1. Bank Table with Operations (Analogy)

Imagine a Bank with a Customer Database:

The database (table) named bank stores information about customers, including:

- S.No - Serial Number (like a customer ID)
- Cust_Name - Customer Name
- Acc_no - Account Number
- Balance - Account Balance
- Cus_Branch - Branch Name

Code and Analogy:

a. Create Table (Setting Up the Database):

SQL

```
CREATE TABLE bank (
  S.No INT PRIMARY KEY,
  Cust_Name VARCHAR(255),
  Acc_no INT,
  Balance DECIMAL(10,2),
  Cus_Branch VARCHAR(255)
);
```

Analogy: This code is like creating a filing cabinet (table) with labeled folders (columns) to store customer information sheets (rows).

b. Simple Select (Viewing All Customers):

SQL

```
SELECT * FROM bank;
```

Analogy: This code is like opening the entire filing cabinet (table) and retrieving all customer information sheets (rows) at once.

c. Select with Where Clause (Finding Specific Customer):

SQL

```
SELECT * FROM bank WHERE S.No = 2;
```

Analogy: This code is like searching the filing cabinet for a specific customer information sheet by their ID (S.No = 2).

d. Select with Comparison Operator > (Finding Rich Customers):

SQL

```
SELECT * FROM bank WHERE Balance > 200000;
```

Analogy: This code is like searching the filing cabinet for customer information sheets where the balance is greater than a certain amount (200000).

e. Select with Between (Finding Customers in Balance Range):

SQL

```
SELECT * FROM bank WHERE Balance BETWEEN 100000 AND 200000;
```

Analogy: This code is like searching the filing cabinet for customer information sheets where the balance falls within a specific range (between 100000 and 200000).

f. Update Cus_Branch (Changing Branch Information):

SQL

```
UPDATE bank SET Cus_Branch = 'Poonamallee' WHERE S.No = 2;
```

Analogy: This code is like finding a specific customer information sheet (S.No = 2) and updating the branch information on that sheet to 'Poonamallee'.

2. CASE Statement for Income Level (Analogy)

Imagine a Scenario for Income Levels:

A program needs to categorize income levels based on monthly income values.

SQL

```sql
SELECT monthly_value,
  CASE
    WHEN monthly_value = 4000 THEN 'Low Income'
    WHEN monthly_value = 5000 THEN 'Avg Income'
    ELSE 'High Income'
  END AS income_level
FROM your_table;
```

Analogy: This code is like a decision tree. It checks the monthly_value and assigns an income level based on specific conditions:

- If monthly_value is 4000, it's categorized as 'Low Income'.
- If monthly_value is 5000, it's categorized as 'Avg Income'.
- For any other value, it's categorized as 'High Income'.

3. Trigger for Employee Changes (Analogy)

Imagine an Audit System for Employee Data:

You want to track changes made to employee information in a table named EMPLOYEES.

Code Breakdown:

1. **Trigger Creation:** This involves defining a trigger, a special program that automatically runs before specific events (like updates) on a table.
2. **BEFORE UPDATE:** This specifies that the trigger will run before any update operation is performed on the EMPLOYEES table.
3. **Log Changes:** The trigger code will insert a new record into a separate table named EMPLOYEES_AUDIT whenever an update occurs. The new record will capture details like the employee ID, previous name, previous email, updated name, and updated email.

Analogy: This is like having an automatic security camera that monitors the EMPLOYEES table. Whenever someone tries to update an employee record (like changing name or email), the camera captures a snapshot of the change (previous and updated values) and stores it in a separate audit log table (EMPLOYEES_AUDIT).

Note: Creating triggers involves writing more complex code, so it's beyond the scope of a basic analogy here.

Here's the code to create a trigger in MySQL to log changes to the EMPLOYEES table:

SQL

```sql
DELIMITER // -- Define delimiter



CREATE TABLE EMPLOYEES_AUDIT (

  id INT PRIMARY KEY,

  employee_id INT REFERENCES EMPLOYEES(ID) ON DELETE CASCADE,

  name VARCHAR(255),

  email VARCHAR(255),

  old_name VARCHAR(255),

  old_email VARCHAR(255),

  change_time DATETIME DEFAULT CURRENT_TIMESTAMP,

   operation VARCHAR(20)
);



CREATE TRIGGER employee_update_audit BEFORE UPDATE ON EMPLOYEES

FOR EACH ROW

BEGIN

  INSERT INTO EMPLOYEES_AUDIT (employee_id, name, email, old_name,
old_email, operation)

  VALUES (OLD.ID, NEW.Name, NEW.Email, OLD.Name, OLD.Email, 'UPDATE');

END // -- End delimiter



DELIMITER ;  -- Reset delimiter
```

Explanation:

1. **Create EMPLOYEES_AUDIT Table:**
    - This table stores the audit trail for the EMPLOYEES table.
    - `id`: An auto-incrementing integer primary key for the audit records.

- `employee_id`: Foreign key referencing the ID in the EMPLOYEES table.
- `name`: Stores the new name of the employee after the update.
- `email`: Stores the new email of the employee after the update.
- `old_name`: Stores the old name of the employee before the update (obtained from the OLD row).
- `old_email`: Stores the old email of the employee before the update (obtained from the OLD row).
- `change_time`: Automatically stores the date and time of the change using `CURRENT_TIMESTAMP`.
- `operation`: Stores the type of operation performed (in this case, 'UPDATE').

2. **Create Trigger (employee_update_audit):**

   - This trigger is named `employee_update_audit`.
   - It fires BEFORE UPDATE on the EMPLOYEES table.
   - `FOR EACH ROW` specifies that the trigger actions happen for each row affected by the update statement.

3. **Trigger Body:**

   - An INSERT statement is executed within the trigger.
   - This statement inserts a new row into the EMPLOYEES_AUDIT table.
   - `OLD.ID`: Refers to the employee ID before the update (from the OLD row).
   - `NEW.Name` and `NEW.Email`: Refer to the updated name and email after the update (from the NEW row).
   - `OLD.Name` and `OLD.Email`: Retrieved from the OLD row to store the

previous values.
   - `'UPDATE'`: Sets the operation type as 'UPDATE'.
4. **Delimiter:**
   - The code uses delimiters (`//`) to differentiate the trigger creation script from regular SQL statements.
   - It defines the delimiter, creates the tables and trigger, and then resets the delimiter back to the semicolon (`;`).

Analogy:

Imagine a logbook for employee changes. This code creates a new logbook (EMPLOYEES_AUDIT table) to track updates to the employee list (EMPLOYEES table). The trigger acts like an automatic log recorder. Whenever someone updates an employee's name or email, the trigger captures the old and new values and records them in the logbook along with the date and time of the update. This provides a history of changes for audit purposes.

4. Procedure to Display Customer Records (Analogy)

Imagine a Customer Report Generation:

You want a procedure to retrieve and display all customer information from the

Absolutely! Here's the code with an analogy for displaying all records from the customer table:

Code:

SQL

```sql
DELIMITER // -- Define delimiter


CREATE PROCEDURE display_all_customers()
BEGIN
  SELECT * FROM Customer;
END // -- End delimiter


DELIMITER ;  -- Reset delimiter


CALL display_all_customers();  -- Call the procedure
```

Analogy:

Imagine a customer database in a store. This database holds information about all the customers, including their names, addresses, purchase history, etc.

The Code as a Special Process:

1. **DELIMITER // -- Define delimiter (Setting Up):** This line acts like defining a special instruction format for a specific task. Instead of regular SQL statements, you're creating a stored procedure with a `//` symbol as a marker to differentiate it.

2. **CREATE PROCEDURE display_all_customers() (Defining the Task):** This line

is like naming the task and its purpose. It defines a stored procedure named `display_all_customers` that specifically retrieves and displays all customer information.

3. **BEGIN (Starting the Task):** This marks the beginning of the actual steps involved in the task.

4. **SELECT * FROM Customer; (Performing the Task):** This line is like the core instruction of the task. It performs a selection operation on the `Customer` table, retrieving all columns (`*`) for all rows in the table. This essentially retrieves information about every customer.

5. **END // -- End delimiter (Finishing Up):** This signifies the end of the specific task instructions defined with the `//` delimiter.

6. **DELIMITER ; -- Reset delimiter (Back to Normal):** This resets the delimiter back to the standard semicolon (`;`) for regular SQL statements outside the stored procedure.

7. **CALL display_all_customers(); -- Call the procedure (Running the Task):** This line is like executing the defined task. It calls the `display_all_customers` stored procedure, which retrieves and displays all customer information from the database.

Overall, this code creates a reusable stored procedure that simplifies retrieving and displaying all customer information in a single step. You can call this procedure anytime to get the latest customer data without needing to write the entire selection statement each time. This makes your code more concise and easier to manage.