1. Student Table Alterations, Description, Copy, Delete, and Drop (Analogy Included)

Code:

SQL

```sql
CREATE TABLE Student (
  S_No INT PRIMARY KEY,
  Name VARCHAR(255),
  Subject VARCHAR(255),
  Mark INT
);

-- Insert sample data (same as provided)
INSERT INTO Student (S_No, Name, Subject, Mark)
VALUES (1, 'Somu', 'OS', 75),
       (2, 'Sankar', 'DBMS', 66),
       (3, 'Mahesh', 'MOBILE COMPUTING', 92);

-- Add Grade column
ALTER TABLE Student ADD COLUMN Grade VARCHAR(255);

-- Modifying the Name column (Caution):**
-- This modification is not recommended as it alters existing data.
Consider creating a new column with a different name instead.
-- (Example provided for demonstration purposes only)
-- ALTER TABLE Student MODIFY Name VARCHAR(255) CHARACTER SET utf8mb4
COLLATE utf8mb4_unicode_ci;

-- Describe table structure (focusing on Subject column)
DESCRIBE Student SELECT * FROM INFORMATION_SCHEMA.COLUMNS WHERE
TABLE_NAME = 'Student' AND COLUMN_NAME = 'Subject';

-- Copy table Student as sub (assuming sub doesn't exist)
CREATE TABLE sub (
  LIKE Student
);

-- Delete second row
DELETE FROM Student WHERE S_No = 2;

-- Drop table Student (**Caution: This action is permanent!**)
DROP TABLE Student;
```

Analogy:

Imagine a student database. You create a table named Student to store information like student ID (S_No), name, subject, and marks.

- **Adding Grade column:** This is like adding a new field (Grade) to the student record, similar to including a grade letter for each student in the database.
- **Modifying Name Column (Caution):** Modifying existing columns like Name can be risky as it might alter existing data. It's generally better to create a new column with a different name for additional data points. This analogy is similar to potentially corrupting existing student information if you try to change the format or meaning of the "Name" field.
- **Describing the table Subject:** This is like checking the specific details (data type, size, etc.) of the 'Subject' column, similar to examining the properties of the "Subject" field in the student database.
- **Copying the Table:** This is like creating a duplicate of the student database table with a new name (sub). Imagine making a backup copy of the student data for reference or further processing.
- **Deleting a Row:** This is like removing a specific student record from the database. In this case, you're deleting the second row (which corresponds to Sankar).
- **Dropping the Table (Caution):** This is like permanently deleting the entire student database table. Use this with caution as it cannot be undone! Imagine completely erasing the student data.

2. CASE Statement Function for Income Level (Analogy Included)

Code:

SQL

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

  CASE monthly_value
    WHEN 4000 THEN SET income_level = 'Low Income';
    WHEN 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 based on pre-defined thresholds.

- **monthly_value (parameter):** This represents the monthly income amount.
- **income_level (variable):** This stores the determined income level based on the CASE conditions.
- **CASE statement:** This defines different categories based on the monthly income value.
  - **WHEN conditions:** These specify the income values for different categories (Low Income, Avg Income, High Income).
  - **SET income_level:** This assigns the corresponding income level based on the matched condition.
- **Function return:** This returns the determined income level based on the monthly value.

3. Procedure to List High Earners (Analogy Included)

Refer to explanation and code in section 1.3 for a procedure listing high earners.

Analogy:

Imagine the employee table as a filing cabinet with employee

Sources
1. https://github.com/192120020/DBMS

Here are four commonly used aggregate functions with explanations and code examples:

1. **COUNT:** This function calculates the total number of rows in a table or the number of rows that meet a specific condition.

Analogy: Imagine a library with books on different shelves. The `COUNT` function is like counting the total number of books in the library or the number of books in a specific genre on a particular shelf.

Code Example:

SQL

```
SELECT COUNT(*) AS total_students
FROM Student;
```

This code would count the total number of students in the `Student` table.

2. **SUM:** This function calculates the sum of a numeric column in a table.

Analogy: Imagine a store that sells shirts. The `SUM` function is like calculating the total revenue from shirt sales by adding up the prices of all the shirts sold.

Code Example:

SQL

```
SELECT SUM(Mark) AS total_marks
FROM Student;
```

This code would calculate the total marks scored by all students in the `Student` table.

3. **AVG:** This function calculates the average value of a numeric column in a table.

Analogy: Imagine a class with students who took a test. The `AVG` function is like calculating the average score of all the students in the class.

Code Example:

SQL

```
SELECT AVG(Mark) AS average_mark
FROM Student;
```

This code would calculate the average mark scored by students in the `Student` table.

4. **MAX:** This function finds the highest value in a numeric column in a table.

Analogy: Imagine a basketball game with players of different heights. The `MAX` function is like finding the tallest player on the team.

Code Example:

SQL

```
SELECT MAX(Mark) AS highest_mark
FROM Student;
```

This code would find the highest mark scored by any student in the `Student` table.