1. Bank Table, Queries, and Update (Analogy Included)

Code:

SQL

```
CREATE TABLE Bank (
  S_No INT PRIMARY KEY,
  Cust_Name VARCHAR(255),
  Acc_No INT,
  Balance DECIMAL(10,2),
  Cus_Branch VARCHAR(255)
);

-- Insert sample data (same as provided)
INSERT INTO Bank (S_No, Cust_Name, Acc_No, Balance, Cus_Branch)
VALUES (1, 'Ramesh', 12378, 100000, 'Adyar'),
       (2, 'Sam', 12367, 152500, 'Mylapore'),
       (3, 'Harish', 12345, 250000, 'Anna Salai');

-- Simple Select (Display all columns from all rows)
SELECT * FROM Bank;

-- Select with WHERE clause (Find customer with account number 12378)
SELECT * FROM Bank WHERE Acc_No = 12378;

-- Select with comparison operator > (Find customers with balance
above 150000)
SELECT * FROM Bank WHERE Balance > 150000;

-- Select with BETWEEN (Find customers with balance between 100000 and
200000)
SELECT * FROM Bank WHERE Balance BETWEEN 100000 AND 200000;

-- Update (Change branch for customer with account number 12367)
UPDATE Bank SET Cus_Branch = 'Poonamallee' WHERE Acc_No = 12367;

-- Verify update (Optional)
SELECT * FROM Bank WHERE Acc_No = 12367;
```

Analogy:

Imagine a bank with a customer database. You create a table named Bank to store information like customer details (name, account number), account balance, and branch.

- **Simple Select:** This is like requesting a complete customer list with all details from the bank.
- **Select with WHERE clause:** This is like searching for a specific customer by their account number, similar to how a bank teller might look up a customer's account based on its number.
- **Select with comparison operator > :** This is like filtering customers based on a criterion, such as finding customers with balances exceeding a certain amount. Imagine checking which accounts have high balances.
- **Select with BETWEEN:** This is like specifying a range for filtering, such as finding customers with balances within a specific range. Imagine identifying customers whose account balances fall within a particular bracket.
- **Update:** This is like modifying a customer's information in the database. In this case, you're updating the branch information for a specific customer.

2. CASE Statement for Income Level (Analogy Included)

Code:

SQL

```sql
CREATE FUNCTION determine_income_level(monthly_value INT)
RETURNS VARCHAR(255)
BEGIN
  DECLARE income_level VARCHAR(255);

  CASE WHEN monthly_value = 4000 THEN
    SET income_level = 'Low Income';
  WHEN monthly_value = 5000 THEN
    SET income_level = 'Avg Income';
  ELSE
    SET income_level = 'High Income';
  END CASE;

  RETURN income_level;
END;

-- Example usage: Call the function with a monthly value of 4500
SELECT determine_income_level(4500);
```

Analogy:

Imagine a system that categorizes income levels based on monthly income. This function acts like a decision-making tool.

- **monthly_value (parameter):** This represents the monthly income amount you want to classify.
- **income_level (variable):** This stores the income category based on the CASE conditions.
- **CASE statement:** This checks the monthly_value against different thresholds (4000 and 5000) and assigns the corresponding income level ('Low Income', 'Avg Income', or 'High Income').
- **Function return:** This returns the determined income level based on the monthly_value.

3. CASE Statement with More Levels (Analogy Included)

Code:

SQL

```sql
CREATE FUNCTION determine_income_level2(monthly_value INT)
RETURNS VARCHAR(255)
BEGIN
  DECLARE income_level VARCHAR(255);

  CASE WHEN monthly_value <= 3000 THEN
    SET income_level = 'Low Income';
  WHEN monthly_value <=
```

Sources
1. https://github.com/subi15github/DBMS-Experiments

Unfortunately, MySQL itself doesn't directly support recursion within functions. Recursion involves a function calling itself within its definition. While MySQL offers powerful functionalities, recursion isn't a built-in capability.

Here's an alternative approach to calculate factorial using a loop within a function:

Code:

SQL

```sql
CREATE FUNCTION calculate_factorial(number INT)
RETURNS INT
BEGIN
  DECLARE factorial INT DEFAULT 1;
  DECLARE counter INT;

  IF number < 0 THEN
    SET factorial = -1; -- Handle negative input (optional)
  ELSEIF number = 0 THEN
    SET factorial = 1;
  ELSE
    SET counter = 2;
    WHILE counter <= number DO
      SET factorial = factorial * counter;
      SET counter = counter + 1;
    END WHILE;
  END IF;

  RETURN factorial;
END;

-- Example usage: Call the function to calculate factorial of 5
SELECT calculate_factorial(5);
```

Analogy:

Imagine calculating factorial as multiplying a number by all positive integers less than itself. This function simulates that process iteratively:

- **number (parameter):** This represents the number for which you want to calculate the factorial.
- **factorial (variable):** This variable accumulates the product as the loop iterates.
- **counter (variable):** This variable keeps track of the numbers used for multiplication.
- **IF statements:** These handle different scenarios: negative input (optional), zero factorial (1), and positive factorials.

- **WHILE loop:** This loop iterates from 2 up to the input number, multiplying `factorial` by `counter` in each iteration.

This approach achieves the factorial calculation using a loop within a function, avoiding recursion.