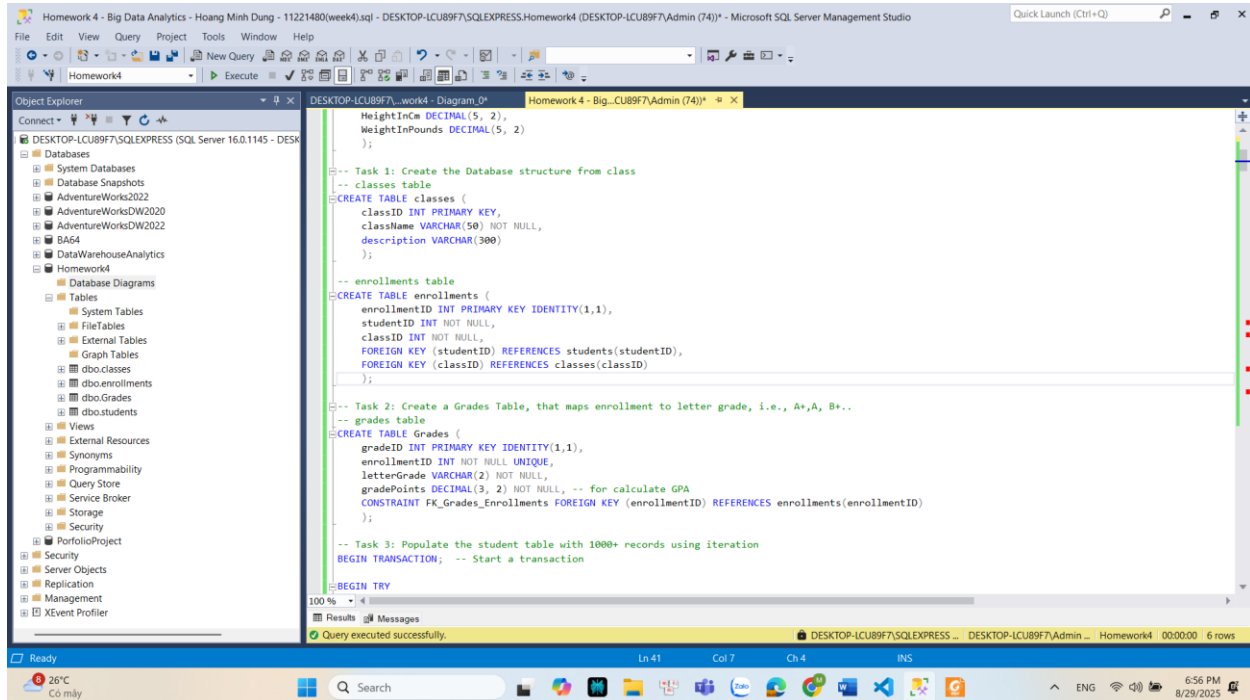


Homework 4 – [GitHub link SQL script](#)

Task 1 and 2:

1. Please create the Database structure from class

2. Create a Grades Table, that maps enrollment to letter grade, i.e., A+,A, B+ ...

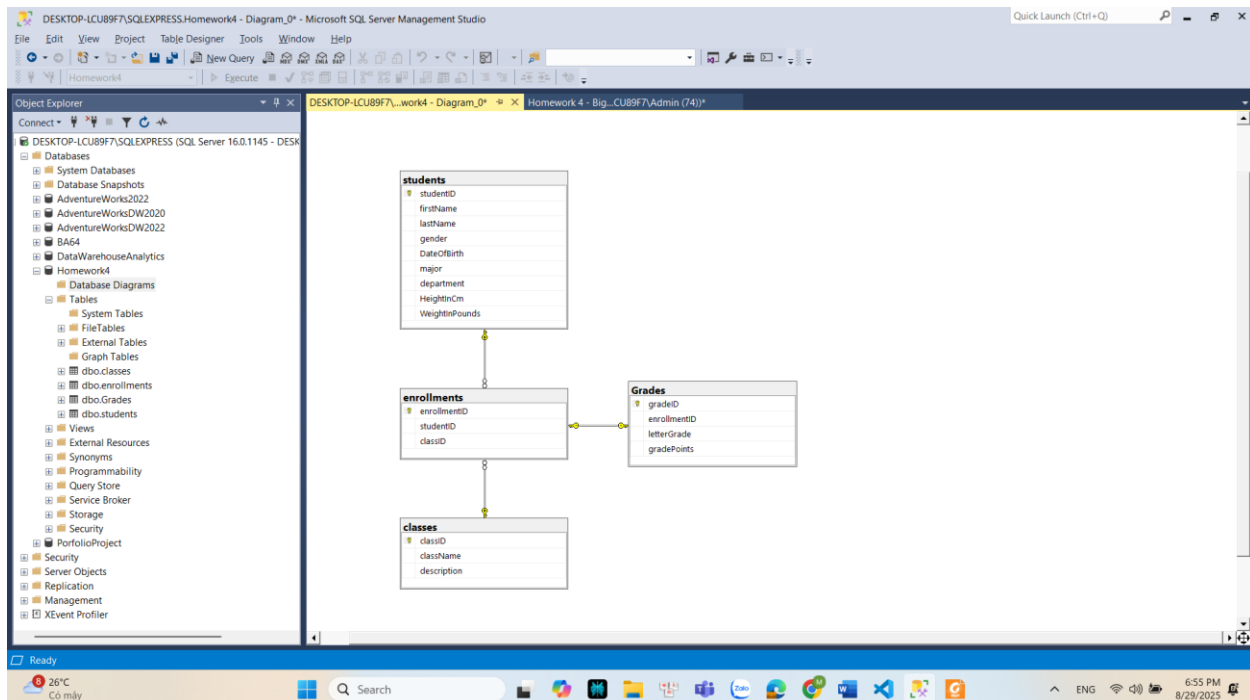


```
-- Task 1: Create the Database structure from class
-- classes table
CREATE TABLE classes (
    classID INT PRIMARY KEY,
    className VARCHAR(50) NOT NULL,
    description VARCHAR(300)
);

-- enrollments table
CREATE TABLE enrollments (
    enrollmentID INT PRIMARY KEY IDENTITY(1,1),
    studentID INT NOT NULL,
    classID INT NOT NULL,
    FOREIGN KEY (studentID) REFERENCES students(studentID),
    FOREIGN KEY (classID) REFERENCES classes(classID)
);

-- Task 2: Create a Grades Table, that maps enrollment to letter grade, i.e., A+,A, B+..
-- grades table
CREATE TABLE Grades (
    gradeID INT PRIMARY KEY IDENTITY(1,1),
    enrollmentID INT NOT NULL UNIQUE,
    letterGrade VARCHAR(2) NOT NULL,
    gradePoints DECIMAL(3, 2) NOT NULL, -- for calculate GPA
    CONSTRAINT FK_Grades_Enrollments FOREIGN KEY (enrollmentID) REFERENCES enrollments(enrollmentID)
);

-- Task 3: Populate the student table with 1000+ records using iteration
BEGIN TRANSACTION; -- Start a transaction
-- (The rest of the script is truncated in the image)
```



Task 3: Populate the student table with 1000+ records using iteration. Please use the example from class as a reference but make sure that your data is well varied and not uniform

```
-- Task 3: Populate the student table with 1000+ records using iteration
BEGIN TRANSACTION; -- Start a transaction

-- STEP 1: Generate 1000 students
-- =====
DECLARE @counter INT = 1;
DECLARE @gender VARCHAR(10);
DECLARE @firstName VARCHAR(50);
DECLARE @lastName VARCHAR(50);
DECLARE @dob DATE;
DECLARE @major VARCHAR(50);
DECLARE @department VARCHAR(50);
DECLARE @height DECIMAL(5,2);
DECLARE @weight DECIMAL(5,2);

WHILE @counter <= 1000
BEGIN
    -- Gender
    SET @gender = CASE WHEN ABS(CHECKSUM(NEWID())) % 2 = 0 THEN 'Male' ELSE 'Female' END;

    -- Names - Alternative approach using modulo with better bounds checking
    DECLARE @randomNum1 INT = ABS(CHECKSUM(NEWID()));
    DECLARE @randomNum2 INT = ABS(CHECKSUM(NEWID()));

    -- Ensure we get values 1-12 for firstName (12 options)
    DECLARE @firstNameIndex INT = (@randomNum1 % 12) + 1;
    -- Ensure we get values 1-10 for lastName (10 options)
    DECLARE @lastNameIndex INT = (@randomNum2 % 10) + 1;

    SET @firstName = CHOOSE(@firstNameIndex,
        'Anh', 'Linh', 'Chi', 'Long', 'Dung', 'Ngoc', 'Trang', 'Tuan', 'Minh', 'Huyen', 'Ha', 'Son');

    -- (Additional logic for other fields would follow here)

    @counter = @counter + 1;
END

-- Query executed successfully.
DESKTOP-LCU89F7\SQLEXPRESS ... Desktop-LCU89F7\Admin ... Homework4 00:00:00 1,020 rows
```

```
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION; -- Rollback if any error occurs
    PRINT 'Error occurred. Transaction rolled back.';
    SELECT ERROR_NUMBER() AS ErrorNumber, ERROR_MESSAGE() AS ErrorMessage;
END CATCH

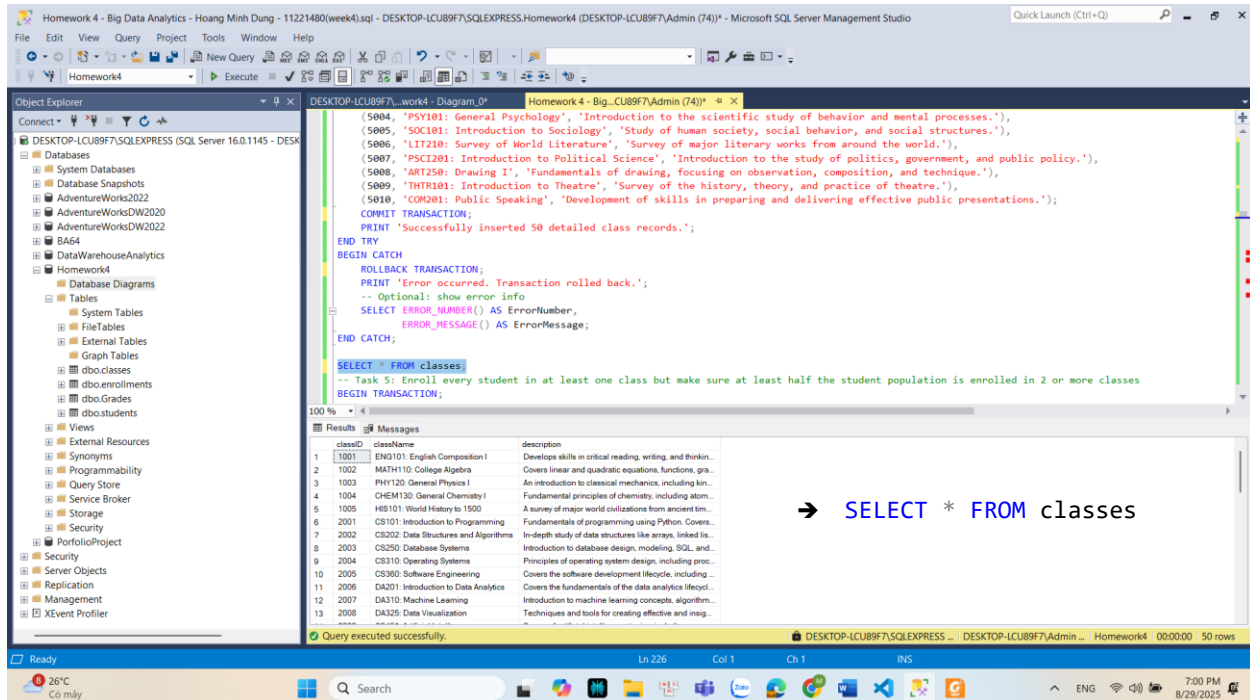
-- Print completion message
PRINT 'Successfully inserted 1000 students with diverse random data.';

SELECT TOP 20 * FROM students ORDER BY DateOfBirth ASC;
SELECT * FROM students;

-- Task 4: POPULATE 50 DETAILED CLASS RECORDS
BEGIN TRANSACTION;
```

studentID	firstName	lastName	gender	DateOfBirth	major	department	HeightCm	WeightPounds
5403	Son	Hoang	Female	2000-09-08	NULL	School of Social Sciences and Humanities	155.00	118.00
5259	Ngoc	Dang	Female	2000-09-09	Computer Science	School of Technology	168.00	142.00
5510	Son	Dang	Male	2000-09-11	NULL	School of Social Sciences and Humanities	173.00	194.00
5452	Huyen	Pham	Male	2000-09-12	Business Administration	School of Economics	173.00	126.00
5511	Long	Do	Male	2000-09-14	Medicine	School of Social Sciences and Humanities	177.00	127.00
5885	Chi	Mai	Male	2000-09-14	Medicine	School of Social Sciences and Humanities	183.00	161.00
5192	Huyen	Tran	Female	2000-09-15	Data Analytics	School of Technology	164.00	111.00
5100	Anh	Pham	Male	2000-09-19	Medicine	School of Social Sciences and Humanities	170.00	198.00
5798	Ngoc	Do	Female	2000-09-19	Medicine	School of Social Sciences and Humanities	157.00	143.00
5158	Trang	Tran	Female	2000-09-21	Computer Science	School of Technology	168.00	142.00
5558	Linh	Mai	Male	2000-09-22	Arts	School of Social Sciences and Humanities	165.00	197.00
5063	Anh	Do	Male	2000-09-24	NULL	School of Social Sciences and Humanities	180.00	128.00
5237	Dung	Nguyen	Male	2000-09-24	Engineering	School of Technology	166.00	197.00
5272	Dung	Hoang	Male	2000-10-02	Computer Science	School of Technology	183.00	180.00
5751	Tuan	Bui	Male	2000-10-05	Computer Science	School of Technology	167.00	181.00
5623	Ngoc	Pham	Female	2000-10-06	NULL	School of Social Sciences and Humanities	162.00	146.00
5302	Linh	Bui	Male	2000-10-13	Engineering	School of Technology	172.00	184.00
5972	Anh	Do	Female	2000-10-14	Business Administration	School of Economics	163.00	106.00
5691	Lono	Mai	Female	2000-10-18	Business Administration	School of Economics	167.00	111.00

Task 4: Populate the classes table with 50 classes



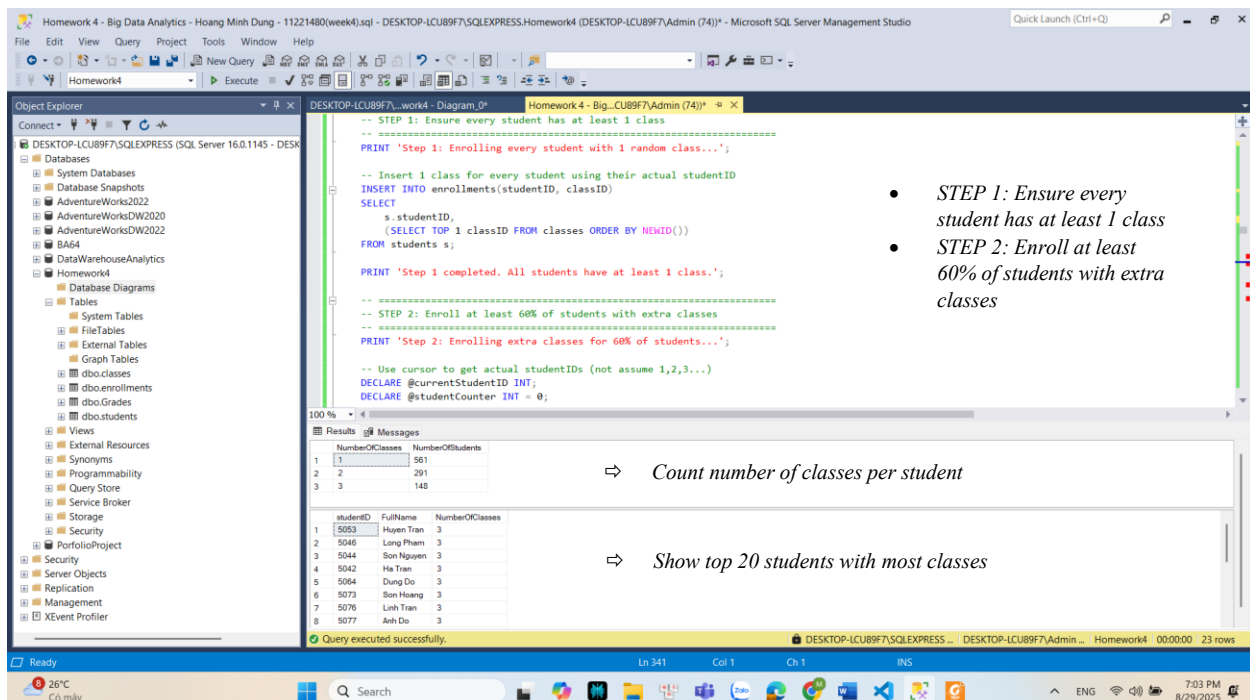
```
-- (5004, 'PSY101: General Psychology', 'Introduction to the scientific study of behavior and mental processes.'),
-- (5005, 'SOC101: Introduction to Sociology', 'Study of human society, social behavior, and social structures.'),
-- (5006, 'LIT210: Survey of World Literature', 'Survey of major literary works from around the world.'),
-- (5007, 'PSCI201: Introduction to Political Science', 'Introduction to the study of politics, government, and public policy.'),
-- (5008, 'ART250: Drawing I', 'Fundamentals of drawing, focusing on observation, composition, and technique.'),
-- (5009, 'THR101: Introduction to Theatre', 'Survey of the history, theory, and practice of theatre.'),
-- (5010, 'COM201: Public Speaking', 'Development of skills in preparing and delivering effective public presentations.'),
COMMIT TRANSACTION;
PRINT 'Successfully inserted 50 detailed class records.';
END TRY
BEGIN CATCH
ROLLBACK TRANSACTION;
PRINT 'Error occurred. Transaction rolled back.';
-- Optional: show error info
SELECT ERROR_NUMBER() AS ErrorNumber,
       ERROR_MESSAGE() AS ErrorMessage;
END CATCH;

SELECT * FROM classes;

-- Task 5: Enroll every student in at least one class but make sure at least half the student population is enrolled in 2 or more classes
BEGIN TRANSACTION;
```

→ **SELECT * FROM classes**

Task 5: Enroll every student in at least one class but make sure at least half the student population is enrolled in 2 or more classes



```
-- STEP 1: Ensure every student has at least 1 class
-- =====
PRINT 'Step 1: Enrolling every student with 1 random class...';

-- Insert 1 class for every student using their actual studentID
INSERT INTO enrollments(studentID, classID)
SELECT
    s.studentID,
    (SELECT TOP 1 classID FROM classes ORDER BY NEWID())
FROM students s;

PRINT 'Step 1 completed. All students have at least 1 class.';

-- STEP 2: Enroll at least 60% of students with extra classes
-- =====
PRINT 'Step 2: Enrolling extra classes for 60% of students...';

-- Use cursor to get actual studentIDs (not assume 1,2,3...)
DECLARE @currentStudentID INT;
DECLARE @studentCounter INT = 0;
```

⇒ **Count number of classes per student**

studentID	FullName	NumberOfClasses	
1	5053	Huyen Tran	3
2	5045	Long Pham	3
3	5044	Son Nguyen	3
4	5042	Ha Tran	3
5	5064	Dung Do	3
6	5073	Bon Hoang	3
7	5076	Linh Tran	3
8	5077	Anh Do	3

⇒ **Show top 20 students with most classes**

6. Assign grades for every class enrolled in

The screenshot shows the SQL Server Management Studio interface. The Object Explorer on the left displays the database structure. The main window shows a SQL script for Task 6: Assign grades for every class enrolled in. The script includes a transaction, a print statement, a delete statement to clear existing grades, variable declarations, a cursor to iterate over enrollments, and an insert statement. The Results pane shows the output of the script, including a table of grades and a summary table.

```
-- Task 6: Assign grades for every class enrolled in
SELECT enrollmentID FROM enrollments;
BEGIN TRANSACTION;

--BEGIN TRY
PRINT 'Starting Assigning grades for all enrollments...';

-- First, clear existing grades if any
DELETE FROM Grades;

-- Declare variables
DECLARE @enrollmentID INT;
DECLARE @letterGrade VARCHAR(2);
DECLARE @gradePoints DECIMAL(3,2);

-- Cursor to iterate over all enrollments
DECLARE enroll_cursor CURSOR FOR
SELECT enrollmentID FROM enrollments;

OPEN enroll_cursor;
```

Results:

gradeID	enrollmentID	letterGrade	gradePoints
1	1589	A+	4.00
2	1615	A+	4.00
3	1621	A+	4.00
4	1626	A+	4.00
5	1645	A+	4.00
6	1653	A+	4.00

CountPerGrade:

letterGrade	CountPerGrade
B	392
B+	385
C+	256
A	223
A+	173
C	106

⇒ SELECT * FROM Grades Order by gradePoints desc;

⇒ CountPerGrade

7. Write a stored procedure to compute the GPA

The screenshot shows the SQL Server Management Studio interface. The Object Explorer on the left displays the database structure. The main window shows a SQL script for Task 7: Write a stored procedure to compute the GPA. The script includes a procedure definition, a set nocount on, a select statement to compute the GPA for each student, and an exec statement to run the procedure. The Results pane shows the output of the script, including a table of student GPA.

```
-- Task 7: Write a stored procedure to compute the GPA
CREATE PROCEDURE ComputeGPA
@StudentID INT = NULL
AS
BEGIN
SET NOCOUNT ON;

SELECT
s.studentID,
s.firstName + ' ' + s.lastName AS FullName,
CAST(AVG(g.gradePoints) AS DECIMAL(3,2)) AS GPA
FROM students s
JOIN enrollments e ON s.studentID = e.studentID
JOIN Grades g ON e.enrollmentID = g.enrollmentID
WHERE (@StudentID IS NULL OR s.studentID = @StudentID)
GROUP BY s.studentID, s.firstName, s.lastName
ORDER BY GPA DESC;
END;
GO

EXEC ComputeGPA;
EXEC ComputeGPA @StudentID = 5035;
```

Results:

studentID	FullName	GPA
1	5062 Dung Pham	4.00
2	5062 Dung Pham	4.00
3	5135 Ha Tran	4.00
4	5136 Long Tran	4.00
5	5283 Trang Vu	4.00
6	5301 Long Bui	4.00
7	5304 Tuan Le	4.00
8	5309 Huyen Tran	4.00
9	5312 Minh Le	4.00
10	5361 Minh Tran	4.00

studentID FullName GPA

1	5035 Long Do	3.00
---	--------------	------

8. Write a stored procedure to compute the descriptive statistics of the student population, viz. Mean , Variance, Standard Deviation, Mode of the following Attributes

a.) G.P.A b.) HeightInCm c.) WeightInPounds

Hint: G.P.A is a derived attribute of every student, it depends on their grade in enrolled classes

The screenshot shows the Microsoft SQL Server Management Studio interface. The Object Explorer on the left displays the database structure. The main window shows a SQL script for a stored procedure named `ComputeDescriptiveStatistics`. The script includes comments for Task 8 and a hint about G.P.A. The procedure is designed to calculate mean, variance, standard deviation, and mode for GPA, Height, and Weight, with an optional filter by gender. The results pane shows the output of the procedure, displaying statistics for GPA, Height, and Weight. The status bar indicates the query was executed successfully.

```
-- Task 8: Write a stored procedure to compute the descriptive statistics of the student population, viz. Mean , Variance, Standard Deviation, Mode of the following Attribute:
-- a.) G.P.A b.) HeightInCm c.) WeightInPounds
-- Hint: G.P.A is a derived attribute of every student, it depends on their grade in enrolled classes
"/

--CREATE OR ALTER PROCEDURE ComputeDescriptiveStatistics
--@Gender VARCHAR(10) = NULL -- Optional parameter to filter by gender
--AS
--BEGIN
--    SET NOCOUNT ON;

--    -- Step 1: Use a CTE to calculate GPA and store the results in a temporary table (#StudentStatsData).
--    -- The temp table exists only for this session and will be dropped automatically if not dropped manually.
--    IF OBJECT_ID('tempdb..#StudentStatsData') IS NOT NULL
--        DROP TABLE #StudentStatsData;

--    WITH StudentStats AS (
--        SELECT
--            s.studentID,
--            s.HeightInCm,
--            s.WeightInPounds,
--            s.gender,
--            CAST(AVG(g.gradePoints) AS DECIMAL(10, 2)) AS GPA
--        FROM
--            Students s
--            JOIN Enrollments e ON s.studentID = e.studentID
--            JOIN Grades g ON e.enrollmentID = g.enrollmentID
--    )

--    EXEC ComputeDescriptiveStatistics@Gender;
```

⇒ EXEC ComputeDescriptiveStatistics;

⇒ EXEC ComputeDescriptiveStatistics@Gender = 'Male';

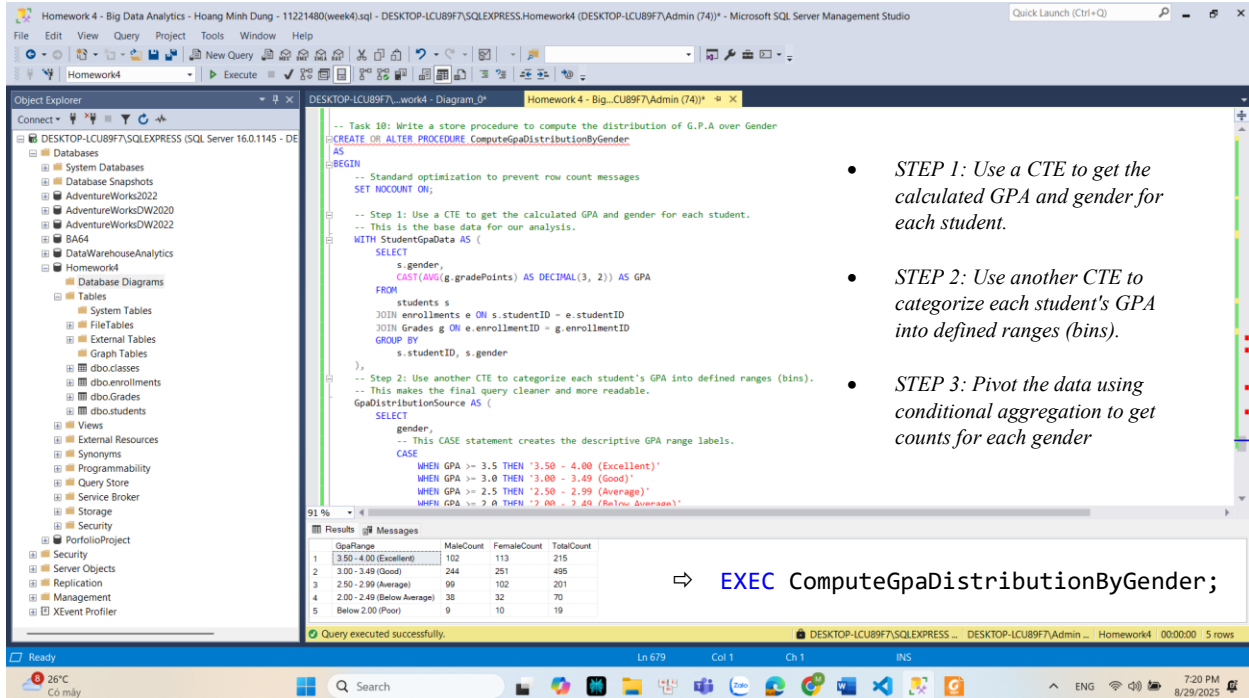
9. Create a VIEW to show how height distributes over Gender

The screenshot shows the Microsoft SQL Server Management Studio interface. The Object Explorer on the left displays the database structure. The main window shows a SQL script for a view named `vHeightDistributionByGender`. The script includes comments for Task 9 and a hint about using TOP 100 PERCENT. The view is designed to show the distribution of height ranges for males and females, along with the total count. The results pane shows the output of the view, displaying height ranges, male counts, female counts, and total counts. The status bar indicates the query was executed successfully.

```
-- Task 9: Create a VIEW to show how height distributes over Gender
-- Hint: Using TOP 100 PERCENT is a common technique to allow ORDER BY in a VIEW definition,
-- ensuring the default output is always sorted logically.
--CREATE OR ALTER VIEW vHeightDistributionByGender
--AS
--SELECT TOP 100 PERCENT
--    HeightRange,
--    MaleCount,
--    FemaleCount,
--    TotalCount
--FROM (
--    -- This inner query performs the aggregation (counting and grouping).
--    SELECT
--        HeightRange,
--        SortOrder,
--        COUNT(CASE WHEN gender = 'Male' THEN 1 END) AS MaleCount,
--        COUNT(CASE WHEN gender = 'Female' THEN 1 END) AS FemaleCount,
--        COUNT(*) AS TotalCount
--    FROM (
--        -- This innermost query categorizes each student's height into a specific bin.
--        SELECT
--            gender,
--            HeightInCm,
--            CASE
--                WHEN HeightInCm < 150 THEN 'Below 150cm'
--                WHEN HeightInCm < 160 THEN '150cm - 159.9cm'
--                WHEN HeightInCm < 170 THEN '160cm - 169.9cm'
--                WHEN HeightInCm < 180 THEN '170cm - 179.9cm'
--                WHEN HeightInCm < 190 THEN '180cm - 189.9cm'
--                WHEN HeightInCm < 200 THEN '190cm and Above'
--            END AS HeightRange
--        FROM
--            Students
--    )
--    GROUP BY
--        HeightRange,
--        SortOrder
--    ORDER BY
--        SortOrder
--)
```

⇒ SELECT * FROM vHeightDistributionByGender;

10. Write a store procedure to compute the distribution of G.P.A over Gender



The screenshot displays the Microsoft SQL Server Management Studio interface. The main window shows a SQL script for creating and executing a stored procedure named `ComputeGpaDistributionByGender`. The script includes comments for each step and uses CTEs to calculate GPA and categorize it into bins. The results pane shows the output of the procedure, which is a table with columns: `GpaRange`, `MaleCount`, `FemaleCount`, and `TotalCount`.

```
-- Task 10: Write a store procedure to compute the distribution of G.P.A over Gender
-- Create or alter the procedure
CREATE OR ALTER PROCEDURE ComputeGpaDistributionByGender
AS
BEGIN
    -- Standard optimization to prevent row count messages
    SET NOCOUNT ON;

    -- Step 1: Use a CTE to get the calculated GPA and gender for each student.
    -- This is the base data for our analysis.
    WITH StudentGpaData AS (
        SELECT
            s.gender,
            CAST(AVG(g.gradePoints) AS DECIMAL(3, 2)) AS GPA
        FROM
            students s
        JOIN enrollments e ON s.studentID = e.studentID
        JOIN Grades g ON e.enrollmentID = g.enrollmentID
        GROUP BY
            s.studentID, s.gender
    ),
    -- Step 2: Use another CTE to categorize each student's GPA into defined ranges (bins).
    -- This makes the final query cleaner and more readable.
    GpaDistributionSource AS (
        SELECT
            gender,
            -- This CASE statement creates the descriptive GPA range labels.
            CASE
                WHEN GPA >= 3.5 THEN '3.50 - 4.00 (Excellent)'
                WHEN GPA >= 3.0 THEN '3.00 - 3.49 (Good)'
                WHEN GPA >= 2.5 THEN '2.50 - 2.99 (Average)'
                WHEN GPA >= 2.0 THEN '2.00 - 2.49 (Below Average)'
                WHEN GPA < 2.0 THEN 'Below 2.00 (Poor)'
            END AS GpaRange
        FROM StudentGpaData
    )
    -- Final query to pivot the data using conditional aggregation
    SELECT
        GpaRange,
        SUM(CASE WHEN gender = 'M' THEN TotalCount ELSE 0 END) AS MaleCount,
        SUM(CASE WHEN gender = 'F' THEN TotalCount ELSE 0 END) AS FemaleCount,
        SUM(TotalCount) AS TotalCount
    FROM GpaDistributionSource
    GROUP BY GpaRange;
```

⇒ EXEC ComputeGpaDistributionByGender;

GpaRange	MaleCount	FemaleCount	TotalCount
3.50 - 4.00 (Excellent)	102	113	215
3.00 - 3.49 (Good)	244	251	495
2.50 - 2.99 (Average)	99	102	201
2.00 - 2.49 (Below Average)	38	32	70
Below 2.00 (Poor)	9	10	19

11. Describe what you learned from this homework

This homework assignment provided a comprehensive, hands-on journey through the entire lifecycle of creating and managing a relational database system. Moving from foundational concepts to advanced analytical procedures, I have solidified my understanding of SQL not just as a query language, but as a powerful platform for data engineering and analysis.

First and foremost, I gained practical experience in database schema design and implementation. The initial tasks required me to translate a conceptual model into a physical one by creating a series of interconnected tables: students, classes, enrollments, and Grades. This process reinforced the importance of defining primary keys for uniqueness, foreign keys to enforce referential integrity, and appropriate data types to ensure data quality. Designing the Grades table with a UNIQUE constraint on enrollmentID was a key insight into building robust, error-resistant schemas.

The second major area of learning was in procedural SQL and large-scale data generation. Rather than performing trivial, manual INSERT statements, I wrote a script to programmatically generate over 1,000 student records. This required using WHILE loops, variables, and randomization functions like CHECKSUM(NEWID()). More importantly, I learned to create realistic, non-uniform data by implementing conditional logic, such as varying height and weight based on gender, and assigning specific departments based on a student's major. This elevated the task from simple data entry to a simulation of real-world data.

Subsequently, the assignment transitioned into advanced data analysis and reporting, which was the most challenging and rewarding part. I learned to encapsulate complex logic into reusable

database objects. For instance, I developed a flexible ComputeGPA stored procedure with an optional parameter, allowing it to calculate GPA for either a single student or the entire population. The ComputeDescriptiveStatistics procedure was a deeper dive, compelling me to implement a custom solution for calculating the Mode—a function not natively available in SQL Server—in addition to using built-in aggregate functions like AVG, STDEV, and VAR.

Furthermore, I learned to differentiate between and apply two powerful reporting concepts: summary statistics and frequency distributions. For the latter, I created both a VIEW (vHeightDistributionByGender) and a Stored Procedure (ComputeGpaDistributionByGender). These tasks taught me the powerful technique of conditional aggregation (COUNT(CASE WHEN ...)), which allows for pivoting data to create clear, insightful comparison reports directly within the database engine. I also learned the practical use of a VIEW as a "virtual table" to simplify complex queries for end-users.

Finally, this entire process was wrapped in SQL best practices. I consistently used TRANSACTIONS to ensure atomicity, rolling back changes if errors occurred. I employed SET NOCOUNT ON for optimization and utilized Common Table Expressions (CTEs) and temporary tables to structure complex queries, making them more readable and maintainable.

So in conclusion, this homework has many lots of useful exercises that a little bit help me imagine how to bridge the gap between theoretical database concepts and practical, real-world applications. I have moved from simply querying data to engineering it, automating its processing, and deriving complex analytical insights directly from the database itself, though it still consumes errors, and optimization.

Thank you for reviewing the whole of my work!