SQL Task

Task:

Create a database and tables to manage a simple e-commerce system.

The system should have three tables: customers, orders, and products.

Requirements:

- Create a database named ecommerce.
- Create three tables: customers, orders, and products.
- Insert some sample data into the tables.

Table Structure:

customers

id (primary key, auto-increment): unique identifier for each customer
name: customer's name
email: customer's email address
address: customer's address

orders

id (primary key, auto-increment): unique identifier for each order
customer_id (foreign key referencing customers.id): a customer who placed the order
order_date: date the order was placed
total_amount: total amount of the order

products

id (primary key, auto-increment): unique identifier for each product
name: product's name
price: product's price
description: product's description

Queries to Write:

- Retrieve all customers who have placed an order in the last 30 days.

  SELECT * FROM orders JOIN customers ON orders.customer_id=customers.customer_id WHERE order_date >= CURDATE()-INTERVAL 30 DAY;

Using the JOIN Clause: The JOIN clause combines data from the orders and customers tables based on a common column (in this case, customer_id). This allows us to retrieve relevant information from both tables, such as customer details and their corresponding order data, to create a meaningful result set.

Using the WHERE Clause with INTERVAL: The WHERE clause is used to filter rows based on specific conditions. The INTERVAL keyword represents periods of time (e.g., days, months, years) that can be added to or subtracted from a date. This helps define time-based conditions, such as filtering records from the past 30 days.

Using INTERVAL with CURDATE(): The CURDATE() function returns the current date, and the INTERVAL keyword can modify it by adding or subtracting time units (like days, months, or years). This functionality is flexible and works with DATE, TIME, or DATETIME values, enabling dynamic date-based queries.

- Get the total amount of all orders placed by each customer.

  SELECT customer_id, SUM(total_amount) AS total_amount FROM orders GROUP BY customer_id;

Using the SUM function, the query calculates the total total_amount for each customer_id.

The GROUP BY clause organizes the data into groups based on customer_id, ensuring that the total is calculated separately for each customer.

The result will be a list showing each customer_id along with the total amount of all the orders they have placed.

- Update the price of Product C to 45.00.

  UPDATE products SET product_price = 45 WHERE product_name = 'Product C' ;

The purpose of the query is to update the price of a specific product in the products table.

Specifically, the SET clause updates the product_price column to 45 for the product whose product_name is 'Product C'. The WHERE clause ensures that only the row corresponding to

'Product C' is affected by this update. The query uses the UPDATE statement to modify the products table.

- Add a new column discount to the products table.

  ALTER TABLE products ADD discount INT(50);

The purpose of this query is to add a new column named discount to the products table. This new column will store discount values, and the data type for the column is defined as INT, which is used to store integer values.

The ALTER TABLE statement is used to modify the structure of the table, and the ADD clause is used to introduce the new discount column.

- Retrieve the top 3 products with the highest price.

  SELECT * FROM products ORDER BY product_price DESC LIMIT 3;

The SELECT * retrieves all columns (such as product_id, product_name, product_price, etc.) from the products table for each product.

The ORDER BY product_price DESC orders the products in descending order based on the product_price, ensuring that the highest-priced products appear first.

The LIMIT 3 restricts the output to only the top 3 products with the highest prices, displaying only the most expensive items.

- Get the names of customers who have ordered Product A.

  ```
  ALTER TABLE orders
  ADD product_id INT NOT NULL,
  ADD FOREIGN KEY (product_id) REFERENCES products(product_id);

  UPDATE orders
  SET product_id = 1
  WHERE product_id =3;

  SELECT DISTINCT c.customer_name
  FROM customers c
  JOIN orders o ON c.customer_id = o.customer_id
  JOIN products p ON o.product_id = p.product_id
  WHERE p.product_name = 'Product A';
  ```

The ALTER TABLE statement adds a new column product_id to the orders table, ensuring that each order is linked to a product. The column is marked as NOT NULL, and a foreign key constraint is added to reference the product_id in the products table.

The UPDATE statement modifies orders with a product_id of 3, changing it to 1. This ensures that all rows in the orders table have a valid product_id that corresponds to an existing product in the products table.

The SELECT DISTINCT query retrieves unique customer names who have ordered "Product A" by joining the customers, orders, and products tables. It filters the results to only include those customers who ordered "Product A".

- Join the orders and customers tables to retrieve the customer's name and order date for each order.

  SELECT customers.customer_name, orders.order_date FROM customers JOIN orders ON orders.customer_id=customers.customer_id;

The query selects the customer_name from the customers table and the order_date from the orders table to show which customers placed orders and when.

The JOIN operation links the customers table to the orders table using the customer_id field, ensuring that customer details are paired with their corresponding orders.

The query displays the order date for each customer, providing the specific date when each order was placed.

- Retrieve the orders with a total amount greater than 150.00.

  SELECT * FROM orders WHERE total_amount >150;

The query selects all columns from the orders table where the total_amount is greater than 150.

The WHERE clause filters the rows, only including orders with a total_amount greater than 150.

The query returns the details of orders that meet the specified condition, showing

all their attributes.

- Normalize the database by creating a separate table for order items and updating the orders table to reference the order_items table.

```
CREATE TABLE order_items (
    order_item_id INT NOT NULL AUTO_INCREMENT,
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT NOT NULL,
    PRIMARY KEY (order_item_id),
    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id)
) ENGINE=InnoDB;
```

The code creates a new order_items table to store individual items in each order, allowing multiple products per order.

Foreign keys are used to link the order_items table to the orders and products tables, ensuring referential integrity between orders and products.

The quantity column in the order_items table stores the number of each product in an order, facilitating accurate tracking of product quantities for each order.

- Retrieve the average total of all orders.

```
SELECT AVG(total_amount) AS avg_total_amount FROM orders;
```

The AVG() function computes the average of the total_amount column in the orders table.

The result is returned with the alias avg_total_amount, making it clear that it represents the average total of all orders.

The query calculates the overall average total amount across all orders in the orders table without any grouping.

Note:

- Use MySQL syntax and formatting for your queries.
- Provide clear and concise comments for each query explaining what it does.