatggttgaagctggggatt'),
(DEFAULT, 'Samuel', 'Hinkston', 4, 2, 110000, 'cgaagcgctagatgctgtgttgtag'),
(DEFAULT, 'Jeff', 'Spender', 1, 9, 10000, 'gactaatgtcttcgtggattgcaga'),
(DEFAULT, 'Sam', 'Parkhill', 2, 12, 125000, 'gttactttgcgaaagccgtggctac'),
(DEFAULT, 'Elma', 'Parkhill', 3, 8, 137000, 'gcaggaatggaagcaactgccatat'),
(DEFAULT, 'Melissa', 'Lewis', 1, 1, 145250, 'cttcgatcgtcaatggagttccggac'),
(DEFAULT, 'Mark', 'Watney', 3, NULL, 121100, 'gacacgaggcgaactatgtcgcggc'),
(DEFAULT, 'Beth', 'Johanssen', 1, 1, 130000, 'cttagactaggtgtgaaacccgtta'),
(DEFAULT, 'Chris', 'Beck', 4, NULL, 125000, 'ggggggttacgacgaggaatccat'),
(DEFAULT, 'Nathaniel', 'York', 4, 2, 105000, 'ggtctccctgggcgggatattggatg'),
(DEFAULT, 'Elon', 'Musk', 2, NULL, 155800, 'atctgcttggatcaatagcgctgcg'),
(DEFAULT, 'John', 'Carter', NULL, 8, 129500, 'ccaatcgtgcgagtcgcgatagtct');

INSERT INTO supply
(supply_id, name, description, quantity)
VALUES
(DEFAULT, 'Solar Panel', 'Standard 1x1 meter cell', 912),
(DEFAULT, 'Water Filter', 'This takes things out of your water so it''s drinkable.', 6),
(DEFAULT, 'Duct Tape', 'A 10 meeter roll of duct tape for ALL your repairs', 951),
(DEFAULT, 'Ketchup', 'It''s ketchup', 206),
(DEFAULT, 'Battery Cell', 'Standard 1000 kAH battery cell for power grid (heavy item).', 17),
(DEFAULT, 'USB 6.0 Cable', 'Carbon fiber cooated / 10 TBps spool', 42),
(DEFAULT, 'Fuzzy Duster', 'It gets dusty around here. Be prepared!', 19),
(DEFAULT, 'Mars Bars', 'The ORIGINAL nutirent bar made with the finest bioengineered ingredients.', 3801),
(DEFAULT, 'Air Filter', 'Removes 99% of all Martian dust from your ventilation unit', 23),
(DEFAULT, 'Famous Ray''s Frozen Pizza', 'This Martian favourite is covered in all your favourite toppings. 1 flavour only.', 823);

INSERT INTO base
(base_id, base_name, founded)
VALUES
(DEFAULT, 'Tharsisland', '2037-06-03'),
(DEFAULT, 'Valles Marineris 2.0', '2040-12-01'),
(DEFAULT, 'Gale Cratertown', '2014-08-16'),
(DEFAULT, 'New New New York', '2042-02-10'),
(DEFAULT, 'Olympus Mons Spa & Casino', NULL);

INSERT INTO visitor
(visitor_id, host_id, first_name, last_name)
VALUES
(DEFAULT, 7, 'George', 'Ambrose'),
(DEFAULT, 1, 'Kris', 'Cardenas'),
(DEFAULT, 9, 'Priscilla', 'Lane'),
(DEFAULT, 11, 'Jane', 'Thornton'),
(DEFAULT, NULL, 'Doug', 'Stavenger'),
(DEFAULT, NULL, 'Jamie', 'Waterman'),
(DEFAULT, 8, 'Martin', 'Humphries');
             INSERT INTO inventory
             (base_id, supply_id, quantity)
                    VALUES
                    (1, 1, 8),
                    (1, 3, 5),
                    (1, 5, 1),
                    (1, 6, 2),
                    (1, 8, 12),
```

```
                    (1, 9, 1),
                    (2, 4, 5),
                    (2, 8, 62),
                    (2, 10, 37),
                    (3, 2, 11),
                    (3, 7, 2),
                    (4, 10, 91);
Select data from a table
SELECT * FROM employees;
SELECT first_name, last_name FROM employees;
SELECT last_name, first_name FROM employees;
SELECT * FROM employees WHERE employee_id = 1;
SELECT COUNT(*) last_name FROM employees;
SELECT * FROM employees WHERE hourly_pay >=15;
SELECT * FROM employees WHERE employee_id !=1;
SELECT * FROM employees WHERE hire_date IS NULL;

Q: How many rows are in the 'earthquake' table?
SELECT * FROM earthquake;
This will find the row's in the table
SELECT COUNT(*) FROM earthquake;
Select columns from a table
SELECT latitude, depth, magnitude FROM earthquake;
Select all count of a specific column
SELECT COUNT(*) place FROM earthquake;
Selects specific table with row (WHERE selects the row)
SELECT * FROM earthquake WHERE occurred_on >= '2000-01-01';
How to use AND and ORDER BY and DESC for descending order and LIMIT to 1 row
SELECT * FROM earthquake WHERE occurred_on >= '2010-01-01' AND occurred_on <= '2010-12-31' ORDER BY magnitude DESC LIMIT 1;
Select all columns from table but LIMIT to 1 row
SELECT * FROM earthquake LIMIT 1;
Selects the MIN and MAX from a column
SELECT MIN(occurred_on), MAX(occurred_on) FROM earthquake;
Q:what magnitude range is covered by the 'earthquake' table?| A:5.5+
SELECT MIN(magnitude), MAX(magnitude) FROM earthquake;
Select column and removing duplicates
SELECT DISTINCT cause FROM earthquake;
Q:How many earthquakes were natural earthquakes?| A: 22,942
SELECT COUNT(*) FROM earthquake WHERE cause = 'earthquake';
Query: Find most recent earthquake caused by a nuclear explosion
SELECT place, magnitude, occurred_on FROM earthquake WHERE cause = 'nuclear explosion' ORDER BY occurred_on DESC LIMIT 1;
Q:What were the 10 largest earthquakes from 1969 - 2018?
SELECT place, magnitude, occurred_on FROM earthquake ORDER BY magnitude DESC LIMIT 10;
Q: How can we count the number of aftershocks?|Idea: Find quakes with "Honshu" and "Japan" in the 'place' text & occurred within a week of the initial quake
SELECT COUNT(*) FROM earthquake WHERE place LIKE '%Honshu%Japan%' AND occurred_on BETWEEN '2011-03-11' AND '2011-03-18';

Update data from a table
UPDATE employees
       SET hourly_pay = 10.25
       WHERE employee_id = 7;
UPDATE employees
       SET hourly_pay = 10.50,hire_date = "2023-01-07"
       WHERE employee_id = 7;
UPDATE employees
       SET hire_date = NULL
       WHERE employee_id = 7;

SELECT * FROM secret_user;
INSERT INTO secret_user(user_id, first_name, last_name, code_name, country, organization, salary,knows_kong_fu) VALUES(4, 'John','Doe', 'Jd','USA','Kobra Kai',10000,true);
SELECT * FROM secret_user;
UPDATE secret_user SET first_name = 'James' WHERE user_id = 1;
SELECT * FROM secret_user;
UPDATE secret_user SET first_name = 'James' WHERE user_id = 2;
SELECT * FROM secret_user ORDER BY user_id;
UPDATE secret_user SET code_name ='Neo 2.0', salary = 115000 WHERE first_name ='John' AND last_name = 'Doe';
SELECT * FROM secret_user ORDER BY user_id;
UPDATE secret_user SET salary = 20000 WHERE organization = 'Kobra Kai';
SELECT * FROM secret_user ORDER BY user_id;
UPDATE secret_user SET knows_kong_fu = TRUE WHERE user_id IN (1,3,4);
SELECT * FROM secret_user ORDER BY user_id;
```

```sql
UPDATE secret_user SET salary = 1.1 * salary;
SELECT * FROM secret_user ORDER BY user_id;
SELECT SUM(salary) FROM secret_user;


Delete data from a table
DELETE FROM employees WHERE employee_id = 6;

SELECT COUNT(*) FROM song WHERE year_released =0;
SELECT * FROM song WHERE year_released =0 LIMIT 10;
DELETE removes rows not columns, without WHERE it will delete all rows!
DELETE FROM song WHERE year_released = 0;
SELECT COUNT(*) FROM song;
SELECT DISTINCT year_released FROM song ORDER BY year_released;
SELECT MIN(tempo), MAX(tempo) FROM song;
SELECT COUNT(*) FROM song WHERE tempo =0;
SELECT * FROM song WHERE tempo =0;
DELETE FROM song WHERE tempo =0;
SELECT COUNT(*) FROM song WHERE tempo = 0;
SELECT MIN(tempo), MAX(tempo) FROM song;
SELECT COUNT(*) FROM song;
SELECT MIN(duration), MAX(duration) FROM song;
SELECT MIN(loudness), MAX(loudness), MAX(loudness) FROM song;
SELECT COUNT(*) FROM song WHERE loudness > 0;
SELECT * FROM song WHERE loudness > 0;
DELETE FROM song WHERE loudness > 0;
Q: Has tempo changed over time?
SELECT year_released, ROUND(AVG(tempo)) FROM song GROUP BY year_released ORDER BY year_released;
SELECT * FROM song;


Autocommit
SET AUTOCOMMIT = OFF;

Commit
COMMIT;
DELETE FROM employees;

Rollback
ROLLBACK;


Using Current_Date() & Current_Time()
CREATE TABLE test(
            my_date DATE,
            my_time TIME,
            my_datetime DATETIME
            );
INSERT INTO test VALUES(CURRENT_DATE(), CURRENT_TIME(), NOW());
INSERT INTO test VALUES(CURRENT_DATE() - 1 , NULL, NULL);


Constraints
Unique constraint (ensures that all values in column are different, can be added the constraint before or after creating column)
CREATE TABLE products (
            product_id INT,
            product_name VARCHAR(25) UNIQUE,
            price DECIMAL(4,2)
            );
            to add it later: ALTER TABLE products ADD CONSTRAINT UNIQUE(product_name); <<---if you forgot to add it before handler
            INSERT INTO products
                  VALUES (100, "hamburger", 3.9),
                         (101, "fries", 1.89),
                         (102, "soda", 1.00),
                         (103, "ice cream", 1.49);
Not Null Constraint (so the value in that column can't be NULL when adding values)
CREATE TABLE products(
            product_id INT,
            product_name VARCHAR(25),
            price DECIMAL(4,2) NOT NULL
```

```
            );
            If the column is created already this is how you add NOT NULL
            ALTER TABLE products MODIFY price DECIMAL(4, 2) NOT NULL;
            INSERT INTO products VALUES (104,"cookie",0);
            ALTER TABLE employees ADD CONSTRAINT chk_hourly_pay CHECK(hourly_pay >= 10.00);
            INSERT INTO employees VALUES (6,"Sheldon", "Plankton", 5.00, "2023-01-07");
            ALTER TABLE employees DROP CHECK chk_hourly_pay;


Adding Default Value
INSERT INTO products
       VALUES (105, "straw",0.00),
              (106, "napkin",0.00),
              (107, "fork",0.00),
              (108, "spoon",0.00);
DELETE FROM products WHERE product_id >= 104;
CREATE TABLE products (
            product_id INT,
            product_name VARCHAR(25),
            price DECIMAL(4, 2) DEFAULT 0
            );
ALTER TABLE products ALTER price SET DEFAULT 0;

INSERT INTO products(product_id, product_name)
       VALUES (104, "straw"),
              (105, "napkin"),
              (106, "fork"),
              (107, "spoon");

CREATE TABLE transactions(
            transaction_id INT,
            amount DECIMAL(5, 2),
            transaction_date DATETIME DEFAULT NOW()
            );

INSERT INTO transactions (transaction_id, amount) VALUES (1, 4.99);
DROP TABLE transactions;
SELECT * FROM transactions;


Primary Keys
can be applied to a column, that must be unique and null, it is typically used as a unique identifier, like identifying by social security number, in matrix we have many
JOHN SMITH's from MATRIX but we can make a difference between them through key identifier, only one primary key per table is aloud
CREATE TABLE transactions(
            transaction_id INT PRIMARY KEY,
            amount DECIMAL(5, 2)
            );
             this will be added if the primary key was not added when you created the table
             ALTER TABLE transactions ADD CONSTRAINT PRIMARY KEY (transaction_id);
             so transaction has to be null and unique
             INSERT INTO transactions VALUES(1002, 3.38);
             SELECT * FROM transactions;
             SELECT amount FROM transactions WHERE transaction_id = 1002;


Auto Increment
can only applied to a column that is set as a key
DROP TABLE transactions;

CREATE TABLE transactions (
            transaction_id INT PRIMARY KEY AUTO_INCREMENT,
            amount DECIMAL(5, 2)
            );
SELECT * FROM transactions;

INSERT INTO transactions (amount) VALUES(4.99);
SELECT * FROM transactions;

ALTER TABLE transactions AUTO_INCREMENT = 1000;
           delete all the rows
```

```
            DELETE FROM transactions;
            SELECT * FROM transactions;
            INSERT INTO transactions (amount) VALUES(4.99);
            SELECT * FROM transactions;

Foreign Keys
will create a link that prevents any action to destroy the link between them
CREATE TABLE customers(
             customer_id INT PRIMARY KEY AUTO_INCREMENT,
             first_name VARCHAR(50),
             last_name VARCHAR(50)
             );
SELECT * FROM customers;

INSERT INTO customers (first_name, last_name)
       VALUES ("Fred", "Fish"),
              ("Larry", "Lobster"),
              ("Bubble", "Bass");
SELECT * FROM customers;

DROP TABLE transactions;
CREATE TABLE transactions (
             transactions_id INT PRIMARY KEY AUTO_INCREMENT,
             amount DECIMAL(5, 2),
             customer_id INT,
             FOREIGN KEY(customer_id) REFERENCES customers(customer_id)
             );
SELECT * FROM transactions;
         this will drop the foreign key
         ALTER TABLE transactions DROP FOREIGN KEY transactions_ibfk_1;
         This will add a name for the constraint
         ALTER TABLE transactions ADD CONSTRAINT fk_customer_id FOREIGN KEY(customer_id) REFERENCES customers(customer_id);
         DELETE FROM transactions;
         SELECT * FROM transactions;
         ALTER TABLE transactions AUTO_INCREMENT = 1000;

         INSERT INTO transactions (amount, customer_id)
                VALUES (4.00, 3),
                       (2.89, 2),
                       (3.38, 3),
                       (4.99, 1);
                       SELECT * FROM transactions;
                       DELETE FROM customers WHERE customer_id =3;
                       SELECT * FROM transactions;


Joins
Inner, Left, Right
INSERT INTO transactions (amount, customer_id) VALUES(1.00, NULL);
SELECT * FROM transactions;
SET AUTOCOMMIT = ON;
DELETE FROM transactions where transactions_id = 1007;
SELECT * FROM transactions;
INSERT INTO customers (first_name, last_name) VALUES("Poppy", "Puff");
SELECT * FROM customers;
SELECT transactions_id, amount, first_name, last_name FROM transactions INNER JOIN customers ON transactions.customer_id = customers.customer_id;
SELECT * FROM transactions RIGHT JOIN customers ON transactions.customer_id = customers.customer_id;

Q: Create report of Martian names and their base, A: JOIN martian & base tables by base_id
SELECT * FROM martian INNER JOIN base ON martian.base_id = base.base_id;
create only tables that we need
SELECT * FROM martian INNER JOIN base ON martian.base_id = base.base_id;
SELECT first_name,last_name, base_name FROM martian INNER JOIN base ON martian.base_id = base.base_id;
SELECT martian.martian_id, base.base_id, base.base_name FROM martian INNER JOIN base ON martian.base_id = base.base_id;
using alias
SELECT m.martian_id, b.base_id, b.base_name FROM martian AS m INNER JOIN base AS b ON m.base_id = b.base_id;


Functions
SELECT * FROM transactions;
```

```sql
SELECT COUNT(amount) AS "Today's transaction" FROM transactions;
SELECT MAX(amount) AS maximum FROM transactions;
SELECT MIN(amount) AS minimum FROM transactions;
SELECT AVG(amount) AS average FROM transactions;
SELECT SUM(amount) AS sum FROM transactions;
SELECT * FROM employees;
SELECT CONCAT(first_name," ", last_name) AS full_name FROM employees;


Logical operators
used to check more than one condition
ALTER TABLE employees ADD COLUMN job VARCHAR(25) AFTER hourly_pay;
SELECT * FROM employees;
UPDATE employees SET job = "janitor" WHERE employee_id = 6;
DELETE FROM employees WHERE employee_id = 7;
SELECT * FROM employees;
SELECT * FROM employees WHERE hire_date < "2023-01-5" AND job = "cook";
SELECT * FROM employees WHERE job = "cook" OR job = "cashier";
        reverse everything you say
        SELECT * FROM employees WHERE NOT job = "manager"; --
        SELECT * FROM employees WHERE NOT job = "manager" AND NOT job = "asst. manager";
        SELECT * FROM employees WHERE hire_date BETWEEN "2023-01-04" AND "2023-01-07";
        SELECT * FROM employees WHERE job IN ("cook", "cashier", "janitor");


Wild cards
( % or _ )
used to substitute one or more character in a string, it is like the TV game when turning back the letters to find the words
        you must write LIKE and s% will return what begins with s
        SELECT * FROM employees WHERE first_name LIKE "s%";
        SELECT * FROM employees WHERE hire_date LIKE "2023%";
        finishes with the letter r
        SELECT * FROM employees WHERE last_name LIKE "%r";
        _ is one random character
        SELECT * FROM employees WHERE job LIKE "_ook";
        SELECT * FROM employees WHERE hire_date LIKE "____-01-__";
        you can combine them also
        SELECT * FROM employees WHERE job LIKE "_a%";


Order by
ASC is the default no need to write it
SELECT * FROM employees ORDER BY last_name;
SELECT * FROM employees ORDER BY last_name DESC;
order by more than column if they share both amount
SELECT * FROM transactions ORDER BY amount, customer_id;


Limit clause
Limit clause is used to limit the number of records
Useful if you're working with a lot of data
Can be used to display a large data on pages (pagination)
       limit with order works well
       SELECT * FROM customers ORDER BY last_name LIMIT 2;
       will return the 2nd one but only that
       SELECT * FROM customers ORDER BY last_name LIMIT 2,1;


Unions
combines the results of two or more SELECT statements, but both needs to have same number of column
if you use UNION ALL then it will take also duplicates
SELECT first_name, last_name FROM employees UNION SELECT first_name, last_name FROM customers;


SELF Join
join another copy of a table to itself, used to compare rows of the same table, helps to display a hierarchy of data
ALTER TABLE customers ADD referral_id INT;
UPDATE customers SET referral_id = 2 WHERE customer_id = 4;
SELECT * FROM customers;
```

```sql
SELECT a.customer_id, a.first_name, a.last_name,
       CONCAT(b.first_name," ", b.last_name) AS "referred_by"

FROM customers AS a
     INNER JOIN customers AS b
     ON a.referral_id = b.customer_id;
SELECT * FROM employees;

ALTER TABLE employees ADD supervisor_id INT;
SELECT * FROM employees;
UPDATE employees SET supervisor_id = 1 WHERE employee_id = 5;
SELECT * FROM employees;

SELECT a.first_name, a.last_name,
       CONCAT(b.first_name," ", b.last_name) AS "reports to"
       FROM employees AS a
       INNER JOIN employees AS b
       ON a.supervisor_id = b.employee_id;


Views
it is not a real table but behaves as one, so any change in the original table will affect the virtual one, you can interact with it as it was real
SELECT * FROM employees;
CREATE VIEW employee_attendance AS
       SELECT first_name, last_name
       FROM employees;

SELECT * FROM employee_attendance ORDER BY last_name ASC;
DROP VIEW employee_attendance;
SELECT * FROM customers;
ALTER TABLE customers ADD COLUMN email VARCHAR(50);
SELECT * FROM customers;

UPDATE customers
       SET email = "BBass@gmail.com"
       WHERE customer_id = 4;

SELECT * FROM customers;
CREATE VIEW customer_emails AS SELECT email FROM customers;
SELECT * FROM customer_emails;
INSERT INTO customers VALUES(5,"PEARL","Krabs",NULL,"PKrabs@gmail.com");
SELECT * FROM customers;
SELECT * FROM customer_emails

Write a query returning the data to appear in the view
SELECT * FROM martian_confidential;
CREATE VIEW martian_public AS SELECT martian_id, first_name, last_name, base_id, super_id FROM martian_confidential;
SELECT * FROM martian_public;
will give error as like it was not there
SELECT salary, dna-id FROM martian_public;
Task: Create view of ALL people in Mars, required columns(first name, last name, unique ID, martian or visitor?)
      Use Union to combine 2 queries (combine 2 results in 1 set, columns and type must be the same)
      SELECT martian_id, first_name, last_name,  'Martian' AS status FROM martian_public UNION SELECT visitor_id, first_name, last_name,  'Visitor' AS status FROM visitor;
      we will do this as some visitors and martians have the same id so we concat with a prefix to know which one is martian or visitor
      SELECT CONCAT('m', martian_id) AS id, first_name, last_name,  'Martian' AS status FROM martian_public UNION SELECT CONCAT('v', visitor_id), first_name,
last_name,  'Visitor' AS status FROM visitor;
      add view to make it easier to handle
      CREATE VIEW people_on_mars AS SELECT CONCAT('m', martian_id) AS id, first_name, last_name,  'Martian' AS status FROM martian_public UNION SELECT CONCAT('v',
visitor_id), first_name, last_name,  'Visitor' AS status FROM visitor;
       SELECT * FROM people_on_mars ORDER BY last_name;
Task: Create view, name:base_storage, contents:Supply quantities at each base
      COALESCE returns 0 instead of NULL and use AS to have ALIAS quantity instead of COALESCE
      CREATE VIEW base_storage AS SELECT b.base_id, s.supply_id, s.name,COALESCE((SELECT quantity FROM inventory WHERE base_id = b.base_id AND supply_id = s.supply_id), 0)
AS quantity FROM base AS b CROSS JOIN supply AS s;
      SELECT * FROM base_storage;


Indexes
takes less time to search a column, it is a BTree data structure BUT updates will take longer
Search faster with Index, scan faster, it is like index of a book, you go to index and find the list
```

```
better use here as we do not add customers every day but at transaction table that is updated more so would not have sense to use index
SELECT * FROM customers;
will show index of a table
CREATE INDEX last_name_idx

SHOW INDEXES FROM customers;
       ON customers(last_name);

see indexes
SHOW INDEXES FROM customers;
this will be speed up by the index created, this was a single column index
SELECT * FROM customers WHERE last_name = "Puff";
multi column index
CREATE INDEX last_name_first_name_idx ON customers(last_name, first_name);
SHOW INDEXES FROM customers;
ALTER TABLE customers DROP INDEX last_name_idx;
SHOW INDEXES FROM customers;
SELECT * FROM customers WHERE last_name = "Puff" AND first_name = "Poppy";

SELECT * FROM song;
Example: Create index on 'artist' column of 'song' table -- idx means index
CREATE INDEX song_artist_idx ON song(artist);
88 msec
SELECT COUNT(*) FROM song WHERE artist = 'U2';
create single index
CREATE INDEX song_artist_idx ON song(artist);
72 msec <-Improvement of time search
SELECT COUNT(*) FROM song WHERE artist = 'U2';
multi index, order matters
CREATE INDEX person_first_name_last_name_idx ON person(last_name, first_name);
SELECT COUNT(*) FROM person WHERE last_name = 'Williams' AND first_name = 'John';


Subqueries
query within another query
SELECT * FROM employees;
SELECT first_name, last_name, hourly_pay, (SELECT AVG(hourly_pay) FROM employees) as avg_pay FROM employees;
SELECT first_name, last_name, hourly_pay FROM employees WHERE hourly_pay > (SELECT AVG(hourly_pay) FROM employees);
SELECT * FROM transactions;

SELECT first_name, last_name FROM customers
       WHERE customer_id
       IN (SELECT DISTINCT customer_id
       FROM transactions
       every person that placed an order
       WHERE customer_id IS NOT NULL);


Group by
this aggregate all rows by a specific column and used often with aggregate functions ex. SUM(),MAX(),MIN(),AVG(),COUNT()
SELECT * FROM transactions;
DROP TABLE IF EXISTS transactions;

CREATE TABLE transactions (
    transaction_id INT PRIMARY KEY AUTO_INCREMENT,
    amount DECIMAL(5, 2),
    customer_id INT,
    order_date DATE,
    FOREIGN KEY (customer_id)
    REFERENCES customers(customer_id)
);

INSERT INTO transactions
       VALUES (1000, 4.99, 3, "2023-01-01"),
              (1001, 2.89, 2, "2023-01-01"),
              (1002, 3.38, 3, "2023-01-02"),
              (1003, 4.99, 1, "2023-01-02"),
              (1004, 1.00, NULL, "2023-01-03"),
              (1005, 2.49, 4, "2023-01-03"),
              (1006, 5.48, NULL, "2023-01-03");
```

```sql
SELECT * FROM transactions;
what was the sum amount on each day
SELECT SUM(amount), order_date
FROM transactions GROUP BY order_date;
max transactions that occurred on that days
SELECT COUNT(amount), order_date
       FROM transactions
       GROUP BY order_date;
SELECT COUNT(amount), customer_id
       FROM transactions
       GROUP BY customer_id
       customer id that visited more than once
       HAVING COUNT(amount) > 1 AND customer_id IS NOT NULL;


Rollup
extension of the Group By clause, makes another row and sows the grand total, it is called super-aggregate value
SELECT * FROM transactions;
SELECT SUM(amount), order_date
       FROM transactions
       GROUP BY order_date WITH ROLLUP;

SELECT COUNT(transaction_id), order_date
       FROM transactions
       GROUP BY order_date WITH ROLLUP;

SELECT COUNT(transaction_id) AS "# of orders", customer_id
       FROM transactions
       GROUP BY customer_id WITH ROLLUP;

SELECT SUM(hourly_pay) AS "hourly pay", employee_id
       FROM employees
       GROUP BY employee_id WITH ROLLUP;


OnDelte
On Delete Set Null = When FK is deleted, replace FK with NULL
On Delete Cascade = When a FK is deleted, delete row
SET foreign_key_checks = 0;
DELETE FROM customers WHERE customer_id = 4;
SELECT * FROM customers;
SET foreign_key_checks = 1;
SELECT * FROM transactions;
INSERT INTO customers VALUES(4, "Poppy","Puff",2 ,"PPuff@gmail.com");
SELECT * FROM customers;

IF we create new table we can add here the ON DELETE
CREATE TABLE transactions (
             transaction_id INT PRIMARY KEY,
             amount DECIMAL(5,2),
             customer_id INT,
             order_date DATE,
             FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
             ON DELETE SET NULL);

ALTER TABLE transactions DROP FOREIGN KEY fk_customer_id;

ALTER TABLE transactions
            ADD CONSTRAINT fk_customer_id
            FOREIGN KEY(customer_id)
            REFERENCES customers(customer_id)
            ON DELETE SET NULL;

SELECT * FROM transactions;
DELETE FROM customers WHERE customer_id = 4;
SELECT * FROM transactions;
INSERT INTO customers VALUES(4, "Poppy","Puff",2 ,"PPuff@gmail.com");
SELECT * FROM customers;
ALTER TABLE transactions DROP FOREIGN KEY fk_customer_id;
```

```sql
UPDATE transactions
       SET customer_id = 4
       WHERE transaction_id = 1005;

SELECT * FROM transactions;

ALTER TABLE transactions
      ADD CONSTRAINT fk_transactions_id
      FOREIGN KEY(customer_id) REFERENCES customers(customer_id)
      ON DELETE CASCADE;

DELETE FROM customers WHERE customer_id = 4;
SELECT * FROM transactions;
```

Stored Procedures

```sql
DELIMITER $$
CREATE PROCEDURE get_customers()
BEGIN
SELECT * FROM customers;
END $$
DELIMITER ;

CALL get_customers();
DROP PROCEDURE get_customers;

DELIMITER $$
CREATE PROCEDURE find_customer(IN id INT)
BEGIN
    SELECT *
    FROM customers
    WHERE customer_id = id;
END $$
DELIMITER ;

CALL find_customer(5);
DROP PROCEDURE find_customer;

DELIMITER $$
CREATE PROCEDURE find_customer(IN f_name VARCHAR(50), IN l_name VARCHAR(50))
BEGIN
    SELECT *
    FROM customers
    WHERE first_name = f_name AND last_name = l_name;
END $$
DELIMITER ;

CALL find_customer("Larry", "Lobster");
```

so stored procedure reduces network traffic, increase performance, secure as admin can grant permission to use but it increases usage of memory of every connection

Triggers

when an event happens like INSERT,UPDATE ot DELETE, do something like check data, handle error, audit tables

```sql
SELECT * FROM employees;
ALTER TABLE employees
       ADD COLUMN salary DECIMAl(10, 2)
       AFTER hourly_pay;
SELECT * FROM employees;

UPDATE employees
       SET salary = hourly_pay * 2080;
SELECT * FROM employees;

CREATE TRIGGER before_hourly_pay_update
       BEFORE UPDATE ON employees
       FOR EACH ROW
       SET NEW.salary = (NEW.hourly_pay * 2080);

SHOW TRIGGERS;
```

```sql
UPDATE employees
       SET hourly_pay = 50
       WHERE employee_id = 1;
SELECT * FROM employees;

UPDATE employees
SET hourly_pay = hourly_pay + 1;
SELECT * FROM employees;

DELETE FROM employees
       WHERE employee_id = 6;
SELECT * FROM employees;

CREATE TRIGGER before_hourly_pay_insert
       BEFORE INSERT ON employees
       FOR EACH ROW
       SET NEW.salary = (NEW.hourly_pay * 2080);

INSERT INTO employees VALUES(6, "Sheldon", 10, NULL, "janitor", "2023-01-07",5);
SELECT * FROM employees;

CREATE TABLE expenses(
    expense_id INT PRIMARY KEY,
    expense_name VARCHAR(50),
    expense_total DECIMAL(10,2)
);
SELECT * FROM expenses;

INSERT INTO expenses
       VALUES (1, "salaries", 0),
              (2, "supplies", 0),
              (3, "taxes", 0);
SELECT * FROM expenses;

UPDATE expenses
       this will be a nested query
       SET expense_total = (SELECT SUM(salary) FROM employees)
       WHERE expense_name = "salaries";
SELECT * FROM expenses;

CREATE TRIGGER after_salary_delete
       AFTER DELETE ON employees
       FOR EACH ROW
       UPDATE expenses
       SET expense_total = expense_total - OLD.salary
       WHERE expense_name = "salaries";

DELETE FROM employees WHERE employee_id =6;
SELECT * FROM expenses;

CREATE TRIGGER after_salary_insert
       AFTER INSERT ON employees
       FOR EACH ROW
       UPDATE expenses
       SET expense_total = expense_total + NEW.salary
       WHERE expense_name = "salaries";

INSERT INTO employees
       VALUES (6, "Sheldon", "Plankton", 10, NULL, "janitor","2023 -01-07",5);
SELECT * FROM expenses;

CREATE TRIGGER after_salary_update
       AFTER UPDATE ON employees
       FOR EACH ROW
       UPDATE expenses
       SET expense_total = expense_total + (NEW.salary - OLD.salary)
       WHERE expense_name = "salaries";
UPDATE employees SET hourly_pay = 100 WHERE employee_id = 1;SELECT * FROM expenses;
```

OVERVIEW

SQL

- Often times, in order for us to build the most functional website we can, we depend on a **database** to store information.
- If you've ever used Microsoft Excel or Google Spreadsheets (among others), odds are you're familiar with the notion of a database: a hierarchically organized set of tables, each of which contains a set of rows and columns.

SQL

- SQL (the *Structured Query Language*) is a programming language whose purpose is to **query** a database.
- **MySQL** is one open-source platform on which you can establish the type of relational database that SQL is most adept at working with.
  - **SQLite** is another, which we've actually use in CS50 since 2016.
- Many installations of SQL come with a GUI tool called **phpMyAdmin** which can be used to execute database queries in a more user-friendly way.

SQL

- After you create a database, the next thing you'll most likely want to do is create a **table**.
  - The syntax for doing this is actually a bit awkward to do programmatically, at least at the outset, and so this is where phpMyAdmin will come in handy.
- As part of the process of creating a table, you'll be asked to specify all of the **columns** in that table.
- Thereafter, all your queries will refer to **rows** of the table.

SQL

- Each column of your SQL table is capable of holding data of a particular data type.

INT	SMALLINT	TINYINT	MEDIUMINT	BIGINT
DECIMAL	FLOAT	BIT	DATE	TIME
DATETIME	TIMESTAMP	CHAR	VARCHAR	BINARY
BLOB	TEXT	ENUM	GEOMETRY	LINESTRING

SQL

- Unlike in C, the CHAR data type in SQL does not refer to a single character. Rather, it is a fixed-length string.
  - In most relational databases, including MySQL, you actually specify the fixed-length as part of the type definition, e.g. `CHAR(10)`.
- A VARCHAR refers to a variable-length string.
  - VARCHARs also require you to specify the **maximum** possible length of a string that could be stored in that column, e.g. `VARCHAR(99)`.

SQL

- SQLite has these data types as well, but affiliates each with a "type affinity" to simplify things.

NULL	INTEGER	REAL	TEXT	BLOB

SQL

- One other important consideration when constructing a table in SQL is to choose one column to be your **primary key**.
- Primary keys enable rows of a table to be uniquely and quickly identified.
  - Choosing your primary key appropriately can make subsequent operations on the table much easier.
- It is also possible to establish a joint primary key – a combination of two columns that is always guaranteed to be unique.

SQL

- SQL is a programming language, but its vocabulary is fairly limited.
- We will primarily consider just **four** operations that one may perform on a table.

```
INSERT
SELECT
UPDATE
DELETE
```

SQL

users

idnum	username	password	fullname
10	jerry	fus!!!!	Jerry Seinfeld
11	gcostanza	b0sc0	George Costanza

moms

username	mother
jerry	Helen Seinfeld
gcostanza	Estelle Costanza

SQL

- **INSERT**
  - Add information to a table.

```
INSERT INTO
<table>
(<columns>)
VALUES
(<values>)
```

SQL

- When defining the column that ultimately ends up being your table's primary key, it's usually a good idea to have that column be an integer.

- Moreover, so as to eliminate the situation where you may accidentally forget to specify a real value for the primary key column, you can configure that column to **autoincrement**, so it will pre-populate that column for you automatically when rows are added to the table.

SQL

- **INSERT**
  - Add information to a table.

```
INSERT INTO
users
(username, password, fullname)
VALUES
('newman', 'USMAIL', 'Newman')
```

SQL

- **INSERT**
  - Add information to a table.

```
INSERT INTO
moms
(username, mother)
VALUES
('kramer', 'Babs Kramer')
```

SQL

users

idnum	username	password	fullname
10	jerry	fus!!!!	Jerry Seinfeld
11	gcostanza	b0sc0	George Costanza
12	newman	USMAIL	Newman

moms

username	mother
jerry	Helen Seinfeld
gcostanza	Estelle Costanza

SQL

users

idnum	username	password	fullname
10	jerry	fus!!!!	Jerry Seinfeld
11	gcostanza	b0sc0	George Costanza
12	newman	USMAIL	Newman

moms

username	mother
jerry	Helen Seinfeld
gcostanza	Estelle Costanza

SQL

- **SELECT**
  - Extract information from a table.

```
SELECT
<columns>
FROM
<table>
WHERE
<condition>
ORDER BY
<column>
```

SQL

- **SELECT**
  - Extract information from a table.

```
SELECT
idnum, fullname
FROM
users
```

SQL

users

idnum	username	password	fullname
10	jerry	fus!!!!	Jerry Seinfeld
11	gcostanza	b0sc0	George Costanza
12	newman	USMAIL	Newman

moms

username	mother
jerry	Helen Seinfeld
gcostanza	Estelle Costanza
kramer	Babs Kramer

SQL

- **SELECT**
  - Extract information from a table.

```
SELECT
password
FROM
users
WHERE
idnum < 12
```

SQL

users

idnum	username	password	fullname
10	jerry	fus!!!!	Jerry Seinfeld
11	gcostanza	b0sc0	George Costanza
12	newman	USMAIL	Newman

moms

username	mother
jerry	Helen Seinfeld
gcostanza	Estelle Costanza
kramer	Babs Kramer

SQL

- **SELECT**
  - Extract information from a table.

```
SELECT
*
FROM
moms
WHERE
username = 'jerry'
```

SQL

users

idnum	username	password	fullname
10	jerry	fus!!!!	Jerry Seinfeld
11	gcostanza	b0sc0	George Costanza
12	newman	USMAIL	Newman

moms

username	mother
jerry	Helen Seinfeld
gcostanza	Estelle Costanza
kramer	Babs Kramer

SQL

- Databases empower us to organize information into tables efficiently.
  - We don't always need to store every possible relevant piece of information in the same table, but can use relationships across the tables to let us pull information from where we need it.

- What if we now find ourselves in a situation where we need to get a user's full name (from the *users* table) and their mother's name (from the *mother* table).

SQL

users

idnum	username	password	fullname
10	jerry	fus!!!!	Jerry Seinfeld
11	gcostanza	b0sc0	George Costanza
12	newman	USMAIL	Newman

moms

username	mother
jerry	Helen Seinfeld
gcostanza	Estelle Costanza
kramer	Babs Kramer

SQL

- **SELECT (JOIN)**
  - Extract information from multiple tables.

```
SELECT
<columns>
FROM
<table1>
JOIN
<table2>
ON
<predicate>
```

SQL

- **SELECT (JOIN)**
  - Extract information from multiple tables.

```
SELECT
users.fullname, moms.mother
FROM
users
JOIN
moms
ON
users.username = moms.username
```

SQL

users & moms

users.idnum	users.username moms.username	users.password	users.fullname	moms.mother
10	jerry	fus!!!!	Jerry Seinfeld	Helen Seinfeld
11	gcostanza	b0sc0	George Costanza	Estelle Costanza

SQL

- **UPDATE**
  - Modify information in a table.

```
UPDATE
<table>
SET
<column> = <value>
WHERE
<predicate>
```

SQL

- **UPDATE**
  - Modify information in a table.

```
UPDATE
users
SET
password = 'yadayada'
WHERE
idnum = 10
```

SQL

users

idnum	username	password	fullname
10	jerry	yadayada	Jerry Seinfeld
11	gcostanza	b0sc0	George Costanza
12	newman	USMAIL	Newman

moms

username	mother
jerry	Helen Seinfeld
gcostanza	Estelle Costanza
kramer	Babs Kramer

SQL

- **DELETE**
  - Remove information from a table.

```
DELETE FROM
<table>
WHERE
<predicate>
```

SQL

- **DELETE**
  - Remove information from a table.

```
DELETE FROM
users
WHERE
username = 'newman'
```

SQL

users

idnum	username	password	fullname
10	jerry	yadayada	Jerry Seinfeld
11	gcostanza	b0sc0	George Costanza

moms

username	mother
jerry	Helen Seinfeld
gcostanza	Estelle Costanza
kramer	Babs Kramer

SQL

- All of these operations are pretty easy to do in the graphical interface of phpMyAdmin.
- We want a way to do this programmatically, not just typing SQL commands into the "SQL" tab of phpMyAdmin.
- Fortunately, SQL integrates with other programming languages such as Python or PHP very easily.

create a database called "testDB"

```
CREATE DATABASE testDB;
```

drop the existing database "testDB"

```
DROP DATABASE testDB;
```

create a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City

```
CREATE TABLE Persons (
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
```

create a new table called "TestTables" (which is a copy of the "Customers" table)

```
CREATE TABLE TestTable AS SELECT customername, contactname FROM customers;
```

drop the existing table "Shippers"

```
DROP TABLE Shippers;
```

delete the data inside a table, but not the table itself
```
TRUNCATE TABLE table_name;
```

add an "Email" column to the "Customers" table
```
ALTER TABLE Customers ADD Email varchar(255);
```

drop the "Email" column from the "Customers" table
```
ALTER TABLE Customers DROP COLUMN Email;
```

add a column named "DateOfBirth" in the "Persons" table
```
ALTER TABLE Persons ADD DateOfBirth date;
```

change the data type of the column named "DateOfBirth" in the "Persons" table
```
ALTER TABLE Persons MODIFY COLUMN DateOfBirth year;
```

delete the column named "DateOfBirth" in the "Persons" table
```
ALTER TABLE Persons DROP COLUMN DateOfBirth;
```

create a CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that the age of a person must be 18, or older
```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CHECK (Age>=18)
);
```

naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns
```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255),
    CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sandnes')
);
```

create a CHECK constraint on the "Age" column when the table is already created
```
ALTER TABLE Persons ADD CHECK (Age>=18);
```

allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns
```
ALTER TABLE Persons ADD CONSTRAINT CHK_PersonAge CHECK (Age>=18 AND City='Sandnes');
```

drop a CHECK constraint
```
ALTER TABLE Persons DROP CHECK CHK_PersonAge;
```

set a DEFAULT value for the "City" column when the "Persons" table is created
```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255) DEFAULT 'Sandnes'
);
```

DEFAULT constraint can also be used to insert system values, by using functions like CURRENT_DATE()
```
CREATE TABLE Orders (
    ID int NOT NULL,
    OrderNumber int NOT NULL,
    OrderDate date DEFAULT CURRENT_DATE()
);
```

create a DEFAULT constraint on the "City" column when the table is already created
```
ALTER TABLE Persons ALTER City SET DEFAULT 'Sandnes';
```

drop a DEFAULT constraint

```
ALTER TABLE Persons ALTER City DROP DEFAULT;
```

create an index named "idx_lastname" on the "LastName" column in the "Persons" table

```
CREATE INDEX idx_lastname ON Persons (LastName);
```

create an index on a combination of columns, you can list the column names within the parentheses, separated by commas

```
CREATE INDEX idx_pname ON Persons (LastName, FirstName);
```

DROP INDEX statement is used to delete an index in a table

```
ALTER TABLE table_name DROP INDEX index_name;
```

define the "Personid" column to be an auto-increment primary key field in the "Persons" table

```
CREATE TABLE Persons (
    Personid int NOT NULL AUTO_INCREMENT,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    PRIMARY KEY (Personid)
);
```

let the AUTO_INCREMENT sequence start with another value

```
ALTER TABLE Persons AUTO_INCREMENT=100;
```

insert a new record into the "Persons" table, we do NOT have to specify a value for the "Personid" column (a unique value will be added automatically)

```
INSERT INTO Persons (FirstName,LastName) VALUES ('Lars','Monsen');
```

- `DATE` - format YYYY-MM-DD
- `DATETIME` - format: YYYY-MM-DD HH:MI:SS
- `TIMESTAMP` - format: YYYY-MM-DD HH:MI:SS
- `YEAR` - format YYYY or YY

select the records with an OrderDate of "2008-11-11" from the table

```
SELECT * FROM Orders WHERE OrderDate='2008-11-11'
```

use the same SELECT

```
SELECT * FROM Orders WHERE OrderDate='2008-11-11'
```

create a view that shows all customers from Brazil

```
CREATE VIEW [Brazil Customers] AS SELECT CustomerName, ContactName FROM Customers WHERE Country = 'Brazil';
```

query the view above

```
SELECT * FROM [Brazil Customers];
```

create a view that selects every product in the "Products" table with a price higher than the average price

```
CREATE VIEW [Products Above Average Price] AS SELECT ProductName, Price FROM Products WHERE Price > (SELECT AVG(Price) FROM Products);
```

query the view above

```
SELECT * FROM [Products Above Average Price];
```

add the "City" column to the "Brazil Customers" view

```
CREATE OR REPLACE VIEW [Brazil Customers] AS SELECT CustomerName, ContactName, City FROM Customers WHERE Country = 'Brazil';
```

delete with the DROP VIEW

```
DROP VIEW [Brazil Customers];
```

String Data Types

Data type	Description
CHAR(size)	A FIXED length string (can contain letters, numbers, and special characters). The *size* parameter specifies the column length in characters - can be from 0 to 255. Default is 1
VARCHAR(size)	A VARIABLE length string (can contain letters, numbers, and special characters). The *size* parameter specifies the maximum column length in characters - can be from 0 to 65535
BINARY(size)	Equal to CHAR(), but stores binary byte strings. The *size* parameter specifies the column length in bytes. Default is 1
VARBINARY(size)	Equal to VARCHAR(), but stores binary byte strings. The *size* parameter specifies the maximum column length in bytes.
TINYBLOB	For BLOBs (Binary Large OBjects). Max length: 255 bytes
TINYTEXT	Holds a string with a maximum length of 255 characters
TEXT(size)	Holds a string with a maximum length of 65,535 bytes
BLOB(size)	For BLOBs (Binary Large OBjects). Holds up to 65,535 bytes of data
MEDIUMTEXT	Holds a string with a maximum length of 16,777,215 characters
MEDIUMBLOB	For BLOBs (Binary Large OBjects). Holds up to 16,777,215 bytes of data
LONGTEXT	Holds a string with a maximum length of 4,294,967,295 characters
LONGBLOB	For BLOBs (Binary Large OBjects). Holds up to 4,294,967,295 bytes of data
ENUM(val1, val2, val3, ...)	A string object that can have only one value, chosen from a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted. The values are sorted in the order you enter them
SET(val1, val2, val3, ...)	A string object that can have 0 or more values, chosen from a list of possible values. You can list up to 64 values in a SET list

Numeric Data Types

Data type	Description
BIT(*size*)	A bit-value type. The number of bits per value is specified in *size*. The *size* parameter can hold a value from 1 to 64. The default value for *size* is 1.
TINYINT(*size*)	A very small integer. Signed range is from -128 to 127. Unsigned range is from 0 to 255. The *size* parameter specifies the maximum display width (which is 255)
BOOL	Zero is considered as false, nonzero values are considered as true.
BOOLEAN	Equal to BOOL
SMALLINT(*size*)	A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The *size* parameter specifies the maximum display width (which is 255)
MEDIUMINT(*size*)	A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The *size* parameter specifies the maximum display width (which is 255)
INT(*size*)	A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The *size* parameter specifies the maximum display width (which is 255)
INTEGER(*size*)	Equal to INT(size)
BIGINT(*size*)	A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The *size* parameter specifies the maximum display width (which is 255)
FLOAT(*size*, *d*)	A floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. This syntax is deprecated in MySQL 8.0.17, and it will be removed in future MySQL versions
FLOAT(*p*)	A floating point number. MySQL uses the *p* value to determine whether to use FLOAT or DOUBLE for the resulting data type. If *p* is from 0 to 24, the data type becomes FLOAT(). If *p* is from 25 to 53, the data type becomes DOUBLE()
DOUBLE(*size*, *d*)	A normal-size floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter
DOUBLE PRECISION(*size*, *d*)	
DECIMAL(*size*, *d*)	An exact fixed-point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. The maximum number for *size* is 65. The maximum number for *d* is 30. The default value for *size* is 10. The default value for *d* is 0.
DEC(*size*, *d*)	Equal to DECIMAL(size,d)

Date and Time Data Types

Data type	Description
DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'
DATETIME(*fsp*)	A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time
TIMESTAMP(*fsp*)	A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT_TIMESTAMP and ON UPDATE CURRENT_TIMESTAMP in the column definition
TIME(*fsp*)	A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'
YEAR	A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000. MySQL 8.0 does not support year in two-digit format.

MySQL String Functions

Function	Description
ASCII	Returns the ASCII value for the specific character
CHAR_LENGTH	Returns the length of a string (in characters)
CHARACTER_LENGTH	Returns the length of a string (in characters)
CONCAT	Adds two or more expressions together
CONCAT_WS	Adds two or more expressions together with a separator
FIELD	Returns the index position of a value in a list of values
FIND_IN_SET	Returns the position of a string within a list of strings
FORMAT	Formats a number to a format like "#,###,###.##", rounded to a specified number of decimal places
INSERT	Inserts a string within a string at the specified position and for a certain number of characters
INSTR	Returns the position of the first occurrence of a string in another string
LCASE	Converts a string to lower-case
LEFT	Extracts a number of characters from a string (starting from left)
LENGTH	Returns the length of a string (in bytes)
LOCATE	Returns the position of the first occurrence of a substring in a string
LOWER	Converts a string to lower-case
LPAD	Left-pads a string with another string, to a certain length
LTRIM	Removes leading spaces from a string
MID	Extracts a substring from a string (starting at any position)
POSITION	Returns the position of the first occurrence of a substring in a string
REPEAT	Repeats a string as many times as specified
REPLACE	Replaces all occurrences of a substring within a string, with a new substring
REVERSE	Reverses a string and returns the result
RIGHT	Extracts a number of characters from a string (starting from right)
RPAD	Right-pads a string with another string, to a certain length
RTRIM	Removes trailing spaces from a string
SPACE	Returns a string of the specified number of space characters
STRCMP	Compares two strings
SUBSTR	Extracts a substring from a string (starting at any position)
SUBSTRING	Extracts a substring from a string (starting at any position)
SUBSTRING_INDEX	Returns a substring of a string before a specified number of delimiter occurs
TRIM	Removes leading and trailing spaces from a string
UCASE	Converts a string to upper-case
UPPER	Converts a string to upper-case

- `NOT NULL` - Ensures that a column cannot have a NULL value
- `UNIQUE` - Ensures that all values in a column are different
- `PRIMARY KEY` - A combination of a `NOT NULL` and `UNIQUE`. Uniquely identifies each row in a table
- `FOREIGN KEY` - Prevents actions that would destroy links between tables
- `CHECK` - Ensures that the values in a column satisfies a specific condition
- `DEFAULT` - Sets a default value for a column if no value is specified
- `CREATE INDEX` - Used to create and retrieve data from the database very quickly

the "ID", "LastName", and "FirstName" columns will NOT accept NULL values when the "Persons" table is created

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int
);
```

create a `NOT NULL` constraint on the "Age" column when the "Persons" table is already created

```
ALTER TABLE Persons MODIFY Age int NOT NULL;
```

create a `UNIQUE` constraint on the "ID" column when the "Persons" table is created

```
CREATE TABLE Persons (
    ID int NOT NULL,
```

```
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    UNIQUE (ID)
);
```

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT UC_Person UNIQUE (ID,LastName)
);
```

create a UNIQUE constraint on the "ID" column when the table is already created

```
ALTER TABLE Persons ADD UNIQUE (ID);
```

name a UNIQUE constraint, and define a UNIQUE constraint on multiple columns

```
ALTER TABLE Persons ADD CONSTRAINT UC_Person UNIQUE (ID,LastName);
```

drop a UNIQUE constraint

```
ALTER TABLE Persons DROP INDEX UC_Person;
```

create a PRIMARY KEY on the "ID" column when the "Persons"

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    PRIMARY KEY (ID)
);
```

allow naming of a PRIMARY KEY constraint, and define a PRIMARY KEY constraint on multiple columns

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT PK_Person PRIMARY KEY (ID,LastName)
);
```

Note: In the example above there is only ONE PRIMARY KEY (PK_Person). However, the VALUE of the primary key is made up of TWO COLUMNS (ID + LastName).

create a FOREIGN KEY on the "PersonID" column when the "Orders" table is created

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID)
    REFERENCES Persons(PersonID)
);
```

FOREIGN KEY constraint on the "PersonID" column when the "Orders" table is already created

```
ALTER TABLE Orders ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns

```sql
ALTER TABLE Orders ADD CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

drop a FOREIGN KEY constraint
```sql
ALTER TABLE Orders DROP FOREIGN KEY FK_PersonOrder;
```

select all records from a table
```sql
SELECT * FROM Customers;
```

select data from a database
```sql
SELECT CustomerName, City, Country FROM Customers;
```

extract only those records that fulfill a specified condition
```sql
SELECT * FROM Customers WHERE Country = 'Mexico';
```

select all fields from "Customers" where country is "Germany" AND city is "Berlin"
```sql
SELECT * FROM Customers WHERE Country = 'Germany' AND City = 'Berlin';
```

select all fields from "Customers" where city is "Berlin" OR "Stuttgart"
```sql
SELECT * FROM Customers WHERE City = 'Berlin' OR City = 'Stuttgart';
```

select all fields from "Customers" where country is NOT "Germany" and NOT "USA"
```sql
SELECT * FROM Customers WHERE NOT Country = 'Germany';
```

select all fields from "Customers" where country is "Germany" AND city must be "Berlin" OR "Stuttgart" (use parenthesis to form complex expressions)
```sql
SELECT * FROM Customers WHERE Country = 'Germany' AND (City = 'Berlin' OR City = 'Stuttgart');
```

select all customers from the "Customers" table, sorted by the "Country" column
```sql
SELECT * FROM Customers ORDER BY Country;
```

select all customers from the "Customers" table, sorted DESCENDING by the "Country" column
```sql
SELECT * FROM Customers ORDER BY Country DESC;
```

select all customers from the "Customers" table, sorted by the "Country" and the "CustomerName" column. This means that it orders by Country, but if some rows have the same Country, it orders them by CustomerName
```sql
SELECT * FROM Customers ORDER BY Country, CustomerName;
```

select all customers from the "Customers" table, sorted ascending by the "Country" and descending by the "CustomerName" column
```sql
SELECT * FROM Customers ORDER BY Country ASC, CustomerName DESC;
```

insert a new record in the "Customers" table
```sql
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country)
       VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');
```

insert a new record, but only insert data in the "CustomerName", "City", and "Country" columns (CustomerID will be updated automatically)
```sql
INSERT INTO Customers (CustomerName, City, Country) VALUES ('Cardinal', 'Stavanger', 'Norway');
```

list all customers with a NULL value in the "Address" field
```sql
SELECT CustomerName, ContactName, Address FROM Customers WHERE Address IS NULL;
```

list all customers with a value in the "Address" field
```sql
SELECT CustomerName, ContactName, Address FROM Customers WHERE Address IS NOT NULL;
```

update the first customer (CustomerID = 1) with a new contact person and a new city
```sql
UPDATE Customers SET ContactName = 'Alfred Schmidt', City = 'Frankfurt' WHERE CustomerID = 1;
```

update the PostalCode to 00000 for all records where country is "Mexico"
```sql
UPDATE Customers SET PostalCode = 00000 WHERE Country = 'Mexico';
```

delete the customer "Alfreds Futterkiste" from the "Customers" table
```sql
DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';
```

delete all rows in the "Customers" table, without deleting the table
```sql
DELETE FROM Customers;
```

select the first three records from the "Customers" table
```sql
SELECT * FROM Customers LIMIT 3;
```

select the first three records from the "Customers" table, where the country is "Germany"

```
SELECT * FROM Customers WHERE Country='Germany'LIMIT 3;
```

find the price of the cheapest product

```
SELECT MIN(Price) AS SmallestPrice FROM Products;
```

find the price of the most expensive product

```
SELECT MAX(Price) AS LargestPrice FROM Products;
```

find the number of products

```
SELECT COUNT(ProductID) FROM Products;
```

find the average price of all products

```
SELECT AVG(Price) FROM Products;
```

find the sum of the "Quantity" fields in the "OrderDetails" table

```
SELECT SUM(Quantity)FROM OrderDetails;
```

Here are some examples showing different `LIKE` operators with '%' and '_' wildcards:

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that start with "a"
WHERE CustomerName LIKE '%a'	Finds any values that end with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%'	Finds any values that start with "a" and are at least 2 characters in length
WHERE CustomerName LIKE 'a__%'	Finds any values that start with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that start with "a" and ends with "o"

select all customers with a CustomerName starting with "a"

```
SELECT * FROM Customers WHERE CustomerName LIKE 'a%';
```

select users with name starting with the letter N and deletes it

```
DELETE FROM users WHERE name LIKE 'N%';
```

select all customers with a CustomerName ending with "a"

```
SELECT * FROM Customers WHERE CustomerName LIKE '%a';
```

select all customers with a CustomerName that have "or" in any position

```
SELECT * FROM Customers WHERE CustomerName LIKE '%or%';
```

select all customers with a CustomerName that have "r" in the second position

```
SELECT * FROM Customers WHERE CustomerName LIKE '_r%';
```

select all customers with a CustomerName that starts with "a" and are at least 3 characters in length

```
SELECT * FROM Customers WHERE CustomerName LIKE 'a__%';
```

select all customers with a ContactName that starts with "a" and ends with "o"

```
SELECT * FROM Customers WHERE ContactName LIKE 'a%o';
```

select all customers with a CustomerName that does NOT start with "a"

```
SELECT * FROM Customers WHERE CustomerName NOT LIKE 'a%';
```

Symbol	Description	Example
%	Represents zero or more characters	bl% finds bl, black, blue, and blob
_	Represents a single character	h_t finds hot, hat, and hit

The wildcards can also be used in combinations!

Here are some examples showing different `LIKE` operators with '%' and '_' wildcards:

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that starts with "a"
WHERE CustomerName LIKE '%a'	Finds any values that ends with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%_%'	Finds any values that starts with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that starts with "a" and ends with "o"

select all customers with a City starting with "ber"
```
SELECT * FROM Customers WHERE City LIKE 'ber%';
```

select all customers with a City containing the pattern "es"
```
SELECT * FROM Customers WHERE City LIKE '%es%';
```

select all customers with a City starting with "L", followed by any character, followed by "n", followed by any character, followed by "on"
```
SELECT * FROM Customers WHERE City LIKE 'L_n_on';
```

select all customers that are located in "Germany", "France" or "UK"
```
SELECT * FROM Customers WHERE Country IN ('Germany', 'France', 'UK');
```

select all customers that are NOT located in "Germany", "France" or "UK"
```
SELECT * FROM Customers WHERE Country NOT IN ('Germany', 'France', 'UK');
```

select all customers that are from the same countries as the suppliers
```
SELECT * FROM Customers WHERE Country IN (SELECT Country FROM Suppliers);
```

select all products with a price between 10 and 20
```
SELECT * FROM Products WHERE Price BETWEEN 10 AND 20;
```

display the products outside the range of the previous example, use `NOT BETWEEN`
```
SELECT * FROM Products WHERE Price NOT BETWEEN 10 AND 20;
```

select all products with a price between 10 and 20. In addition; do not show products with a CategoryID of 1,2, or 3
```
SELECT * FROM Products WHERE Price BETWEEN 10 AND 20 AND CategoryID NOT IN (1,2,3);
```

select all products with a ProductName between "Carnarvon Tigers" and "Mozzarella di Giovanni"
```
SELECT * FROM Products WHERE ProductName BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni' ORDER BY ProductName;
```

select all products with a ProductName between "Carnarvon Tigers" and "Chef Anton's Cajun Seasoning"
```
SELECT * FROM Products WHERE ProductName BETWEEN "Carnarvon Tigers" AND "Chef Anton's Cajun Seasoning" ORDER BY ProductName;
```

select all products with a ProductName not between "Carnarvon Tigers" and "Mozzarella di Giovanni"
```
SELECT * FROM Products WHERE ProductName NOT BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni' ORDER BY ProductName;
```

select all orders with an OrderDate between '01-July-1996' and '31-July-1996'
```
SELECT * FROM Orders WHERE OrderDate BETWEEN '1996-07-01' AND '1996-07-31';
```

remove bro from the row of the column name of the table users
```
UPDATE users SET name = REPLACE(name, 'bro', '');
```

create two aliases, one for the CustomerID column and one for the CustomerName column
```
SELECT CustomerID AS ID, CustomerName AS Customer FROM Customers;
```

create two aliases, one for the CustomerName column and one for the ContactName column. *Note:* ***Single or double quotation marks are required if the alias name contains spaces***

```
SELECT CustomerName AS Customer, ContactName AS "Contact Person" FROM Customers;
```

create an alias named "Address" that combine four columns (Address, PostalCode, City and Country)

```
SELECT CustomerName, CONCAT_WS(', ', Address, PostalCode, City, Country) AS Address FROM Customers;
```

select all the orders from the customer with CustomerID=4 (Around the Horn). We use the "Customers" and "Orders" tables, and give them the table aliases of "c" and "o" respectively (Here we use aliases to make the SQL shorter)

```
SELECT o.OrderID, o.OrderDate, c.CustomerName FROM Customers AS c, Orders AS o WHERE c.CustomerName='Around the Horn' AND c.CustomerID=o.CustomerID;
```

same as above, but without aliases

```
SELECT Orders.OrderID, Orders.OrderDate, Customers.CustomerName FROM Customers, Orders WHERE Customers.CustomerName='Around the Horn' AND Customers.CustomerID=Orders.CustomerID;
```

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19
10310	77	1996-09-20

Then, look at a selection from the "Customers" table:

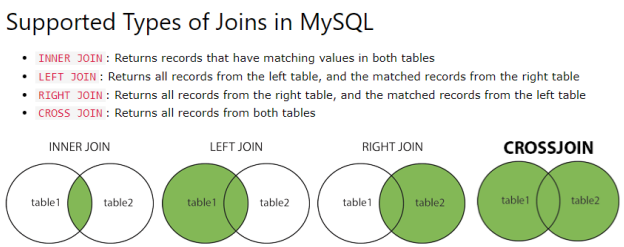
CustomerID	CustomerName	ContactName	Country
1	Alfreds Futterkiste	Maria Anders	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mexico

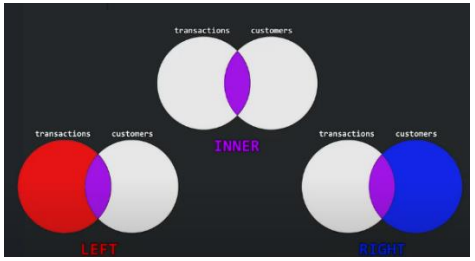
Notice: ***that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column. Then, we can create the following SQL statement (that contains an*** `INNER JOIN`***), that selects records that have matching values in both tables:***

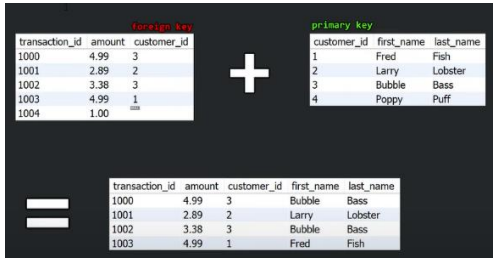
```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate FROM Orders INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

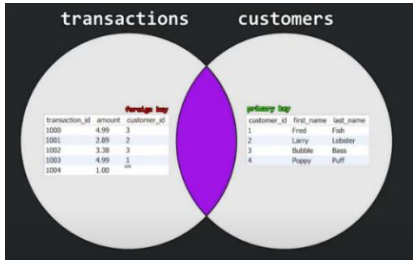
and it will produce something like this:

OrderID	CustomerName	OrderDate
10308	Ana Trujillo Emparedados y helados	9/18/1996
10365	Antonio Moreno Taquería	11/27/1996
10383	Around the Horn	12/16/1996
10355	Around the Horn	11/15/1996
10278	Berglunds snabbköp	8/12/1996



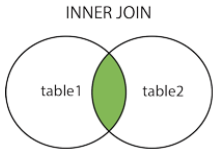


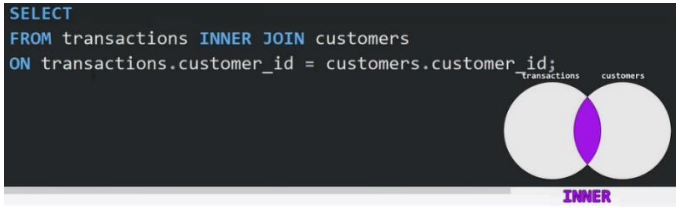




MySQL INNER JOIN Keyword

The `INNER JOIN` keyword selects records that have matching values in both tables.





Below is a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

And a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

select all orders with customer information

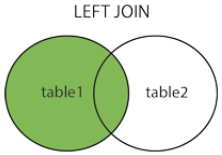
```
SELECT Orders.OrderID, Customers.CustomerName FROM Orders INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

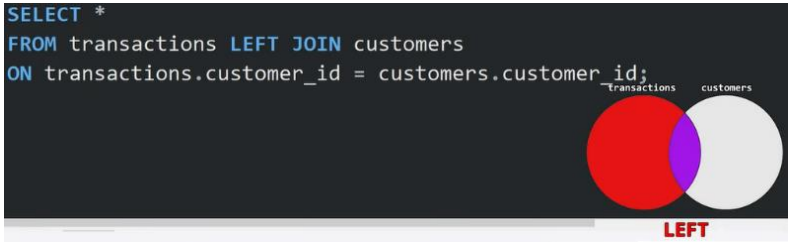
select all orders with customer and shipper information

```
SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName FROM ((Orders INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID) INNER JOIN Shippers ON Orders.ShipperID =
Shippers.ShipperID);
```

MySQL LEFT JOIN Keyword

The `LEFT JOIN` keyword returns all records from the left table (table1), and the matching records (if any) from the right table (table2).





Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

And a selection from the "Orders" table:

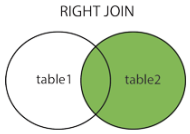
OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

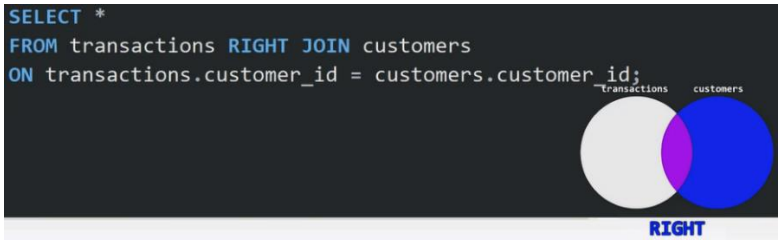
select all customers, and any orders they might have

```
SELECT Customers.CustomerName, Orders.OrderID FROM Customers LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID ORDER BY Customers.CustomerName;
```

MySQL RIGHT JOIN Keyword

The `RIGHT JOIN` keyword returns all records from the right table (table2), and the matching records (if any) from the left table (table1).





Below is a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

And a selection from the "Employees" table:

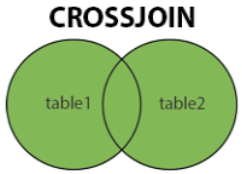
EmployeeID	LastName	FirstName	BirthDate	Photo
1	Davolio	Nancy	12/8/1968	EmpID1.pic
2	Fuller	Andrew	2/19/1952	EmpID2.pic
3	Leverling	Janet	8/30/1963	EmpID3.pic

below to return all employees, and any orders they might have placed

```
SELECT Orders.OrderID, Employees.LastName, Employees.FirstName FROM Orders RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID ORDER BY Orders.OrderID;
```

SQL CROSS JOIN Keyword

The `CROSS JOIN` keyword returns all records from both tables (table1 and table2).



Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

And a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

select all customers, and all orders

```
SELECT Customers.CustomerName, Orders.OrderID FROM Customers CROSS JOIN Orders;
```

Note: The `CROSS JOIN` keyword returns all matching records from both tables whether the other table matches or not. So, if there are rows in "Customers" that do not have matches in "Orders", or if there are rows in "Orders" that do not have matches in "Customers", those rows will be listed as well.

If you add a `WHERE` clause (if table1 and table2 has a relationship), the `CROSS JOIN` will produce the same result as the `INNER JOIN` clause

```
SELECT Customers.CustomerName, Orders.OrderID FROM Customers CROSS JOIN Orders WHERE Customers.CustomerID=Orders.CustomerID;
```

Below is a selection from the "Customers" table:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico

match customers that are from the same city

```
SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.City FROM Customers A, Customers B WHERE A.CustomerID <> B.CustomerID AND A.City = B.City ORDER BY A.City;
```

return the cities (only distinct values) from both the "Customers" and the "Suppliers" table

```
SELECT City FROM Customers UNION SELECT City FROM Suppliers ORDER BY City;
```

return the cities (duplicate values also) from both the "Customers" and the "Suppliers" table

```
SELECT City FROM Customers UNION ALL SELECT City FROM Suppliers ORDER BY City;
```

return the German cities (only distinct values) from both the "Customers" and the "Suppliers" table

```
SELECT City, Country FROM Customers WHERE Country='Germany' UNION SELECT City, Country FROM Suppliers WHERE Country='Germany' ORDER BY City;
```

return the German cities (duplicate values also) from both the "Customers" and the "Suppliers" table

```
SELECT City, Country FROM Customers WHERE Country='Germany' UNION ALL SELECT City, Country FROM Suppliers WHERE Country='Germany' ORDER BY City;
```

list all customers and suppliers

```
SELECT 'Customer' AS Type, ContactName, City, Country FROM Customers UNION SELECT 'Supplier', ContactName, City, Country FROM Suppliers;
```

list the number of customers in each country

```
SELECT COUNT(CustomerID), Country FROM Customers GROUP BY Country;
```

list the number of customers in each country, sorted high to low

```
SELECT COUNT(CustomerID), Country FROM Customers GROUP BY Country ORDER BY COUNT(CustomerID) DESC;
```

Below is a selection from the "Orders" table in the Northwind sample database:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10248	90	5	1996-07-04	3
10249	81	6	1996-07-05	1
10250	34	4	1996-07-08	2

And a selection from the "Shippers" table:

ShipperID	ShipperName
1	Speedy Express
2	United Package
3	Federal Shipping

MySQL Compound Operators

Operator	Description
+=	Add equals
-=	Subtract equals
*=	Multiply equals
/=	Divide equals
%=	Modulo equals
&=	Bitwise AND equals
^-=	Bitwise exclusive equals
|*=	Bitwise OR equals

MySQL Logical Operators

Operator	Description
ALL	TRUE if all of the subquery values meet the condition
AND	TRUE if all the conditions separated by AND is TRUE
ANY	TRUE if any of the subquery values meet the condition
BETWEEN	TRUE if the operand is within the range of comparisons
EXISTS	TRUE if the subquery returns one or more records
IN	TRUE if the operand is equal to one of a list of expressions
LIKE	TRUE if the operand matches a pattern
NOT	Displays a record if the condition(s) is NOT TRUE
OR	TRUE if any of the conditions separated by OR is TRUE
SOME	TRUE if any of the subquery values meet the condition

list the number of orders sent by each shipper

```
SELECT Shippers.ShipperName, COUNT(Orders.OrderID) AS NumberOfOrders FROM Orders LEFT JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID GROUP BY ShipperName;
```

list the number of customers in each country. Only include countries with more than 5 customers

```
SELECT COUNT(CustomerID), Country FROM Customers GROUP BY Country HAVING COUNT(CustomerID) > 5;
```

list the number of customers in each country, sorted high to low (Only include countries with more than 5 customers)

```
SELECT COUNT(CustomerID), Country FROM Customers GROUP BY Country HAVING COUNT(CustomerID) > 5 ORDER BY COUNT(CustomerID) DESC;
```

list the employees that have registered more than 10 orders

```
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders FROM (Orders INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID) GROUP BY LastName HAVING COUNT(Orders.OrderID) > 10;
```

list if the employees "Davolio" or "Fuller" have registered more than 25 orders

```
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders FROM Orders INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID WHERE LastName = 'Davolio' OR LastName = 'Fuller'
GROUP BY LastName HAVING COUNT(Orders.OrderID) > 25;
```

return TRUE and lists the suppliers with a product price less than 20

```
SELECT SupplierName FROM Suppliers WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID = Suppliers.supplierID AND Price < 20);
```

return TRUE and lists the suppliers with a product price equal to 22

```
SELECT SupplierName FROM Suppliers WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID = Suppliers.supplierID AND Price = 22);
```

list the ProductName if it finds ANY records in the OrderDetails table has Quantity equal to 10 (this will return TRUE because the Quantity column has some values of 10)

```
SELECT ProductName FROM Products WHERE ProductID = ANY(SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
```

list the ProductName if it finds ANY records in the OrderDetails table has Quantity larger than 99 (this will return TRUE because the Quantity column has some values larger than 99)

```
SELECT ProductName FROM Products WHERE ProductID = ANY(SELECT ProductID FROM OrderDetails WHERE Quantity > 99);
```

list the ProductName if it finds ANY records in the OrderDetails table has Quantity larger than 1000 (this will return FALSE because the Quantity column has no values larger than 1000)

```
SELECT ProductName FROM Products WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity > 1000);
```

list ALL the product names

```
SELECT ALL ProductName FROM Products WHERE TRUE;
```

list the ProductName if ALL the records in the OrderDetails table has Quantity equal to 10. This will of course return FALSE because the Quantity column has many different values (not only the value of 10)

```
SELECT ProductName FROM Products WHERE ProductID = ALL (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
```

copy "Suppliers" into "Customers" (the columns that are not filled with data, will contain NULL)

```
INSERT INTO Customers (CustomerName, City, Country) SELECT SupplierName, City, Country FROM Suppliers;
```

copy "Suppliers" into "Customers" (fill all columns)

```
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country) SELECT SupplierName, ContactName, Address, City, PostalCode, Country FROM Suppliers;
```

copy "Suppliers" into "Customers" (fill all columns)

```
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode, Country) SELECT SupplierName, ContactName, Address, City, PostalCode, Country FROM Suppliers;
```

copy only the German suppliers into "Customers"

```
INSERT INTO Customers (CustomerName, City, Country) SELECT SupplierName, City, Country FROM Suppliers WHERE Country='Germany';
```

below to go through conditions and returns a value when the first condition is met

```
SELECT OrderID, Quantity, CASE WHEN Quantity > 30 THEN 'The quantity is greater than 30' WHEN Quantity = 30 THEN 'The quantity is 30' ELSE 'The quantity is under 30' END AS QuantityText
FROM OrderDetails;
```

order the customers by City. However, if City is NULL, then order by Country

```
SELECT CustomerName, City, Country FROM Customers ORDER BY (CASE WHEN City IS NULL THEN Country ELSE City END);
```

"UnitsOnOrder" column is optional, and may contain NULL values

```
SELECT ProductName, UnitPrice * (UnitsInStock + UnitsOnOrder FROM Products;
```

return an alternative value if an expression is NULL

```
SELECT ProductName, UnitPrice * (UnitsInStock + IFNULL(UnitsOnOrder, 0)) FROM Products;
```

use the `COALESCE()` function

```
SELECT ProductName, UnitPrice * (UnitsInStock + COALESCE(UnitsOnOrder, 0)) FROM Products;
```

single-line comment as an explanation

```
-- Select all: SELECT * FROM Customers;
```

use a single-line comment to ignore the end of a line

```
SELECT * FROM Customers -- WHERE City='Berlin';
```

MySQL

Connect with python

```python
""" Connect and check connection"""
import mysql.connector

head_office = mysql.connector.connect(
    host = "localhost",
    user = "root",
    passwd = "Sitaru3!",
)

print(head_office)



""" Create a db (it has to be created manually not like sqlite automatically)"""
import mysql.connector

head_office = mysql.connector.connect(
    host = "localhost",
    user = "root",
    passwd = "Sitaru3!",
)
manager = head_office.cursor()
manager.execute("CREATE DATABASE warehouse_dubai")



""" Read from databases """
import mysql.connector

head_office = mysql.connector.connect(
    host = "localhost",
    user = "root",
    passwd = "Sitaru3!",
)
manager = head_office.cursor()
manager.execute("SHOW DATABASES")
for warehouse in manager:
    print(warehouse[0])



""" Create a table in database """
import mysql.connector
```

```
head_office = mysql.connector.connect(
    host = "localhost",
    user = "root",
    passwd = "Sitaru3!",
    database  = "warehouse_dubai",# telling the programme that we are going to use just this dataase
)
manager = head_office.cursor()

manager.execute("CREATE TABLE users(name VARCHAR(255), email VARCHAR(255), age INTEGER(10), user_id INTEGER AUTO_INCREMENT PRIMARY KEY)")
manager.execute("SHOW TABLES")
for table in manager:
    print(table)



""" Check what tables we created only in warehouse_dubai """
import mysql.connector

head_office = mysql.connector.connect(
    host = "localhost",
    user = "root",
    passwd = "Sitaru3!",
    database  = "warehouse_dubai",# telling the programme that we are going to use just this dataase
)
manager = head_office.cursor()

manager.execute("SHOW TABLES")
for table in manager:
    print(table[0])



""" Insert 1 row only in warehouse_dubai database"""
import mysql.connector

head_office = mysql.connector.connect(
    host = "localhost",
    user = "root",
    passwd = "Sitaru3!",
    database  = "warehouse_dubai",
)
manager = head_office.cursor()
add_on_shelf = "INSERT INTO users(name,email,age) VALUES (%s,%s,%s)"
shelf_1 = ("Harvey Specter","harvey.specter@suits.com",50)
manager.execute(add_on_shelf, shelf_1)
head_office.commit()



""" Insert multiple entries manually with single variable"""
import mysql.connector

head_office = mysql.connector.connect(
    host = "localhost",
    user = "root",
    passwd = "Sitaru3!",
    database  = "warehouse_dubai",
)
manager = head_office.cursor()
add_on_shelf = "INSERT INTO users(name,email,age) VALUES (%s,%s,%s)"
shelf_1 = [
    ("Wes","wes@brown.com",50),
    ("Steph", "steph@Kuewa.com",40),
    ("Dan","dan@pas.com",40)]
manager.executemany(add_on_shelf, shelf_1)
head_office.commit()
```

```
""" Deleting a record or a table """
manager_Delete_query = """Delete from users WHERE user_id = 13"""
manager.execute(manager_Delete_query)
head_office.commit()



import mysql.connector

head_office = mysql.connector.connect(
    host = "localhost",
    user = "root",
    passwd = "Sitaru3!",
    database  = "warehouse_dubai",
)
manager = head_office.cursor()

first_name = input("insert your name: ")

name = "INSERT INTO users (name) VALUES (%s)"
manager.execute(name, (first_name, ))
head_office.commit()
```

PostgreSQL

“”” Postgres Elephants never forget “””

```
Connect with python

""" Read with pandas """

import psycopg2
import pandas as po
import numpy as np

head_office = psycopg2.connect(
    host = "localhost",
    database = "postgres",
    user = "postgres",
    password = "Sitaru3!",
)
manager = head_office.cursor()

manager = ''' SELECT * FROM earthquake'''
command = po.read_sql_query(manager, head_office)
command.head()

# In[ ]:

""" Measure speed of INSERT queries
INsert 10,000 rows in 2 different ways
1.Insert 10,000 rows one at a time
2.Insert 10,000 ros in a single batch"""
import psycopg2
import time

head_office = psycopg2.connect(
```

```
    host = "localhost",
    database = "postgres",
    user = "postgres",
    password = "Sitaru3!",
)
manager = head_office.cursor()

#Number of rows to add in each batch
n = 10000

# Generate single INSERT INTO query
single_query = """ INSERT INTO POST (user_id, post_text)
                VALUES (1, 'ALL work and no play makes Jack a dull boy.');"""

# Generate one BIG query
big_query = "INSERT INTO post (user_id, post_text) VALUES "
for i in range(n):
    big_query += "(1, 'ALL work and no play makes Jack a dull boy.'),"
big_query = big_query.strip(',') + ';' # Replace trailing ',' with ';'

# Time the 'n' individual queries
start_time = time.time()
for i in range(n):
    manager.execute(single_query)
head_office.commit()
stop_time = time.time()
print("{0} individual queries took {1} secounds.".format(n, stop_time - start_time))s

#Time the BIG Query
start_time = time.time()
manager.execute(big_query)
head_office.commit()
stop_time = time.time()
print("The query with {0} rows took {1} secounds.".format(n, stop_time - start_time))

#Close both cursor and connection to databse
manager.close()
head_office.close()
```

```
""" Find out the log in details in PostgeSQL (must be run inside PgAdmin)"""
select pid as process_id,
       usename as username,
       datname as database_name,
       client_hostname,
       application_name,
       backend_start,
       state,
       wait_event,
       state_change
from pg_stat_activity WHERE state = 'active';
```

SQLite

Connect with python 1

```
""" Create a table (table is like excel sheet), will connect and create db automatically """
import sqlite3
#manager = sqlite3.connect(':memory:') -> to temporarry use database without saving
warehouse = sqlite3.connect("C:\\gitMain\\warehouse.db")
manager = warehouse.cursor()

manager.execute("""
CREATE TABLE IF NOT EXISTS customers(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    first_name TEXT,
    last_name TEXT,
    email_address TEXT)""")
warehouse.commit()


""" Insert single value """
insert_single = "INSERT INTO customers (first_name) VALUES (?)"
manager.execute(insert_single, ("aloha",))
warehouse.commit()



""" Insert values in single line """
manager.execute("INSERT INTO customers VALUES (47,'Johnyo','Elder','john@codemy')")
warehouse.commit()



""" Insert with repeated task """
manager.execute(insert_single, ("testing",))
warehouse.commit()



""" Insert multiple entries with single variable"""
insert_multiple = "INSERT INTO customers (first_name, last_name, email_address) VALUES (?, ?, ?)"
manager.execute(insert_multiple, ("MACHINE LEARNING", "programming",1))
warehouse.commit()



""" Insert multiple entries with multiple variables"""
name = 'lulu'
phone = '3366858'
```

```
email = 'user@example.com'
manager.execute('''INSERT INTO customers(first_name, last_name, email_address)
                  VALUES(?,?,?)''', (name,phone, email))
warehouse.commit()



""" Insert multiple entries manually with single variable"""
import sqlite3
warehouse = sqlite3.connect("C:\\gitMain\\warehouse.db")
manager = warehouse.cursor()
insert_many =   [
                (29,"Wes","Brown","wes@brown.com"),
                (30,"Steph", "Kuwea", "steph@Kuewa.com"),
                (31,"Dan","Pas","dan@pas.com")
                ]
manager.executemany("INSERT INTO customers VALUES(?,?,?,?)",insert_many)
warehouse.commit()



""" Insert input from the user """
import sqlite3

warehouse = sqlite3.connect("C:\\gitMain\\warehouse.db")
manager = warehouse.cursor()

first_name = input("insert your name: ")

name = "INSERT INTO customers (first_name) VALUES (?)"
manager.execute(name, (first_name, ))
warehouse.commit()



""" Read from table """
manager.execute("SELECT * FROM customers")
print(manager.fetchone()[1])
print(manager.fetchmany())
print(manager.fetchall()[11][3])
data = manager.fetchall()
for item in data:
    if item[1] == "aloha":
        print(f"the item has the word aloha!!! at count {item[0]}")

manager.execute("SELECT * FROM customers WHERE last_name='Elder'")
manager.execute("SELECT * FROM customers WHERE last_name LIKE 'El%'")
manager.execute("SELECT rowid, * FROM customers WHERE last_name LIKE 'El%' AND rowid = 3")
manager.execute("SELECT rowid, * FROM customers WHERE last_name LIKE 'El%' OR rowid = 3")
manager.execute("SELECT rowid, * FROM customers LIMIT 2")
manager.execute("SELECT rowid, * FROM customers ORDER BY rowid DESC LIMIT 2")
data = manager.fetchall()
for item in data:
    print(item)

#To check how many id's I have
courses_count = manager.execute("SELECT count(id) FROM customers").fetchone()
print("Number of courses : {}".format(*courses_count))



""" Read with pandas """
import pandas as po
import numpy as np
import sqlite3 as location

location = ('c:\\gitMain\\warehouse.db')
warehouse = sql.connect(location)
manager = ''' SELECT * from customers'''
command = po.read_sql_query(manager, warehouse)
```

```
command.head()



""" Update table """
manager.execute("""UPDATE customers SET first_name = 'Bob'
                WHERE last_name = 'Elder'
                """)

manager.execute("""UPDATE customers SET first_name = 'John'
                WHERE rowid = 35
                """)
warehouse.commit()



""" Deleting a record or a table """
manager.execute("DELETE from customers WHERE rowid=47")
anager.execute("DROP TABLE customers")



""" write and load from a different jupyter file """
%%writefile c:\gitmain\customer.py
get_ipython().run_line_magic('load', 'c:\\gitmain\\customer.py')
get_ipython().run_line_magic('run', 'c:\\gitMain\\customer.py')
get_ipython().run_line_magic('save', 'c:\\gitMain\\customer.py %history 2-4')
get_ipython().run_line_magic('save', '-a c:\\gitMain\\customer.py %history 2-4')
get_ipython().run_line_magic('history', '')

# In[ ]:

get_ipython().run_cell_magic('writefile', 'c:\\\\gitMain\\\\customer.py ', '# keep this the first line otherwise will give error\n""" Save as function to use in other file to show
everything"""\nimport sqlite3\n\ndef show_all():\n    warehoue = sqlite3.connect(\'c:\\\\gitMain\\\\warehouse.db\')\n    manager = conn.cursor()\n    manager.execute("SELECT * FROM
customers")\n    items = c.fetchall()\n    \n    for item in items:\n        print(item)\n    warehoue.commit()\n    warehouse.close() ')



""" Run function from different file to just read entries"""
get_ipython().run_line_magic('run', 'C:\\\\gitMain\\\\customer.py')
show_all()



get_ipython().run_cell_magic('writefile', 'c:\\\\gitMain\\\\customer.py', '""" Create a function to use in other file to add 1 record to our database """\nimport sqlite3\n\ndef
add_one(first_name, last_name,email_address):\n    warehouse = sqlite3.connect(\'c:\\\\gitMain\\\\warehouse.db\')\n    command = "INSERT INTO customers (first_name, last_name,email_address)
VALUES (?,?,?)"\n    manager = warehouse.cursor()\n    manager.execute(command, (first_name, last_name,email_address))\n    warehouse.commit()\n    warehouse.close() ')



""" Run function from different file to add entry"""
get_ipython().run_line_magic('run', 'C:\\\\gitMain\\\\customer.py')
add_one("Ronaldo","naldo","ronaldo@yahoo.com")
show_all()



get_ipython().run_cell_magic('writefile', 'c:\\\\gitMain\\\\customer.py', '""" Create a function to use in other file to delete record to our database """\nimport sqlite3\n\ndef
delete_one(id):\n    warehouse = sqlite3.connect(\'c:\\\\gitMain\\\\warehouse.db\')\n    command = "DELETE from customers WHERE rowid=(?)"\n    manager =
warehouse.cursor()\n    manager.execute(command, (id))\n    warehouse.commit()\n    warehouse.close() ')



""" Run function from different file to delete"""
get_ipython().run_line_magic('run', 'C:\\\\gitMain\\\\customer.py')
delete_one('2')
show_all()
```

```
get_ipython().run_cell_magic('writefile', 'c:\\\\gitMain\\\\customer.py', '""" Create a function to use in other file and add multiple records to our database """\nimport sqlite3\n\ndef
add_many(list):\n    warehouse = sqlite3.connect(\'c:\\\\gitMain\\\\warehouse.db\')\n    manager = warehouse.cursor()\n    manager.executemany("INSERT INTO customers Values
(?,?,?,?)",(list))\n    warehouse.commit()\n    warehouse.close() ')



""" Run function from different file to add multiple entries"""
get_ipython().run_line_magic('run', 'C:\\\\gitMain\\\\customer.py')
new_records = [
    (53,'lo','mititica','lulu@lulu.com'),
    (54,'lolo','paulica','paulicuta@gmail.com')
]
add_many(new_records)
show_all()



warehouse.close()
```

```
Connect with python 2


"""Reading from sqlite total count of a specific column"""

import csv

from cs50 import SQL

db = SQL("sqlite:///favorites.db")#open database on disk

language = input("Language: ").strip()# take input

rows = db.execute("SELECT COUNT(*) AS counter FROM favorites WHERE language LIKE ?", language)

row = rows[0]

print(row["counter"])
```

```
"""Reading from sqlite all from a column """

import csv

from cs50 import SQL

db = SQL("sqlite:///favorites.db")#open database on disk

language = input("Language: ").strip()# take input

rows = db.execute("SELECT language FROM favorites WHERE language LIKE ?", language)

for row in  rows:
   print(row["language"])
```

Example 2

```
import sqlite3

connection = sqlite3.connect("gta.db") #create empty database and in the bracket select name for the database

# in order to use SQL commands we need to create something called cursor object
cursor = connection.cursor() # this object is in charge to all our communication with the databse so any CRUD will be executed on cursor object

# to create a brand new table in our database:
cursor.execute("create table if not exists gta (release_year integer primary key, release_name text not null, city text not null)")#pass in the brackets our SQL commands and inside again the () will be the name of the columns but add also the type od the data
(string=text,int=integer,float=real,binary=blob)

release_list = [
```

```
    (1997, "Grand Theft Auto", "state of New Guernsey"),
    (1999, "Grand Theft Auto 2", "Anywhere, USA"),
    (2001, "Grand Theft Auto III", "Liberty City"),
    (2002, "Grand Theft Auto: Vice City", "Vice City"),
    (2004, "Grand Theft Auto: San Andreas", "state of San Andreas"),
    (2008, "Grand Theft Auto IV", "Liberty City"),
    (2013, "Grand Theft Auto V", "Los Santos")
]

# fill table with data, after we define our release list because it has to exist before we use it
# since we are going to insert multiple raws of data in the same time we re going to proceed like below
cursor.executemany("insert into gta values(?,?,?)", release_list)#instead of just execute

#print database rows
for row in cursor.execute("select * from gta"):# so select all values from gta
    print(row)

#print specific rows
print("********************")
cursor.execute("select * from gta where city=:c", {"c":"Liberty City"})#so we are focusing on the last column and when we find an instance of the liberty city we bring back the entire row
gta_search = cursor.fetchall()
print(gta_search)

cursor.execute("create table if not exists cities (gta_city text not null, real_city text not null)")#instead of just execute
cursor.execute("insert into cities values (?,?)",("Liberty City", "New York"))#database we assigned it to a variable called connection, does not matter what name it is)
cursor.execute("select * from cities where gta_city=:c", {"c":"Liberty City"})
cities_search = cursor.fetchall()
print(cities_search)

#manipulate database data
print("--------------------")
for i in gta_search:
    adjusted = [cities_search[0][1] if value == cities_search[0][0] else value for value in i]
    print(adjusted)

connection.close()
```

- **So cursor is like a mouse cursor that keeps track where you are in the database table (functions:execute, fetch,fetchall)**

```
Connect with python 3

import sqlite3

# Create a database in RAM
# db = sqlite3.connect(':memory:')

# Creates or opens a file called mydb with a SQLite3 DB
db = sqlite3.connect('db.sqlite3')

##########
# CREATE #
##########
cursor = db.cursor()
cursor.execute('''
    CREATE TABLE IF NOT EXISTS users(
        id INTEGER PRIMARY KEY,
        name TEXT,
        phone TEXT,
        email TEXT,
        password TEXT
    )
''')
db.commit()

##########
# INSERT #
##########
```

```
'''
If you need values from Python variables it is recommended to use the "?" placeholder.
Never use string operations or concatenation to make your queries because is very insecure.
'''
cursor = db.cursor()
name = 'Andres'
phone = '3366858'
email = 'user@example.com'
password = '12345'
cursor.execute('''INSERT INTO users(name, phone, email, password)
                  VALUES(?,?,?,?)''', (name,phone, email, password))
db.commit()
'''
The values of the Python variables are passed inside a tuple.
Another way to do this is passing a dictionary using the ":keyname" placeholder:
'''
cursor = db.cursor()
cursor.execute('''INSERT INTO users(name, phone, email, password)
                  VALUES(:name,:phone, :email, :password)''',
                  {'name':name, 'phone':phone, 'email':email, 'password':password})
db.commit()

# If you need to insert several users use executemany and a list with the tuples:
users = [('a','1', 'a@b.com', 'a1'),
        ('b','2', 'b@b.com', 'b1'),
        ('c','3', 'c@b.com', 'c1'),
        ('c','3', 'c@b.com', 'c1')]
cursor.executemany(''' INSERT INTO users(name, phone, email, password) VALUES(?,?,?,?)''', users)
db.commit()

# ????
# If you need to get the id of the row you just inserted use lastrowid:
id = cursor.lastrowid
print('Last row id: %d' % id)

##########
# SELECT #
##########
# To retrieve data, execute the query against the cursor object
# and then use fetchone() to retrieve a single row or fetchall() to retrieve all the rows.

cursor.execute('''SELECT name, email, phone FROM users''')
user1 = cursor.fetchone() #retrieve the first row
print(user1[0])
all_rows = cursor.fetchall()
for row in all_rows:
    # row[0] returns the first column in the query (name), row[1] returns email column.
    print('{0} : {1}, {2}'.format(row[0], row[1], row[2]))
# The cursor object works as an iterator, invoking fetchall() automatically:
cursor.execute('''SELECT name, email, phone FROM users''')
for row in cursor:
    print('{0} : {1}, {2}'.format(row[0], row[1], row[2]))

# To retrive data with conditions, use again the "?" placeholder:
user_id = 3
cursor.execute('''SELECT name, email, phone FROM users WHERE id=?''', (user_id,))
user = cursor.fetchone()
db.commit()

##########
# UPDATE #
##########
# The procedure to update data is the same as inserting data:
newphone = '3113093164'
userid = 1
cursor.execute('''UPDATE users SET phone = ? WHERE id = ? ''', (newphone, userid))
db.commit()
```

```
##########
# DELETE #
##########
# The procedure to delete data is the same as inserting data:
delete_userid = 2
cursor.execute('''DELETE FROM users WHERE id = ? ''', (delete_userid,))
db.commit()


### About commit() and rollback():
'''
Using SQLite Transactions:
Transactions are an useful property of database systems.
It ensures the atomicity of the Database.
Use commit to save the changes.
Or rollback to roll back any change to the database since the last call to commit:
'''
cursor.execute('''UPDATE users SET phone = ? WHERE id = ? ''', (newphone, userid))
# The user's phone is not updated
db.rollback()

'''
Please remember to always call commit to save the changes.
If you close the connection using close or the connection to the file is lost
(maybe the program finishes unexpectedly), not committed changes will be lost.
'''

### Exception Handling:
try:
    db = sqlite3.connect('db.sqlite3')
    cursor = db.cursor()
    cursor.execute('''CREATE TABLE IF NOT EXISTS
                    users(id INTEGER PRIMARY KEY, name TEXT, phone TEXT, email TEXT, password TEXT)''')
    db.commit()
except Exception as e:
    # This is called a catch-all clause.
    # This is used here only as an example.
    # In a real application you should catch a specific exception such as IntegrityError or DatabaseError

    # Roll back any change if something goes wrong
    db.rollback()
    raise e
finally:
    db.close()

### SQLite Row Factory and Data Types
'''
The following table shows the relation between SQLite datatypes and Python datatypes:

    None type is converted to NULL
    int type is converted to INTEGER
    float type is converted to REAL
    str type is converted to TEXT
    bytes type is converted to BLOB
'''

# The row factory class sqlite3.Row is used to access the columns of a query by name instead of by index:
db = sqlite3.connect('db.sqlite3')
db.row_factory = sqlite3.Row
cursor = db.cursor()
cursor.execute('''SELECT name, email, phone FROM users''')
for row in cursor:
    # row['name'] returns the name column in the query, row['email'] returns email column.
    print('{0} -> {1}, {2}'.format(row['name'], row['email'], row['phone']))
db.close()
```

```
########
# DROP #
########
db = sqlite3.connect('db.sqlite3')
cursor = db.cursor()
cursor.execute('''DROP TABLE users''')
db.commit()

# When we are done working with the DB we need to close the connection:
db.close()
```

We will use:
1 - express.js
2 - body-parser
3 - mongoose
##############

// 1. go to the desktop
cmd prompt: cd desktop

// 2. create a new folder
cmd prompt: mkdir html-mongo

// 3. go into the new folder
cmd prompt: cd html-mongo

// 4.create 2 files
Create 2 files in the html-mongo **server.js** & **index.html**

// 5. initialize npm, it will initialize the project and create package.json file
cmd prompt: npm init
npm is the package manager for NodeJS, it puts modules in place so that node can find them, it is used to publish, discover,install and develop node programs

// 6. Select default settings
Click **Enter** for default settings

****Next install needed npm packages****
// 7. install express
cmd prompt: npm i express mongoose body-parser

- **express. Js: is a free and open-source web application framework for Node.js, helps in handling requests and views, it is hosted in Node JS runtime environment,it is a back end web application framework for Node.js express.js is a Node.js application server framework**
- **NodeJs: is not a framework, it is not a programming language but it allows you to use JS**
- **Mongoose: is a Node. js based Object Data Modeling library for MongoDB, it allows to define schemas with strongly typed data.Once a schema is defined, Mongoose lets you create a Model based on a specific schema, then mapped to a MongoDB document via the Model's schema definition**
- **mongodb: is the native driver for interacting with mongodb instance and mongoose is an Object modeling tool for MongoDB, mongoose is built on top of the mongodb driver to provide programmers with a way to model their data.**
- **Express body-parser: mis and npm library used to process data sent through an HTTP request body, so backend service will accept POST request with text in the request body**

// 8. run our server
cmd prompt: nodemon server.js

// 9.Open folder in vsCode and configure the express server
##
server,js
****
*** Database used notesDB**
*** Collection used notes**

* schema required for this collection: - title = type String
- content = type String
* schema and data model has to be created in server.js file

```
// 10. require express, calls the express function and puts a new express application inside the app variable(to start a new express application)
const express = require("express");

// 11. create our app that will be used in this express
const app = express();

// 12. create mongoose that will require mongoose
const mongoose = require ("mongoose");

// 13. we will require body parser so backend service will accept POST request with text
const bodyParser = require("body-parser");

// 14. configure the above
app.use(bodyParser.urlencoded({extended: true}));

// 18. connect to mongoose (uri from the AtlasDB site => just specify the name of the database we want to use in this project "notesDB" + add password)
mongoose.connect("mongodb+srv://demo:Mamamiapizzapia3!!!@cluster0.cpk97.mongodb.net/notesDB")

// 19. create a data schema
// it will be an object of title that is string and content the same
const notesSchema = {
  title: String,
  content: String
}

// 20. create model named Note, will be referred to as Note and will be using the notesSchema => so now in our app.post method we can use this model to save or to perform crud operations in this model
const Note = mongoose.model('Note', notesSchema);

// 17. we use the app.get method and the route route and the function that we will send//Check server
app.get('/', function(req, res) {
  // if we change res.send("express is working") - it will print this message to check server
  // 22. render the file by using res.sendFile....and specify the location of the file
  res.sendFile(__dirname + '/index.html')
})

// 23. app.post
app.post('/', function(req, res) {
  // 24. we will create new note that will be used in Note model (const Note), that is an object of title and content
  let newNote = new Note({
    // we will use the name that were given in HTML
    title: req.body.title,
    content: req.body.content
  });

  // 25. take the new note and save it
  newNote.save();
  // 26. redirect to the URL derived from the specified path.The default status is "302 Found"
  res.redirect('/');
})

// 15. make our app listen at 3000
app.listen(3000, function() {
  // 16. log that our server is running on 3000
  console.log("server running");
})
```

#####
HTML
#####
* We will need to render this file in server.js through `res.sendFile(__dirname + '/index.html')`

```
//21.create html
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  // copy from: bootstrap maxcdn
  <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
  <title>Document</title>
</head>
<body>
  <h1>Posting data from HTML to MongoDB Atlas</h1>

  // because we will be posting below info to our server then to mongoDb and the action will be at the route raute
  <form class="container" method="post" action="/">

    // easiest way to add structure to forms
    <div class="form-group">

      // we take the input from html to mongodb =>we need to provide the name for each input from which we want to take this data and through the body-parser we can take now the values that we enter through the reference to the title and the content
(down mentioned)
      <input class ="form-control" name="title">
    </div>
    <div class="form-group">
      // form-control tweaks input appearance and setting width
      <textarea class ="form-control" name="content"></textarea>
</div>
    <button>ADD TO MONGODB</button>
  </form>

</body>
</html>
```

We will use:

1 - express.js
2 - ejs
3 - mongoose
##############

// 1. go to the desktop
cmd prompt: **cd desktop**

// 2. create a new folder
cmd prompt: **mkdir mongo-html**

// 3. go into the new folder
cmd prompt: **cd mongo-mongo-html**

// 4.create 1 file
Create 1 file in the html-mongo **server.js**

// 5. initialize npm, it will initialize the project and create package.json file
cmd prompt: **npm init**
npm is the package manager for NodeJS, it puts modules in place so that node can find them, it is used to publish, discover,install and develop node programs

// 6. Select default settings
Click **Enter** for default settings

****Next install needed npm packages****
// 7. install express
cmd prompt: **npm i express ejs mongoose**

// 8. run our server
cmd prompt: **nodemon server.js**

// 9.Open folder in vsCode and configure the express server
##

#######
server,js
#######

```
// 10.
const express = require('express');

// 11.
const mongoose = require('mongoose');
```

```
// 12.
const app = express();
// 13. create also ejs
const ejs = require('ejs');

// 14. with ejs you have to set the view engine to ejs = configure the app
app.set('view engine', 'ejs');

// 15. connect mongoose to atlasDB (you have to connect to the database moviesDB - this is what you put in the link above)
mongoose.connect('mongodb+srv://demo:Mamamiapizzapia3!!!@cluster0.cpk97.mongodb.net/moviesDB?retryWrites=true&w=majority');

// 20. this will have an object title, genre and year of String
const moviesSchema={
   title: String,
   genre: String,
   year:String
}

// 21. we create our movie model that we use to get all the movies here
const Movie = mongoose.model('Movie', moviesSchema);

Obs - diff between html.index and html.ejs:
index.html - you can simply send from html.index to server by : res.sendFile(__dirname + '/index.html') and in the index.html file you just write simple <h1>HTML Here</h1>
index.ejs - you have to create separate file called views or it will not work and inside this file called views you create a file that is called index.ejs, you will have same info from index.html BUT ejs format will allow us to incorporate JS inside this html file and in
order to RENDER this file we will use res.render without to need to specify the location rather just the name of the file because it knows that the file will be ready to be taken from the file views so just specify the name of the file = it will do same as index.html
// 18. check that everything is working the same first line + res.send('working')
app.get('/', (req, res)=>{
   // 22. we take our movie model and find all movies in database and access to the function where we will have all those find movies then we want to pass them to our index.ejs and put them into an HTML table
   Movie.find({}, function(err, movies){
      // 19.ejs will allow you to specify a second argument as a js object with any field
      res.render('index', {
         moviesList:movies // so: app.get('/',(req,res)=>{let name='Alan'; res.render('index',{ userName:name});}) => so we create
      }) name that we said it is Alan, we pass it to userName then we pass it to index.ejs file
   }) So we can access it in ejs file by writing <h1><%= userName %>Here</h1> => so it will say Alan Here
}) With this logic you can pass any variable from our server to html.

// 16. we will have the app listen at port 4000
app.listen(4000, function(){

// 17. message to print that server is running
console.log('server is running');
})
```

#####
HTML
#####
* Database used moviesDB
* Collection used movies
* schema required for this collection: - title = type String
                                       - genre = type String
                                       - year = type String
* schema and data model has to be created in server.js file

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Document</title>
   <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@3.3.7/dist/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
</head>
<body>

   <table class="table">
```

```
    <thead>
      <tr>
        <th scope="col">ID</th>
        <th scope="col">Title</th>
        <th scope="col">Genre</th>
        <th scope="col">Year</th>
      </tr>
    </thead>
    <tbody>
      // ejs allows to write this file like html but with js elements
      <%moviesList.forEach(movie =>{%>
        <tr>
          <th scope="row"><%= movie._id %></th>
          <td><%= movie.title %></td>
          <td><%= movie.genre %></td>
          <td><%= movie.year %></td>
        </tr>
        <%})%>
    </tbody>
  </table>
</body>
</html>
```

Part 1: Connecting & CRUD Operations
CRUD = create, read, update, delete

- open VSCode
- create file NODEJS-DEMO
- check if node installed >**$ node -v**
- install mongodb node.js driver = node driver allows to easily interact with mongodb databases from within node.js applications, we need the driver to connect to the database and execute queries > **$npm install mongodb**
- we can check version of mongodb > **npm list mongodb**
- create a mongodb database > use mongodb Atlas > Atlas is fully managed database as a service (otherwise you will have to manage the database yourself)
- create new project in Atlas > create a new cluster > cluster is set of nodes where a copy of the database will be stored
- after creating cluster > load sample data
- connecting the database > click connect > click add ip 2 times > create user & pass > click choose connection method > connect your application > driver:Node.js > click copy connection string > close = DATABASE WAS SET UP

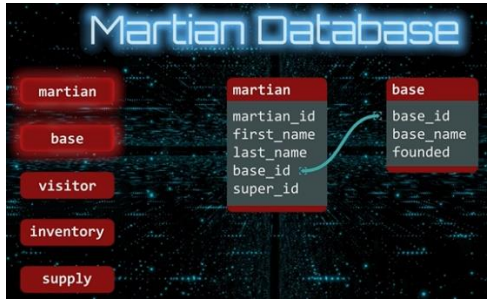
➤ **We need to write a node.js script that will connect the Atlas cluster we have created and list all the databases in that cluster**
- create a new file > **demo.js**

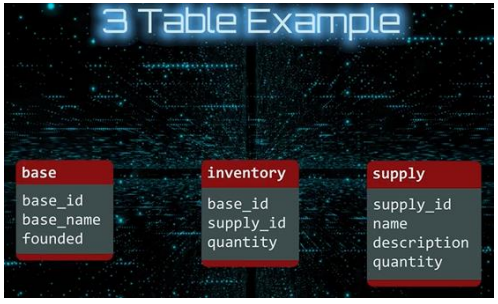
Part 2: CODING

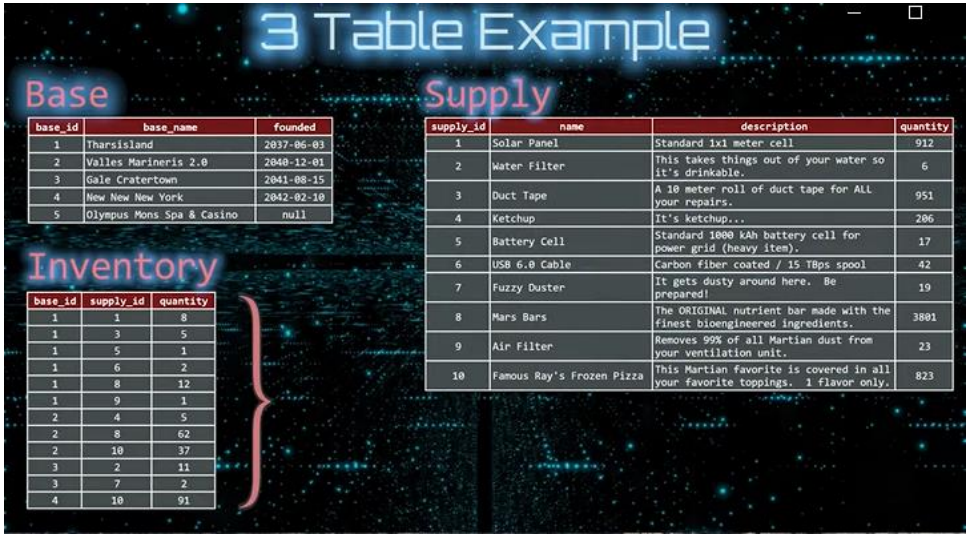
```
// 1.get the MongoClient first//MongoClient is what we are gonna use to connect to mongodb database
const {MongoClient}= require ('mongodb')

// 2.inside this function we are going to connect to mongodb cluster, call functions that were created the database and disconnected from the cluster// So here we call function
async function main(){

// 3.first create a constant for my connection uri (copy paste the link copied from atlas and updated the username and password we have created)
const uri = "mongodb+srv://demo:Sitaru3!@cluster0.cth6r.mongodb.net/myFirstDatabse?retryWrites=true&w=majority";

 // 4.after the uri we can create an instance of MangoClient
const client=new MongoClient(uri);
try{

// 5.we will now connect our cluster // client.connect returns a promise so away keyword is used that indicates that we will block further execution until that operation has been
await.client.connect(); completed

// 21.create items
//await createListing(client,  {
//name:"Lovely loft",
//summary:"A charming loft in Paris",
//bedrooms:1,
//bathrooms:1
//})

// 24.create items
await createMultipleListings(client, [
{
bed rooms:4
beds:7
},
{last_review:new Date(),
name:"Beautiful Beach"
}
]);

//27
await findOneListByName(client,"Infinite Views");
```

```
// 16.we will pass the mango client
//await listDatabases(client);

// 6.client.connect could through an error so we will have to wrap a try/catch around it
}catch (e){
// 7.we will just print the error if it's there
console.error(e)
// 8.we will make sure we close the connection to the cluster so we will close the try,catch with a finally statement
  } finally  {
     await client.close();
   }
});
// 9.we need to call main
main().catch(console.error

// 26.parameters are mongo client and names we want to find
async function findOneListingByName(client, nameListing){
const result=await client.db("sample_airbnb").collection("listingAndReviews").findOne({name:nameOfListing});
if (result){
console.log)`Found a listing in the collection with the name'${nameOfListing}'`);
console.log(result);
} else{
console.log(`No listings found with the name '${nameOfListing}'`);
}

// 23. create multiple listings
async function createMultipleListings(client, newListings){
const results = await client.db("sample_airbnb").collection("listingsAndReviews").insertMany(newListings);
console.log(`$result.insertedCount} new listings created with the following id(s):`);
console.log(result.insertedIds);
}

// 18. Create a function to create a new listing
async function createListing(client, newListing){
// 19. So we are choosing sample_airbnb database and inside this database we need to access the collection listingAnd Reviews, from that collection we can call insert one and we'll pass our listing into insert one
const result = await client.db("sample_airbnb").collection("listingAndReviews").insertOne(newListing);
// 20.log id of the new created document
console.log(`New listing created with the following id: ${result.insertedId}`);
}

// 11.we are gonna set the client as our parameter//we will get our list of databases through this
async function listDatabases(client){
// 12.we will use client.db().admin() to get the list and we use await before continuing and assign the results to a constant named databasesList
const databasesList = await client.db().admin().listDatabses();
// 13.we will print them all
console.log("Databases:" );
// 14.we will print each one of them
databasesList.databases.forEach(db =>{
// 15.we will log each database name => next step we need to call it
console.log(`- ${db.name}`);
})
}

Part 3: Terminal
// 10. if nothing happens that means all ok
    $ node demo.js
// 17. will list the list of databases
    $ node demo.js
// 22. will create a new listing
    $ node demo.js
// 25. will create multiple listings
    $ node demo.js
// 28.

    clear
// 29.
    $ node demo.js
```

1. Create a SQLite Database (and a Table)







- First, let us understand how create a SQLite database with couple of tables, populate some data, and view those records.
  The following example creates a database called company.db. This also creates an employee table with 3 columns (id, name and title), and a department table in the company.db database. We've purposefully missed the deptid column in the employee table. We'll see how to add that later.

  - **relational database is called sql**
  - **flat file database:simple rows and columns file**
  - **So in SQL you capitalizing all the special sql keywords and lowercasing the column names and the table names which is a convention and helps reads the sql commands**

- to clear window in unix => ctrl + l
- ctr +d to get out
- to see the design of the table you use .schema
- get into the database: sqlite3 favorites.db
- sqlite> .timer on

```
.mode csv
.import FILE TABLE
```

```
$ sqlite3 favorites.db
SQLite version 3.36.0 2021-06-18 18:36:39
Enter ".help" for usage hints.
sqlite> .mode csv
sqlite> .import favorites.csv favorites
sqlite>

$
```

- If done properly then the data has been uploaded into sql

- .schema = created a table if it did not exist named favorites with 3 columns and agreed to be text

```
# sqlite3 company.db
sqlite> create table employee(empid integer,name varchar(20),title varchar(10));
sqlite> create table department(deptid integer,name varchar(20),location varchar(10));
sqlite> .quit
```

- Note: To exit from the SQLite commandline “sqlite>” prompt, type “.quit” as shown above.

A SQLite database is nothing but a file that gets created under your current directory as shown below.

```
# ls -l company.db
-rw-r--r--. 1 root root 3072 Sep 19 11:21 company.db
```

2. Insert Records

- The following example populates both employee and department table with some sample records.

You can execute all the insert statements from the sqlite command line, or you can add those commands into a file and execute the file as shown below.

First, create a insert-data.sql file as shown below.

```
# vi insert-data.sql
insert into employee values(101,'John Smith','CEO');
insert into employee values(102,'Raj Reddy','Sysadmin');
insert into employee values(103,'Jason Bourne','Developer');
insert into employee values(104,'Jane Smith','Sale Manager');
insert into employee values(105,'Rita Patel','DBA');

insert into department values(1,'Sales','Los Angeles');
insert into department values(2,'Technology','San Jose');
insert into department values(3,'Marketing','Los Angeles');
```

- The following will execute all the commands from the insert-data.sql in the company.db database

```
# sqlite3 company.db < insert-data.sql
```

3. View Records

- Once you’ve inserted the records, view it using select command as shown below.

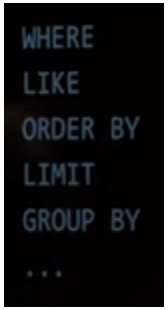
```
# sqlite3 company.db
sqlite> select * from employee;
101|John Smith|CEO
102|Raj Reddy|Sysadmin
```

```
103|Jason Bourne|Developer
104|Jane Smith|Sale Manager
105|Rita Patel|DBA

sqlite> select * from department;
1|Sales|Los Angeles
2|Technology|San Jose
3|Marketing|Los Angeles
```

- you can operate certain operations on the column such as:


```
SELECT DISTINCT (language) FROM favorites;
sqlite> SELECT DISTINCT (UPPER(language)) FROM favorites;
sqlite> SELECT COUNT(language) FROM favorites;
```

- **more filtration:**




```
sqlite> SELECT language FROM favorites LIMIT 10;
sqlite> SELECT language FROM favorites WHERE language LIKE "python";
sqlite> SELECT language FROM favorites WHERE language LIKE "%C%";
sqlite> SELECT COUNT(language) FROM favorites WHERE language LIKE "python";
sqlite> SELECT (language) FROM favorites WHERE language LIKE "%python%";
sqlite> DELETE FROM favorites WHERE language LIKE "%python%";
sqlite> UPDATE favorites SET language = "the programme C" Where language = "C";
```

```
Select * from people;//show all people from the shows.db
sqlite> SELECT * FROM people WHERE name = "Choi Han"; // select a name from the whole list
sqlite> SELECT * FROM shows WHERE title = "The Office";
```

SELECT == It orders the computer to include or select each content from the database name(table) .
(*) == means all {till here code means include all from the database.}
FROM == It refers from where we have to select the data.
example_table == This is the name of the database from where we have to select data.
The overall meaning is :
- include all data from the databse whose name is example_table.

SELECT refers to attributes that you want to have displayed in your final query result. There are different 'SELECT' statements such as '**SELECT DISTINCT**' which returns only unique values (if there were duplicate values in the original query result)
FROM basically means from which table you want the data. There can be one or many tables listed under the 'FROM' statement.
WHERE means the condition you want to satisfy. You can also do things like ordering the list by using 'order by **DESC**' (no point using order by **ASC** as SQL orders values in ascending order after you use the order by clause)

```
sqlite> SELECT * FROM shows WHERE title = "The Office" AND year = 2005;
```

4. Rename a Table

- The following example renames department table to dept using the alter table command.

```
sqlite> alter table department rename to dept;
```

5. Add a Column to an Existing Table

- The following examples adds deptid column to the existing employee table;

```
sqlite> alter table employee add column deptid integer;
```

- Update the department id for the employees using update command as shown below.

```
update employee set deptid=3 where empid=101;
update employee set deptid=2 where empid=102;
update employee set deptid=2 where empid=103;
update employee set deptid=1 where empid=104;
update employee set deptid=2 where empid=105;
```

- Verify that the deptid is updated properly in the employee table.

```
sqlite> select * from employee;
101|John Smith|CEO|3
102|Raj Reddy|Sysadmin|2
103|Jason Bourne|Developer|2
104|Jane Smith|Sale Manager|1
105|Rita Patel|DBA|2
```

6. View all Tables in a Database

- Execute the following command to view all the tables in the current database. The folowing example shows that there are two tables in the current database.

```
sqlite> .tables
dept     employee
```

7. Create an Index

- The following example creates an unique index called empidx on the empid field of employee table.

```
sqlite> create unique index empidx on employee(empid);
```

- Once an unique index is created, if you try to add another record with an empid that already exists, you'll get an error as shown below.

```
sqlite> insert into employee values (101,'James Bond','Secret Agent',1);
Error: constraint failed
```

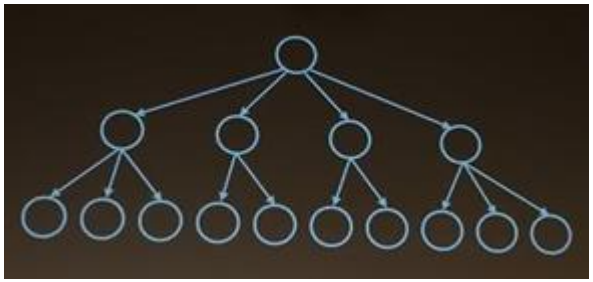
Indexes review:

- It is like giving the database a clue in advance that I'm going to Make a search there so do something to speed stuff up

sqlite> CREATE INDEX "title_index" ON "shows"("title"); //create an index to move faster in search

often tree used it is called a B-tree

sqlite> CREATE INDEX person_index ON stars (person_id);
sqlite> CREATE INDEX show_index ON stars (show_id);
sqlite> CREATE INDEX name_index ON PEOPLE (name);



8. Create a Trigger

- For this example, first add a date column called “updatedon” on employee table.

```
sqlite> alter table employee add column updatedon date;
```

- Next, create a file that has the trigger definition. The following trigger will update the “updatedon” date column with the current timestamp whenever you perform an update on this table.

```
# vi employee_update_trg.sql
create trigger employee_update_trg after update on employee
begin
  update employee set updatedon = datetime('NOW') where rowid = new.rowid;
end;
```

- Create the trigger on the company.db database as shown below.

```
# sqlite3 company.db < employee_update_trg.sql
```

- Now anytime you update any record in the employee table, the “updatedon” date column will be updated with the current timestamp as shown below. The following example updates the “updatedon” timestamp for empid 104 through trigger.

```
# sqlite3 company.db
sqlite> update employee set title='Sales Manager' where empid=104;

sqlite> select * from employee;
101|John Smith|CEO|3|
102|Raj Reddy|Sysadmin|2|
103|Jason Bourne|Developer|2|
104|Jane Smith|Sales Manager|1|2012-09-15 18:29:28
105|Rita Patel|DBA|2|
```

9. Create a View

- The following example creates a view called “empdept”, which combines fields from both employee and dept table.

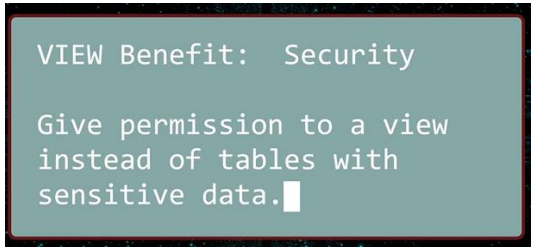
```
sqlite> create view empdept as select empid, e.name, title, d.name, location from employee e, dept d where e.deptid = d.deptid;
```

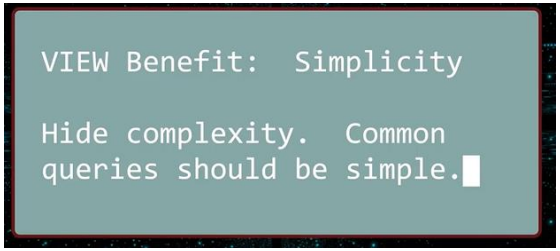
- Now you can execute select command on this view just like a regular table.

```
sqlite> select * from empdept;
101|John Smith|CEO|Marketing|Los Angeles
102|Raj Reddy|Sysadmin|Technology|San Jose
103|Jason Bourne|Developer|Technology|San Jose
104|Jane Smith|Sales Manager|Sales|Los Angeles
105|Rita Patel|DBA|Technology|San Jose
```

- After creating a view, if you execute .tables, you’ll also see the view name along with the tables.

```
sqlite> .tables
dept     empdept   employee
```





10. SQLite Savepoint, Rollback, Commit

- Currently dept table has the following 3 records.

```
sqlite> select * from dept;
1|Sales|Los Angeles
2|Technology|San Jose
3|Marketing|Los Angeles
```

- Now, create a savepoint called “major”, and perform some transactions on the dept table. As you see below, we’ve added two records, deleted one record, after creating a savepoint called “major”.

```
sqlite> savepoint major;
sqlite> insert into dept values(4,'HR','Los Angeles');
sqlite> insert into dept values(5,'Finance','San Jose');
sqlite> delete from dept where deptid=1;
sqlite> select * from dept;
2|Technology|San Jose
3|Marketing|Los Angeles
4|HR|Los Angeles
5|Finance|San Jose
```

- Now for some reason, if we don’t want the above transactions, we can rollback the changes to a particular savepoint. In this example, we are rolling back all the changes we’ve made after the “major” savepoint.

```
sqlite> rollback to savepoint major;
sqlite> select * from dept;
1|Sales|Los Angeles
2|Technology|San Jose
3|Marketing|Los Angeles
```

- If you don’t want your savepoints anymore, you can erase it using release command.

```
sqlite> release savepoint major;
```

11. Additional Date Functions

- By default, the date columns values displayed in UTC time. To display in the local time, use the datetime command on the date column as shown below.

```
sqlite> select empid,datetime(updatedon,'localtime') from employee;
104|2012-09-15 11:29:28
```

- You can also use strftime to display the date column in various output.

```
sqlite> select empid,strftime('%d-%m-%Y %w %W',updatedon) from employee;
104|19-09-2012 3 38
```

- The following are the possible modifers you can use in the strftime function.
  - %d day of month: 00
  - %f fractional seconds: SS.SSS
  - %H hour: 00-24
  - %j day of year: 001-366
  - %J Julian day number
  - %m month: 01-12
  - %M minute: 00-59
  - %s seconds since 1970-01-01
  - %S seconds: 00-59
  - %w day of week 0-6 with Sunday==0
  - %W week of year: 00-53
  - %Y year: 0000-9999
  - %% %

12. Dropping Objects

- You can drop all the above created objects using the appropriate drop command as shown below.
  Since we are dropping objects for testing purpose, copy the company.db to a test.db and try these commands on the test.db

```
# cp company.db test.db

# sqlite3 test.db
sqlite> .tables
dept     empdept   employee

sqlite> drop index empidx;
sqlite> drop trigger employee_update_trg;
sqlite> drop view empdept;
sqlite> drop table employee;
sqlite> drop table dept;
```

- All the tables and views from the test.db are now deleted.

```
sqlite> .tables
sqlite>
```

Note: When you drop a table all the indexes and triggers for that table are also dropped.

13. Operators

- The following are the possible operators you can use in SQL statements.
  - | |
  - * / %
  - + –
  - << >> & |
  - < >=
  - = == != <> IS IS NOT IN LIKE GLOB MATCH REGEXP
  - AND OR

For example:

```
sqlite> select * from employee where empid >= 102 and empid  select * from dept where location like 'Los%';
1|Sales|Los Angeles
3|Marketing|Los Angeles
```

14. Explain Query Plan

- Execute “explain query plan”, to get information about the table that is getting used in a query or view. This is very helpful when you are debugging a complex query with multiple joins on several tables.

```
sqlite> explain query plan select * from empdept;
0|0|TABLE employee AS e
1|1|TABLE dept AS d
```

- For a detailed trace, just execute “explain” followed by the query to get more performance data on the query. This is helpful for debugging purpose when the query is slow.

```
sqlite> explain select empid,strftime('%d-%m-%Y %w %W',updatedon) from employee;
0|Trace|0|0|0||00|
1|Goto|0|12|0||00|
2|OpenRead|0|2|0|4|00|
3|Rewind|0|10|0||00|
4|Column|0|0|1||00|
5|String8|0|3|0|%d-%m-%Y %w %W|00|
6|Column|0|3|4||00|
7|Function|1|3|2|strftime(-1)|02|
8|ResultRow|1|2|0||00|
9|Next|0|4|0||01|
10|Close|0|0|0||00|
11|Halt|0|0|0||00|
12|Transaction|0|0|0||00|
13|VerifyCookie|0|19|0||00|
14|TableLock|0|2|0|employee|00|
15|Goto|0|2|0||00|
```

15. Attach and Detach Database

- When you have multiple database, you can use attach command to execute queries across database.
  For example, if you have two database that has the same table name with different data, you can create a union query across the database to view the combined records as explained below.
  In this example, we have two company database (company1.db and company2.db). From the sqlite prompt, attach both these database by giving alias as c1 and c2 as shown below.

```
# sqlite3
sqlite> attach database 'company1.db' as c1;
sqlite> attach database 'company2.db' as c2;
```

- Execute “.database” command which will display all the attached databases.

```
sqlite> .database
seq  name             file
---  ---------------  ----------------------
0    main
2    c1               /root/company1.db
3    c2               /root/company2.db
```

- Now, you can execute a union query across these databases to combine the results.

```
sqlite> select empid, name, title from c1.employee union select empid, name, title from c2.employee;
101|John Smith|CEO
102|Raj Reddy|Sysadmin
103|Jason Bourne|Developer
104|Jane Smith|Sales Manager
105|Rita Patel|DBA
201|James Bond|Secret Agent
202|Spider Man|Action Hero
```

- After attaching a database, from the current sqlite session, if you want to detach it, use detach command as shown below.

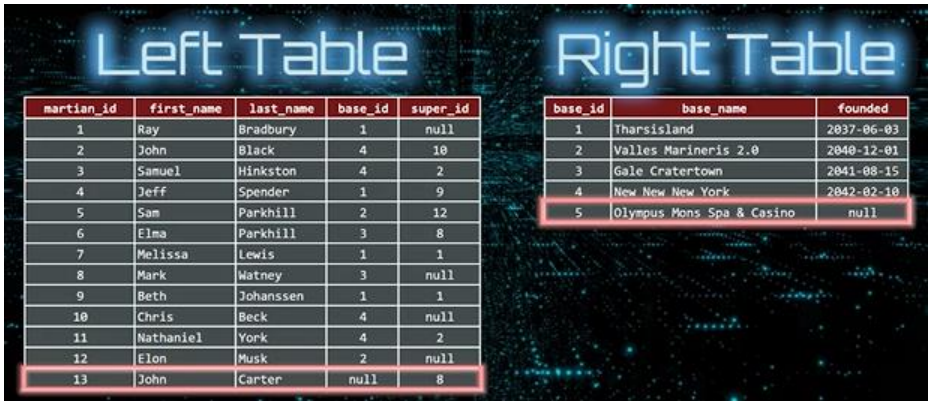
```
sqlite> detach c1;

sqlite> .database
seq  name             file
---  ---------------  ----------------
0    main
2
```

16. Combine Columns

```
sqlite> SELECT title FROM people, stars, shows
   ...> WHERE people.id = stars.person_id
   ...> AND stars.show_id = shows.id
    ...> AND name = "Steve Carell";
```

17. Joining

```
sqlite> SELECT * FROM people WHERE name = "Steve Carell";
sqlite> SELECT id FROM people WHERE name = "Steve Carell";
sqlite> SELECT show_id FROM stars WHERE person_id = (SELECT id FROM people WHERE name = "Steve Carell");
sqlite> SELECT title FROM shows WHERE id IN (SELECT show_id FROM stars WHERE person_id = (SELECT id FROM people WHERE name = "Steve Carell"));
```



martian_id	first_name	last_name	base_id	super_id
1	Ray	Bradbury	1	null
2	John	Black	4	10
3	Samuel	Hinkston	4	2
4	Jeff	Spender	1	9
5	Sam	Parkhill	2	12
6	Elma	Parkhill	3	8
7	Melissa	Lewis	1	1
8	Mark	Watney	3	null
9	Beth	Johanssen	1	1
10	Chris	Beck	4	null
11	Nathaniel	York	4	2
12	Elon	Musk	2	null
13	John	Carter	null	8

base_id	base_name	founded
1	Tharsisland	2037-06-03
2	Valles Marineris 2.0	2040-12-01
3	Gale Cratertown	2041-08-15
4	New New New York	2042-02-10
5	Olympus Mons Spa & Casino	null

What if you do not have an entry?

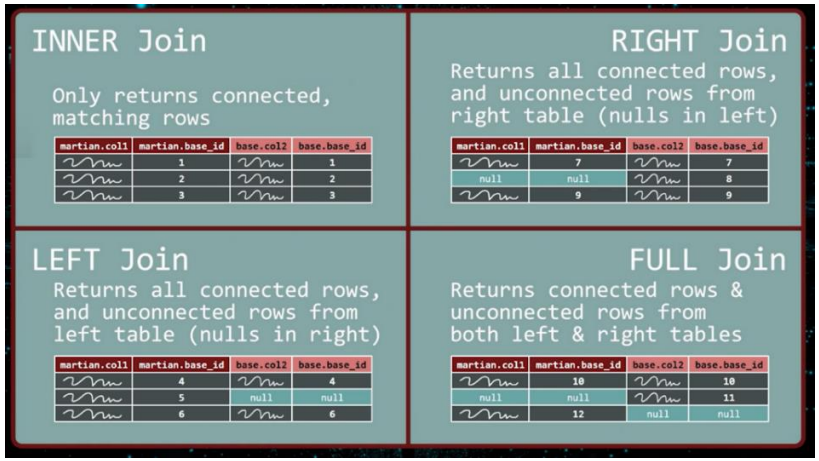
martian_id	first_name	last_name	base_id	super_id
1	Ray	Bradbury	1	null
2	John	Black	4	10
3	Samuel	Hinkston	4	2
4	Jeff	Spender	1	9
5	Sam	Parkhill	2	12
6	Elma	Parkhill	3	8
7	Melissa	Lewis	1	1
8	Mark	Watney	3	null
9	Beth	Johanssen	1	1
10	Chris	Beck	4	null
11	Nathaniel	York	4	2
12	Elon	Musk	2	null
13	John	Carter	null	8

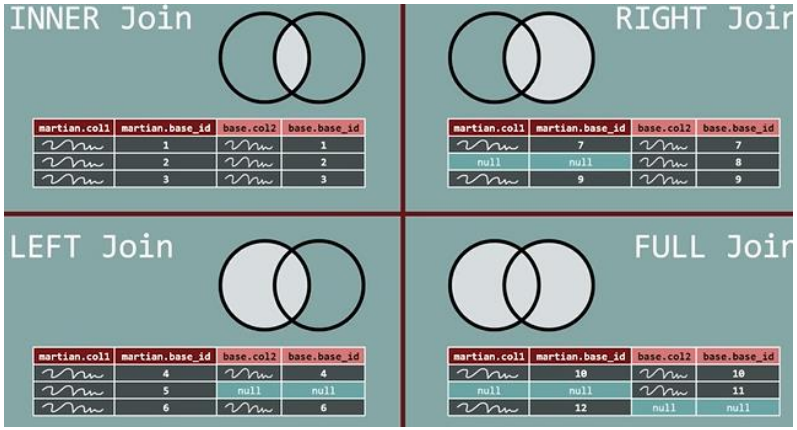
base_id	base_name	founded
1	Tharsisland	2037-06-03
2	Valles Marineris 2.0	2040-12-01
3	Gale Cratertown	2041-08-15
4	New New New York	2042-02-10
5	Olympus Mons Spa & Casino	null

```
SELECT column1, column2, ...
FROM martian
________ JOIN base
ON martian.base_id = base.base_id
WHERE condition(s)
ORDER BY value
```

martian_id	first_name	last_name	base_id	super_id
1	Ray	Bradbury	1	null
2	John	Black	4	10
3	Samuel	Hinkston	4	2
4	Jeff	Spender	1	9
5	Sam	Parkhill	2	12
6	Elma	Parkhill	3	8
7	Melissa	Lewis	1	1
8	Mark	Watney	3	null
9	Beth	Johanssen	1	1
10	Chris	Beck	4	null
11	Nathaniel	York	4	2
12	Elon	Musk	2	null
13	John	Carter	null	8

base_id	base_name	founded
1	Tharsisland	2037-06-03
2	Valles Marineris 2.0	2040-12-01
3	Gale Cratertown	2041-08-15
4	New New New York	2042-02-10
5	Olympus Mons Spa & Casino	null






```sql
SELECT b.base_id, s.supply_id, s.name
FROM base AS b
CROSS JOIN supply AS s;
```

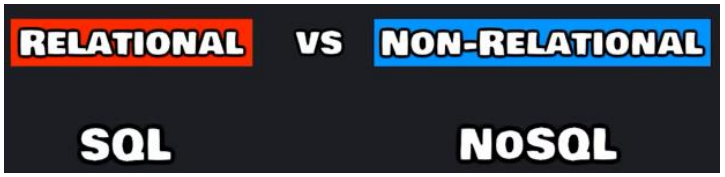
Data Output | Explain | Messages | Notifications

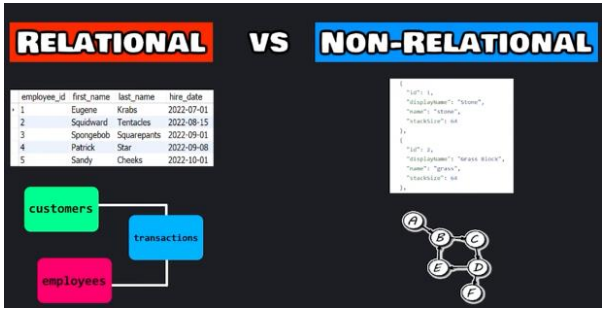
	base_id integer	supply_id integer	name character varying
1	1	1	Solar Panel
2	1	2	Water Filter
3	1	3	Duct Tape
4	1	4	Ketchup
5	1	5	Battery Cell
6	1	6	USB 6.0 Cable
7	1	7	Fuzzy Duster
8	1	8	Mars Bars
9	1	9	Air Filter
10	1	10	Famous Ray's Froze...
11	2	1	Solar Panel

CROSS JOIN:
Performs cross product between two tables.

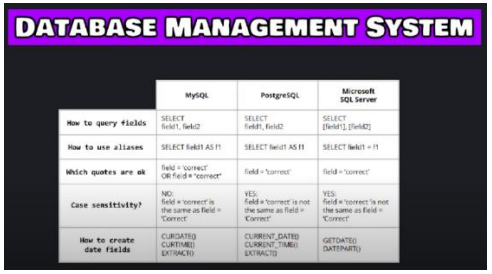
Connects each row in the left table with each row in the right table.

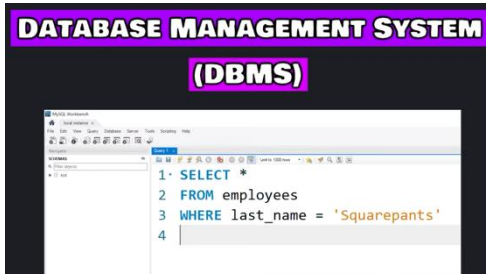

DATABASE


RELATIONAL
VS
NON-RELATIONAL
SQL
NoSQL


RELATIONAL
VS
NON-RELATIONAL
employee_id first_name last_name hire_date
1 Eugene Krabs 2022-07-01
2 Squidward Tentacles 2022-08-15
3 Spongebob Squarepants 2022-09-01
4 Patrick Star 2022-09-08
5 Sandy Cheeks 2022-10-01
customers
transactions
employees

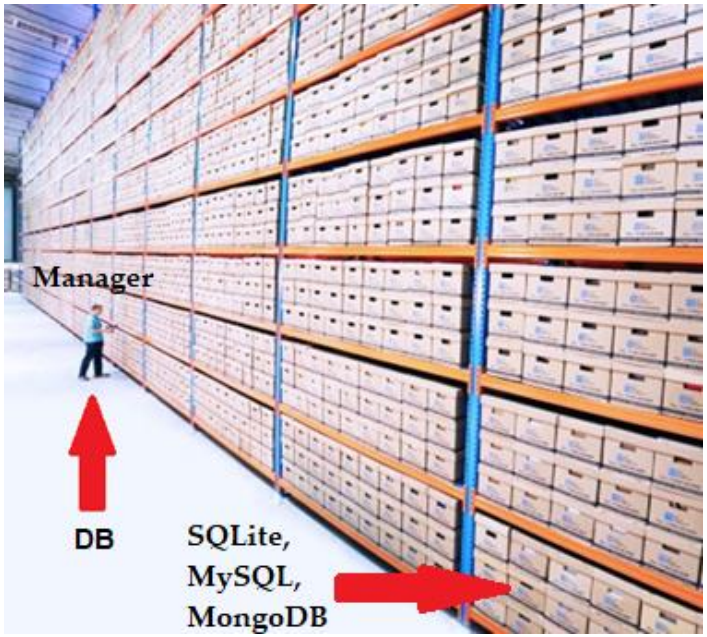

DATABASE MANAGEMENT SYSTEM
(DBMS)
MySQL
Microsoft SQL Server
ORACLE
PostgreSQL


DATABASE MANAGEMENT SYSTEM
MySQL
PostgreSQL
Microsoft SQL Server
How to query fields
How to use aliases
Which quotes are ok
Case sensitivity?
How to create date fields

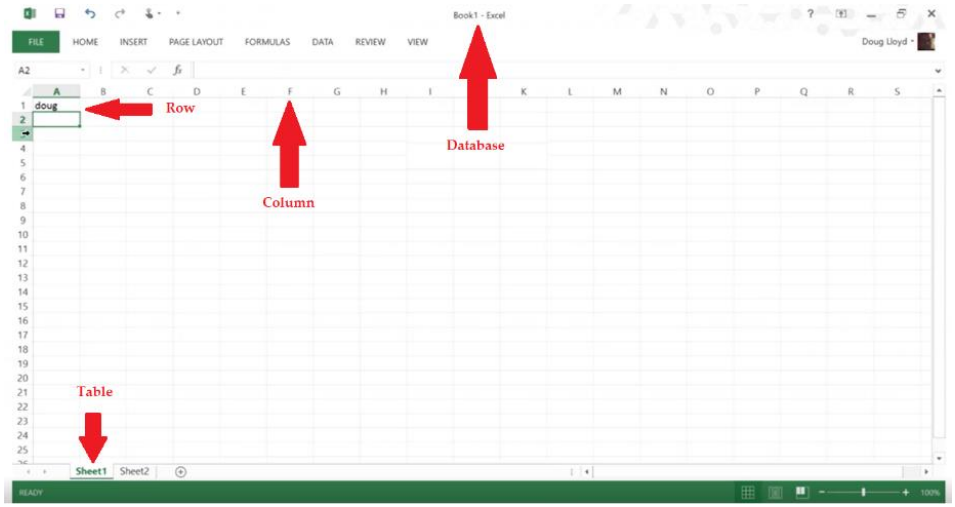

DATABASE MANAGEMENT SYSTEM
(DBMS)
1 SELECT *
2 FROM employees
3 WHERE last_name = 'Squarepants'
4


Ownership
MySQL
MariaDB
ORACLE
Many companies & individuals

Database vs Warehouse vs Excel File



=



- **You own a building!**
  Imagine you are a wealthy landlord. And that you have a ten-story building at some specific location. You have rented out each floor for different services.

- **The warehouse floor with the manager!**
  One floor is for a warehouse. Since this block has many rooms, and it could use management, you hired a manager. The manager oversees pretty much everything: the arrangement of objects in each room, when and how to add, remove, replace and repair items, and so much more. The manager is pretty much in charge of everything. If you, as the owner, want something to be done, you can talk to the manager. He/she will happily comply as long as you can prove you are the owner. The manager is a bit out of the ordinary, so you need a specific set of communication rules to talk to him about business-related subjects. Otherwise, he/she won't comply. And to accomplish this might duty of his, he might use a handful of tools like a forklift.

- **How you reach the manager!**
  You also have many ways of sending the instructions. You can either call or video chat. Whatever is convenient to you will do the trick.

- **The server!**
  Starting with the building, it would resemble a server. The server could be virtual, shared, or just a large data center. The location would mirror your server's domain name or public IP address. If you want to store your things and want them to be accessible, let interested parties know about the location. You, however, have to be more specific than that. Giving the building's address won't be enough. You also have to provide the floor number; the equivalent of that would be a port number. Whoever is accessing the server, the port number will tell which program to talk to.

- **The DBMS!!**
  Your warehouse floor couldn't by itself be a warehouse if it wasn't for a manager, right? You require the same thing with databases. The server alone can't store anything. At least, it will need an operating system with some sort of file system. That, however, isn't very efficient for the type of data we use databases. So you will have to install some program on top of the hardware, on top of the os. There are many options, but today we will talk about MySQL.

- **MySQL!!!**
  MySQL is a relational database management system. Its name even says it. **It is equivalent to the manager. This database uses relations to store the items on tables (shelves),** hence the name relational. **Tables in the same room can make relations. And a room is equivalent to one database**. The same way the manager could handle over one room in the same warehouse (the floor), MySQL can control over one database in the same server. MySQL wraps around the hardware and the Os and facilitates the data storage. You, **as the owner, can command the manager to add, remove and update items in-store, even shelves in a room**. The same goes with MySQL. You can tell it to do anything you want.

- **The commands!!**
  As mentioned, you need a specific set of instructions, but **first, prove yourself. You can identify who you are with your password or private keys. It doesn't matter if you are a fully privileged owner or a user with some restricted access. The manager will talk to you if you prove yourself. But even then, you have to use specific keywords to communicate since the manager is weird** (for lack of a better word). The same goes with MySQL: **you have to use SQL (a structured query language). Only then can you communicate.**

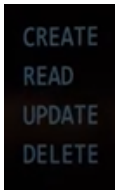
- **But first the channel!!**
  Sending the message comes first before understanding takes place. That is why you need channels. After knowing the address of your server and MySQL's listening port, you need a portal to take you there. You can talk to the manager over the phone or over a video. That is equivalent to a simple command-line interface using ssh or a more user-friendly GUI like Phpmyadmin and MySQL workbench, which will use the internet as their road.

- **The storage engine!!**

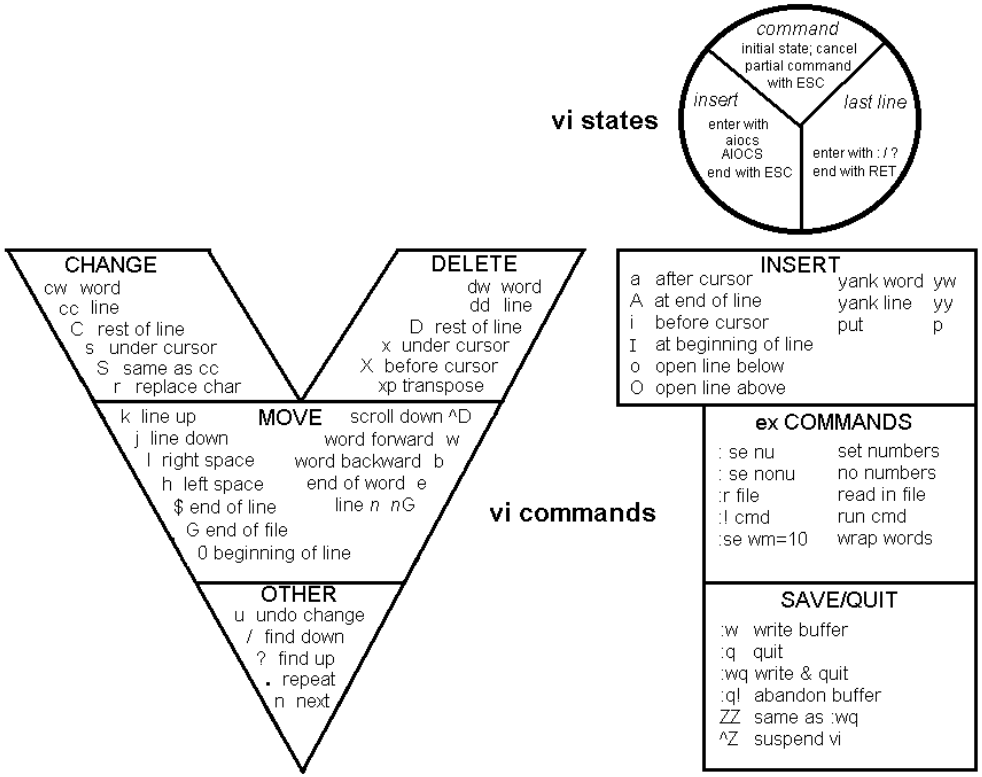
After channels are established and communications made, the manager could use tools like the forklift to do the dirty work. The equivalent would be storage engines. Storage engines will do the dirty work of actually storing, deleting, and updating data on the os's filesystem
. They take the command from MySQL and do the communication with the os. To understand this more, you can make your own simple storage engine for a command-line-based DBMS.But in the end the storage engine stores data and tables in file types like '.frm', which facilitate the different functions of MySQL.

Pretty much that is all about databases there is. But sometimes the server might be your computer, in which case it is called localhost. Both the owner/user and the database have the same IP address. Every analogy still holds, though.

SQL = Structures Query Language



VI Editor Basic Commands



- **The vi editor tool is** an interactive tool as it displays changes made in the file on the screen while you edit the file**. In vi editor you can insert, edit or remove a word as cursor moves throughout the file. Commands are specified for each function like to delete it's x or dd. The vi editor is case-sensitive.**

Basics

- VI stands for visual editor.
- Can handle text files.
- Its case-sensitive. Needs to be dealt with care as no error messages appear like while executing SQL queries or UNIX commands.

VI editor modes of operation

-> Command Mode

- In this mode all the keys pressed by the user are considered as executable commands. Once the Vi editor is invoked it enters into the command mode. To return to command mode from any of the below mode press 'ESC' key.

-> Insert Mode

- This mode permits insertion of new text, editing of the existing text & replacing of the existing text for the file.
- To enter in the insert mode by selecting any of the below keys.
- I, i, A, a, O, o, R, r, C, c

-> Ex Command Mode

- This mode permits the user to give commands at the command line (the bottom line of the vi editor screen).

- The command line is used to display messages & commands.
- All block commands are executed in this mode
- Any commands proceeded with the : (colon) symbol are given in this mode.

Invoking vi editor

- $vi file1
- The vi editor shows the full screen view of the file. If the file isn't long enough to fill the screen; vi editor shows tildes(~) on the blank lines beyond the EOF.

Moving the cursor

- h-moves the cursor one character left
- 2h-moves the cursor 2 characters left
- l-moves the cursor one character right
- 3l-moves the cursor 3 characters right
- j-moves the cursor one character below
- 4j- moves the cursor 4 characters below
- k-moves the cursor one character above
- 5k-moves the cursor 5 characters above
- w- moves cursor one word forward
- b- moves the cursor one word backward
- e- end of the current word
- 0- moves cursor at the beginning of the current line
- $-moves cursor at the end of the current line
- +-moves cursor below the beginning of the next line.
- --moves the cursor above the beginning of the previous lines
- H- go to the first line on the screen
- M-go to the middle line on the screen
- L- go to the last line on the screen
- G-go to the last line of the file

Scrolling the screen

- Ctrl+f – Scroll forward one window
- Ctrl+b – Scroll backward one window

Inserting text w.r.t cursor position

- i- Inserts text before the current position
- I-Inserts text at the beginning of the file
- a- Inserts text after the current position
- A- Inserts text at the end of the file
- o -Inserts a blank line below the current position
- O-Inserts a blank line above the current position
- c –Change current object
- C-Change from current position till end of line.
- r –Replace character at current position
- R- Replace all characters until <ESC> is pressed.

Deleting text w.r.t cursor position

- dw –Delete current word
- dd –Delete current line
- d0- Delete from current position to the beginning of the line
- d$ -Delete from current position to the end of line
- x- Deletes the character directly under the current position
- <n>x- Deletes n characters
- <n>dw –Deletes n words
- <n>dd- Deletes n lines
- J – Join the EOL character. Join the current & next line.

- <n>J –Join the next n lines

Undo change w.r.t cursor position

- u –Undo the effect of the last command
- U –Undo all changes to the current line since the cursor was moved to this line.
- ~ - Changes the character in the current position from upper to lower & vice-versa
- :sh –Temporarily returns to the shell to perform some shell commands. Type 'exit' to return to the vi editor.

Searching patters w.r.t cursor position

- /<string> -Search forward to the next occurrence of the string
- ?<string> -Search backward to the next occurrence of the string
- ^<string>- Search for all the lines which begin with the string
- <string>$-Search for all the lines which end with the string
- \<<string>-Search for all the words which begin with the string
- <string>\>-Search for all the words which end with the string

You may use metacharacters to represent the <string>

Quitting Vi editor

- :q! –Will terminate the file whether or not the changes made in the buffer were written
- :wq –Write all changes & quit editor.
- :w file1 –Write all changes to file1 & quit editor
- :q –Quits editor if the changes made were written to a file

Block commands in vi editor

- First press the 'Esc' key to enter the command mode.
- Then the ':' key to begin with block commands.
- To display line numbers enter the command 'set number' after following the above commands.
- To turn off the numbering type command 'set nonumber' after following the above commands.

- :4,12d -Lines 4 to 12 should be deleted from the current.
- :5 mo 6 -Moves line 5 after line 6
- :5,7 mo 9 -Moves lines 5 to 7 after line 9
- :10 co 11 -Copies line 10 after line 11
- :10-15 co 16 –Copies line 10 to 15 after line 16
- :21,31 w file1- Writes lines 21 to 31 to file1
- :21,31 w>>file1 –Appends file1 with lines 21 to 31