**DBI202 – LEARNING CENTER MANAGEMENT SYSTEM**

**Lecturer:** DO THI THU NGA

**Class:** AI2003

**Group**: 2

**Authors:**

Le anh duc | HE200666

tran xuan tung | HE200683

tran minh dung | HE200667

nguyen tien dat | HE199203

vu duc minh | HE201162

**TABLE OF CONTENTS**

[I. Problem Statement 3](#_Toc204075563)

[1. Current State 3](#_Toc204075564)

[2. Objectives 3](#_Toc204075565)

[II. Functional Specification 4](#_Toc204075566)

[1. Table-level Operations 4](#_Toc204075567)

[2. System Operational Logic 4](#_Toc204075568)

[a. Course and Document Management: 4](#_Toc204075569)

[b. Classroom and Teacher Management: 5](#_Toc204075570)

[c. Student Management and Enrollment: 5](#_Toc204075571)

[d. Examination and Grading Operations: 5](#_Toc204075572)

[e. Tuition Payment Procedure: 6](#_Toc204075573)

[3. Attribute Definition 6](#_Toc204075574)

[III. ER model of the system 7](#_Toc204075575)

[IV. Data Dictionary 7](#_Toc204075576)

[1. Teacher 7](#_Toc204075577)

[2. Student 8](#_Toc204075578)

[3. Course 9](#_Toc204075579)

[4. Course\_Material 10](#_Toc204075580)

[5. Class 10](#_Toc204075581)

[6. Class\_Student 11](#_Toc204075582)

[7. Exam 11](#_Toc204075583)

[8. Exam\_Result 12](#_Toc204075584)

[9. Payment 12](#_Toc204075585)

[V. Physical Implementation 13](#_Toc204075586)

[1. Views 13](#_Toc204075587)

[a. V\_StudentContactInfo 13](#_Toc204075588)

[b. V\_Class\_Details 13](#_Toc204075589)

[c. V\_Student\_Grades 13](#_Toc204075590)

[d. V\_Course\_Summary 13](#_Toc204075591)

[e. V\_Teacher\_Workload 13](#_Toc204075592)

[f. V\_StudentClassStats 13](#_Toc204075593)

[g. V\_Student\_Enrollments 13](#_Toc204075594)

[h. V\_Payment\_History 13](#_Toc204075595)

[i. V\_LatestExamResults 13](#_Toc204075596)

[2. Triggers/ Procedures/ Functions 14](#_Toc204075597)

[2.1. Functions 14](#_Toc204075598)

[a. fn\_GetStudentFullName 14](#_Toc204075599)

[b. fn\_CalculateStudentAge 14](#_Toc204075600)

[c. Fn\_GetClassesByTeacher 14](#_Toc204075601)

[d. fn\_CanPayForClas 14](#_Toc204075602)

[e. fn\_GetStudentClasses 14](#_Toc204075603)

[f. fn\_GetClassStudents 14](#_Toc204075604)

[g. fn\_CourseAverageGrade 14](#_Toc204075605)

[h. fn\_IsStudentEnrolledCourse 14](#_Toc204075606)

[2.2. Procedures 14](#_Toc204075607)

[a. usp\_GetStudentEnrollments 14](#_Toc204075608)

[b. usp\_UpdateStudentBalance 14](#_Toc204075609)

[c. usp\_ProcessCoursePayment 14](#_Toc204075610)

[d. usp\_GetStudentsByCourseAndBalance 14](#_Toc204075611)

[e. usp\_GetCoursePaymentSummary 14](#_Toc204075612)

[f. usp\_TransferStudentBalance 14](#_Toc204075613)

[g. usp\_GetStudentsWithHighestTotalPayment 14](#_Toc204075614)

[h. usp\_VerifyUserLogin 14](#_Toc204075615)

[2.3. Triggers 15](#_Toc204075616)

[a. trg\_UpdateCourseLastModified 15](#_Toc204075617)

[b. trg\_LogStudent (Creation, Update, Deletion) 15](#_Toc204075618)

[c. trg\_SetInitialStudentBalance 15](#_Toc204075619)

[d. trg\_DeleteExamsOnClassDelete 15](#_Toc204075620)

[e. trg\_HashPassword 15](#_Toc204075621)

[3. SQL Queries 16](#_Toc204075622)

[a. Find classes with the same teacher. 16](#_Toc204075623)

[b. Reports student performance in a specific course. 16](#_Toc204075624)

[c. Top 2 classes with the highest and lowest average scores. 17](#_Toc204075625)

[d. Teacher performance summary 17](#_Toc204075626)

[e. Find students who scored above the class average 18](#_Toc204075627)

[f. List high-rerforming and active students 18](#_Toc204075628)

[g. Teacher's star student report 18](#_Toc204075629)

[h. Student loyalty and course path analysis 19](#_Toc204075630)

[i. Course profitability ranking 19](#_Toc204075631)

[j. At-risk student tdentification 20](#_Toc204075632)

[k. Exam difficulty analysis 20](#_Toc204075633)

[l. Student academic journey & teacher influence 21](#_Toc204075634)

[VI. Conclusion 22](#_Toc204075635)

# Problem Statement

## Current State

In the development of educational centers, especially foreign language centers, information management has become increasingly complex and requires a lot of accuracy and consistency. However, many centers today are still using manual management methods such as Excel spreadsheets, paper records or separate software that is not integrated with each other. This leads to a series of problems:

* Student information is scattered, difficult to control, and prone to loss or distortion of data such as scores, class schedules, or tuition payment status.
* Scheduling, assigning classes, and coordinating instructors become more complicated as the number of classes and students increases, leading to overlapping hours, empty classrooms, or teacher overload.
* Asynchronous teacher management causes tracking of teaching information, student feedback, or teaching quality to be missed or performed ineffectively.
* Difficulty in updating, organizing courses, curriculum, and accompanying documents.
* Collecting tuition fees and tracking finances largely by hand makes reporting time-consuming, error-prone, and lacking transparency.
* Difficulty in communication between teachers and students, leading to a situation where some students do not receive information.
* Manual systems can’t meet the needs as the center expands new learning programs.
* From these practical issues, building a centralized and effective management system is needed. EducateDB is proposed as a solution to better automate the management process of the center, reduces administrative work, increases data accuracy and improves the experience for both students and staffs in the center.

## Objectives

The main goal of the project is to build a good database system capable of managing the operations of a learning center. We need to:

* Design and implement an database system that supports the management of all important components of the center: students, teachers, classes, courses, schedules, finances.
* Manage student information including data, class schedules, grades, and tuition status.
* Manage teacher information such as personal profiles, teaching schedules, class assignments, and student feedback.
* Organize and manage courses at each level, including curriculum, materials, and entry requirements.
* Automate class placement, teaching assignments, and student progress tracking.
* Record and analyze scores and attendance in a transparent and easy-to-reference manner.
* Manage tuition collection, monitor and report on financial status accurately and promptly.
* Facilitate student reporting, activity reporting and administrative statistics easily for the center's management board.
* Minimize dependence on manual operations, minimizing unnecessary errors.
* Aim to build a scalable system, ready to meet the needs when the center grows in the number of classes, subjects, or new branches in the future.

# Functional Specification

## Table-level Operations

* **Student, teacher**: Has attributes such as id, name, age, and personal account to manage personal information of students/ teachers.
* **course, course\_material**: Manage courses infomation and teaching materials needed for those courses.
* **class, class\_student**: Manage enrollment and data from each student and their class.
* **exam, grade**: Manage information about scores and assignments/tests.
* **payment:** Manage accounting information/ tuition of each student.

## System Operational Logic

### Course and Document Management:

**Description**: The center needs to create and manage a list of English courses (eg IELTS, TOEIC). Each course has detailed information such as description, tuition, and can be updated. Each course comes with specific learning materials (books, videos, audio files).

**In the database:**

* The Course table stores information about each course.
* The Course\_Material table stores the materials, and the course\_id column in this table creates a 1-Many relationship: a Course can have many Course\_Materials.

### Classroom and Teacher Management:

**Description**: To implement a course, the center opens specific classes. Each class is a version of a course, has its own schedule, room, start/end date, and is taught by a teacher. One teacher can teach many different classes.

**In the database:**

* The Teacher table stores their personal information.
* The Class table stores information about classes. The course\_id column associates a class with a Course (1-Many relationship from Course to Class), and the teacher\_id column associates a class with a Teacher (1-Many relationship from Teacher to Class).

### Student Management and Enrollment:

**Description**: Student information management center. Students can register (enroll) in many different classes, and a class can also have many students participating.

**In the database:**

* The Student table stores their personal information.
* The Many-To-Many relationship between Student and Class is resolved using the intermediate table Class\_Student. Each row in this table is an enrollment record, showing which students are enrolled in which classes.

### Examination and Grading Operations:

**Description**: During the course, a class will have tests (midterm, final, quiz). After each exam, each participating student will receive a corresponding score.

**In the database:**

* The Exam table stores exam information and is linked to a Class (1-Many relationship from Class to Exam).
* The Grade table stores scores of each student during a given test. It has a Many-to-1 relationship with both Student and Exam, meaning a score must belong to a specific student and a specific exam.

### Tuition Payment Procedure:

**Description**: Students make tuition payments for the classes they enroll in. The system needs to record these payments, including amount, method, status, and associate the payment with the corresponding student and classes.

**In the database:**

* The student\_id and class\_id columns in this table create a Many-to-1 relationship with Student and Class, indicating which student paid for which class, and thus, which course.

## Attribute Definition

* Teacher: **id (PK)**, first\_name, last\_name, date\_birth, gender, email, phone, address, city, description, user\_name, password
* Student: **id (PK**), first\_name , last\_name, date\_birth, gender, email, phone, address, city, user\_name, password, balance, created\_date
* Course: **id (PK)**, description, last\_modified
* Course\_Material: **id (PK)**, **course\_id (FK),** description, material\_type, material\_url, date\_add
* Class: **id (PK)**, start\_date, end\_date, **teacher\_id (FK), course\_id (FK)**, schedule\_info, room\_number, tuition\_fee
* Class\_Student**: class\_id (PK, FK)**, **student\_id (PK, FK),** enrollment\_date
* Exam: **id (PK)**, date, description, **class\_id (FK)**, exam\_type, duration\_minutes
* Exam\_Result: **exam\_id (FK, PK), student\_id (FK, PK),** value, date
* Payment: **id (PK)**, payment\_date, amount , status, **student\_id (FK), class\_id (FK)**, payment\_method, notes

# A diagram of a flowchartER model of the system

# Data Dictionary

## Teacher

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Check** | **Key/Index/Constraint** |
| id | Varchar(5) | LIKE 'TE[0-9][0-9][0-9]' | PK, Not null |
| first\_name | Nvarchar(50) |  | Not null |
| last\_name | Nvarchar(50) |  | Not null |
| date\_birth | Date |  |  |
| gender | Nvarchar(3) | IN (N'Nam', N'Nữ') |  |
| e-mail | Varchar(100) | LIKE '%\_@\_\_%.\_\_%' | Unique, Not null |
| phone | Varchar(20) |  |  |
| address | Nvarchar(255) |  |  |
| city | Nvarchar(50) |  |  |
| description | Nvarchar(255) |  |  |
| user\_name | Varchar(50) |  | Unique, Not null |
| password | Varchar(255) |  | Not null |

* **SQL:**

CREATE TABLE Teacher (

id VARCHAR(5) PRIMARY KEY,

first\_name NVARCHAR(50) NOT NULL,

last\_name NVARCHAR(50) NOT NULL,

date\_birth DATE,

gender NVARCHAR(3),

email VARCHAR(100) UNIQUE NOT NULL,

phone VARCHAR(20),

address NVARCHAR(255),

city NVARCHAR(50) DEFAULT N'Hà Nội',

description NVARCHAR(255),

user\_name VARCHAR(50) UNIQUE NOT NULL,

password VARCHAR(255) NOT NULL,

CONSTRAINT CK\_Teacher\_ID CHECK (id LIKE 'TE[0-9][0-9][0-9]'),

CONSTRAINT CK\_Teacher\_Gender CHECK (gender IN (N'Nam', N'Nữ')),

CONSTRAINT CK\_Teacher\_Email CHECK (email LIKE '%\_@\_\_%.\_\_%')

## Student

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Check** | **Key/Index/Constraint** |
| id | Varchar(5) | LIKE 'ST[0-9][0-9][0-9]' | PK, Not null |
| first\_name | Nvarchar(50) |  | Not null |
| last\_name | Nvarchar(50) |  | Not null |
| date\_birth | Date |  |  |
| gender | Nvarchar(3) | IN (N'Nam', N'Nữ') |  |
| e-mail | Varchar(100) | LIKE '%\_@\_\_%.\_\_%' | Unique, Not null |
| phone | Varchar(20) |  |  |
| address | Nvarchar(255) |  |  |
| city | Nvarchar(50) |  |  |
| user\_name | Varchar(50) |  | Unique, Not null |
| password | Varchar(255) |  | Not null |
| balance | Decimal(12,2) | >= 0 |  |
| created\_date | Date |  |  |

* **SQL:**

CREATE TABLE Student (

id VARCHAR(5) PRIMARY KEY,

first\_name NVARCHAR(50) NOT NULL,

last\_name NVARCHAR(50) NOT NULL,

date\_birth DATE,

gender NVARCHAR(3),

email VARCHAR(100) UNIQUE NOT NULL,

phone VARCHAR(20),

address NVARCHAR(255),

city NVARCHAR(50) DEFAULT N'Hà Nội',

user\_name VARCHAR(50) UNIQUE NOT NULL,

password VARCHAR(255) NOT NULL,

balance DECIMAL(12,2),

created\_date DATE,

CONSTRAINT CK\_Student\_ID CHECK (id LIKE 'ST[0-9][0-9][0-9]'),

CONSTRAINT CK\_Student\_Gender CHECK (gender IN (N'Nam', N'Nữ')),

CONSTRAINT CK\_Student\_Email CHECK (email LIKE '%\_@\_\_%.\_\_%'),

CONSTRAINT CK\_Student\_Balance CHECK (balance >= 0))

## Course

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Check** | **Key/Index/Constraint** |
| id | Nvarchar(50) |  | PK, Not null |
| description | Nvarchar(MAX) |  |  |
| last\_modified | Datetime2 |  |  |

* **SQL:**

CREATE TABLE Course(

id NVARCHAR(50) PRIMARY KEY,

description NVARCHAR(MAX),

last\_modified DATETIME2,

tuition\_fee DECIMAL(12,2),

CONSTRAINT CK\_Course\_TuitionFee CHECK (tuition\_fee >= 0)

);

## Course\_Material

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Check** | **Key/Index/Constraint** |
| id | Varchar(5) | LIKE 'CM[0-9][0-9][0-9]' | PK, Not null |
| course\_id | Nvarchar(50) |  | FK references Course(id), Not null |
| description | Nvarchar(MAX) |  |  |
| material\_type | Nvarchar(50) | IN (N'Sách giáo trình', N'Sách bài tập', N'Tệp âm thanh', N'Video Links', N'Học liệu', N'Sách luyện đề', N'Tài liệu tham khảo') |  |
| material\_url | Varchar(255) |  |  |
| date\_add | Date |  |  |

* **SQL:**

CREATE TABLE Course\_Material (

id VARCHAR(5) PRIMARY KEY,

course\_id NVARCHAR(50) NOT NULL,

description NVARCHAR(MAX),

material\_type NVARCHAR(50),

material\_url VARCHAR(255),

date\_add DATE,

FOREIGN KEY (course\_id) REFERENCES Course(id),

CONSTRAINT CK\_Course\_Material\_ID CHECK (id LIKE 'CM[0-9][0-9][0-9]'),

CONSTRAINT CK\_Course\_Material\_Type CHECK (material\_type IN (N'Sách giáo trình', N'Sách bài tập', N'Sách từ vựng', N'Tệp âm thanh', N'Video Links', N'Học liệu', N'Sách luyện đề', N'Tài liệu tham khảo')));

## Class

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Check** | **Key/Index/Constraint** |
| id | Nvarchar(20) |  | PK, Not null |
| start\_date | Date |  |  |
| end\_date | Date | >= start\_date |  |
| teacher\_id | Varchar(5) |  | FK references Teacher(id) |
| course\_id | Nvarchar(50) |  | FK references Course(id), Not null |
| schedule\_info | Nvarchar(100) |  |  |
| room\_number | Nvarchar(20) | LIKE 'P[1-3][0-9][0-9]' |  |

|  |  |  |  |
| --- | --- | --- | --- |
| tuition\_fee | Decimal(12,2) | >= 0 |  |

* **SQL:**

CREATE TABLE Class (

id NVARCHAR(20) PRIMARY KEY,

start\_date DATE,

end\_date DATE,

teacher\_id VARCHAR(5),

course\_id NVARCHAR(50) NOT NULL,

schedule\_info NVARCHAR(100),

room\_number NVARCHAR(20),

tuition\_fee DECIMAL(12,2) NOT NULL,

FOREIGN KEY (teacher\_id) REFERENCES Teacher(id),

FOREIGN KEY (course\_id) REFERENCES Course(id),

CONSTRAINT CK\_Course\_TuitionFee CHECK (tuition\_fee >= 0),

CONSTRAINT CK\_Class\_Dates CHECK (end\_date >= start\_date),

CONSTRAINT CK\_Class\_RoomNumber CHECK (room\_number LIKE 'P[1-3][0-9][0-9]'));

## Class\_Student

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Check** | **Key/Index/Constraint** |
| class\_id | Nvarchar(20) |  | PK, FK references Class(id), Not null |
| student\_id | Varchar(5) |  | PK, FK references Student(id), Not null |
| enrollment\_date | Date |  |  |

* **SQL:**

CREATE TABLE Class\_Student (

class\_id NVARCHAR(20) NOT NULL,

student\_id VARCHAR(5) NOT NULL,

enrollment\_date DATE,

PRIMARY KEY (class\_id, student\_id),

FOREIGN KEY (class\_id) REFERENCES Class(id),

FOREIGN KEY (student\_id) REFERENCES Student(id))

## Exam

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Check** | **Key/Index/Constraint** |
| id | Varchar(5) | LIKE 'EX[0-9][0-9][0-9]' | PK, Not null |
| date | Date |  |  |
| description | Nvarchar(MAX) |  |  |
| class\_id | Nvarchar(20) |  | FK references Class(id), Not null |
| exam\_type | Nvarchar(50) | IN (N'Midterm', N'Final', N'Quiz', N'Mock Test', N'Speaking Test') |  |
| duration\_minutes | Int | > 0 |  |

* **SQL:**

CREATE TABLE Exam (

id VARCHAR(5) PRIMARY KEY,

date DATE,

description NVARCHAR(MAX),

class\_id NVARCHAR(20) NOT NULL,

exam\_type NVARCHAR(50),

duration\_minutes INT,

FOREIGN KEY (class\_id) REFERENCES Class(id),

CONSTRAINT CK\_Exam\_ID CHECK (id LIKE 'EX[0-9][0-9][0-9]'),

CONSTRAINT CK\_Exam\_Type CHECK (exam\_type IN (N'Midterm', N'Final', N'Quiz', N'Mock Test', N'Speaking Test')),

CONSTRAINT CK\_Exam\_Duration CHECK (duration\_minutes > 0))

## Exam\_Result

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Check** | **Key/Index/Constraint** |
| student\_id | Varchar(5) |  | PK, FK references Student(id), Not null |
| exam\_id | Varchar(5) |  | PK, FK references Exam(id), Not null |
| value | Decimal(4,2) | [0, 10] | Not null |
| date | Date |  |  |

* **SQL:**

CREATE TABLE Exam\_Result (

student\_id VARCHAR(5) NOT NULL,

exam\_id VARCHAR(5) NOT NULL,

value DECIMAL(4,2) NOT NULL,

date DATE,

PRIMARY KEY (student\_id, exam\_id),

FOREIGN KEY (student\_id) REFERENCES Student(id),

FOREIGN KEY (exam\_id) REFERENCES Exam(id),

CONSTRAINT CK\_Grade\_Value CHECK (value >= 0.00 AND value <= 10.00));

## Payment

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Check** | **Key/Index/Constraint** |
| id | Varchar(5) | LIKE 'PA[0-9][0-9][0-9]' | PK, Not null |
| payment\_date | Date |  |  |
| amount | Decimal(12,2) |  | > 0, Not null |
| status | Nvarchar(20) | IN (N'Success', N'Failed') | Not null |
| student\_id | Varchar(5) |  | FK references Student(id), Not null |
| class\_id | Nvarchar(50) |  | FK references Class(id), Not null |
| payment\_method | Nvarchar(50) | IN (N'Tiền mặt', N'Chuyển khoản', N'Thẻ tín dụng', N'Momo') |  |
| notes | Nvarchar(255) |  |  |

* **SQL:**

CREATE TABLE Payment (

id VARCHAR(5) PRIMARY KEY,

payment\_date DATE,

amount DECIMAL(12,2) NOT NULL,

status NVARCHAR(20) NOT NULL,

student\_id VARCHAR(5) NOT NULL,

course\_id NVARCHAR(50) NOT NULL,

payment\_method NVARCHAR(50),

notes NVARCHAR(255),

FOREIGN KEY (student\_id) REFERENCES Student(id),

FOREIGN KEY (course\_id) REFERENCES Course(id),

CONSTRAINT CK\_Payment\_ID CHECK (id LIKE 'PA[0-9][0-9][0-9]'),

CONSTRAINT CK\_Payment\_Amount CHECK (amount > 0),

CONSTRAINT CK\_Payment\_Status CHECK (status IN (N'Success', N'Failed')),

CONSTRAINT CK\_Payment\_Method CHECK (payment\_method IN (N'Tiền mặt', N'Chuyển khoản', N'Thẻ tín dụng', N'Momo'))

# Physical Implementation

## Views

### V\_StudentContactInfo

**Purpose:** To get personal and contact details for every student into a single, easy-to-query source, separating it from academic or financial data.

**Usecase:** This view is essential for administrative staff to quickly retrieve contact information for communication.

CREATE VIEW V\_StudentContactInfo

AS

SELECT

id AS StudentID,

last\_name + ' ' + first\_name AS FullName,

date\_birth AS DateOfBirth,

gender AS Gender,

email AS EmailAddress,

phone AS PhoneNumber,

address AS HomeAddress,

city AS City

FROM

Student;

GO

### V\_Class\_Details

* **Purpose:** To provide an easy-to-read, detailed overview of each class's information.
* **Usecase:** Helps counselors, academic administrators, or teachers quickly look up information about a class without having to write complex JOIN statements between the Class, Course, and Teacher tables.

CREATE VIEW V\_Class\_Details AS

SELECT

cl.id AS ClassID,

cl.schedule\_info AS Schedule,

cl.room\_number AS Room,

co.id AS CourseID,

co.description AS CourseDescription,

t.last\_name + N' ' + t.first\_name AS TeacherFullName,

cl.tuition\_fee AS TuitionFee

FROM Class cl

JOIN Course co ON cl.course\_id = co.id

LEFT JOIN Teacher t ON cl.teacher\_id = t.id

### V\_Student\_Grades

* **Purpose:** Generate a detailed student score report, combining information from the student, exam, and score.
* **Usecase:** Useful for teachers and academic departments when they need to view or export student transcripts.

CREATE VIEW V\_Student\_Grades AS

SELECT

s.id AS StudentID,

s.last\_name + N' ' + s.first\_name AS StudentFullName,

e.description AS ExamDescription,

e.exam\_type AS ExamType,

e.date AS ExamDate,

g.value AS GradeValue

FROM Exam\_Result g

JOIN Student s ON g.student\_id = s.id

JOIN Exam e ON g.exam\_id = e.id;

### V\_Course\_Summary

* **Purpose:** Provide a summary report on the status of each course, including key metrics such as number of classes and number of students.
* **Usecase**: Helps manage the center to evaluate the effectiveness and popularity of the courses, making business decisions such as opening more classes, or changing tuition fees.

CREATE VIEW V\_Course\_Summary AS

SELECT

c.id AS CourseID,

c.description AS CourseDescription,

MIN(cl.tuition\_fee) AS MinTuitionFee,

MAX(cl.tuition\_fee) AS MaxTuitionFee,

AVG(cl.tuition\_fee) AS AvgTuitionFee,

COUNT(DISTINCT cl.id) AS NumberOfClasses,

COUNT(DISTINCT cs.student\_id) AS TotalEnrollments

FROM Course c

LEFT JOIN Class cl ON c.id = cl.course\_id

LEFT JOIN Class\_Student cs ON cl.id = cs.class\_id

GROUP BY c.id, c.description;

### V\_Teacher\_Workload

* **Purpose:** To aggregate and display the workload of each teacher.
* **Usecase:** Based on this View, we can quickly know which teachers are teaching multiple classes, which teachers have free schedules, or which teachers are teaching different courses to plan assignments for the next semesters.

CREATE VIEW V\_Teacher\_Workload AS

SELECT

t.id AS TeacherID,

t.last\_name + N' ' + t.first\_name AS TeacherFullName,

ISNULL(COUNT(DISTINCT cl.id), 0) AS AssignedClasses,

ISNULL(COUNT(DISTINCT cl.course\_id), 0) AS DistinctCoursesTaught

FROM Teacher t

LEFT JOIN Class cl ON t.id = cl.teacher\_id

GROUP BY t.id, t.first\_name, t.last\_name

### V\_StudentClassStats

**Purpose**: To provide a summary of each student's academic activity and performance.

**Usecase**: Used to create student leaderboards, identify students who may need academic support.

CREATE VIEW V\_StudentClassStats AS

SELECT s.id, s.first\_name + ' ' + s.last\_name AS full\_name,

COUNT(DISTINCT cs.class\_id) AS class\_count,

AVG(g.value) AS avg\_grade

FROM Student s

LEFT JOIN Class\_Student cs ON s.id = cs.student\_id

LEFT JOIN Exam\_Result g ON s.id = g.student\_id

GROUP BY s.id, s.first\_name, s.last\_name;

### V\_Student\_Enrollments

**Purpose**: Display a detailed list of every class that each student is enrolled in.

**Usecase**: A student can quickly see their upcoming classes, times, and locations.

CREATE VIEW V\_Student\_Enrollments AS

SELECT

s.id AS StudentID,

s.last\_name + N' ' + s.first\_name AS StudentFullName,

co.description AS Course,

cl.schedule\_info AS Schedule,

cl.room\_number AS Room

FROM Class\_Student cs

JOIN Student s ON cs.student\_id = s.id

JOIN Class cl ON cs.class\_id = cl.id

JOIN Course co ON cl.course\_id = co.id;

### V\_Payment\_History

**Purpose**: To create a history of all payment transactions, linking each payment to the student who made it.

**Usecase**: This can be used by the accounting department to generate reports or by students to view their personal payment history.

CREATE VIEW V\_Payment\_History AS

SELECT

p.id AS PaymentID,

s.id AS StudentID,

s.last\_name + N' ' + s.first\_name AS StudentFullName,

p.amount AS AmountPaid,

p.payment\_date AS PaymentDate

FROM Payment p

JOIN Student s ON p.student\_id = s.id

### V\_LatestExamResults

**Purpose:** To create a list of every exam result, linking each score to the corresponding student, exam, class, and course information.

**Usecase:** Teachers or advisors can use it to track a specific student's complete performance history across all subjects.

CREATE VIEW V\_LatestExamResults

AS

SELECT

S.id AS StudentID,

S.first\_name + ' ' + S.last\_name AS StudentName,

E.id AS ExamID,

E.exam\_type AS ExamType,

E.description AS ExamDescription,

ER.value AS Score,

ER.date AS ResultDate,

C.id AS ClassID,

Co.description AS CourseName

FROM Exam\_Result AS ER

JOIN Student AS S ON ER.student\_id = S.id

JOIN Exam AS E ON ER.exam\_id = E.id

JOIN Class AS C ON E.class\_id = C.id

JOIN Course AS Co ON C.course\_id = Co.id;

## Triggers/ Procedures/ Functions

### Functions

#### fn\_GetStudentFullName

* **Purpose:** Takes in StudentID and returns the student's full name in "Full Name" format (eg: "Nguyen Van An").
* **Usecase:** Ensure consistency in name display across the system and simplify queries.

CREATE FUNCTION fn\_GetStudentFullName (@StudentID VARCHAR(5))

RETURNS NVARCHAR(101)

AS

BEGIN

DECLARE @FullName NVARCHAR(101);

SELECT @FullName = last\_name + N' ' + first\_name

FROM Student WHERE id = @StudentID;

RETURN @FullName;

END

#### fn\_CalculateStudentAge

* **Purpose:** Calculate a person's current age based on their date of birth.
* **Usecase:** Useful when needing to create demographic statistics reports or filter data by age.

CREATE FUNCTION dbo.fn\_CalculateStudentAge (@DateOfBirth DATE)

RETURNS INT

AS

BEGIN

RETURN DATEDIFF(YEAR, @DateOfBirth, GETDATE()) -

CASE WHEN (MONTH(@DateOfBirth) > MONTH(GETDATE())) OR

(MONTH(@DateOfBirth) = MONTH(GETDATE()) AND

DAY(@DateOfBirth) > DAY(GETDATE())) THEN 1 ELSE 0 END;

END

#### Fn\_GetClassesByTeacher

* **Purpose:** Get a detailed list of classes that a particular teacher is teaching.
* **Usecase:** Provide flexible information for teachers and administrators to view and analyze each teacher's class list.

CREATE FUNCTION dbo.fn\_GetClassesByTeacher (@TeacherID VARCHAR(5))

RETURNS TABLE

AS

RETURN

(

SELECT

cl.id AS ClassID,

cl.schedule\_info AS Schedule,

co.description AS CourseDescription

FROM Class cl

JOIN Course co ON cl.course\_id = co.id

WHERE cl.teacher\_id = @TeacherID);

#### fn\_CanPayForClas

* **Purpose**: Takes a StudentID and a ClassID to determine if a student's current balance is sufficient to cover the class’s tuition fee.
* **Usecase**: It allows the system to instantly check if a payment is possible before the student enrolls.

CREATE FUNCTION fn\_CanPayForClass(@StudentID VARCHAR(5), @ClassID NVARCHAR(20))

RETURNS NVARCHAR(20)

AS

BEGIN

DECLARE @balance DECIMAL(12,2);

DECLARE @fee DECIMAL(12,2);

DECLARE @result NVARCHAR(20);

SELECT @balance = balance FROM Student WHERE id = @StudentID;

SELECT @fee = tuition\_fee FROM Class WHERE id = @ClassID;

IF @balance IS NULL OR @fee IS NULL SET @result = N'Not Affordable'

ELSE IF @balance >= @fee SET @result = N'Affordable'

ELSE SET @result = N'Not Affordable';

RETURN @result;

END

#### fn\_GetStudentClasses

**Purpose**: Retrieves a detailed table of all specific classes a given StudentID is enrolled in, including the class ID, schedule, and room number.

**Usecase:** Essential for building a student's personal dashboard or "My Schedule" page, providing them with the exact information needed to attend their classes.

CREATE FUNCTION fn\_GetStudentClasses(@StudentID VARCHAR(5))

RETURNS TABLE

AS RETURN

(

SELECT cs.class\_id, c.course\_id, c.schedule\_info, c.room\_number

FROM Class\_Student cs

JOIN Class c ON cs.class\_id = c.id

WHERE cs.student\_id = @StudentID)

GO

#### fn\_GetClassStudents

**Purpose**: Takes a ClassID and returns a complete list of all students enrolled in that specific class, including their names and enrollment dates.

**Usecase**: Used to generate a class roster for taking attendance or to check the current enrollment count for a specific class.

CREATE FUNCTION fn\_GetClassStudents(@ClassID NVARCHAR(20))

RETURNS TABLE

AS

RETURN

(

SELECT s.id, s.first\_name, s.last\_name, cs.enrollment\_date

FROM Class\_Student cs

JOIN Student s ON cs.student\_id = s.id

WHERE cs.class\_id = @ClassID)

GO

#### fn\_CourseAverageGrade

**Purpose**: Calculates the overall average grade for a specific CourseID by aggregating the grades from all students across all classes and exams related to that course.

**Usecase**: Serves as a high-level analytical tool for academic managers to assess course difficulty and overall student performance in a subject, helping to identify trends or areas needing curriculum improvement.

CREATE FUNCTION fn\_CourseAverageGrade(@CourseID NVARCHAR(50))

RETURNS DECIMAL(5,2)

AS

BEGIN

DECLARE @avg DECIMAL(5,2);

SELECT @avg = AVG(er.value) FROM Exam\_Result er

JOIN Exam e ON er.exam\_id = e.id

JOIN Class c ON e.class\_id = c.id

WHERE c.course\_id = @CourseID;

RETURN @avg;

END

#### fn\_IsStudentEnrolledCourse

**Purpose**: Takes a StudentID and a CourseID and returns 'Yes' or 'No' to indicate if the student is currently enrolled in any class belonging to that course.

**Usecase**: Acts as a quick validation check. It can be used to prevent a student from accidentally enrolling in the same course twice or to control access to course-specific online content.

CREATE FUNCTION fn\_IsStudentEnrolledCourse(

@StudentID VARCHAR(5), @CourseID NVARCHAR(50))

RETURNS NVARCHAR(3)

AS BEGIN

DECLARE @result NVARCHAR(3);

IF EXISTS (

SELECT 1

FROM Class\_Student cs

JOIN Class c ON cs.class\_id = c.id

WHERE cs.student\_id = @StudentID AND c.course\_id = @CourseID

)

SET @result = N'Yes'

ELSE

SET @result = N'No'

RETURN @result

END

### Procedures

#### usp\_GetStudentEnrollments

* **Purpose:** Get a list of all classes that a particular student is enrolled in.
* **Usecase:** Helps student portal or counselors quickly look up a student's class schedule.

CREATE PROCEDURE usp\_GetStudentEnrollments @StudentID VARCHAR(5)

AS

BEGIN

SELECT ClassID, Schedule, CourseDescription, TeacherFullName

FROM V\_Class\_Details

WHERE ClassID IN (SELECT class\_id FROM Class\_Student

WHERE student\_id = @StudentID);

END

#### usp\_UpdateStudentBalance

* **Purpose:** To provide a secure way to fund student accounts.
* **Usecase:** Assist accounting staff in managing student balances, ensuring accuracy in financial transactions.

CREATE PROCEDURE usp\_UpdateStudentBalance

@StudentID VARCHAR(5),

@AmountToAdd DECIMAL(12,2)

AS

BEGIN

DECLARE @NewBalance DECIMAL(12,2);

IF NOT EXISTS (SELECT 1 FROM Student WHERE id = @StudentID)

BEGIN

PRINT N'Lỗi: Không tìm thấy sinh viên với ID ' + @StudentID;

RETURN;

END

UPDATE Student SET balance = ISNULL(balance, 0) + @AmountToAdd WHERE id = @StudentID;

SELECT @NewBalance = balance FROM Student WHERE id = @StudentID;

PRINT N'Đã cập nhật số dư cho sinh viên ' + @StudentID + N'. Số dư mới: ' + CAST(@NewBalance AS VARCHAR);

END

#### usp\_ProcessCoursePayment

* **Purpose:** Encapsulate the entire complex business logic of paying tuition for a course.
* **Usecase:** Very important for the finance department, helping to process payment transactions consistently, check balances and record transaction status.

CREATE OR ALTER PROCEDURE usp\_ProcessCoursePayment

@PaymentID VARCHAR(5),

@StudentID VARCHAR(5),

@ClassID NVARCHAR(20),

@PaymentAmount DECIMAL(12,2),

@PaymentMethod NVARCHAR(50) = NULL,

@TransactionNotes NVARCHAR(255) = NULL

AS

BEGIN

SET NOCOUNT ON;

IF NOT EXISTS (SELECT 1 FROM Class WHERE id = @ClassID)

BEGIN

PRINT N'Giao dịch thanh toán ' + @PaymentID + N' cho sinh viên ' + @StudentID + N' - Trạng thái: Failed. Lỗi: ClassID ''' + @ClassID + ''' không tồn tại.';

RETURN

END

IF NOT EXISTS (SELECT 1 FROM Student WHERE id = @StudentID)

BEGIN

PRINT N'Giao dịch thanh toán ' + @PaymentID + N' cho sinh viên ' + @StudentID + N' - Trạng thái: Failed. Lỗi: StudentID ''' + @StudentID + ''' không tồn tại.';

RETURN

END

DECLARE @StudentBalance DECIMAL(12,2), @ClassTuition DECIMAL(12,2), @PaymentStatus NVARCHAR(20);

DECLARE @CurrentPaymentDate DATE = GETDATE();

DECLARE @FinalNotes NVARCHAR(MAX) = ISNULL(@TransactionNotes, N'');

SELECT @ClassTuition = tuition\_fee FROM Class WHERE id = @ClassID;

SELECT @StudentBalance = ISNULL(balance, 0) FROM Student WHERE id = @StudentID;

IF @StudentBalance < @PaymentAmount

BEGIN

SET @PaymentStatus = N'Failed';

SET @FinalNotes = @FinalNotes + N'; Lý do: Số dư không đủ.';

END

ELSE IF @PaymentAmount <> @ClassTuition

BEGIN

SET @PaymentStatus = N'Failed';

SET @FinalNotes = @FinalNotes + N'; Lý do: Số tiền thanh toán không khớp học phí.';

END

ELSE

BEGIN

BEGIN TRANSACTION;

BEGIN TRY

UPDATE Student

SET balance = balance - @PaymentAmount

WHERE id = @StudentID;

INSERT INTO Class\_Student (class\_id, student\_id, enrollment\_date)

VALUES (@ClassID, @StudentID, GETDATE());

COMMIT TRANSACTION;

SET @PaymentStatus = N'Success';

SET @FinalNotes = @FinalNotes + N'; Thanh toán và ghi danh thành công.';

END TRY

BEGIN CATCH

ROLLBACK TRANSACTION;

SET @PaymentStatus = N'Failed';

SET @FinalNotes = @FinalNotes + N'; Lỗi hệ thống: ' + ERROR\_MESSAGE();

END CATCH

END

INSERT INTO Payment (id, payment\_date, amount, status, student\_id, class\_id, payment\_method, notes)

VALUES (@PaymentID, @CurrentPaymentDate, @PaymentAmount, @PaymentStatus, @StudentID, @ClassID, @PaymentMethod, LTRIM(STUFF(@FinalNotes, 1, 1, '')));

PRINT N'Giao dịch thanh toán ' + @PaymentID + N' cho sinh viên ' + @StudentID + N' - Trạng thái: ' + @PaymentStatus;

END

#### usp\_GetStudentsByCourseAndBalance

* **Purpose:** To retrieve a list of students who are enrolled in a specific course and meet a minimum balance requirement.
* **Usecase:** Useful for marketing or administrative tasks, such as identifying students in an advanced course who can afford to purchase supplementary materials or to send targeted promotions for upcoming workshops.

CREATE PROCEDURE usp\_GetStudentsByCourseAndBalance

@CourseID NVARCHAR(50),

@MinBalance DECIMAL(12,2)

AS

BEGIN

SELECT S.id AS StudentID, S.last\_name + N' ' + S.first\_name AS StudentFullName, S.balance

FROM Student S

JOIN Class\_Student CS ON S.id = CS.student\_id

JOIN Class CL ON CS.class\_id = CL.id

WHERE CL.course\_id = @CourseID AND S.balance >= @MinBalance

ORDER BY StudentFullName;

END

#### usp\_GetCoursePaymentSummary

* **Purpose:** To generate a financial summary for a specific course, including the total number of payments, total amount paid, average payment, and the date of the most recent payment.
* **Usecase:** An essential tool for the finance department to quickly analyze the revenue and payment activity for any given course. It helps in tracking financial performance and identifying popular or underperforming courses.

CREATE PROCEDURE usp\_GetCoursePaymentSummary @CourseID NVARCHAR(50)

AS

BEGIN

SELECT C.description AS CourseName, COUNT(P.id) AS TotalPayments,

ISNULL(SUM(P.amount), 0) AS TotalAmountPaid,

ISNULL(AVG(P.amount), 0) AS AveragePaymentAmount,

MAX(P.payment\_date) AS LastPaymentDate

FROM Course C

LEFT JOIN Class CL ON C.id = CL.course\_id

LEFT JOIN Payment P ON CL.id = P.class\_id

WHERE C.id = @CourseID

GROUP BY C.description;

END

#### usp\_TransferStudentBalance

* **Purpose:** To transfer a specified amount from one student's balance to another's.
* **Usecase:** Useful for staff to handle cases such as a family member paying for another, correcting a mistaken payment.

CREATE PROCEDURE usp\_TransferStudentBalance

@SenderStudentID VARCHAR(5),

@ReceiverStudentID VARCHAR(5),

@TransferAmount DECIMAL(12,2)

AS

BEGIN

SET NOCOUNT ON;

DECLARE @SenderBalance DECIMAL(12,2);

IF @TransferAmount <= 0

BEGIN

PRINT N'Lỗi: Số tiền chuyển phải lớn hơn 0.';

RETURN;

END;

IF NOT EXISTS (SELECT 1 FROM Student WHERE id = @SenderStudentID)

BEGIN

PRINT N'Lỗi: Sinh viên gửi với ID ' + @SenderStudentID + N' không tồn tại.';

RETURN;

END;

IF NOT EXISTS (SELECT 1 FROM Student WHERE id = @ReceiverStudentID)

BEGIN

PRINT N'Lỗi: Sinh viên nhận với ID ' + @ReceiverStudentID + N' không tồn tại.';

RETURN;

END;

IF @SenderStudentID = @ReceiverStudentID

BEGIN

PRINT N'Lỗi: Không thể chuyển tiền cho chính mình.'

RETURN

END

BEGIN TRY

BEGIN TRANSACTION

SELECT @SenderBalance = ISNULL(balance, 0) FROM Student WHERE id = @SenderStudentID

IF @SenderBalance < @TransferAmount

BEGIN

PRINT N'Lỗi: Số dư của sinh viên ' + @SenderStudentID + N' không đủ để thực hiện giao dịch này.'

ROLLBACK TRANSACTION

RETURN

END;

UPDATE Student

SET balance = balance - @TransferAmount

WHERE id = @SenderStudentID

UPDATE Student

SET balance = ISNULL(balance, 0) + @TransferAmount

WHERE id = @ReceiverStudentID

COMMIT TRANSACTION

PRINT N'Chuyển khoản thành công từ ' + @SenderStudentID + N' sang ' + @ReceiverStudentID + N' số tiền ' + CAST(@TransferAmount AS VARCHAR) + N'.'

END TRY

BEGIN CATCH

ROLLBACK TRANSACTION

PRINT N'Lỗi trong quá trình chuyển khoản: ' + ERROR\_MESSAGE();

THROW;

END CATCH

END

#### usp\_GetStudentsWithHighestTotalPayment

* **Purpose:** To generate a "Top N" leaderboard of students based on their total payment amount across all courses.
* **Usecase:** It can be used to identify high-value students for loyalty programs, special offers.

CREATE PROCEDURE usp\_GetStudentsWithHighestTotalPayment @TopN INT = 5

AS

BEGIN

SET NOCOUNT ON;

SELECT TOP (@TopN)

S.id AS StudentID,

S.last\_name + N' ' + S.first\_name AS StudentFullName,

ISNULL(SUM(P.amount), 0) AS TotalAmountPaid

FROM Student S

JOIN Payment P ON S.id = P.student\_id

GROUP BY S.id, S.first\_name, S.last\_name

ORDER BY TotalAmountPaid DESC

END

#### usp\_VerifyUserLogin

* Purpose: To provide a authentication mechanism for both students and teachers. It validates a user's credentials by comparing a hashed password attempt against the securely stored hash and unique salt in the database, preventing plaintext password exposure.
* Usecase: The backend system will call this procedure, passing the username and password submitted by the user.

CREATE OR ALTER PROCEDURE usp\_VerifyUserLogin

@Username VARCHAR(50),

@PasswordAttempt NVARCHAR(255)

AS

BEGIN

SET NOCOUNT ON;

DECLARE @StoredHash NVARCHAR(128)

DECLARE @UserType NVARCHAR(10)

DECLARE @UserID VARCHAR(5)

SELECT @StoredHash = password, @UserID = id, @UserType = 'Student' FROM Student WHERE user\_name = @Username

IF @UserID IS NULL

BEGIN

SELECT @StoredHash = password, @UserID = id, @UserType = 'Teacher' FROM Teacher WHERE user\_name = @Username

END

IF @StoredHash IS NOT NULL

BEGIN

IF CONVERT(NVARCHAR(128), HASHBYTES('SHA2\_512', @PasswordAttempt + @Username), 2) = @StoredHash

BEGIN

SELECT 'Login Successful' AS Status, @UserID AS UserID, @UserType AS UserType

RETURN

END

END

SELECT 'Invalid Username or Password' AS Status, NULL AS UserID, NULL AS UserType

END

### Triggers

#### trg\_UpdateCourseLastModified

* **Activation table:** Course\_Material
* **Purpose:** Automatically update a course's last\_modified field whenever any information about that course's documents is changed.
* **Practical Operations:** Helps the system always have a trace of the last update related to a course, supporting management and testing.

CREATE TRIGGER trg\_UpdateCourseLastModified ON Course\_Material

AFTER UPDATE

AS

BEGIN

SET NOCOUNT ON;

UPDATE Course SET last\_modified = GETDATE()

WHERE id IN (SELECT course\_id FROM inserted);

END

#### trg\_LogStudent (Creation, Update, Deletion)

* **Activation table:** Student
* **Purpose:** Automatically record important actions (create, update, delete) on student data into a separate AuditLog table.
* **Practical Operations:** Create a basic auditing mechanism, helping administrators track important student data changes

CREATE TRIGGER trg\_LogStudentCreation ON Student

AFTER INSERT

AS

BEGIN

SET NOCOUNT ON;

INSERT INTO AuditLog (TableName, RecordID, ActionType, ChangeDetails)

SELECT 'Student', i.id, 'INSERT', 'A new student was created: ' + i.last\_name + N' ' + i.first\_name

FROM inserted i;

END

GO

CREATE TRIGGER trg\_LogStudentUpdate

ON Student

AFTER UPDATE

AS

BEGIN

SET NOCOUNT ON

IF UPDATE(balance)

BEGIN

INSERT INTO AuditLog (TableName, RecordID, ActionType, ChangeDetails)

SELECT

'Student', i.id, 'UPDATE',

'Balance changed from ' + ISNULL(CONVERT(NVARCHAR(30), d.balance, 1), 'NULL') +

' to ' + ISNULL(CONVERT(NVARCHAR(30), i.balance, 1), 'NULL') + '.'

FROM inserted i JOIN deleted d ON i.id = d.id

WHERE ISNULL(d.balance, -1) <> ISNULL(i.balance, -1)

END

IF UPDATE(email)

BEGIN

INSERT INTO AuditLog (TableName, RecordID, ActionType, ChangeDetails)

SELECT 'Student', i.id, 'UPDATE', 'Email changed from ''' + d.email + ''' to ''' + i.email + '''.'

FROM inserted i JOIN deleted d ON i.id = d.id

WHERE d.email <> i.email

END

IF UPDATE(phone)

BEGIN

INSERT INTO AuditLog (TableName, RecordID, ActionType, ChangeDetails)

SELECT 'Student', i.id, 'UPDATE', 'Phone changed from ''' + ISNULL(d.phone, 'NULL') + ''' to ''' + ISNULL(i.phone, 'NULL') + '''.'

FROM inserted i JOIN deleted d ON i.id = d.id

WHERE ISNULL(d.phone, '') <> ISNULL(i.phone, '')

END

END

GO

CREATE TRIGGER trg\_LogStudentDeletion ON Student

AFTER DELETE

AS

BEGIN

SET NOCOUNT ON;

INSERT INTO AuditLog (TableName, RecordID, ActionType, ChangeDetails)

SELECT 'Student', d.id, 'DELETE', 'Student record deleted. Name: ' + d.last\_name + N' ' + d.first\_name + N', Email: ' + d.email

FROM deleted;

END

GO

#### trg\_SetInitialStudentBalance

* **Activation table**: Student
* **Purpose:** If a new student record is created without a specified balance, this trigger automatically sets it to 0.
* **Practical Operations:** This ensures that the balance column always contains a valid numerical value.

CREATE TRIGGER trg\_SetInitialStudentBalance

ON Student

AFTER INSERT

AS

BEGIN

SET NOCOUNT ON;

UPDATE Student

SET balance = 0

WHERE id IN (SELECT id FROM inserted) AND balance IS NULL

END

#### trg\_DeleteExamsOnClassDelete

* **Activation table:** Class
* **Purpose:** When a class is deleted, this trigger also deletes all exams and grades associated with that class.
* **Practical Operations:** If a class is cancelled and removed, this ensures that all its related academic records are also cleaned up.

CREATE TRIGGER trg\_DeleteExamsOnClassDelete

ON Class

INSTEAD OF DELETE

AS

BEGIN

SET NOCOUNT ON

DELETE Payment FROM Payment p

JOIN deleted d ON p.class\_id = d.id

DELETE Exam\_Result FROM Exam\_Result g

JOIN Exam e ON g.exam\_id = e.id

JOIN deleted d ON e.class\_id = d.id

DELETE Exam FROM Exam e

JOIN deleted d ON e.class\_id = d.id

DELETE Class\_Student FROM Class\_Student cs

JOIN deleted d ON cs.class\_id = d.id

DELETE Class FROM Class c

JOIN deleted d ON c.id = d.id

END

#### trg\_HashPassword

* **Purpose:** These triggers automatically intercept INSERT operations on the Student and Teacher tables. They take the provided plaintext password, combine it with the user's unique username (which acts as a "salt"), generate a secure SHA2\_512 hash, and then store the resulting hash string in the password column.
* **Usecase:** Enforce the security policy, meaning the application layer doesn't need to contain complex hashing logic.

CREATE TRIGGER trg\_HashStudentPassword

ON Student

INSTEAD OF INSERT

AS

BEGIN

SET NOCOUNT ON;

INSERT INTO Student (id, first\_name, last\_name, user\_name, password, date\_birth, gender, email, phone, address, city, balance, created\_date)

SELECT

i.id, i.first\_name, i.last\_name, i.user\_name,

CONVERT(NVARCHAR(128), HASHBYTES('SHA2\_512', i.password + i.user\_name), 2),

i.date\_birth, i.gender, i.email, i.phone, i.address, i.city, i.balance, i.created\_date

FROM inserted i;

END

GO

CREATE TRIGGER trg\_HashTeacherPassword

ON Teacher

INSTEAD OF INSERT

AS

BEGIN

SET NOCOUNT ON;

INSERT INTO Teacher (id, first\_name, last\_name, user\_name, password, date\_birth, gender, email, phone, address, city, description)

SELECT

i.id, i.first\_name, i.last\_name, i.user\_name,

CONVERT(NVARCHAR(128), HASHBYTES('SHA2\_512', i.password + i.user\_name), 2),

i.date\_birth, i.gender, i.email, i.phone, i.address, i.city, i.description

FROM inserted i;

END

GO

## SQL Queries

### Find classes with the same teacher.

* **Requirement:** Make a list of pairs of different classes taught by the same teacher. The report should display TeacherFullName, ClassID\_1, and ClassID\_2.

SELECT

t.last\_name + N' ' + t.first\_name AS TeacherFullName,

c1.id AS ClassID\_1,

c2.id AS ClassID\_2

FROM Class c1

JOIN Class c2 ON c1.teacher\_id = c2.teacher\_id

JOIN Teacher t ON c1.teacher\_id = t.id

WHERE c1.id < c2.id AND c1.teacher\_id IS NOT NULL;

### Reports student performance in a specific course.

* **Requirement:** For the course 'IELTS\_70', generate a report that displays the name of each enrolled student, their average score, highest score, and lowest score across all exams taken in this course.

WITH StudentGradesInCourse AS (

SELECT

cs.student\_id,

er.value

FROM Class\_Student cs

JOIN Class cl ON cs.class\_id = cl.id

JOIN Exam e ON cl.id = e.class\_id

JOIN Exam\_Result er ON e.id = er.exam\_id AND cs.student\_id = er.student\_id

WHERE cl.course\_id = N'IELTS\_70'

)

SELECT

s.last\_name + N' ' + s.first\_name AS StudentFullName,

AVG(sg.value) AS AverageScore,

MAX(sg.value) AS HighestScore,

MIN(sg.value) AS LowestScore

FROM StudentGradesInCourse sg

JOIN Student s ON sg.student\_id = s.id

GROUP BY s.id, s.first\_name, s.last\_name

ORDER BY AverageScore DESC

### Top 2 classes with the highest and lowest average scores.

* **Requirement:** Create a report with two parts: Top 2 classes with the highest average student scores and Top 2 classes with the lowest average student scores.

WITH ClassAverages AS (

SELECT

cl.id AS ClassID,

co.description AS CourseDescription,

t.last\_name + N' ' + t.first\_name AS TeacherFullName,

AVG(er.value) AS AverageGrade

FROM Exam\_Result er

JOIN Exam e ON er.exam\_id = e.id

JOIN Class cl ON e.class\_id = cl.id

JOIN Course co ON cl.course\_id = co.id

LEFT JOIN Teacher t ON cl.teacher\_id = t.id

GROUP BY cl.id, co.description, t.last\_name, t.first\_name

),

Highest AS (

SELECT TOP 2 WITH TIES \* FROM ClassAverages ORDER BY AverageGrade DESC

),

Lowest AS (

SELECT TOP 2 WITH TIES \* FROM ClassAverages ORDER BY AverageGrade ASC

)

SELECT \* FROM Highest

UNION ALL

SELECT \* FROM Lowest

ORDER BY AverageGrade DESC

### Teacher performance summary

* **Requirement:** Create a report that includes the teacher's name, the total number of distinct classes they teach, the total number of unique students they have taught across all their classes, and the overall average grade they have given.

SELECT

t.id AS TeacherID,

t.last\_name + N' ' + t.first\_name AS TeacherName,

COUNT(DISTINCT c.id) AS TotalClasses,

COUNT(DISTINCT cs.student\_id) AS TotalStudentsTaught,

AVG(er.value) AS AvgGradeGiven

FROM Teacher t

LEFT JOIN Class c ON t.id = c.teacher\_id

LEFT JOIN Class\_Student cs ON c.id = cs.class\_id

LEFT JOIN Exam e ON c.id = e.class\_id

LEFT JOIN Exam\_Result er ON e.id = er.exam\_id AND cs.student\_id = er.student\_id

GROUP BY t.id, t.last\_name, t.first\_name

ORDER BY AvgGradeGiven DESC

### Find students who scored above the class average

* **Requirement:** Create a report that list the student's name, the exam ID, the student's individual grade, and the calculated class average for that same exam.

WITH ExamAverages AS (

SELECT exam\_id, AVG(value) as AvgGrade

FROM Exam\_Result

GROUP BY exam\_id

)

SELECT

s.last\_name + N' ' + s.first\_name AS FullName,

er.exam\_id,

e.description AS ExamDescription,

er.value AS StudentGrade,

ea.AvgGrade AS ClassAverage

FROM Exam\_Result er

JOIN Student s ON er.student\_id = s.id

JOIN ExamAverages ea ON er.exam\_id = ea.exam\_id

JOIN Exam e ON er.exam\_id = e.id

WHERE er.value > ea.AvgGrade

ORDER BY ClassAverage DESC, StudentGrade DESC)

### List high-rerforming and active students

* **Requirement:** Generate a list of students that have taken at least three exams and maintained an overall average score of greater than 7 across all their exams.

SELECT

s.id AS StudentID,

s.last\_name + N' ' + s.first\_name AS FullName,

COUNT(er.exam\_id) AS ExamCount,

AVG(er.value) AS AvgScore

FROM Student s

JOIN Exam\_Result er ON s.id = er.student\_id

GROUP BY s.id, s.last\_name, s.first\_name

HAVING COUNT(er.exam\_id) >= 3 AND AVG(er.value) > 7

ORDER BY AvgScore DESC

### Teacher's star student report

* **Requirement:** Identify the top-performing student (by average grade) for each teacher. The report should list the teacher's name, the student's name, and their average grade in that teacher's classes.

WITH TeacherStudentPerformance AS (

SELECT

c.teacher\_id,

cs.student\_id,

AVG(er.value) AS AvgGradeInTeacherClasses

FROM Class c

JOIN Class\_Student cs ON c.id = cs.class\_id

JOIN Exam e ON c.id = e.class\_id

JOIN Exam\_Result er ON e.id = er.exam\_id AND cs.student\_id = er.student\_id

WHERE c.teacher\_id IS NOT NULL

GROUP BY c.teacher\_id, cs.student\_id

),

RankedStudents AS (

SELECT

teacher\_id,

student\_id,

AvgGradeInTeacherClasses,

RANK() OVER (PARTITION BY teacher\_id ORDER BY AvgGradeInTeacherClasses DESC) as RankNum

FROM TeacherStudentPerformance

)

SELECT

t.last\_name + N' ' + t.first\_name AS TeacherName,

s.last\_name + N' ' + s.first\_name AS StarStudentName,

rs.AvgGradeInTeacherClasses

FROM RankedStudents rs

JOIN Teacher t ON rs.teacher\_id = t.id

JOIN Student s ON rs.student\_id = s.id

WHERE rs.RankNum = 1

ORDER BY TeacherName

### Student loyalty and course path analysis

* **Requirement:** Generate a list of "loyal" students who have enrolled in more than one distinct course. The report should show the student's name, the count of distinct courses they've taken, and a comma-separated list of those course IDs.

WITH StudentCourseHistory AS (

SELECT

cs.student\_id,

c.course\_id

FROM Class\_Student cs

JOIN Class c ON cs.class\_id = c.id

GROUP BY cs.student\_id, c.course\_id

)

SELECT

s.last\_name + N' ' + s.first\_name AS StudentName,

COUNT(sch.course\_id) AS DistinctCoursesTaken,

STRING\_AGG(sch.course\_id, ', ') AS CoursePath

FROM StudentCourseHistory sch

JOIN Student s ON sch.student\_id = s.id

GROUP BY s.id, s.last\_name, s.first\_name

HAVING COUNT(sch.course\_id) > 1

ORDER BY DistinctCoursesTaken DESC, StudentName

### Course profitability ranking

**Requirement:** Rank all courses based on the total revenue generated from successful payments. The report should include the course name, the number of unique paying students, the total revenue, and its profitability rank.

SELECT

c.description AS CourseName,

COUNT(DISTINCT p.student\_id) AS NumberOfPayingStudents,

SUM(p.amount) AS TotalRevenue,

AVG(p.amount) AS AveragePayment,

RANK() OVER (ORDER BY SUM(p.amount) DESC) AS ProfitabilityRank

FROM Course c

JOIN Class cl ON c.id = cl.course\_id

JOIN Payment p ON cl.id = p.class\_id

WHERE p.status = 'Success'

GROUP BY c.description

ORDER BY ProfitabilityRank

### At-risk student tdentification

**Requirement:** Find students who are currently enrolled in a class but have not yet received a grade for any exam in that class. This helps identify students who may need follow-up.

SELECT

s.id AS StudentID,

s.last\_name + N' ' + s.first\_name AS StudentName,

cs.class\_id AS EnrolledClassID,

c.description AS CourseDescription

FROM Student s

JOIN Class\_Student cs ON s.id = cs.student\_id

JOIN Class cl ON cs.class\_id = cl.id

JOIN Course c ON cl.course\_id = c.id

WHERE NOT EXISTS (

SELECT 1

FROM Exam e

JOIN Exam\_Result er ON e.id = er.exam\_id

WHERE e.class\_id = cs.class\_id AND er.student\_id = s.id

)

ORDER BY s.id, cs.class\_id

GO

### Exam difficulty analysis

**Requirement**: Rank all exams from hardest to easiest based on the average score achieved by students. The report should only include exams taken by at least 3 students to ensure statistical significance.

SELECT

e.description AS ExamName,

cl.id AS ClassID,

e.exam\_type AS ExamType,

AVG(er.value) AS AverageScore,

COUNT(er.student\_id) AS Participants,

MIN(er.value) AS LowestScore,

MAX(er.value) AS HighestScore,

RANK() OVER (ORDER BY AVG(er.value) ASC) as DifficultyRank

FROM Exam e

JOIN Exam\_Result er ON e.id = er.exam\_id

JOIN Class cl ON e.class\_id = cl.id

GROUP BY e.id, e.description, cl.id, e.exam\_type

HAVING COUNT(er.student\_id) >= 3 -- Only consider exams with at least 3 participants

ORDER BY DifficultyRank

### Student academic journey & teacher influence

**Requirement**: Generate a detailed report for each student who has taken more than one exam. The report must track their academic progression from their very first exam to their most recent one. It should calculate their overall performance, their score improvement, and identify the teachers who taught their first and last classes where they were tested.

WITH StudentExamHistory AS (

SELECT

er.student\_id,

er.value,

cl.teacher\_id,

ROW\_NUMBER() OVER(PARTITION BY er.student\_id ORDER BY e.date ASC, e.id ASC) as FirstExamRank,

ROW\_NUMBER() OVER(PARTITION BY er.student\_id ORDER BY e.date DESC, e.id DESC) as LastExamRank

FROM Exam\_Result er

JOIN Exam e ON er.exam\_id = e.id

JOIN Class cl ON e.class\_id = cl.id

),

StudentSummary AS (

SELECT

student\_id,

AVG(value) as AverageScore,

COUNT(value) as TotalExams,

MIN(CASE WHEN FirstExamRank = 1 THEN value END) as FirstScore,

MIN(CASE WHEN LastExamRank = 1 THEN value END) as LastScore,

MIN(CASE WHEN FirstExamRank = 1 THEN teacher\_id END) as FirstTeacherID,

MIN(CASE WHEN LastExamRank = 1 THEN teacher\_id END) as LastTeacherID

FROM StudentExamHistory

GROUP BY student\_id

)

SELECT

s.id AS StudentID,

dbo.fn\_GetStudentFullName(s.id) AS StudentName,

ss.TotalExams,

ss.FirstScore,

ss.LastScore,

(ss.LastScore - ss.FirstScore) AS ScoreImprovement,

ss.AverageScore,

ISNULL(ft.last\_name + N' ' + ft.first\_name, 'N/A') AS FirstTeacher,

ISNULL(lt.last\_name + N' ' + lt.first\_name, 'N/A') AS LastTeacher,

CASE

WHEN (ss.LastScore - ss.FirstScore) > 1.0 THEN 'Significant Improvement'

WHEN (ss.LastScore - ss.FirstScore) > 0 THEN 'Improved'

WHEN (ss.LastScore - ss.FirstScore) = 0 THEN 'Consistent'

ELSE 'Declined'

END AS PerformanceJourney

FROM StudentSummary ss

JOIN Student s ON ss.student\_id = s.id

LEFT JOIN Teacher ft ON ss.FirstTeacherID = ft.id

LEFT JOIN Teacher lt ON ss.LastTeacherID = lt.id

WHERE ss.TotalExams > 1

ORDER BY ScoreImprovement DESC, AverageScore DESC

# Conclusion

* EducateDB project is an answer to modernizing the management process of learning and tutoring centers, especially language center. With the creation and design of a networked database system, the project has solved many issues still encountered in traditional management methods such as data errors, duplication of scheduling, difficulties in handling finances and communication.
* Not only does the system enable efficient management of students, teachers, courses, schedules, documents, examinations and fees, but also pursues automation, scalability and ease of use. The project assists in improving the operational effectiveness of the center, reducing administrative load and improving learning.
* In the future, EducateDB can be improved even more to incorporate more advanced features such as online learning, assessment of teaching quality, or compatibility with other learning systems, aptly fitting management and training needs during the digital era.