```
-- -------------------- 0.Data Base Processing Types ---------------------

-- OLTP = Online Transaction Processing(For day to day work)
-- OLAT = Online Analytical Processing(For analysing)

-- ---------------- 1.Retrieving Data From a Single Table ----------------

-- Lesson 1: SELECT
SHOW DATABASES;
USE sql_store;
-- * means every thing
SELECT * FROM customers;
-- WHERE clause
SELECT * FROM customers WHERE customer_id = 1;
-- ORDER BY clause
SELECT * FROM customers ORDER BY first_name;

-- ------------------------------------------------------------------------

-- Lesson 2: SELECT clause
USE sql_store;
SELECT first_name, last_name, points, (points * 10) + 10 AS 'calculation' FROM
customers;
-- DISTINCT
SELECT DISTINCT state FROM customers;
-- opposite of DISTINCT is ALL and in MYSQL , default is ALL
SELECT ALL state FROM customers;

-- Exercise: return all the products -> name, unit_price, new_price(unit_price * 1.1)
SELECT name, unit_price, unit_price * 1.1 AS new_price FROM products;

-- ------------------------------------------------------------------------

-- Lesson 3: WHERE clause
USE sql_store;
SELECT * FROM customers WHERE points > 3000;
-- <, >, =, != or <>, <=, >=
SELECT * FROM customers WHERE birth_date > '1990-01-01';
SELECT * FROM customers WHERE state = 'VA';
```

```
-- Exercise: get the orders that are placed in this year
SELECT * FROM orders WHERE order_date > '2019-01-01';

-- -------------------------------------------------------------------

-- Lesson 4: AND, NOT, OR
USE sql_store;
-- AND
SELECT * FROM customers WHERE birth_date > '1990-01-01' AND points > 2000;
-- OR
SELECT * FROM customers WHERE birth_date > '1990-01-01' OR points > 2000;
-- AND, OR => AND has higher priority than OR
SELECT * FROM customers WHERE birth_date > '1990-01-01' OR points > 2000 AND
state='VA';
-- NOT
SELECT * FROM customers WHERE NOT (birth_date > '1990-01-01' OR points > 2000);
-- means: birth_date <= '1990-01-01' AND points <= 2000

-- Exercise: from order_items get items with order_id = 6 and total_price > 30
SELECT * FROM order_items WHERE order_id = 6 AND (quantity * unit_price) > 30;

-- -------------------------------------------------------------------

-- Lesson 5: IN operator
USE sql_store;
-- with OR
SELECT * FROM customers WHERE state = 'VA' OR state = 'GA' OR state = 'FL';
-- with IN operator
SELECT * FROM customers WHERE state IN('VA', 'GA', 'FL');

-- Exercise: return products with quantity in stock equal to 49, 38, 72
SELECT * FROM products WHERE quantity_in_stock IN(49, 38, 72);

-- -------------------------------------------------------------------

-- Lesson 6: BETWEEN operator
USE sql_store;
-- with AND
```

```
SELECT * FROM customers WHERE points >= 2000 AND points <= 3000;
-- with BETWEEN
SELECT * FROM customers WHERE points BETWEEN 2000 AND 3000;

-- Exercise: return customers born BETWEEN 1990-01-01 AND 2000-01-01
SELECT * FROM customers WHERE birth_date BETWEEN '1990-01-01' AND '2000-01-
01';

-- ---------------------------------------------------------------------

-- Lesson 7: LIKE operator
USE sql_store;
-- % : means any number of characters
-- _ : means single character
SELECT * FROM customers WHERE last_name LIKE 'b%';
SELECT * FROM customers WHERE last_name LIKE 'brush%';
SELECT * FROM customers WHERE last_name LIKE '_____y';
SELECT * FROM customers WHERE last_name LIKE 'b____y';

-- Exercise: get customers whose address contains TRAIL or AVENUE
SELECT * FROM customers WHERE address LIKE '%TRAIL%' OR address LIKE
'%AVENUE%';

-- Exercise: get customers whose phone numbers end with 9
SELECT * FROM customers WHERE phone LIKE '%9';

-- ---------------------------------------------------------------------

-- Lesson 8: REGEXP operator
USE sql_store;
-- ^ : means start of string
-- $ : means end of string
-- | : means logical or
-- [gio] : means have one of g or i or o
-- [a-z] : means have one of a to z
SELECT * FROM customers WHERE last_name REGEXP 'field$|mac|^rose|[gim]e';

-- Exercise: get the customers whose first_name are ELKA or AMBUR
SELECT * FROM customers WHERE first_name REGEXP 'AMBUR|ELKA';
```

```
-- Exercise: get the customers last_name ends with EY or ON
SELECT * FROM customers WHERE last_name REGEXP 'EY$|ON$';
-- Exercise: get the customers last_name starts with MY or contains SE
SELECT * FROM customers WHERE last_name REGEXP '^MY|SE';
-- Exercise: get the customers last_name contains B followed by R or U
SELECT * FROM customers WHERE last_name REGEXP 'B[RU]';

-- ---------------------------------------------------------------------

-- Lesson 9: IS NULL operator
USE sql_store;
SELECT * FROM customers WHERE phone IS NULL;

-- Exercise: get the orders that are not shipped
SELECT * FROM orders WHERE shipper_id IS NULL OR shipped_date IS NULL;

-- ---------------------------------------------------------------------

-- Lesson 10: ORDER BY clause
USE sql_store;
-- ASC which is the default
SELECT * FROM customers ORDER BY first_name ASC;
-- DESC
SELECT * FROM customers ORDER BY first_name DESC;
SELECT * FROM customers ORDER BY state, last_name;
SELECT * FROM customers ORDER BY state ASC, last_name DESC;

-- Exercise: from order_items WHERE order_id = 2 and order by total price
SELECT * FROM order_items WHERE order_id = 2 ORDER BY (unit_price * quantity)
DESC;
-- using alias(Only works in MYSQL)
SELECT order_id, product_id, (unit_price * quantity) AS total_price FROM order_items
WHERE order_id = 2 ORDER BY total_price DESC;

-- ---------------------------------------------------------------------

-- Lesson 11: LIMIT clause
USE sql_store;
SELECT * FROM customers LIMIT 10;
```

```
-- Ignore 5 first and show 10 items after 5 => 5 - 15
SELECT * FROM customers LIMIT 5, 10;

-- Exercise: get the top 3 loyal customers
SELECT * FROM customers ORDER BY points DESC LIMIT 3;

-- --------- 2. Retrieving Data From Multiple Tables ----------------------

-- Lesson 12: Inner join
USE sql_store;
SELECT order_id, first_name, last_name, orders.customer_id FROM orders
JOIN customers c on orders.customer_id = c.customer_id;

SELECT order_id, first_name, last_name, o.customer_id FROM orders o
JOIN customers c on o.customer_id = c.customer_id;

SELECT  order_id, oi.product_id, quantity, oi.unit_price FROM order_items oi
JOIN products p on oi.product_id = p.product_id;

-- ------------------------------------------------------------------------

-- Lesson 13: Joining across multiple tables
use sql_store;
SELECT * FROM order_items oi
JOIN sql_inventory.products p ON oi.product_id = p.product_id;

-- ------------------------------------------------------------------------

-- Lesson 14: Self joins
USE sql_hr;
SELECT e.employee_id,
      e.first_name,
      e.last_name,
      m.first_name  AS manager_first_name,
      m.last_name   AS manager_last_name,
      m.employee_id AS manager_id
FROM employees e
       JOIN employees m ON e.reports_to = m.employee_id;
```

```
-- ---------------------------------------------------------------------

-- Lesson 15: Joining Multiple Tables
USE sql_store;
SELECT o.order_id, o.order_date, c.first_name, c.last_name, os.name AS status
FROM orders o
	JOIN customers c ON o.customer_id = c.customer_id
	JOIN order_statuses os ON o.status = os.order_status_id;

-- Exercise: use sql_invoicing database : join payments with clients and join payments
with payment_methods
USE sql_invoicing;
SELECT p.date, p.invoice_id, p.amount, c.name AS client, pm.name AS method
FROM payments p
	JOIN clients c ON p.client_id = c.client_id
	JOIN payment_methods pm ON p.payment_method = pm.payment_method_id;

-- ---------------------------------------------------------------------

-- Lesson 16: Compound Join Condition

USE sql_store;

SELECT *
FROM order_items oi
	JOIN order_item_notes oin ON oi.order_id = oin.order_Id
   AND oi.product_id = oin.product_id;

-- ---------------------------------------------------------------------

-- Lesson 17: Implicit Join

-- It's better to not use implicit join because if we forget WHERE it will be cross join
USE sql_store;

-- Explicit Join
SELECT *
FROM orders o
	JOIN customers c ON o.customer_id = c.customer_id;
```

```
-- Implicit Join
SELECT *
FROM orders o,
    customers c
WHERE o.customer_id = c.customer_id;


-- ---------------------------------------------------------------------

-- Lesson 18: Outer Join

-- left: the columns in left side have value if right have not does not matter
-- right: the columns in right side have value if left have not does not matter
USE sql_store;

SELECT c.customer_id, c.first_name, o.order_id
FROM customers c
       JOIN orders o ON c.customer_id = o.customer_id
ORDER BY c.customer_id;

-- if any field in a row was null(like order_id in this example) that row will not show in
INNER JOIN
-- LEFT OUTER JOIN = LEFT JOIN
-- RIGHT OUTER JOIN = RIGHT JOIN

-- LEFT in here is customers
SELECT c.customer_id, c.first_name, o.order_id
FROM customers c
       LEFT JOIN orders o ON c.customer_id = o.customer_id
ORDER BY c.customer_id;

-- RIGHT here is order because order_id have null values so we can't see all the
customers
SELECT c.customer_id, c.first_name, o.order_id
FROM customers c
       RIGHT JOIN orders o ON c.customer_id = o.customer_id
ORDER BY c.customer_id;
```

```
-- Exercise: product_id, name, quantity join products table with order_items
SELECT *
FROM products;

SELECT *
FROM order_items;

SELECT p.product_id, p.name, oi.quantity
FROM products p
      LEFT JOIN order_items oi ON p.product_id = oi.product_id
ORDER BY p.product_id;

-- ---------------------------------------------------------------------

-- Lesson 19: Outer Joins Between Multiple Tables

USE sql_store;

-- Try not to use right join for simplicity
SELECT c.customer_id, c.first_name, o.order_id, s.name AS shipper
FROM customers c
       LEFT JOIN orders o ON c.customer_id = o.customer_id
       LEFT JOIN shippers s ON o.shipper_id = s.shipper_id
ORDER BY c.customer_id;

-- Exercise: order_date, order_id, first_name, shipper, status
SELECT o.order_id, o.order_date, os.name, c.first_name, s.name AS shipper
FROM orders o
       JOIN customers c ON o.customer_id = c.customer_id
       LEFT JOIN shippers s ON o.shipper_id = s.shipper_id
       LEFT JOIN order_statuses os ON o.status = os.order_status_id;

-- ---------------------------------------------------------------------

-- Lesson 20: Self Outer Joins

USE sql_hr;
-- Inner Join
SELECT e.employee_id, e.first_name, m.first_name AS manager
```

```
FROM employees e
	JOIN employees m ON e.reports_to = m.employee_id;

-- Outer Join
SELECT e.employee_id, e.first_name, m.first_name AS manager
FROM employees e
	LEFT JOIN employees m ON e.reports_to = m.employee_id;

-- ---------------------------------------------------------------------

-- Lesson 21: USING clause

USE sql_store;
-- if column name is exactly the same across these 2 tables we can utilize USING()

SELECT o.order_id, c.first_name, s.name AS shipper
FROM orders o
	JOIN customers c USING (customer_id)
	LEFT JOIN shippers s USING (shipper_id);
-- ON o.customer_id = c.customer_id

SELECT *
FROM order_items oi
	JOIN order_item_notes oin ON oi.order_id = oin.order_id AND oi.product_id =
oin.product_id;

SELECT *
FROM order_items oi
	JOIN order_item_notes oin USING (order_id, product_id);

-- Exercise: use sql_invoicing data base and from payments table: date, client, amount,
name
USE sql_invoicing;
SELECT p.date, c.name AS client, p.amount, pm.name AS payment_method
FROM payments p
	JOIN clients c USING (client_id)
	JOIN payment_methods pm ON p.payment_method = pm.payment_method_id;

-- ---------------------------------------------------------------------
```

```
-- Lesson 22: NATURAL JOIN
-- NATURAL JOIN: join based on common column

USE sql_store;

SELECT o.order_id, c.first_name
FROM orders o
	NATURAL JOIN customers c;

-- ---------------------------------------------------------------------

-- Lesson 23: CROSS JOIN

USE sql_store;

-- Explicit
SELECT c.first_name AS customer, p.name AS product
FROM customers c
	CROSS JOIN products p
ORDER BY c.first_name;

-- Implicit
SELECT c.first_name AS customer, p.name AS product
FROM customers c,
    products p
ORDER BY c.first_name;

-- Exercise: Do a cross join between shippers and products using implicit syntax and
explicit one
-- Implicit
SELECT s.name AS shipper, p.name AS product
FROM shippers s,
    products p
ORDER BY s.name;
-- Explicit
SELECT s.name AS shippers, p.name AS product
FROM shippers s
	CROSS JOIN products p
```

```
ORDER BY s.name;

-- ---------------------------------------------------------------------

-- Lesson 24: UNION

USE sql_store;

SELECT order_id, order_date, 'Active' AS status
FROM orders
WHERE order_date >= '2019-01-01'
UNION
SELECT order_id, order_date, 'Archived' AS status
FROM orders
WHERE order_date < '2019-01-01';

SELECT first_name AS 'First Name'
FROM customers
UNION
SELECT name
FROM shippers;

-- Exercise: customer_id, first_name, point,
-- type | greater 2000: bronze | between 2000 and 3000: silver | greater than 3000:
gold And sort by first_name
SELECT customer_id, first_name, points, 'Bronze' AS 'type'
FROM customers
WHERE points < 2000
UNION
SELECT customer_id, first_name, points, 'Silver' AS 'type'
FROM customers
WHERE points BETWEEN 2000 AND 3000
UNION
SELECT customer_id, first_name, points, 'Gold' AS 'type'
FROM customers
WHERE customers.points > 3000
ORDER BY first_name;

-- ---------- 3.Inserting, Updating, and Deleting Data ----------
```

```
-- Lesson 25: Column Attributes

-- VARCHAR: variable characters(variable length up to NUM)
-- VARCHAR(NUM)
-- CHAR: character(fix length) -> waste space
-- CHAR(NUM)
-- INT: integer
-- INT(NUM)
-- PK: Primary Key
-- NN: Not Null
-- AI: Auto Increment
-- Default/ Expression: Default value for a column

-- ---------------------------------------------------------------------

-- Lesson 26: Insert a row

USE sql_store;

INSERT INTO customers VALUE (DEFAULT,
                        'ali',
                        'azani',
                        '1998-10-23',
                        NULL,
                        'Karegar st.',
                        'Tehran',
                        'TA',
                        3800);

INSERT INTO customers (first_name,
                    last_name,
                    birth_date,
                    address,
                    city,
                    state,
                    points)
VALUES ('Maria',
       'Rosen',
```

```
        '1996-08-13',
        'karegar st.',
        'Tehran',
        'TA',
        4800);

SELECT * FROM sql_store.customers WHERE city = 'Tehran';

-- ---------------------------------------------------------------------

-- Lesson 27: Insert Multiple Row

USE sql_store;

INSERT INTO shippers (name)
VALUES ('shipper-1'),
       ('shipper-2'),
       ('shipper-3');

SELECT *
FROM shippers;

-- Exercise: Insert 3 rows in products table
INSERT INTO products (name, quantity_in_stock, unit_price)
VALUES ('snake', 10, 23),
       ('t-shirt', 14, 34),
       ('shoes', 42, 50);

SELECT *
FROM products;

-- ---------------------------------------------------------------------

-- Lesson 28: INSERT INTO multiple tables(Inserting Hierarchical Rows)

USE sql_store;

INSERT INTO orders(customer_id, order_date, status)
VALUES (1, '2021-01-12', 1);
```

```
SELECT *
FROM orders;

INSERT INTO order_items(order_id, product_id, quantity, unit_price)
VALUES (LAST_INSERT_ID(), 1, 12, 2.32),
      (LAST_INSERT_ID(), 2, 2, 10.12);

SELECT *
FROM order_items;

-- ---------------------------------------------------------------------

-- Lesson 29: create a copy of table

USE sql_store;

CREATE TABLE IF NOT EXISTS orders_archived AS
SELECT *
FROM orders
WHERE order_date < '2020-01-01';

SELECT *
FROM orders_archived;

-- sub query example
INSERT INTO orders_archived (SELECT * FROM orders WHERE order_date < '2019-01-
01');

SELECT *
FROM orders_archived;

-- Exercise: create a new table invoices_archived and instead of client_id
-- we should have client_name column and only copy the invoices have payment_date
USE sql_invoicing;
CREATE TABLE IF NOT EXISTS invoices_archived AS
SELECT i.invoice_id,
      i.number,
      c.name AS client_name,
```

```
      i.invoice_total,
      i.payment_total,
      i.invoice_date,
      i.payment_date,
      i.due_date
FROM invoices i
       JOIN clients c USING (client_id)
WHERE payment_date IS NOT NULL;

SELECT * FROM invoices_archived;

-- ---------------------------------------------------------------------

-- Lesson 30: Updating a single row

USE sql_invoicing;

UPDATE invoices
SET payment_total = 92.5,
   payment_date='2020-01-01'
WHERE invoice_id = 1;

SELECT *
FROM invoices
WHERE invoice_id = 1;

UPDATE invoices
SET payment_total = DEFAULT,
   payment_date  = NULL
WHERE invoice_id = 1;

SELECT *
FROM invoices
WHERE invoice_id = 1;

UPDATE invoices
SET payment_total = (invoice_total * 0.5),
   payment_date  = due_date
WHERE invoice_id = 1;
```

```
SELECT *
FROM invoices
WHERE invoice_id = 1;

-- ---------------------------------------------------------------------

-- Lesson 31: updating multiple rows

USE sql_invoicing;

UPDATE invoices
SET payment_total = (invoice_total * 0.7),
   payment_date  = due_date
WHERE client_id = 1;

-- to update multiple rows value you should uncheck the safe update
SELECT *
FROM invoices;

-- Exercise: write a query to give any customer born before 1990, 50 extra points
USE sql_store;

UPDATE customers
SET points = points + 50
WHERE birth_date < '1990-01-01';

SELECT * FROM customers;

-- ---------------------------------------------------------------------

-- Lesson 32: Using sub-queries in updating

USE sql_invoicing;

UPDATE invoices
SET payment_total = (invoice_total * 0.7),
   payment_date  = due_date
WHERE client_id IN (SELECT client_id FROM clients WHERE name = 'myworks');
```

```
SELECT *
FROM invoices;

UPDATE invoices
SET payment_total = (invoice_total * 0.7),
   payment_date  = due_date
WHERE client_id IN (SELECT client_id FROM clients WHERE state IN ('CA', 'NY'));

SELECT *
FROM invoices;

-- Exercise: in orders and some of rows have not comments, update the comments for
customers have more than 3000 points
USE sql_store;
UPDATE orders
SET comments = 'Gold Customer'
WHERE customer_id IN (SELECT customer_id FROM customers WHERE points > 3000);

SELECT *
FROM orders;

-- ---------------------------------------------------------------------

-- Lesson 33: DELETE row

-- Lesson 33: Delete row
USE sql_invoicing;

-- Delete all the rows
DELETE
FROM invoices;

DELETE
FROM invoices
WHERE invoice_id = 1;

DELETE
FROM invoices
```

```
WHERE client_id = (SELECT * FROM clients WHERE name = 'myworks');

-- ---------------------- 4.Summarizing Data ----------------------------

-- Lesson 34: Aggregate functions

-- MIN(), MAX(), SUM(), AVG(), COUNT()

USE sql_invoicing;

SELECT MAX(invoice_total)       AS highest,
      MIN(invoice_total)        AS lowset,
      SUM(invoice_total)        AS summarized_data,
      AVG(invoice_total)        AS average,
      COUNT(invoice_total)      AS number_of_invoices,
      COUNT(payment_date)       AS number_of_payment,
      COUNT(*)                  AS total_records,
      COUNT(client_id)          AS all_clients,
      COUNT(DISTINCT client_id) AS distinct_client
FROM invoices
WHERE invoice_date > '2019-07-01';


SELECT MAX(payment_date) AS highest,
      MIN(payment_date) AS lowset
FROM invoices;

-- Exercise: write a query for invoices table : data_range, total_sale, total _payment,
what_we_expect
-- first half of 2019, second half of 2019, total
SELECT 'First Half Of 2019'             AS date_range,
      SUM(invoice_total)                AS total_sale,
      SUM(payment_total)                AS total_payment,
      SUM(invoice_total - payment_total) AS what_we_excpect
FROM invoices
WHERE invoice_date BETWEEN '2019-01-01' AND '2019-06-30'
UNION
SELECT 'Second Half Of 2019'            AS date_range,
      SUM(invoice_total)                AS total_sale,
```

```
    SUM(payment_total)              AS total_payment,
    SUM(invoice_total - payment_total) AS what_we_excpect
FROM invoices
WHERE invoice_date BETWEEN '2019-07-01' AND '2019-12-31'
UNION
SELECT 'total Of 2019'              AS date_range,
    SUM(invoice_total)              AS total_sale,
    SUM(payment_total)              AS total_payment,
    SUM(invoice_total - payment_total) AS what_we_excpect
FROM invoices
WHERE invoice_date BETWEEN '2019-01-01' AND '2019-12-31'

-- -------------------------------------------------------------------------

-- Lesson 35: GROUP BY clause
USE sql_invoicing;

SELECT client_id, SUM(invoice_total) AS total_sale
FROM invoices
WHERE invoice_date >= '2019-07-01'
GROUP BY client_id
ORDER BY total_sale DESC;

SELECT city, state, SUM(invoice_total) AS total_sale
FROM invoices i
      JOIN clients USING (client_id)
GROUP BY city, state;

-- Exercise: date, payment_methods, total_payments
SELECT date,
    pm.name    AS payment_method,
    SUM(amount) AS total_payment
FROM payments p
      JOIN payment_methods pm ON p.payment_method = pm.payment_method_id
GROUP BY date, payment_method
ORDER BY date;

-- -------------------------------------------------------------------------
```

```sql
-- Lesson 36: HAVING clause
-- Filter the data after GROUP BY clause
-- With WHERE , we can filter data before GROUP BY clause
-- With HAVING, we can filter data after GROUP BY clause

USE sql_invoicing;

SELECT SUM(invoice_total) AS total_sale,
     COUNT(*)         AS number_of_invoices
FROM invoices
GROUP BY client_id
HAVING total_sale > 500
  AND number_of_invoices > 5;

-- Exercise: get the customers located in virgina who spent more than 100$
USE sql_store;

SELECT c.customer_id,
     c.first_name,
     c.last_name,
     SUM(oi.quantity * oi.unit_price) AS total_spend_money
FROM customers c
       JOIN orders USING (customer_id)
       JOIN order_items oi USING (order_id)
WHERE state = 'VA'
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING total_spend_money > 100;

-- ---------------------------------------------------------------------

-- Lesson 37: ROLLUP operator
USE sql_invoicing;

SELECT client_id,
     SUM(invoice_total) AS total_sale
FROM invoices i
       JOIN clients USING (client_id)
GROUP BY client_id
WITH ROLLUP;
```

```
SELECT state,
      city,
      SUM(invoice_total) AS total_sale
FROM invoices i
       JOIN clients c USING (client_id)
GROUP BY state, city
WITH ROLLUP;

-- Exercise: payment_method, total
SELECT pm.name     AS payment_method,
      SUM(amount) AS total
FROM payments p
       JOIN payment_methods pm ON p.payment_method = pm.payment_method_id
GROUP BY pm.name
WITH ROLLUP;
-- When we use WITH ROLLUP, we can not use alias

-- -------------------- 5.Writing Complex Query -------------------------

-- Lesson 38: sub-queries

USE sql_store;
-- Find products that are more expensive than lettuce(id = 3)
SELECT *
FROM products
WHERE unit_price > (SELECT unit_price FROM products WHERE product_id = 3);

-- Exercise: in sql_hr: find the employees whose earn more than average

USE sql_hr;

SELECT *
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);

-- ------------------------------------------------------------------------

-- Lesson 39: IN operator
```

```
USE sql_store;

SELECT *
FROM products
WHERE product_id NOT IN (SELECT DISTINCT product_id FROM order_items);

-- Exercise: Find clients without invoices

USE sql_invoicing;
SELECT *
FROM clients
WHERE client_id NOT IN (SELECT DISTINCT client_id FROM invoices);

-- ---------------------------------------------------------------------

-- Lesson 40: sub-queries VS joins

-- use the more readable statement
USE sql_invoicing;

-- JOIN
SELECT *
FROM clients c
	LEFT JOIN invoices i USING (client_id)
WHERE invoice_id IS NULL;

-- Sub-queries
SELECT *
FROM clients
WHERE client_id NOT IN (SELECT DISTINCT client_id FROM invoices);

-- Exercise: find the customers who have ordered lettuce (id = 3)
-- SELECT customer_id, first_name, last_name
USE sql_store;

-- Sub-queries
SELECT customer_id, first_name, last_name
FROM customers
WHERE customer_id IN
```

```
    (SELECT customer_id
     FROM orders
     WHERE order_id IN
         (SELECT order_id
          FROM order_items
          WHERE product_id = 3));

-- JOINS
SELECT DISTINCT c.customer_id,
             c.first_name,
             c.last_name
FROM order_items oi
       LEFT JOIN orders o USING (order_id)
       LEFT JOIN customers c USING (customer_id)
WHERE product_id = 3;

-- Combination
SELECT customer_id, first_name, last_name
FROM customers
WHERE customer_id
        IN (SELECT o.customer_id
           FROM order_items oi
                JOIN orders o USING (order_id)
           WHERE product_id = 3);

-- ---------------------------------------------------------------------

-- Lesson 41: ALL keyword

USE sql_invoicing;

SELECT *
FROM invoices
WHERE invoice_total > (SELECT MAX(invoice_total)
                FROM invoices
                WHERE client_id = 3);
-- ALL
SELECT *
FROM invoices
```

```
WHERE invoice_total > ALL (SELECT invoice_total
                           FROM invoices
                           WHERE client_id = 3);

-- ---------------------------------------------------------------------

-- Lesson 42: ANY keyword

USE sql_invoicing;
SELECT *
FROM clients
WHERE client_id IN (SELECT client_id
                    FROM invoices
                    GROUP BY client_id
                    HAVING COUNT(*) >= 2);

-- ANY
SELECT *
FROM clients
WHERE client_id = ANY (SELECT client_id
                       FROM invoices
                       GROUP BY client_id
                       HAVING COUNT(*) >= 2);

-- ---------------------------------------------------------------------

-- Lesson 43: Correlated sub-queries

USE sql_hr;

-- SELECT employees whose salary is above the average in their office:
-- for each employee
-- calculate the average salary for employee.office
-- return the employee if salary > AVG(salary)
-- sub-query will execute for each row
SELECT *
FROM employees e
WHERE salary > (SELECT AVG(salary)
                FROM employees
```

```
            WHERE e.office_id = office_id);

-- Exercise: get the invoices that are larger than the client average invoice amount
USE sql_invoicing;
SELECT *
FROM invoices i
WHERE invoice_total > (SELECT AVG(invoice_total)
                       FROM invoices
                       WHERE i.client_id = client_id)
ORDER BY client_id;

-- ---------------------------------------------------------------------

-- Lesson 44: EXISTS operator
USE sql_invoicing;

SELECT *
FROM clients
WHERE client_id IN (
   SELECT DISTINCT client_id
   FROM invoices
);

SELECT *
FROM clients c
WHERE EXISTS(
           SELECT client_id
           FROM invoices i
           WHERE c.client_id = i.client_id
        );

-- Exercise: find the products that have never been ordered
USE sql_store;

SELECT *
FROM products p
WHERE NOT EXISTS(SELECT product_id
                 FROM order_items oi
                 WHERE p.product_id = oi.product_id);
```

```
SELECT *
FROM products
WHERE product_id NOT IN (SELECT product_id
                         FROM order_items);

-- ---------------------------------------------------------------------

-- Lesson 45: Sub-queries in the SELECT clause

USE sql_invoicing;

SELECT invoice_id,
    invoice_total,
    (SELECT AVG(invoice_total) FROM invoices) AS invoice_average,
    invoice_total - (SELECT invoice_average)  AS difference
FROM invoices;

-- Exercise: client_id, name, total_sales, average, difference

SELECT client_id,
    name,
    (SELECT SUM(invoice_total)
     FROM invoices
     WHERE client_id = c.client_id)          AS total_sales,
    (SELECT AVG(invoice_total) FROM invoices) AS average,
    (SELECT total_sales - average)          AS difference
FROM clients c;

-- ---------------------------------------------------------------------

-- Lesson 46: Sub-queries in the FROM clause
USE sql_store;

SELECT *
FROM (SELECT client_id,
          name,
          (SELECT SUM(invoice_total)
           FROM invoices
```

```
            WHERE client_id = c.client_id)          AS total_sales,
           (SELECT AVG(invoice_total) FROM invoices) AS average,
           (SELECT total_sales - average)           AS difference
     FROM clients c
    ) AS sales_summary
WHERE total_sales IS NOT NULL;

-- ----------------- 6.Essential MySQL Functions ------------------------

-- Lesson 47: Numeric functions

SELECT ROUND(5.50);
SELECT ROUND(5.76, 1);
SELECT TRUNCATE(5.726, 2);
SELECT CEILING(2.2);
SELECT FLOOR(5.8);
SELECT ABS(-23);
-- Random number between 0 and 1
SELECT RAND();

-- ------------------------------------------------------------------

-- Lesson 48: String functions

SELECT LENGTH('ALI');

SELECT UPPER('ali');
SELECT LOWER('ALI');

SELECT LTRIM('        sky');
SELECT RTRIM('sky     ');
SELECT TRIM('    sky    ');

SELECT LEFT('Kindergarten', 4);
SELECT RIGHT('Kindergarten', 6);

SELECT SUBSTRING('Ali Azani', 5, 4);

SELECT LOCATE('n', 'No name');
```

```
SELECT LOCATE('s', 'No name');
SELECT LOCATE('name', 'No name');

SELECT REPLACE('Kindergarten', 'garten', 'garden');

SELECT CONCAT('FirstName', ' LastName');

USE sql_store;

SELECT customer_id,
    CONCAT(first_name, last_name) AS full_name,
    birth_date
FROM customers;

-- ------------------------------------------------------------------------

-- Lesson 49: Date and Time functions

SELECT NOW(), CURDATE(), CURTIME();

SELECT YEAR(NOW()),
    MONTH(NOW()),
    DAY(NOW()),
    TIME(NOW()),
    HOUR(NOW()),
    MINUTE(NOW()),
    SECOND(NOW());

SELECT DAYNAME(NOW()),
    MONTHNAME(NOW());

SELECT EXTRACT(DAY FROM NOW()),
    EXTRACT(MONTH FROM NOW()),
    EXTRACT(YEAR FROM NOW());

-- Exercise: orders placed in the current year
USE sql_store;
SELECT * FROM orders WHERE YEAR(order_date) = YEAR(NOW()) - 2;
```

```
-- ---------------------------------------------------------------------

-- Lesson 50: Formatting Date and Time

SELECT DATE_FORMAT(NOW(), '%Y/%M/%D');
SELECT DATE_FORMAT(NOW(), '%y/%m/%d');
SELECT DATE_FORMAT(NOW(), '%H:%i %p');

-- ---------------------------------------------------------------------

-- Lesson 51: Calculating Dates and Times

-- Date
SELECT DATE_ADD(NOW(), INTERVAL -4 YEAR ),
      DATE_ADD(NOW(), INTERVAL 2 MONTH ),
      DATE_SUB(NOW(), INTERVAL 10 DAY),
      DATEDIFF('2019-01-01 9:00', '2021-01-01 12:00');

-- Time
SELECT TIME_TO_SEC('9:00') - TIME_TO_SEC('12:00');

-- ---------------------------------------------------------------------

-- Lesson 52: The IFNULL and COALESCE functions

USE sql_store;

SELECT order_id,
      IFNULL(shipper_id, 'Not Assigned') AS shipper_id
FROM orders;

SELECT order_id,
      COALESCE(shipper_id, comments, 'Not Assigned') AS shipper_id
FROM orders;

-- Exercise: customer(full_name) and the phone(unknown)
SELECT CONCAT(first_name, ' ', last_name) AS full_name,
      COALESCE(phone, 'Unknown')        AS phone_number
FROM customers;
```

```
-- ---------------------------------------------------------------------

-- Lesson 53: IF function
USE sql_store;

SELECT order_id,
      order_date,
      IF(YEAR((order_date) = YEAR(NOW())) - 1,
        'Active',
        'Archived') AS status
FROM orders;



-- Exercise: product_id, name, orders,
-- frequency(more than 1: many times, 1: once)
SELECT DISTINCT product_id,
              name,
              COUNT(*)   AS orders,
              IF(COUNT(*) > 1,
                'Many Times',
                'Once') AS frequency
FROM products p
       JOIN order_items oi USING (product_id)
GROUP BY product_id, name;

-- ---------------------------------------------------------------------

-- Lesson 54: CASE operator
USE sql_store;

SELECT order_id,
      order_date,
      CASE
         WHEN YEAR(order_date) = YEAR(NOW())
            THEN 'Active'
         WHEN YEAR(order_date) = YEAR(NOW()) - 1
            THEN 'Last Year'
         WHEN YEAR(order_date) < YEAR(NOW()) - 1
```

```
            THEN 'Archived'
        ELSE 'Future Year'
        END AS category
FROM orders;

-- Exercise: full name customer, points, category -> greater tha 3000: Gold
-- Between 2000 And 3000: Silver
-- Less than 2000: Bronze
SELECT CONCAT(first_name, ' ', last_name) AS full_name,
    points,
    CASE
        WHEN points > 3000 THEN 'Gold'
        WHEN points BETWEEN 2000 AND 3000 THEN 'Silver'
        ELSE 'Bronze'
        END                     AS category
FROM customers
ORDER BY points DESC;

-- ----------------------- 7.Views -------------------------------------

-- Lesson 55: Creating Views

-- Views behave like a virtual table but view don't store data, our data actually stored in
our table
USE sql_invoicing;

CREATE VIEW sales_by_client AS
SELECT c.client_id,
    c.name,
    SUM(invoice_total) AS total_sale
FROM clients c
      JOIN invoices i USING (client_id)
GROUP BY client_id, name;

SELECT *
FROM sales_by_client
ORDER BY name DESC;

SELECT *
```

```
FROM sales_by_client
WHERE total_sale > 500;

-- Exercise: create a view to see the balance for each client
-- client_balance: client_id, name, balance
-- balance = invoice_total - payment_total
CREATE VIEW client_balance AS
SELECT i.client_id,
      c.name,
      SUM(invoice_total - payment_total) AS balance
FROM invoices i
       JOIN clients c USING (client_id)
GROUP BY c.client_id
ORDER BY client_id, name;

-- ---------------------------------------------------------------------

-- Lesson 56: Altering or Dropping Views

USE sql_store;

DROP VIEW client_balance;

CREATE OR REPLACE VIEW client_balance
AS
SELECT i.client_id, c.name, SUM(invoice_total - payment_total) AS balance
FROM invoices i
       JOIN clients c USING (client_id)
GROUP BY c.client_id
ORDER BY client_id, name;

SELECT * FROM client_balance;

-- ---------------------------------------------------------------------

-- Lesson 57: Update Views

-- Updatable View -> Don't have DISTINCT, Aggregate functions(MIN, MAX, AVG, SUM),
GROUP BY, HAVING, UNION
```

```sql
-- INSERT, UPDATE, DELETE
CREATE OR REPLACE VIEW invoice_with_balance
AS
SELECT invoice_id,
    number,
    client_id,
    invoice_total,
    payment_total,
    invoice_total - payment_total AS balance,
    invoice_date,
    due_date,
    payment_date
FROM invoices
WHERE (invoice_total - payment_total) > 0;

SELECT *
FROM invoice_with_balance;

DELETE
FROM invoice_with_balance
WHERE invoice_id = 1;

UPDATE invoice_with_balance
SET due_date = DATE_ADD(due_date, INTERVAL 2 DAY)
WHERE invoice_id = 2;

SELECT *
FROM invoice_with_balance;

-- ---------------------------------------------------------------------

-- Lesson 58: WITH CHECK OPTION clause

USE sql_invoicing;
CREATE OR REPLACE VIEW invoice_with_balance AS
SELECT invoice_id,
    number,
    client_id,
```

```
    invoice_total,
    payment_total,
    invoice_total - payment_total AS balance,
    invoice_date,
    due_date,
    payment_date
FROM invoices
WHERE (invoice_total - payment_total) > 0;

UPDATE invoice_with_balance
SET payment_total = invoice_total
WHERE invoice_id = 3;

SELECT *
FROM invoice_with_balance;

-- get error
CREATE OR REPLACE VIEW invoice_with_balance AS
SELECT invoice_id,
    number,
    client_id,
    invoice_total,
    payment_total,
    invoice_total - payment_total AS balance,
    invoice_date,
    due_date,
    payment_date
FROM invoices
WHERE (invoice_total - payment_total) > 0
WITH CHECK OPTION;

UPDATE invoice_with_balance
SET payment_total = invoice_total
WHERE invoice_id = 3;

SELECT *
FROM invoice_with_balance;

-- ---------------------------------------------------------------------
```

```
-- Lesson 59: Other benefits of Views

-- 1) Simplify queries
-- 2) Reduce the impact of changes
-- 3) Restrict access to the data

-- ---------- 8.Stored Procedures ----------

-- Lesson 60: Creating Stored Procedure

USE sql_invoicing;

DELIMITER $$

CREATE PROCEDURE get_clients()
BEGIN
   SELECT * FROM clients;
END $$

DELIMITER ;

-- Call it in our applications (java, python, ...)
CALL sql_invoicing.get_clients();

-- Exercise: Create a stored procedure called get_invoices_with_balance to return all
the invoices with balance > 0

USE sql_invoicing;

DELIMITER $$

CREATE PROCEDURE get_invoices_with_balance()
BEGIN
   SELECT * FROM invoices WHERE invoice_total - invoices.payment_total > 0;
END $$

DELIMITER ;
```

```
CALL sql_invoicing.get_invoices_with_balance();

-- ---------------------------------------------------------------------

-- Lesson 61: Creating Stored procedure in mysql workbench

-- Right click in stored procedure and click on create stored procedure

-- ---------------------------------------------------------------------

-- Lesson 62: Dropping Stored Procedure

USE sql_invoicing;
-- This is not best way to do this(best practice)
DROP PROCEDURE get_clients;

-- best practice
DROP PROCEDURE IF EXISTS get_clients;

DELIMITER $$

CREATE PROCEDURE IF NOT EXISTS get_clients()
BEGIN
   SELECT *
   FROM clients;
END$$

DELIMITER ;

-- ---------------------------------------------------------------------

-- Lesson 63: Parameters

USE sql_invoicing;

DELIMITER $$

CREATE PROCEDURE get_clients_by_state(state CHAR(2))
BEGIN
```

```
    SELECT *
    FROM clients c
    WHERE c.state = state;
END $$

DELIMITER ;

CALL sql_invoicing.get_clients_by_state('CA');

-- Exercise: write a stored procedure to return invoices for a given client
-- get_invoices_by_client
USE sql_invoicing;

DELIMITER $$

CREATE PROCEDURE get_invoices_by_client(client_id INT)
BEGIN
    SELECT * FROM invoices i WHERE i.client_id = client_id;
END $$

DELIMITER ;

CALL sql_invoicing.get_invoices_by_client(1);

-- ---------------------------------------------------------------------

-- Lesson 64: Parameters with default values

CALL sql_invoicing.get_clients_by_state(NULL);

-- Method1:
DELIMITER $$
CREATE PROCEDURE get_clients_by_state(state CHAR(2))
BEGIN
    IF state IS NULL THEN
        SET state = 'CA';
    END IF;
END $$
```

```
DELIMITER ;

-- Method 2:

DELIMITER $$

CREATE PROCEDURE get_clients_by_state(state CHAR(2))
BEGIN
   IF state IS NULL
   THEN
      SELECT * FROM clients;
   ELSE
      SELECT * FROM clients c WHERE c.state = state;
   END IF;
END $$

DELIMITER ;

-- Method 3:

DELIMITER $$
CREATE PROCEDURE get_clients_by_state(state CHAR(2))
BEGIN
   SELECT * FROM clients c WHERE c.state = IFNULL(state, c.state);
END $$

DELIMITER ;

-- Exercise: write a stored procedure called get_payments with 2 paramters
-- client_id: INT, payment_method_id: TINYINT
-- TINYINT -> 0-255(1 Byte), INT(4 Byte)

DELIMITER $$

CREATE PROCEDURE get_payments(client_id INT, payment_method_id TINYINT)
BEGIN
   SELECT * FROM payments p
      WHERE p.client_id = IFNULL(client_id, p.client_id)
   AND p.payment_method = IFNULL(payment_method_id, p.payment_method);
```

```
END $$

DELIMITER ;

CALL sql_invoicing.get_payments(1, NULL);

-- ------------------------------------------------------------------

-- Lesson 65: Parameter Validation

USE sql_invoicing;

DROP PROCEDURE IF EXISTS new_procedure;

DROP PROCEDURE IF EXISTS sql_invoicing.new_procedure;

DELIMITER $$

CREATE PROCEDURE new_procedure(invoice_id INT,
                               payment_amount DECIMAL(9, 2),
                               payment_date DATE)
BEGIN
   IF payment_amount >= 0 THEN
      UPDATE invoices i
      SET i.payment_total = payment_total
         AND i.payment_date = payment_date;
   ELSE SINGNAL SQLSTATE '22003'
      SET MESSAGE_TEXT = 'Invalid payment_amount';
   END IF;
END $$

DELIMITER ;

CALL sql_invoicing.new_procedure(2, 100, '2019-01-01');

-- ------------------------------------------------------------------

-- Lesson 66: Output Parameters
```

```
USE sql_invoicing;

DELIMITER $$
CREATE PROCEDURE get_unpaid(
   client_id INT,
   OUT invoices_count INT,
   OUT invoices_total DECIMAL(9, 2)
)
BEGIN
   SELECT COUNT(*),
         SUM(invoice_total)
   INTO invoices_count,
      invoices_total
   FROM invoices i
   WHERE i.client_id = client_id
     AND payment_total = 0;
END $$

DELIMITER ;

-- User defined variables
SET @invoices_count = 0;
SET @invoices_total = 0;
CALL sql_invoicing.get_unpaid(2, @invoices_count, @invoices_total);

SELECT @invoices_count, @invoices_total;

-- ------------------------------------------------------------------

-- Lesson 67: Variables

USE sql_invoicing;

DROP TABLE IF EXISTS get_risk_factor;

-- User or Session Variable
SET @invoices_count = 0;

DELIMITER $$
```

```
-- Local Variables
CREATE PROCEDURE get_risk_factor()
BEGIN
   DECLARE risk_factor DECIMAL(9, 2) DEFAULT 0;
   DECLARE invoices_total DECIMAL(9, 2);
   DECLARE invoices_count INT;
   SELECT COUNT(*),
        SUM(invoices_total)
   INTO invoices_count,
      invoices_total
   FROM invoices;
   -- Risk factor - ( invoices_total / invoices_count) * 5
   SET risk_factor = (invoices_total / invoices_count) * 5;
   SELECT risk_factor;
END $$

DELIMITER ;

CALL sql_invoicing.get_risk_factor();

-- -------------------------------------------------------------------------

-- Lesson 68: Functions

USE sql_invoicing;
DROP FUNCTION IF EXISTS get_risk_factor_for_client;

-- Functions attributes:
-- READ SQL DATA
-- DETERMINISTIC
-- MODIFIES SQL DATA

DELIMITER $$

CREATE FUNCTION get_risk_factor_for_client(client_id INT)
   RETURNS INTEGER
   READS SQL DATA
BEGIN
```

```
    DECLARE risk_factor DECIMAL(9, 2) DEFAULT 0;
    DECLARE invoices_total DECIMAL(9, 2);
    DECLARE invoices_count INT;
    SELECT COUNT(*), SUM(invoice_total)
    INTO invoices_count, invoices_total
    FROM invoices i
    WHERE i.client_id = client_id;

    -- risk_factor = ( invoices_total / invoices_count ) * 5
    SET risk_factor = (invoices_total / invoices_count) * 5;
    RETURN COALESCE(risk_factor, 0);
END $$

DELIMITER ;

SELECT client_id,
       name,
       get_risk_factor_for_client(client_id) AS risk_factor
FROM clients;

-- ---------------------------------------------------------------------

-- Lesson 69: Other Conventions
-- camel case -> camelCase
-- procGetRiskFactor
-- fnGetRiskFactor
-- getRiskFactor
-- DELIMITER $$ Or DELIMITER //

-- ---------------------- Triggers and Events --------------------------

-- Lesson 70: Triggers

-- Trigger: A block of sql code that automatically gets executed before or after an insert,
update or delete statement

-- AFTER, BEFORE
-- INSERT, UPDATE, DELETE
-- NEW, OLD
```

```
USE sql_invoicing;

SELECT *
FROM invoices;

-- Trigger
DROP TRIGGER IF EXISTS payments_after_insert;

DELIMITER $$

CREATE TRIGGER payments_after_insert
   AFTER INSERT
   ON payments
   FOR EACH ROW
BEGIN
   UPDATE invoices
   SET payment_total = payment_total + NEW.amount
   WHERE invoice_id = NEW.invoice_id;
END $$

DELIMITER ;

-- Test trigger

SELECT *
FROM sql_invoicing.invoices
WHERE invoice_id = 3;

INSERT INTO payments (payment_id,
                      client_id,
                      invoice_id,
                      date,
                      amount,
                      payment_method)
   VALUE (DEFAULT, 5, 3, CURDATE(), 25, 1);

SELECT *
FROM sql_invoicing.invoices
```

```
WHERE invoice_id = 3;

-- Exercise: create a trigger that gets fired when we delete a payment

USE sql_invoicing;

DROP TRIGGER IF EXISTS payments_after_delete;

DELIMITER $$

CREATE TRIGGER payments_after_delete
   AFTER DELETE
   ON payments
   FOR EACH ROW
BEGIN
   UPDATE invoices
   SET payment_total = payment_total - OLD.amount
   WHERE invoice_id = OLD.invoice_id;
END $$

DELIMITER ;

USE sql_invoicing;

SELECT *
FROM invoices
WHERE invoice_id = 3;

DELETE
FROM payments
WHERE payment_id = 10;

SELECT *
FROM invoices
WHERE invoice_id = 3;

-- ---------------------------------------------------------------------

-- Lesson 71: Viewing Triggers
```

```
USE sql_invoicing;

SHOW TRIGGERS;

SHOW TRIGGERS LIKE 'payments%';

SHOW TRIGGERS WHERE EVENT = 'INSERT';

-- ------------------------------------------------------------------------

-- Lesson 72: Dropping Triggers

DROP TRIGGER IF EXISTS payments_after_insert;

-- ------------------------------------------------------------------------

-- Lesson 73: Using Triggers For Auditing

-- For Logging the changes
USE sql_invoicing;

CREATE TABLE payments_audit
(
   client_id   INT          NOT NULL,
   date        DATE         NOT NULL,
   amount      DECIMAL(9, 2) NOT NULL,
   action_type VARCHAR(50)   NOT NULL,
   action_date DATETIME      NOT NULL
);

-- First Trigger
DROP TRIGGER IF EXISTS payments_after_insert;

DELIMITER $$

CREATE TRIGGER payments_after_insert
   AFTER INSERT
   ON payments
```

```
    FOR EACH ROW
BEGIN
    UPDATE invoices
    SET payment_total = payment_total + NEW.amount
    WHERE invoice_id = NEW.invoice_id;
    INSERT INTO payments_audit
    VALUES (NEW.client_id, NEW.date,
            NEW.amount,
            'INSERT', NOW());
END $$

DELIMITER ;

-- Second Trigger
DROP TRIGGER IF EXISTS payments_after_delete;

DELIMITER $$

CREATE TRIGGER payments_after_delete
    AFTER DELETE
    ON payments
    FOR EACH ROW
BEGIN
    UPDATE invoices
    SET payment_total = payment_total - OLD.amount
    WHERE invoice_id = OLD.invoice_id;
    INSERT INTO payments_audit
    VALUES (OLD.client_id,
            OLD.date,
            OLD.amount,
            'DELETE',
            NOW());
END $$

DELIMITER ;

SELECT *
FROM invoices
WHERE client_id = 3;
```

```
INSERT INTO payments (payment_id, client_id, invoice_id, date, amount,
payment_method)
   VALUE (DEFAULT, 5, 3, CURDATE(), 100, 1);

SELECT *
FROM payments;

SELECT *
FROM invoices
WHERE invoice_id = 3;

DELETE
FROM payments
WHERE payment_id = 12;

SELECT *
FROM invoices
WHERE invoice_id = 3;

SELECT *
FROM payments_audit;

-- ---------------------------------------------------------------------

-- Lesson 74: Events

-- Event: A task (or block of sql code) that gets executed according to a schedule

SHOW VARIABLES LIKE 'event%';

SET GLOBAL EVENT_SCHEDULER = ON;
SHOW VARIABLES LIKE 'event%';

SET GLOBAL EVENT_SCHEDULER = OFF;
SHOW VARIABLES LIKE 'event%';

DELIMITER $$
```

```
-- ONCE
-- AT '2020-01-01'
-- REGULAR BASES: EVER 1 YEAR STARTS '2020-01-01' END '2021-01-01'
-- STARTS and ENDS are optional
-- YEAR, MONTH, WEEK, DAY, HOUR, SECOND
CREATE EVENT yearly_delete_stale_audit_rows
   ON SCHEDULE EVERY 1 YEAR STARTS '2021-01-01' ENDS '2022-01-01'
   DO BEGIN
   DELETE FROM payments_audit WHERE action_date < NOW() - INTERVAL 1 YEAR;
END $$

DELIMITER ;

-- ------------------------------------------------------------------------

-- Lesson 75: Viewing, Dropping And Altering Events

-- Show
SHOW EVENTS;

-- Drop
DROP EVENT IF EXISTS yearly_delete_stale_audit_rows;

-- Alter
ALTER EVENT yearly_delete_stale_audit_rows
   ON SCHEDULE
      EVERY 1 YEAR
   DO BEGIN
   DELETE FROM payments_audit WHERE action_date < NOW() - INTERVAL 1 YEAR ;
END;

DELIMITER ;

-- Disable or Enable

ALTER EVENT yearly_delete_stale_audit_rows DISABLE;
ALTER EVENT yearly_delete_stale_audit_rows ENABLE;

-- ------------------------------------------------------------------------
```

```
-- Lesson 76: Transactions

-- transaction: a group of SQL statement that represent a single unit of work
-- Properties: ACID
-- 1- Atomicity: we can not break these atoms
-- 2- Consistency
-- 3- Isolation
-- 4- Durability

-- ---------------------------------------------------------------------

-- Lesson 77: Creating Transactions

USE sql_store;

START TRANSACTION;

INSERT INTO orders (customer_id, order_date, status)
VALUES (1, CURDATE(), 1);

INSERT INTO order_items
VALUES (LAST_INSERT_ID(), 1, 1, 1);

COMMIT;

-- to undo changes and rollback
USE sql_store;

START TRANSACTION;

INSERT INTO orders (customer_id, order_date, status)
VALUES (1, CURDATE(), 1);

INSERT INTO order_items
VALUES (LAST_INSERT_ID(), 1, 1, 1);

ROLLBACK;
```

```
-- auto commit: whenever we execute a single statement MYSQL put that statement in a transaction and commit it if that statement doesn't raise an error
SHOW VARIABLES LIKE 'autocommit';

-- ---------------------------------------------------------------------

-- Lesson 78: Concurrency and locking

-- Concurrency: two or more users may try to access the same data at the same time
-- if your transaction tries to modify a row or multiple rows, it puts a lock on these rows, and this lock prevents other transactions from modifying these rows
-- until the first transaction is done or committed or rolled back.
-- So we have not to worry about concurrency most of the times

-- Connection 1
USE sql_store;
SELECT *
FROM customers
WHERE customer_id = 1;

START TRANSACTION;
UPDATE customers
SET points = points + 10
WHERE customer_id = 1; -- Run line by line until this line

COMMIT;

SELECT *
FROM customers
WHERE customer_id = 1;

-- Connection 2
USE sql_store;
START TRANSACTION;

UPDATE customers
SET points = points + 10
WHERE customer_id = 1; -- Run line by line until this line: stuck in updating state until changes in first connection commit then it will affected
```

COMMIT;

-- --

-- Lesson 79: Concurrency problems

-- 1- Lost updates: repeatable read isolation solve this problem
-- 2- Dirty reads: Read Committed Isolation solve this problem
-- 3- Non repeating reads: Repeatable reads isolation solve this problem
-- 4- Phantom reads : Serializable isolation solve this problem(Highest level of isolation) --> can hurt performance

-- --

-- Lesson 80: Transaction Isolation Levels

-- Lost update : 2 transaction update the same row and the one that commits last overwrite the changes made earlier
-- Dirty read: If you read uncommitted data
-- Non repeating reads: if you read the same data twice in the transaction but get different result
-- Phantom reads: which happen when we miss one or more rows in our query because another transaction is changing the data and we are not aware of the changes in our transaction

-- Isolation levels:
-- 1- Read uncommitted: our transactions are not isolated from each other and they can read uncommitted changes by each other => don't protect us from any Concurrency problems
-- 2- Read Committed: Only read committed data and this prevents Dirty reads
-- 3- Repeatable Read: At this level we can be confident that multiple reads returns the same result even if data gets changed in the mean time => lost update, dirty reads, non repeating reads
-- 4- Serializable: if data is getting changed in the mean time , our transaction will wait to get the most recent data so this puts overhead on the server => lost update, dirty reads, non repeating reads, phantom reads

-- The more we increase the isolation level the more performance and scalability problems => because of more locks

```
-- lower isolation level gives us more concurrency problems so more users can access the same data at the same time
-- In My SQL the default isolation level is Repeatable read

-- See the default isolation level
SHOW VARIABLES LIKE '%isolation%';
-- Only next transaction:
-- SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- change all future transactions for this session = Connect to data base in applications exp: python
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- change globally for all new sessions:
-- SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED;
SHOW VARIABLES LIKE '%isolation%';

-- ------------------------------------------------------------------------

-- Lesson 81: READ UNCOMMITTED Isolation Level

-- Session 1
SHOW VARIABLES LIKE '%isolation%';
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
USE sql_store;

SELECT points
FROM customers
WHERE customer_id = 1;
-- run until this line after update

-- Session 2
USE sql_store;
START TRANSACTION;

UPDATE customers
SET points = 20
WHERE customer_id = 1; -- run until this line

ROLLBACK;
-- COMMIT;
```

```
-- ---------------------------------------------------------------------

-- Lesson 82: READ COMMITTED isolation level

-- Session 1
SHOW VARIABLES LIKE '%isolation%';
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
USE sql_store;

SELECT points
FROM customers
WHERE customer_id = 1;
-- run until this line after update

-- Session 2
USE sql_store;
START TRANSACTION;

UPDATE customers
SET points = 20
WHERE customer_id = 1; -- run until this line

ROLLBACK;
-- COMMIT;

-- Problem of Repeatable read
-- Session 1
SHOW VARIABLES LIKE '%isolation%';
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
USE sql_store;
START TRANSACTION;

SELECT points
FROM customers
WHERE customer_id = 1; -- run until this line after update

SELECT points
FROM customers
```

```
WHERE customer_id = 1;

COMMIT;

-- Session 2
USE sql_store;
START TRANSACTION;

UPDATE customers
SET points = 1920
WHERE customer_id = 1;

COMMIT;

-- ------------------------------------------------------------------------

-- Lesson 83: REPEATABLE READ Isolation Level

-- Solve repeatable read

-- Session 1
SHOW VARIABLES LIKE '%isolation%';
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
USE sql_store;
START TRANSACTION;

SELECT points
FROM customers
WHERE customer_id = 1; -- run until this line after update

SELECT points
FROM customers
WHERE customer_id = 1;

COMMIT;

-- Session 2
USE sql_store;
START TRANSACTION;
```

```
UPDATE customers
SET points = 2000
WHERE customer_id = 1; -- run until this line

COMMIT;

-- Phantom read problem
-- Session 1 : if we run again after commit new customer will show but because of REPEATABLE READ we should not have inconsistent data in one transaction
-- So new customer will not show
SHOW VARIABLES LIKE '%isolation%';
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
USE sql_store;
START TRANSACTION;

SELECT *
FROM customers
WHERE state = 'VA'; -- run until this line after update

COMMIT;

-- Session 2
USE sql_store;
START TRANSACTION;

UPDATE customers
SET state = 'VA'
WHERE customer_id = 1; -- run until this line

COMMIT;

-- ------------------------------------------------------------------------

-- Lesson 84: SERIALIZABLE Isolation Level
-- SERIALIZABLE: transactions run in sequence one after another so we don't have any concurrency
-- Session 1
SHOW VARIABLES LIKE '%isolation%';
```

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
USE sql_store;
START TRANSACTION;

SELECT *
FROM customers
WHERE state = 'VA'; -- run until this line after update

COMMIT;

-- Session 2
USE sql_store;
START TRANSACTION;

UPDATE customers
SET state = 'VA'
WHERE customer_id = 3; -- run until this line

COMMIT;

-- ---------------------------------------------------------------------

-- Lesson 85: Deadlocks
-- Deadlock : deadlock happen when different transactions cannot complete because each transaction holds a lock that the other needs , so both transactions keep waiting for each other and never release their lock

-- Session 1
USE sql_store;
START TRANSACTION;

UPDATE customers
SET state = 'VA'
WHERE customer_id = 1; -- Until here => this line have lock

UPDATE orders
SET status = 1
WHERE order_id = 1;
```

```
COMMIT;

-- Session 2
USE sql_store;
START TRANSACTION;

UPDATE orders
SET status = 1
WHERE order_id = 1; -- Until here => this line have lock

UPDATE customers
SET state = 'VA'
WHERE customer_id = 1;

COMMIT;

-- If deadlock happens my sql treats this transaction(The transaction that runs at last) as victim and rolls it back
-- minimize deadlocks:
-- 1) If 2 transactions records in reverse order it's likely you are going to have a deadlock
-- 2) keep your transaction small and shor duration so they are not likely to collide with other transactions

-- ---------- Data Types ----------

-- Lesson 86: Introduction
-- Data types:
-- 1) String types
-- 2) Numeric types
-- 3) Date and Time types
-- 4) Blob types: For storing binary(0, 1) data
-- 5) Special types: for storing geographical values

-- ------------------------------------------------------------

-- Lesson 87: String Types

-- 1) CHAR(x)  -> Fix length strings => like state
-- 2) VARCHAR(x) -> variable length => like email, username, password, addresses
```

-- We can use string types to store numeric values like zip code and phone numbers because we don't use these values in mathematical operations and some times these values may contain heyphen(-) or parenthesis (( )) to group a few digits

-- VARCHAR(50) => short strings
-- VARCHAR(255) => medium strings
-- max length for varchar is 64 kb(65535 characters = 2 ^ 16)
-- 3) MEDIUMTEXT => max to 16 MB => json objects or csv strings => 2 ^ 24
-- 4) LONGTEXT => max to 4 GB => storing text book or years of log files => 2 ^ 32
-- 5) TINYTEXT => max 255 character => 2 ^ 8
-- 6) TEXT => max to 64 KB => 2 ^ 16

-- English letter use => 1 byte
-- European Middle-eastern => 2 bytes
-- Asian languages => 3 bytes
-- If the type of a column is char of 10 my sql will reserve 30 bytes for the values in this column

-- ---

-- Lesson 88: Integer Types

-- Integer : Don't have decimal points

-- 1) TINYINT => 1 byte => [-128, 127]
-- 2) UNSIGNED TINYINT=> 1 byte => [0, 255]
-- 3) SMALLINT => 2 bytes => [-32k, 32k]
-- 4) MEDIUMINT => 3 bytes => [-8m, 8m]
-- 5) INT => 4 bytes => [-2b, 2b]
-- 6) BIGINT => 8 bytes => [-9z, 9z]

-- zerofill => int(4) => 0001
-- use the smallest data type that suits your needs

-- ---

-- Lesson 89: Fixed-point and Floating-point Types

-- 1) DECIMAL(p, s) or DEC(p, s) or NUMERIC(p, s) or FIXED(p, s) => monetary values
-- p:precision s:scale example: decimal(4, 3) 1234.123
-- 2) DOUBLE => 8 bytes => work with very large or very small numbers (scientific calculations) and precision is is not important
-- 3) FLOAT => 4 bytes => work with very large or very small numbers (scientific calculations) and precision is is not important

-- ---

-- Lesson 90: Boolean Types

-- 1) BOOL or BOOLEAN Exp: UPDATE posts SET is_published = TRUE(1) or FALSE(0) WHERE condition

-- ---

-- Lesson 91: Enums and Set Types

-- ENUM('x', 'y', 'z') => Example: ENUM('small' , 'medium', 'large')
-- only values we can set for our columns is small, medium and large
-- Enum are generally bad and we should avoid them because changing the member of enum can be expensive And enums are not reusable so if you have another table and you want to reuse these values here , in that table we need to redefine that enum

-- SET() => store multiple values in set

-- ---

-- Lesson 91: Date and Time Types

-- 1) DATE
-- 2) TIME
-- 3) DATETIME => 8 bytes
-- 4) TIMESTAMP => 4 bytes => Problem: we can not store dates after year 2038 because of lack of size(4 bytes)
-- 5) YEAR

-- ---

```
-- Lesson 92: Blob Types
-- To store large amounts of binary data like images, video, pdfs, files, .... , we use blobs
-- TINYBLOB => 255B
-- BLOB => 65 KB
-- MEDIUMBLOB => 16 MB
-- LONGBLOB => 4 GB
-- relational databases like sql are not designed for nonstructural data like images, videos and files
-- Problems:
-- 1) increase database size
-- 2) slower backups
-- 3) performance problem
-- 4) more code to read/write images
-- 5) ...

-- ---------------------------------------------------------------------

-- Lesson 93: Json Type
-- JSON(Javascript Object Notation): lightweight format for storing and transferring data over the Internet
-- Json : { "key": value}

-- Insert JSON
USE sql_store;
-- First way
UPDATE products
SET properties = '
{
        "dimensions": [1, 2, 3],
   "weight": 10,
   "manufacturer": {"name": "sony"}
}'
WHERE product_id = 1;

-- Second way
UPDATE products
SET properties = JSON_OBJECT(
       "weight", 10
   , "dimensions", JSON_ARRAY(4, 5, 6)
```

```
   , "manufacturer", JSON_OBJECT("name", "Google")
   )
WHERE product_id = 2;

SELECT *
FROM products;

-- Extract JSON
-- $ => entire object            . => each key
USE sql_store;
-- First way

SELECT product_id, JSON_EXTRACT(properties, "$.weight")
FROM products; -- JSON_EXTRACT(Object name, "$.key name")

SELECT product_id, JSON_EXTRACT(properties, "$.dimensions[0]")
FROM products;

SELECT product_id, JSON_EXTRACT(properties, "$.manufacturer.name")
FROM products;

-- Second way
SELECT product_id, properties -> "$.manufacturer"
FROM products;

SELECT product_id, properties ->> "$.manufacturer.name"
FROM products
WHERE properties ->> "$.manufacturer.name" = 'sony';

-- Update JSON
UPDATE products
SET properties = JSON_SET(properties, "$.weight", 20, "$.age", 100)
WHERE product_id = 1;

SELECT *
FROM products;

-- Remove JSON
UPDATE products
```

```
SET properties = JSON_REMOVE(properties, "$.age")
WHERE product_id = 1;

SELECT *
FROM products;

-- ---------------------- Designing Databases ---------------------------

-- Lesson 94:  Introduction

-- Design a database from scratch or add new tables to existing database
-- Design a database is very important step
-- Poor design database needs a lot of maintenance and cost of maintaining increases over time

-- ------------------------------------------------------------------------

-- Lesson 95: Data Modeling
-- The process of creating a model for data we want to store in our database it involves 4 steps:
-- 1) Understand the requirements(Most important step)          2) Build a conceptual model( identifying the things, entities in the business and their relationship with each other- A visual representation of these concepts that we use to communicate with stakeholders )          3) Build a logical model( an abstract data model that is independent of database technology, it just show you the tables and columns you need)          4) Build a Physical model (implementation of logical model for a particular database technology) => supported data types, the default values, primary key, views, stored procedure

-- ------------------------------------------------------------------------

-- Lesson 96: Conceptual Models

-- Represent entities and their relations
-- 2 way to build conceptual models:          1) Entity Relationship(ER) **          2) UML(Unified Modeling Language ) diagrams
-- Tools for creating entity relationships:        1) Microsoft Visio          2) draw.io **          3) Lucid charts
```

-- --

-- Lesson 97: Logical Models

-- It's better to break attributes like name to first_name and last_name to speed up the queries faster and simpler
-- Relationship types: 1) One to One 2)One to Many 3) Many to Many

-- --

-- Lesson 98: Physical Models

-- Plural or Singular name for tables and follow the conventions that used earlier or if you want to design new database stick to convention that you use first

-- --

-- Lesson 99: Primary Key

-- A primary key is a column that uniquely identifies each record within table
-- Composite primary key : combination of multiple columns to create a unique column
-- It's better to use a column as primary key with less characters (not use email) And it should not change (a student may change his/her email) in this situations we introduce a new column as primary key which we call id column

-- --

-- Lesson 100: Foreign Key

-- Whenever we add a relationship between 2 tables, one end of relationship is called parent or primary key table and the other end is called the child or foreign key table
-- for figure out which table is parent and which one is child => child can not exist without parent (enrollment can not exist without student)
-- A foreign key is a column in one table that references the primary key of another table
-- benefit of composite primary key (student_id, course_id): it prevents us from accidentally enrolling the same student in the same course twice (in case: primary key = enrollment_id)

-- Problem: if we have another table in the future and there is going to be a relationship between the enrollment and that new table , these 2 keys need to be repeated in that table as the foreign key (enrollment table be parent and new table is child) if we use primary key = enrollment_id , we would just use enrollment_id as foreign key in that table

-- ---

-- Lesson 101: Foreign Key Constraints

-- that basically protects your data from getting corrupted
-- fk_child_parent
-- on update => 1)cascade :automatically change the records in the child table if the primary key changes
-- 2) restrict: reject the update from happening
-- 3) set null: if the primary key update in parent table , set the foreign key to null in the child table and with this we'll end up with a child record that doesn't have parent we call this orphan record(bad data)
-- 4) No action: same as restrict
-- on delete: 1) cascade 2) restrict 3) set null 4) no action
-- We don't change or delete primary key of table usually that's considered a bad practice

-- ---

-- Lesson 102: Normalization

-- our design should be optimal and doesn't allow redundant data and duplicated data because redundancy increase the size of our database and complicates the insert, update and delete operations
-- example if we have a redundant name in several places , we have to update several different places otherwise we have inconsistent data

-- Normalization: the process of reviewing our design and makes sure it follows a few predefined rules that prevents data duplication
-- 7 Rules (7 Normal forms): 1)1st normal form(1NF) 2) 2nd normal form (2NF) 3) 3rd normal form (3NF)

-- ---

-- Lesson 103: First Normal Form(1NF)

-- Each cell should have a single value and we can not have repeated columns
-- tags violated this rule
-- solution : have a separate tags table

-- ---

-- Lesson 104: Link Tables

-- in relational databases , we don't have many to many relationship. we only have 1 to 1 and 1 to many relationships so to implement many to many relationship between 2 tables you need to introduce a new table which we call a link table and we gonna have 2 , 1 to many relationship in that table like enrollment table

-- ---

-- Lesson 105: Second Normal Form (2NF)

-- 1) be in 1NF 2) Not have any non-prime attribute that is dependent on any proper subset of any candidate key of the relation
-- Non-prim attribute of a relation: is an attribute that is not a part of any candidate key of the relation
-- SImple definition: Every table should describe on entity , and every column in that table should describe that table
-- Example: in orders table we should not have customer_name
-- Instructor => violated 2NF

-- ---

-- Lesson 106: Third Normal Form (3NF)

-- 1) be in 2NF 2) All the attributes in a table are determined only by the candidate key of that relation and not by any non-prime attributes

-- Simple Definition: A column in a table should not be derived from other columns

-- Example: we can dervie a column by operation on another columns we have invoice_total , payment_total and balance
-- balance = invoice_total - payment_total so what if we change value of invoice_total or payment_total So we should again update balance(what if we forgot to update that => inconsistent data in our database)
-- Example 2: first_name , last_name , full_name = first_name + last_name

-- ---

-- Lesson 107: My Pragmatic Advice

-- Don't worry about memorizing these normalization rules

-- Just focus on removing redundancy not applying normalization rules to each table and each column

-- Whenever you see duplicated value and these values are not foreign keys like 1, 2, 3, 4 that means your design is not normalized, what normalization form it violates doesn't really matter

-- Don't apply normalization rules blindly. always take the requirements into the account

-- ---

-- Lesson 108: Don't Model the Universe

-- 6th normal form: build a relational database on top of a relational database

-- Important: solve today's problems, not future problems that may never happen
-- build a model for you problem domain , not the real world

-- Simplicity is the ultimate sophistication

-- ---

-- Lesson 109: Forward Engineering a Model

-- To generate scripts of our physical design

-- ---

-- Lesson 110: Synchronizing a Model with a Database

-- to make any changes: if we use database for our personal usage we can make changes easily

-- Production environment: where our users access our application or databases
-- Staging environment: which is close to production environment
-- Testing environment: That is used purely for testing
-- Development environment
-- each environment has one or more severs, so any time we developers want to make any changes to these databases, we should be able to replicate the same changes on other databases, so all these databases that we have in various environments are consistent
-- If we use database in large companies we should synchronize our database:
-- instead of forward engineer , we will use synchronize model
-- we use forward engineer when we don't have database => to generate new database

-- ---

-- Lesson 111: Reverse Engineering a Database

-- To change a database that doesn't have a model we need to use Reverse engineering like sql_store
-- It's better to have separate model for each database
-- we can include multiple databases in one model but that's something we want to do if these databases are highly related and we want to work with them in the same model

-- ---

-- Lesson 112: Creating and Dropping Databases

CREATE DATABASE IF NOT EXISTS sql_store_2;

DROP DATABASE IF EXISTS sql_store_2;

-- ---

```
-- Lesson 113: Creating Tables

CREATE DATABASE IF NOT EXISTS sql_store2;
USE sql_store2;

DROP TABLE IF EXISTS customers;
CREATE TABLE IF NOT EXISTS customers
(
   customer_id INT PRIMARY KEY AUTO_INCREMENT,
   first_name  VARCHAR(50)  NOT NULL,
   last_name   VARCHAR(50)  NOT NULL,
   birth_date  DATE,
   phone       VARCHAR(50)  NOT NULL,
   address     VARCHAR(255) NOT NULL,
   city        VARCHAR(50),
   state       CHAR(2),
   points      INT(11)      NOT NULL DEFAULT 0,
   email       VARCHAR(255) NOT NULL UNIQUE
);

-- ---------------------------------------------------------------------

-- Lesson 114: Altering Tables

USE sql_store2;

-- ADD COLUMN name = ADD name
ALTER TABLE customers
   ADD full_name VARCHAR(255) NOT NULL AFTER last_name,
   MODIFY COLUMN last_name VARCHAR(50) DEFAULT '',
   DROP full_name;

-- ---------------------------------------------------------------------

-- Lesson 115: Creating Relationships

USE sql_store2;
```

```
-- we can not drop customer table because it's part of a relationship with orders and
orders depend on customers so we should first drop orders table
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS customers;

CREATE TABLE IF NOT EXISTS customers
(
   customer_id INT PRIMARY KEY AUTO_INCREMENT,
   first_name  VARCHAR(50)  NOT NULL,
   last_name   VARCHAR(50)  NOT NULL,
   birth_date  DATE,
   phone       VARCHAR(50)  NOT NULL,
   address     VARCHAR(255) NOT NULL,
   city        VARCHAR(50),
   state       CHAR(2),
   points      INT(11)      NOT NULL DEFAULT 0,
   email       VARCHAR(255) NOT NULL UNIQUE
);

CREATE TABLE orders
(
   order_id    INT PRIMARY KEY,
   customer_id INT NOT NULL,
   FOREIGN KEY fk_orders_customers (customer_id)
      REFERENCES customers (customer_id)
      ON UPDATE CASCADE
      ON DELETE RESTRICT
   -- SET NULL                 NO ACTION           RESTRICT
);

-- ---------------------------------------------------------------------

-- Lesson 116: Altering Primary and Foreign Key Constraints

USE sql_store2;
ALTER TABLE orders
   -- DROP PK
   DROP PRIMARY KEY,
   -- ADD PK => ADD PRIMARY KEY (order_id)
```

```
   ADD PRIMARY KEY (order_id, customer_id),
   -- DROP FOREIGN KEY
   DROP FOREIGN KEY fk_orders_customers,
   -- ADD FOREIGN KEY
   ADD FOREIGN KEY fk_orders_customers (customer_id)
      REFERENCES customer (customer_id)
      ON UPDATE CASCADE
      ON DELETE RESTRICT;

-- ------------------------------------------------------------------

-- Lesson 117: Character Sets and Collations

-- Default character set is UTF-8 and default collations is utf8_general_ci        ci: case insensitive
-- MaxLen : means mysql reserve maximum 3(default in utf8) bytes for storing each character
-- Example : CHAR(10) -> 10 * 3 = 30 bytes for each cell

SHOW CHARSET;
-- create database with different character set
CREATE DATABASE IF NOT EXISTS db_name
   CHARACTER SET latin1;

-- change database character set
ALTER DATABASE db_name
   CHARACTER SET latin1;
-- create table
CREATE TABLE t_name
(
   order_id    INT PRIMARY KEY,
   customer_id INT NOT NULL
)
   CHARACTER SET latin1;

-- alter table
ALTER TABLE t_name
   CHARACTER SET latin1;
```

```
-- character set for specific column
CREATE TABLE IF NOT EXISTS customers
(
   customer_id INT PRIMARY KEY AUTO_INCREMENT,
   first_name  VARCHAR(50) CHARACTER SET latin1 NOT NULL,
   last_name   VARCHAR(50) CHARACTER SET latin1 NOT NULL,
   birth_date  DATE,
   phone       VARCHAR(50) CHARACTER SET latin1 NOT NULL,
   address     VARCHAR(255)                NOT NULL,
   city        VARCHAR(50),
   state       CHAR(2),
   points      INT(11)                  NOT NULL DEFAULT 0,
   email       VARCHAR(255)                NOT NULL UNIQUE
);

-- ------------------------------------------------------------------------

-- Lesson 118: Storage Engines

-- in mysql we have several storage engine
-- storage engine determine how the data is stored and what features are available for
us
SHOW ENGINES;
-- innodb is the default

ALTER TABLE customers
   ENGINE = InnoDB;
-- changing the engine of table is expensive because mysql should rebuild the whole
table

-- ---------- Indexing For High Performance ----------

-- Lesson 119: introduction

-- Indexes speed up our queries

-- ------------------------------------------------------------------------

-- Lesson 120: Indexes
```

-- Indexes are data structure that database engine use to quickly find the data as an analogy
-- in a lot of cases indexes are small enough that they can fit into memory that's why it's much faster to use them to find data because reading data from memory is always faster than bringing it from the disk so indexes help us to find data quickly but they come with cost
-- 1) increase the size of database because have to be permanent store next to our tables
-- 2) every time we add, update or delete a record , mysql has to update the corresponding indexes and this will impact the performance of our write operations
-- So we should reserve indexes for performance critical queries
-- Design indexes based on your queries, Not your tables

-- ---

-- Lesson 121: Creating Indexes

```
USE sql_store;

SELECT customer_id
FROM customers
WHERE state = 'CA';

EXPLAIN
SELECT customer_id
FROM customers
WHERE state = 'CA';

SELECT COUNT(*)
FROM customers;

CREATE INDEX idx_state ON customers (state);

EXPLAIN
SELECT customer_id
FROM customers
WHERE state = 'CA';
```

```
-- Exercise: write a query to find customers with more than 1000 points.
EXPLAIN
SELECT points
FROM customers
WHERE points > 1000;

CREATE INDEX idx_points ON customers (points);

EXPLAIN
SELECT points
FROM customers
WHERE points > 1000;

-- ---------------------------------------------------------------------

-- Lesson 122: Viewing Indexes

SHOW INDEXES IN customers;

ANALYZE TABLE customers;
SHOW INDEXES IN customers;

SHOW INDEXES IN orders;

-- ---------------------------------------------------------------------

-- Lesson 123: Prefix Indexes

-- String columns: 1) char      2) varchar      3) text     4) blob => consume a lot of space
and can not fit into memory so we will use prefix of column so our index will be smaller

CREATE INDEX idx_lastname ON customers (last_name(20));

-- it's optional for char and varchar but it's compulsory for blob and text
SELECT COUNT(DISTINCT (LEFT(last_name, 1))),
      COUNT(DISTINCT (LEFT(last_name, 5))),
      COUNT(DISTINCT (LEFT(last_name, 10)))
FROM customers;
```

```
-- ---------------------------------------------------------------------

-- Lesson 124: Full-text Indexes

-- To build fast and flexible search engine
USE sql_blog;
-- Only this exact order: react redux
SELECT *
FROM posts
WHERE title LIKE "%react redux%"
  OR body LIKE "%react reudx%";

CREATE FULLTEXT INDEX idx_title_body ON posts (title, body);

-- fulltext index built-in functions
-- Natural language mode
SELECT *, MATCH(title, body) AGAINST('react redux')
FROM posts
WHERE MATCH(title, body) AGAINST('react redux');
-- Boolean mode
SELECT *, MATCH(title, body) AGAINST('react -redux +form') -- react not redux and must
have form
FROM posts
WHERE MATCH(title, body) AGAINST('react redux' IN BOOLEAN MODE);

-- ---------------------------------------------------------------------

-- Lesson 125: Composite Indexes

-- Only we can use at max 1 index key and for second condition we should use table
search
-- max composite column = 16
USE sql_store;
SHOW INDEXES IN customers;

EXPLAIN
SELECT customer_id
FROM customers
WHERE state = 'CA'
```

```
  AND points > 1000;

-- Composite index
DROP INDEX idx_points ON customers;
DROP INDEX idx_state ON customers;
CREATE INDEX idx_state_points ON customers (state, points);
EXPLAIN
SELECT customer_id
FROM customers
WHERE state = 'CA'
  AND points > 1000;

-- ------------------------------------------------------------------------

-- Lesson 126: Order of Columns in Composite Indexes

-- Rules: 1) Put the most frequently used column first      2) Put the columns with the
higher cardinality first => to narrow down the search
-- Cardinality : number of unique value in the index
SHOW INDEXES IN customers;
ANALYZE TABLE customers;

-- Check the cardinality
SELECT COUNT(DISTINCT (state)),
       COUNT(DISTINCT (last_name))
FROM customers;

CREATE INDEX idx_lastname_state ON customers (last_name, state);
EXPLAIN
SELECT customer_id
FROM customers
WHERE state = 'CA'
  AND last_name LIKE 'a%';

-- = 'CA' is more restrictive than LIKE 'a%'
-- Force mysql to use specific index
CREATE INDEX idx_state_lastname ON customers (state, last_name);

EXPLAIN
```

```
SELECT customer_id
FROM customers
	USE INDEX (idx_lastname_state)
WHERE state = 'CA'
  AND last_name LIKE 'a%';

-- Use for another query
EXPLAIN
SELECT customer_id
FROM customers
	USE INDEX (idx_lastname_state)
WHERE state LIKE 'A%'
  AND last_name LIKE 'a%';

EXPLAIN
SELECT customer_id
FROM customers
	USE INDEX (idx_state_lastname)
WHERE state LIKE 'A%'
  AND last_name LIKE 'a%';

EXPLAIN
SELECT customer_id
FROM customers
	USE INDEX (idx_lastname_state)
WHERE last_name LIKE 'a%';

-- ---------------------------------------------------------------------

-- Lesson 127: Using Indexes For Sorting

EXPLAIN
SELECT customer_id
FROM customers
ORDER BY last_name;
EXPLAIN
SELECT customer_id
FROM customers
ORDER BY state, last_name DESC;
```

```
EXPLAIN
SELECT customer_id
FROM customers
ORDER BY state DESC, last_name DESC;
-- EXPLAIN SELECT customer_id FROM customers ORDER BY first_name;
-- SELECT customer_id FROM customers ORDER BY first_name;
-- SELECT customer_id FROM customers ORDER BY state;
SHOW STATUS LIKE '%last%';

-- (a,b) =>    1)  a        2) a, b      3) a DESC, b DESC

-- ------------------------------------------------------------------------

-- Lesson 128: Covering Indexes
-- Full table scan
EXPLAIN
SELECT *
FROM customers
ORDER BY state;
-- Uses indexes => pk + composite index
EXPLAIN
SELECT customer_id, state, last_name
FROM customers
ORDER BY state;

-- ------------------------------------------------------------------------

-- Lesson 129: Index Maintenance

-- Duplicate Indexes: Indexes on the same set of columns in the same order like abc, abc
-- Redundant indexes: If you have an index on 2 columns like (a , b)  and then create
another index on column a
-- (a, b) => 1) b          2) (b, a) is not redundant

-- Before creating new indexes, check the existing ones.

-- ------------------- Securing Databases -------------------------------

-- Lesson 130: Creating User
```

```
CREATE USER ali@127.0.0.1 IDENTIFIED BY 'Aliazani1378';
CREATE USER ali@localhost IDENTIFIED BY 'Aliazani1378';
CREATE USER ali@aliazani.com IDENTIFIED BY 'Aliazani1378';
-- Use subdomains
CREATE USER ali@'%.aliazani.com'IDENTIFIED BY 'Aliazani1378';

-- ------------------------------------------------------------------------

-- Lesson 131: Viewing Users

SELECT * FROM mysql.user;

-- ------------------------------------------------------------------------

-- Lesson 132: Dropping User

DROP USER ali@127.0.0.1;
SELECT * FROM mysql.user;

-- ------------------------------------------------------------------------

-- Lesson 133: Changing the password

CREATE USER ali IDENTIFIED BY '1234';
SELECT * FROM mysql.user;
SET PASSWORD FOR ali = 'Aliazani1234';

-- ------------------------------------------------------------------------

-- Lesson 134: Viewing Privileges

 SHOW GRANTS FOR ali;
 SHOW GRANTS;

-- ------------------------------------------------------------------------

-- Lesson 135: Revoking Privileges
```

```
GRANT CREATE VIEW ON sql_store.* TO ali;
SHOW GRANTS FOR ali;
REVOKE CREATE VIEW ON sql_store.* FROM ali;
SHOW GRANTS FOR ali;

-- ---------------------------------------------------------------------
```