Transaction

Utility: Identify Underactive Users Who Haven't Logged Meals or Exercises in the Last Week and remove the users

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION;
DECLARE @CutoffDate DATETIME = DATEADD(DAY, -30, GETDATE());
CREATE TABLE #InactiveUsers (
    User_Id INT,
    Username VARCHAR(100)
);
INSERT INTO #InactiveUsers (User_Id, Username)
SELECT
    U.User_Id,
    U.Username
FROM
    User U
LEFT JOIN Meals M ON U.User_Id = M.User_Id AND M.Time >= @CutoffDate
LEFT JOIN Exercises E ON U.User_Id = E.User_Id AND E.Time >= @CutoffDate
WHERE
    M.Meal_Id IS NULL
    AND E.Exercise_Id IS NULL;
DELETE FROM User
WHERE User_Id IN (SELECT User_Id FROM #InactiveUsers);
DROP TABLE #InactiveUsers;
COMMIT TRANSACTION;//
delimiter ;
```

Isolation level: SERIALIZABLE. It ensures that no new meals or exercises are added for users during the transaction, guaranteeing consistency.
Query used: Identify inactive users who haven't logged meals or exercises in the last 30 days using a combination of LEFT JOIN and WHERE conditions.

Utility: Calculate Total Weekly Calorie Intake for a User

```
delimiter //
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN TRANSACTION;

DECLARE @UserId INT = 1;

DECLARE @StartDate DATETIME = DATEADD(DAY, -7, GETDATE());
DECLARE @EndDate DATETIME = GETDATE();

SELECT
    U.Username,
    SUM(F.Calories_Per_Gram * F.Quantity + D.Calories_Per_Gram * D.Quantity) AS Total_Calories
FROM
    User U
INNER JOIN Meals M ON U.User_Id = M.User_Id
LEFT JOIN Food F ON M.Meal_Id = F.Meal_Id
LEFT JOIN Drinks D ON M.Meal_Id = D.Meal_Id
WHERE
    U.User_Id = @UserId
    AND M.Time BETWEEN @StartDate AND @EndDate
GROUP BY
    U.Username;

COMMIT TRANSACTION;//
delimiter ;
```

Isolation level: REPEATABLE READ. Ensures the same rows read during the transaction cannot be modified by others, avoiding non-repeatable reads.
Query used: Uses JOINs to combine related data from multiple tables (e.g., User, Meals, Food, and Drinks), aggregation with SUM to calculate total weekly calories, and dynamic filtering with date range variables for flexible querying. It also utilizes GROUP BY to aggregate data by user, enabling per-user insights, and handles missing data gracefully using LEFT JOIN.

Stored Procedure

1

```sql
CREATE DEFINER=`root`@`%` PROCEDURE `CalculateExerciseTimeForUser`(
    IN p_User_Id INT
)
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE exercise_name VARCHAR(100);
    DECLARE total_time TIME;
    DECLARE exercise_cursor CURSOR FOR
    SELECT DISTINCT Exercise_Name
    FROM Exercises
    WHERE UserId = p_User_Id;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
    CREATE TEMPORARY TABLE IF NOT EXISTS ExerciseSummary (
        Exercise_Name VARCHAR(100),
        Total_Time TIME
    );
    OPEN exercise_cursor;
    exercise_loop: LOOP
        FETCH exercise_cursor INTO exercise_name;
        IF done THEN
            LEAVE exercise_loop;
        END IF;
        SELECT SEC_TO_TIME(SUM(TIME_TO_SEC(Time) * Sets)) INTO total_time
        FROM Exercises
        WHERE UserId = p_User_Id AND ExerciseName = exercise_name;
         INSERT INTO ExerciseSummary (ExerciseName, Total_Time)
         VALUES (exercise_name, total_time);
     END LOOP;
     CLOSE exercise_cursor;
     SELECT * FROM ExerciseSummary;
     DROP TEMPORARY TABLE IF EXISTS ExerciseSummary;
END
```

Query Used:
SELECT DISTINCT retrieves unique exercise names for the specified user (p_User_Id) from the Exercises table.
SEC_TO_TIME and TIME_TO_SEC functions calculate the total time spent on each exercise by multiplying the time duration by the number of sets.
Cursor:
exercise_cursor: Iterates through distinct exercise names for the specified user.
FETCH retrieves each exercise name one by one.
CONTINUE HANDLER ensures that when no more rows are available, the done variable is set to terminate the loop.
Control Structure:

LOOP: Iterates over the distinct exercises for the user, processing each exercise to calculate total time.
IF/LEAVE: Exits the loop when all rows in the cursor have been processed.
Application Utility:
Provides a detailed summary of total time spent on each type of exercise for a specific user, stored temporarily in ExerciseSummary. It also handles multiple exercises for a user dynamically, regardless of the number of exercise types.

2

```
CREATE DEFINER=`root`@`%` PROCEDURE `GetMostCaloricFoodLastWeek`(
    IN p_User_Id INT
)
BEGIN
    SELECT f.FoodName,
           f.CaloriesTotal
    FROM Food f
    JOIN Meals m ON f.MealId = m.MealId
    WHERE m.UserId = p_User_Id
      AND m.Time >= DATE_SUB(NOW(), INTERVAL 7 DAY)
    GROUP BY f.FoodName
    ORDER BY f.CaloriesTotal DESC
    LIMIT 1;
END
```

Query Used:
JOIN: Combines Food and Meals tables to link food items with their corresponding meals based on Meal_Id.
Filtering: Filters data to include only meals consumed by the specified user (p_User_Id) within the last 7 days (DATE_SUB(NOW(), INTERVAL 7 DAY)).
Aggregation: Uses SUM to calculate the total calories for each food item by multiplying Calories_Per_Gram with Quantity.
Sorting: Orders the results by Total_Calories in descending order to identify the most caloric food.
Limiting: Restricts the result to the single most caloric food (LIMIT 1).
Application Utility:
Diet Analysis: Identifies the food item contributing the most calories to a user's diet in the past week and helps users or nutritionists monitor and manage calorie intake effectively.
Dynamic Date Filtering: Enables weekly calorie tracking without requiring static date ranges.

3

```
CREATE DEFINER=`root`@`%` PROCEDURE `GetTotalCaloriesThisWeek`(
    IN UserId INT
)
BEGIN
    SELECT
        COALESCE(SUM(F.CaloriesTotal), 0) + COALESCE(SUM(D.CaloriesTotal), 0)
    FROM Meals M
    LEFT JOIN Food F ON M.MealId = F.MealId
    LEFT JOIN Drink D ON M.MealId = D.MealId
    WHERE M.UserId = UserId
      AND DATE(M.Time) BETWEEN DATE_SUB(CURRENT_DATE, INTERVAL 7 DAY) AND CURRENT_DATE;
END
```

select the CaloriesTotal field for both food and drink, and make sure the meal that contains the food is had within a week.

Trigger

```
delimiter //
CREATE TRIGGER UpdateDailyCalories
AFTER INSERT ON Meals
FOR EACH ROW
BEGIN
   DECLARE meal_calories INT;
   SELECT COALESCE(SUM(Food.CaloriesPerGram * Food.Quantity), 0)
        + COALESCE(SUM(Drinks.CaloriesPerGram * Drinks.Quantity), 0)
   INTO meal_calories
   FROM Food
   LEFT JOIN Drinks ON Food.MealId = Drinks.MealId
   WHERE Food.MealId = NEW.MealId;
   IF CURDATE() = DATE(NEW.Time) THEN
      UPDATE Food
      SET CaloriesTotal = COALESCE(Daily_Calories, 0) + meal_calories
      WHERE User.UserId = NEW.UserId;
   END IF;
END//
delimiter ;
```

Usage: When there is an insert to the meal, calculate the daily calories consumed by the user.

```
delimiter //
CREATE TRIGGER UpdateMealTime
AFTER INSERT ON Meals
FOR EACH ROW
BEGIN
```

```
    IF NEW.MealId IS NOT NULL AND NEW.UserId IS NOT NULL THEN
        UPDATE Meals
        SET Time = CURRENT_TIMESTAMP
        WHERE MealId = NEW.MealId AND UserId = NEW.UserId;
    END IF;
END;
//
delimiter ;
```

Usage: When there is an insert to the meal, store the Time automatically.

Constraints

There are various FK constraints and primary key constraints in our table .

```
CREATE TABLE `User` (
  `UserId` int NOT NULL,
  `UserName` varchar(30) DEFAULT NULL,
  `Height` int DEFAULT NULL,
  `Weight` int DEFAULT NULL,
  `Age` int DEFAULT NULL,
  PRIMARY KEY (`UserId`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

```
CREATE TABLE `Meals` (
  `MealId` int NOT NULL,
  `UserId` int DEFAULT NULL,
  `Time` datetime DEFAULT NULL,
  PRIMARY KEY (`MealId`),
  KEY `UserId` (`UserId`),
  CONSTRAINT `Meals_ibfk_1` FOREIGN KEY (`UserId`) REFERENCES `User` (`UserId`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

```
CREATE TABLE `Food` (
  `FoodName` varchar(100) NOT NULL,
  `NutritionType` varchar(30) DEFAULT NULL,
  `CaloriesPerGram` int DEFAULT NULL,
  `Quantity` int DEFAULT NULL,
  `MealId` int DEFAULT NULL,
  `CaloriesTotal` decimal(10,2) GENERATED ALWAYS AS ((`CaloriesPerGram` * `Quantity`)) STORED,
  PRIMARY KEY (`FoodName`),
  KEY `MealId` (`MealId`),
  KEY `idx_food_calories_per_gram` (`CaloriesPerGram`),
  KEY `idx_food_nutrition_calories` (`NutritionType`,`CaloriesPerGram`),
  KEY `idx_food_name` (`FoodName`),
  KEY `idx_food_calories_quantity` (`CaloriesPerGram`,`Quantity`),
  KEY `idx_food_calories_total` (`FoodName`,`CaloriesTotal`),
  CONSTRAINT `Food_ibfk_1` FOREIGN KEY (`MealId`) REFERENCES `Meals` (`MealId`)
```

```
CREATE TABLE `Drink` (
  `DrinkName` varchar(100) NOT NULL,
  `NutritionType` varchar(30) DEFAULT NULL,
  `CaloriesPerGram` int DEFAULT NULL,
  `Quantity` int DEFAULT NULL,
  `MealId` int DEFAULT NULL,
  `CaloriesTotal` decimal(10,2) GENERATED ALWAYS AS ((`CaloriesPerGram` * `Quantity`)) STORED,
 PRIMARY KEY (`DrinkName`),
 KEY `MealId` (`MealId`),
 KEY `idx_drink_calories_per_gram` (`CaloriesPerGram`),
 KEY `idx_drink_nutrition_calories` (`NutritionType`,`CaloriesPerGram`),
 KEY `idx_drink_name` (`DrinkName`),
 KEY `idx_drink_calories_quantity` (`CaloriesPerGram`,`Quantity`),
 KEY `idx_drink_calories_total` (`DrinkName`,`CaloriesTotal`),
 CONSTRAINT `Drink_ibfk_1` FOREIGN KEY (`MealId`) REFERENCES `Meals` (`MealId`)
```

```
CREATE TABLE `Report` (
  `ReportId` int NOT NULL,
  `UserId` int DEFAULT NULL,
  `Description` varchar(30) DEFAULT NULL,
  `Time` datetime DEFAULT NULL,
 PRIMARY KEY (`ReportId`),
 KEY `UserId` (`UserId`),
 CONSTRAINT `Report_ibfk_1` FOREIGN KEY (`UserId`) REFERENCES `User` (`UserId`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

```
CREATE TABLE `Exercises` (
  `ExerciseId` int NOT NULL,
  `UserId` int DEFAULT NULL,
  `ExerciseName` varchar(100) DEFAULT NULL,
  `Reps` int DEFAULT NULL,
  `Sets` int DEFAULT NULL,
  `Time` time DEFAULT NULL,
 PRIMARY KEY (`ExerciseId`),
 KEY `UserId` (`UserId`),
 CONSTRAINT `Exercises_ibfk_1` FOREIGN KEY (`UserId`) REFERENCES `User` (`UserId`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```