جامعة الإسكندرية
ALEXANDRIA UNIVERSITY
كلية الحاسبات وعلوم البيانات

ADV DATABSE PROJECT Report

May 2025

University Course Enrollment Management database system

Table of content

Team members

MEMBER	ID	ROlE
Toka Mohamed Fawzy	22011608	Member 1 Task
Sarah Sameh Ahmed	22011968	Member 2 Task
Rowan Fayez Mohammed	22011454	Member 3 Task
Esraa Mostafa Ibrahim	22010321	Member 4 Task

Github repo:

https://github.com/tokaM107/University-Course-Enrollment-Management-System

1. Introduction

This database system is designed to manage the complex operations of a university's course registration and academic tracking system. The comprehensive relational database model supports student enrollment, course management, instructor assignments, academic programs, and financial transactions - all critical components of modern higher education administration.

Importance of the Database Tables

The tables in this system serve as the foundational structure for organizing and managing university operations:

1. **Core Entity Tables**: Tables like Students, Instructors, Courses, and Departments form the backbone of the system by storing essential information about the primary entities in a university environment.
2. **Relationship Management**: Tables such as Enrollments, CourseOfferings, and CoursePrerequisites establish and maintain the critical relationships between students, courses, and instructors that enable the academic process.
3. **Academic Tracking**: The AcademicPrograms, StudentStatus, and GPA-related tables allow the university to monitor student progress, program requirements, and academic standing.
4. **Operational Support**: Tables like Classrooms, Waitlists, and CourseOfferings handle the logistical aspects of scheduling courses and managing limited resources.
5. **Financial Management**: The PaymentTransactions, TuitionRates, and related audit tables ensure proper handling of student accounts and financial records.
6. **Historical Tracking**: Audit log tables (StudentNameChangeLog, GPAAuditLog, etc.) provide crucial record-keeping for compliance, troubleshooting, and data analysis.

This normalized database design ensures data integrity through:

- Primary and foreign key relationships
- Data validation constraints
- Comprehensive audit trails
- Appropriate cascading rules for data modifications

The system supports both day-to-day operations (registration, grading) and strategic decision-making (enrollment trends, resource allocation) while maintaining security and data accuracy through its carefully designed table structure.

2. Database Design and Queries

Table design:

```sql
-- Table Creation
CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY IDENTITY(1,1),
    DepartmentName NVARCHAR(100) NOT NULL,
    Building NVARCHAR(50),
    Budget DECIMAL(18,2)
);

CREATE TABLE AcademicPrograms (
    ProgramID INT PRIMARY KEY IDENTITY(1,1),
    ProgramName NVARCHAR(100) NOT NULL,
    DepartmentID INT FOREIGN KEY REFERENCES Departments(DepartmentID) ON DELETE SET NULL,
    TotalCredits INT NOT NULL CHECK (TotalCredits > 0)
);

CREATE TABLE Students (
    StudentID INT PRIMARY KEY IDENTITY(1000,1),
    FirstName NVARCHAR(50) NOT NULL,
    LastName NVARCHAR(50) NOT NULL,
    Email NVARCHAR(100) UNIQUE NOT NULL,
    DateOfBirth DATE,
    EnrollmentDate DATE DEFAULT GETDATE(),
    DepartmentID INT FOREIGN KEY REFERENCES Departments(DepartmentID) ON DELETE SET NULL,
    ProgramID INT FOREIGN KEY REFERENCES AcademicPrograms(ProgramID) ON DELETE SET NULL,
    GPA DECIMAL(3,2),
    Balance DECIMAL(10,2) DEFAULT 0 CHECK (Balance >= 0),
    LoginName AS 'StudentUser' + CAST(StudentID AS NVARCHAR),
    CONSTRAINT CHK_GPA CHECK (GPA <= 4.00)
);
```

```sql
CREATE TABLE StudentStatus (
    StatusID INT PRIMARY KEY IDENTITY(1,1),
    StudentID INT FOREIGN KEY REFERENCES Students(StudentID) ON DELETE CASCADE,
    Status NVARCHAR(50) NOT NULL CHECK (Status IN ('Active', 'Probation', 'Graduated', 'Suspended', 'Honors'))
    StartDate DATE NOT NULL,
    EndDate DATE
);

CREATE TABLE Instructors (
    InstructorID INT PRIMARY KEY IDENTITY(2000,1),
    FirstName NVARCHAR(50) NOT NULL,
    LastName NVARCHAR(50) NOT NULL,
    Email NVARCHAR(100) UNIQUE NOT NULL,
    DepartmentID INT FOREIGN KEY REFERENCES Departments(DepartmentID) ON DELETE SET NULL,
    HireDate DATE,
    Salary DECIMAL(18,2)
);

CREATE TABLE Courses (
    CourseID INT PRIMARY KEY IDENTITY(3000,1),
    CourseCode NVARCHAR(20) UNIQUE NOT NULL,
    CourseName NVARCHAR(100) NOT NULL,
    Credits INT NOT NULL CHECK (Credits > 0),
    DepartmentID INT FOREIGN KEY REFERENCES Departments(DepartmentID) ON DELETE SET NULL,
    Description NVARCHAR(500)
);

CREATE TABLE CoursePrerequisites (
```

```sql

CREATE TABLE CoursePrerequisites (
    PrerequisiteID INT PRIMARY KEY IDENTITY(1,1),
    CourseID INT FOREIGN KEY REFERENCES Courses(CourseID) ON DELETE CASCADE,
    PrerequisiteCourseID INT FOREIGN KEY REFERENCES Courses(CourseID),
    CONSTRAINT CHK_DifferentCourses CHECK (CourseID != PrerequisiteCourseID)
);


CREATE TABLE Classrooms (
    ClassroomID INT PRIMARY KEY IDENTITY(1,1),
    Building NVARCHAR(50),
    RoomNumber NVARCHAR(20),
    Capacity INT NOT NULL CHECK (Capacity > 0)
);

CREATE TABLE CourseOfferings (
    OfferingID INT PRIMARY KEY IDENTITY(4000,1),
    CourseID INT FOREIGN KEY REFERENCES Courses(CourseID) ON DELETE CASCADE,
    InstructorID INT FOREIGN KEY REFERENCES Instructors(InstructorID) ON DELETE SET NULL,
    ClassroomID INT FOREIGN KEY REFERENCES Classrooms(ClassroomID) ON DELETE SET NULL,
    Semester NVARCHAR(20) NOT NULL CHECK (Semester IN ('Fall', 'Spring', 'Summer')),
    Year INT NOT NULL CHECK (Year >= 2000),
    Schedule NVARCHAR(100),
    MaxCapacity INT CHECK (MaxCapacity > 0),
    CurrentEnrollment INT DEFAULT 0 CHECK (CurrentEnrollment >= 0),

);

```

```sql

CREATE TABLE Enrollments (
    EnrollmentID INT PRIMARY KEY IDENTITY(5000,1),
    StudentID INT FOREIGN KEY REFERENCES Students(StudentID) ON DELETE CASCADE,
    OfferingID INT FOREIGN KEY REFERENCES CourseOfferings(OfferingID) ON DELETE CASCADE,
    EnrollmentDate DATE DEFAULT GETDATE(),
    Grade DECIMAL(3,2),
    Status NVARCHAR(20) DEFAULT 'Active' CHECK (Status IN ('Active', 'Completed', 'Dropped')),
    CONSTRAINT UC_Enrollment UNIQUE (StudentID, OfferingID),
    CONSTRAINT CHK_Grade CHECK (Grade <= 4.00)
);

CREATE TABLE Waitlists (
    WaitlistID INT PRIMARY KEY IDENTITY(1,1),
    StudentID INT FOREIGN KEY REFERENCES Students(StudentID) ON DELETE CASCADE,
    OfferingID INT FOREIGN KEY REFERENCES CourseOfferings(OfferingID) ON DELETE CASCADE,
    WaitlistDate DATETIME DEFAULT GETDATE(),
    Position INT NOT NULL CHECK (Position > 0)
);

CREATE TABLE StudentNameChangeLog (
    LogID INT PRIMARY KEY IDENTITY(6000,1),
    StudentID INT NOT NULL,
    OldFirstName NVARCHAR(50),
    OldLastName NVARCHAR(50),
    NewFirstName NVARCHAR(50),
    NewLastName NVARCHAR(50),
    ChangeDate DATETIME DEFAULT GETDATE(),
    ChangedBy NVARCHAR(100)
```

```sql
CREATE TABLE PaymentTransactions (
    TransactionID INT PRIMARY KEY IDENTITY(7000,1),
    StudentID INT FOREIGN KEY REFERENCES Students(StudentID) ON DELETE CASCADE,
    Amount DECIMAL(10,2) CHECK (Amount > 0),
    TransactionDate DATETIME DEFAULT GETDATE(),
    Status NVARCHAR(20) DEFAULT 'Pending' CHECK (Status IN ('Pending', 'Completed', 'Failed'))
);

CREATE TABLE PaymentAuditLog (
    LogID INT PRIMARY KEY IDENTITY(1,1),
    TransactionID INT FOREIGN KEY REFERENCES PaymentTransactions(TransactionID) ON DELETE CASCADE,
    OldStatus NVARCHAR(20),
    NewStatus NVARCHAR(20),
    ChangeDate DATETIME DEFAULT GETDATE(),
    ChangedBy NVARCHAR(100)
);

CREATE TABLE EnrollmentAuditLog (
    LogID INT PRIMARY KEY IDENTITY(1,1),
    EnrollmentID INT FOREIGN KEY REFERENCES Enrollments(EnrollmentID) ON DELETE CASCADE,
    CourseCode NVARCHAR(20),
    OldStatus NVARCHAR(20),
    NewStatus NVARCHAR(20),
    ChangeDate DATETIME DEFAULT GETDATE(),
    ChangedBy NVARCHAR(100)
);

CREATE TABLE GPAAuditLog (
```

```sql
CREATE TABLE GPAAuditLog (
    LogID INT PRIMARY KEY IDENTITY(8000,1),
    StudentID INT,
    OldGPA DECIMAL(3,2),
    NewGPA DECIMAL(3,2),
    ChangeDate DATETIME DEFAULT GETDATE(),
    ChangedBy NVARCHAR(100)
);

CREATE TABLE TuitionRates (
    RateID INT PRIMARY KEY IDENTITY(1,1),
    AcademicYear INT NOT NULL,
    PerCreditRate DECIMAL(10,2) NOT NULL,
    FlatRate DECIMAL(10,2),
    EffectiveDate DATE NOT NULL
);

CREATE TABLE AuditLog (
    AuditID INT PRIMARY KEY,
    Action NVARCHAR(100),
    ActionDate DATETIME DEFAULT GETDATE(),
    UserName NVARCHAR(100)
);

CREATE TABLE AuditLogArchive (
    AuditID INT PRIMARY KEY,
    Action NVARCHAR(100),
    ActionDate DATETIME,
```

```
169
170 CREATE TABLE AuditLogArchive (
171     AuditID INT PRIMARY KEY,
172     Action NVARCHAR(100),
173     ActionDate DATETIME,
174     UserName NVARCHAR(100)
175 );
176
177 CREATE TABLE SecurityAuditLog (
178     SecurityLogID INT PRIMARY KEY IDENTITY(1,1),
179     Action NVARCHAR(100),
180     ActionDate DATETIME DEFAULT GETDATE(),
181     UserName NVARCHAR(100),
182     AffectedRole NVARCHAR(128)
183 );
184
185 CREATE TABLE UserStudentMapping (
186     LoginName NVARCHAR(128) PRIMARY KEY,
187
188     StudentID INT NOT NULL FOREIGN KEY REFERENCES Students(StudentID) ON DELETE CASCADE
189 );
190
191 CREATE TABLE UserInstructorMapping (
192     LoginName NVARCHAR(128) PRIMARY KEY,
193     InstructorID INT NOT NULL FOREIGN KEY REFERENCES Instructors(InstructorID) ON DELETE CASCADE
194 );
195
196
197 IF OBJECT_ID('AuditSequence', 'SO') IS NOT NULL
```

```
181     UserName NVARCHAR(100),
182     AffectedRole NVARCHAR(128)
183 );
184
185 CREATE TABLE UserStudentMapping (
186     LoginName NVARCHAR(128) PRIMARY KEY,
187
188     StudentID INT NOT NULL FOREIGN KEY REFERENCES Students(StudentID) ON DELETE CASCADE
189 );
190
191 CREATE TABLE UserInstructorMapping (
192     LoginName NVARCHAR(128) PRIMARY KEY,
193     InstructorID INT NOT NULL FOREIGN KEY REFERENCES Instructors(InstructorID) ON DELETE CASCADE
194 );
195
196
197 IF OBJECT_ID('AuditSequence', 'SO') IS NOT NULL
198     DROP SEQUENCE AuditSequence;
199
200
201 CREATE SEQUENCE AuditSequence
202     START WITH 1
203     INCREMENT BY 1
204     MINVALUE 1
205     NO MAXVALUE
206     CACHE 10;
207
208
```

Sample data :

```sql
-- Sample Data
INSERT INTO Departments (DepartmentName, Building, Budget) VALUES
('Computer Science', 'Engineering', 1000000.00),
('Mathematics', 'Science', 750000.00),
('Physics', 'Science', 800000.00),
('English', 'Humanities', 500000.00);


INSERT INTO AcademicPrograms (ProgramName, DepartmentID, TotalCredits) VALUES
('BS Computer Science', 1, 120),
('BA Mathematics', 2, 120),
('BS Physics', 3, 124),
('BA English', 4, 120);


INSERT INTO Students (FirstName, LastName, Email, DateOfBirth, EnrollmentDate, DepartmentID, ProgramID, GPA, B
('John', 'Doe', 'john.doe@university.edu', '2000-05-15', '2023-09-01', 1, 1, 3.75, 1500.00),
('Jane', 'Smith', 'jane.smith@university.edu', '1999-08-22', '2023-09-01', 2, 2, 3.90, 0.00),
('Michael', 'Johnson', 'michael.j@university.edu', '2001-02-10', '2024-01-15', 1, 1, 3.45, 2000.00),
('Emily', 'Davis', 'emily.davis@university.edu', '2000-11-30', '2023-09-01', 3, 3, NULL, 500.00),
('Alex', 'Brown', 'alex.brown@university.edu', '2002-03-05', '2024-01-15', 4, 4, 3.20, 0.00);


INSERT INTO StudentStatus (StudentID, Status, StartDate) VALUES
(1000, 'Active', '2023-09-01'),
(1001, 'Honors', '2023-09-01'),
(1002, 'Active', '2024-01-15'),
(1003, 'Active', '2023-09-01'),
```

```sql

INSERT INTO Instructors (FirstName, LastName, Email, DepartmentID, HireDate, Salary) VALUES
('Robert', 'Wilson', 'r.wilson@university.edu', 1, '2010-07-15', 85000.00),
('Sarah', 'Williams', 's.williams@university.edu', 2, '2015-03-10', 75000.00),
('David', 'Lee', 'david.lee@university.edu', 3, '2018-09-01', 78000.00),
('Laura', 'Clark', 'laura.clark@university.edu', 4, '2012-01-20', 72000.00);


INSERT INTO Courses (CourseCode, CourseName, Credits, DepartmentID, Description) VALUES
('CS101', 'Introduction to Programming', 4, 1, 'Basic programming concepts using Python'),
('MATH201', 'Calculus II', 3, 2, 'Advanced calculus topics including integration'),
('PHYS101', 'Mechanics', 4, 3, 'Fundamentals of classical mechanics'),
('ENG201', 'British Literature', 3, 4, 'Study of British literary works'),
('CS202', 'Data Structures', 4, 1, 'Advanced data structures and algorithms');


INSERT INTO CoursePrerequisites (CourseID, PrerequisiteCourseID) VALUES
(3004, 3000); -- CS202 requires CS101


INSERT INTO Classrooms (Building, RoomNumber, Capacity) VALUES
('Engineering', 'ENG-101', 30),
('Science', 'SCI-205', 25),
('Science', 'SCI-101', 20),
('Humanities', 'HUM-301', 35),
('Engineering', 'ENG-102', 30);


```

```
58
59 INSERT INTO CourseOfferings (CourseID, InstructorID, ClassroomID, Semester, Year, Schedule, MaxCapacity, Curre
60 (3000, 2000, 1, 'Fall', 2023, 'MWF 10:00-11:00', 30, 3),
61 (3001, 2001, 2, 'Fall', 2023, 'TTh 13:00-14:30', 25, 2),
62 (3002, 2002, 3, 'Spring', 2024, 'MWF 09:00-10:00', 20, 0),
63 (3003, 2003, 4, 'Fall', 2023, 'TTh 11:00-12:30', 35, 1),
64 (3004, 2000, 5, 'Spring', 2024, 'MWF 11:00-12:00', 30, 0);
65
66
67 INSERT INTO Enrollments (StudentID, OfferingID, EnrollmentDate, Grade, Status) VALUES
68 (1000, 4000, '2023-09-10', 3.80, 'Completed'),
69 (1001, 4001, '2023-09-10', 4.00, 'Completed'),
70 (1001, 4002, '2023-09-10', NULL, 'Active'),
71 (1002, 4000, '2023-09-10', 3.50, 'Completed'),
72 (1003, 4003, '2023-09-10', NULL, 'Active'),
73 (1004, 4001, '2023-09-10', 3.20, 'Completed');
74
75
76 INSERT INTO PaymentTransactions (StudentID, Amount, TransactionDate, Status) VALUES
77 (1000, 1000.00, '2023-09-15', 'Completed'),
78 (1000, 500.00, '2023-10-01', 'Pending'),
79 (1001, 1200.00, '2023-09-20', 'Completed'),
80 (1002, 800.00, '2024-01-20', 'Completed'),
81 (1003, 300.00, '2023-09-25', 'Pending');
82
83
84 INSERT INTO GPAAuditLog (StudentID, OldGPA, NewGPA, ChangeDate, ChangedBy) VALUES
85 (1000, NULL, 3.75, '2023-09-15', 'Registrar'),
```

```
84 INSERT INTO GPAAuditLog (StudentID, OldGPA, NewGPA, ChangeDate, ChangedBy) VALUES
85 (1000, NULL, 3.75, '2023-09-15', 'Registrar'),
86 (1001, NULL, 3.90, '2023-09-15', 'Registrar'),
87 (1002, NULL, 3.45, '2024-01-20', 'Registrar'),
88 (1004, NULL, 3.20, '2024-01-20', 'Registrar');
89
90
91 INSERT INTO TuitionRates (AcademicYear, PerCreditRate, FlatRate, EffectiveDate) VALUES
92 (2023, 500.00, 15000.00, '2023-07-01'),
93 (2024, 525.00, 15750.00, '2024-07-01');
94
95
96 INSERT INTO AuditLog (AuditID, Action, UserName)
97 VALUES (NEXT VALUE FOR AuditSequence, 'Database initialized', SUSER_NAME());
98
99
100 INSERT INTO UserStudentMapping (LoginName, StudentID) VALUES
101 ('StudentUser1000', 1000),
102 ('StudentUser1001', 1001),
103 ('StudentUser1002', 1002),
104 ('StudentUser1003', 1003),
105 ('StudentUser1004', 1004);
106
107
108 INSERT INTO UserInstructorMapping (LoginName, InstructorID) VALUES
109 ('InstructorUser2000', 2000),
110 ('InstructorUser2001', 2001),
111 ('InstructorUser2002', 2002),
```

1. Advanced Query with JOIN, GROUP BY, HAVING

Get the average GPA per department, but only for departments where the average GPA is above 3.

```sql
--Advanced Query with JOIN, GROUP BY, HAVING
SELECT
    D.DepartmentName,
    AVG(S.GPA) AS AverageGPA
FROM
    Students S
JOIN
    Departments D ON S.DepartmentID = D.DepartmentID
WHERE
    S.GPA IS NOT NULL
GROUP BY
    D.DepartmentName
HAVING
    AVG(S.GPA) > 3.0;
```

DepartmentName	AverageGPA
Computer Science	3.600000
English	3.200000
Mathematics	3.900000

3. Functions, Procedures, and Triggers

```sql
---------- 1)Function to get course name by course ID ----------

CREATE FUNCTION dbo.GetCourseNameByID (@CourseID INT)
RETURNS NVARCHAR(100)
AS
BEGIN
    DECLARE @Title NVARCHAR(100);

    SELECT @Title = Title
    FROM Courses
    WHERE CourseID = @CourseID;

    RETURN ISNULL(@Title, 'Course not found');
END;
GO

-- Example usage:
SELECT dbo.GetCourseNameByID(3) AS Course;
GO
```

100 %

Results | Messages

	Course
1	Database Systems

```sql
---------- 2)Stored procedure to enroll a student in a course ----------

CREATE PROCEDURE dbo.EnrollStudentInCourse
    @StudentID INT,
    @OfferingID INT,
    @EnrollmentDate DATE = NULL,
    @ResultMessage NVARCHAR(200) OUTPUT
AS
BEGIN
    SET NOCOUNT ON;

    BEGIN TRY
        -- Validate student exists
        IF NOT EXISTS (SELECT 1 FROM Students WHERE StudentID = @StudentID)
        BEGIN
            SET @ResultMessage = 'Error: Student not found';
            RETURN -1;
        END

        -- Validate course offering exists
        IF NOT EXISTS (SELECT 1 FROM CourseOfferings WHERE OfferingID = @OfferingID)
        BEGIN
            SET @ResultMessage = 'Error: Course offering not found';
            RETURN -2;
        END

-- Check if already enrolled
IF EXISTS (SELECT 1 FROM Enrollments WHERE StudentID = @StudentID AND OfferingID = @OfferingID)
BEGIN
    SET @ResultMessage = 'Error: Student already enrolled in this course';
    RETURN -3;
END

-- Check capacity
DECLARE @CurrentEnrollment INT, @MaxEnrollment INT;

SELECT @CurrentEnrollment = COUNT(*)
FROM Enrollments
WHERE OfferingID = @OfferingID;

SELECT @MaxEnrollment = MaxEnrollment
FROM CourseOfferings
WHERE OfferingID = @OfferingID;

IF @CurrentEnrollment >= @MaxEnrollment
BEGIN
    SET @ResultMessage = 'Error: Course is at maximum capacity';
    RETURN -4;
END
```

```
-- Check prerequisites
DECLARE @CourseID INT;
SELECT @CourseID = CourseID FROM CourseOfferings WHERE OfferingID = @OfferingID;

IF EXISTS (
    SELECT 1 FROM CoursePrerequisites p
    WHERE p.CourseID = @CourseID
    AND NOT EXISTS (
        SELECT 1 FROM Enrollments e
        JOIN CourseOfferings co ON e.OfferingID = co.OfferingID
        WHERE e.StudentID = @StudentID
        AND co.CourseID = p.CourseID
        AND (p.MinimumGrade IS NULL OR e.Grade >= p.MinimumGrade)
    )
)
BEGIN
    SET @ResultMessage = 'Error: Student does not meet prerequisites for this course';
    RETURN -5;
END

-- Set default enrollment date
IF @EnrollmentDate IS NULL
    SET @EnrollmentDate = GETDATE();

-- Enroll the student
INSERT INTO Enrollments (StudentID, OfferingID, EnrollmentDate)
VALUES (@StudentID, @OfferingID, @EnrollmentDate);


        SET @ResultMessage = 'Student successfully enrolled in the course';
        RETURN 0;
    END TRY
    BEGIN CATCH
        SET @ResultMessage = 'Error: ' + ERROR_MESSAGE();
        RETURN -99;
    END CATCH
END;
GO

-- Example usage:
DECLARE @ResultMessage NVARCHAR(200);
DECLARE @ReturnCode INT;
EXEC @ReturnCode = dbo.EnrollStudentInCourse
     @StudentID = 4,
     @OfferingID = 1,
     @ResultMessage = @ResultMessage OUTPUT;
SELECT @ReturnCode AS ReturnCode, @ResultMessage AS ResultMessage;
SELECT * From Enrollments;
GO
```

Results Messages

	ReturnCode	ResultMessage
1	0	Student successfully enrolled in the course

	EnrollmentID	StudentID	OfferingID	EnrollmentDate	Grade	Status
1	1	1	1	2023-08-15	A	Completed
2	2	1	2	2023-08-15	B	Completed
3	3	2	1	2023-08-16	C	Completed
4	4	2	3	2024-01-10	NULL	Enrolled
5	5	3	4	2023-08-17	A	Completed
6	6	3	5	2024-01-11	NULL	Enrolled
7	7	4	2	2023-08-18	B	Completed
8	8	4	4	2023-08-18	A	Completed
9	9	5	1	2023-08-19	C	Completed
10	10	5	3	2024-01-12	NULL	Enrolled
11	11	4	1	2025-05-13	NULL	Enrolled

```
---------- 3)Trigger to log student name changes ----------

CREATE TRIGGER trg_StudentNameChange
ON Students
AFTER UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    -- Only log if first or last name changed
    IF UPDATE(FirstName) OR UPDATE(LastName)
    BEGIN
        INSERT INTO StudentNameChangeLog (
            StudentID,
            OldFirstName,
            OldLastName,
            NewFirstName,
            NewLastName
        )
        SELECT
            i.StudentID,
            d.FirstName AS OldFirstName,
            d.LastName AS OldLastName,
            i.FirstName AS NewFirstName,
            i.LastName AS NewLastName
        FROM inserted i
        JOIN deleted d ON i.StudentID = d.StudentID
        WHERE i.FirstName <> d.FirstName OR i.LastName <> d.LastName;
    END
END;
GO


-- Example usage:
UPDATE Students SET LastName = 'Ahmed' WHERE StudentID = 4;
SELECT * FROM StudentNameChangeLog;
GO
```

Results | Messages

	LogID	StudentID	OldFirstName	OldLastName	NewFirstName	NewLastName	ChangeDate	ChangedBy
1	1	4	Sarah	Sameh	Sarah	Ahmed	2025-05-13 19:03:11.350	DESKTOP-G5AVUN4\diwan

```sql
---------- 4)Instead of trigger to prevent deletion of required courses ----------

CREATE TRIGGER trg_PreventRequiredCourseDeletion
ON Courses
INSTEAD OF DELETE
AS
BEGIN
    SET NOCOUNT ON;

    -- Check if any of the courses to be deleted are required as prerequisites
    IF EXISTS (
        SELECT 1 FROM deleted d
        JOIN CoursePrerequisites p ON d.CourseID = p.PrerequisiteCourseID
    )
    BEGIN
        RAISERROR('Cannot delete courses that are required as prerequisites for other courses', 16, 1);
        RETURN;
    END


    -- If no conflicts, proceed with deletion
    DELETE FROM Courses
    WHERE CourseID IN (SELECT CourseID FROM deleted);
END;
GO

-- Example usage:
DELETE FROM Courses WHERE CourseID = 4;
GO
```

Messages

```
Msg 50000, Level 16, State 1, Procedure trg_PreventRequiredCourseDeletion, Line 16 [Batch Start Line 394]
Cannot delete courses that are required as prerequisites for other courses

(1 row affected)

Completion time: 2025-05-13T19:25:53.9567963+03:00
```

4. Transactions and Concurrency

1. Introduction to Task

This Task explores transactional control, concurrency issues, and isolation mechanisms within the UniversityCourseSystem database. The goal is to simulate real-world behaviors and demonstrate how SQL Server ensures data consistency, integrity, and safe concurrent access.

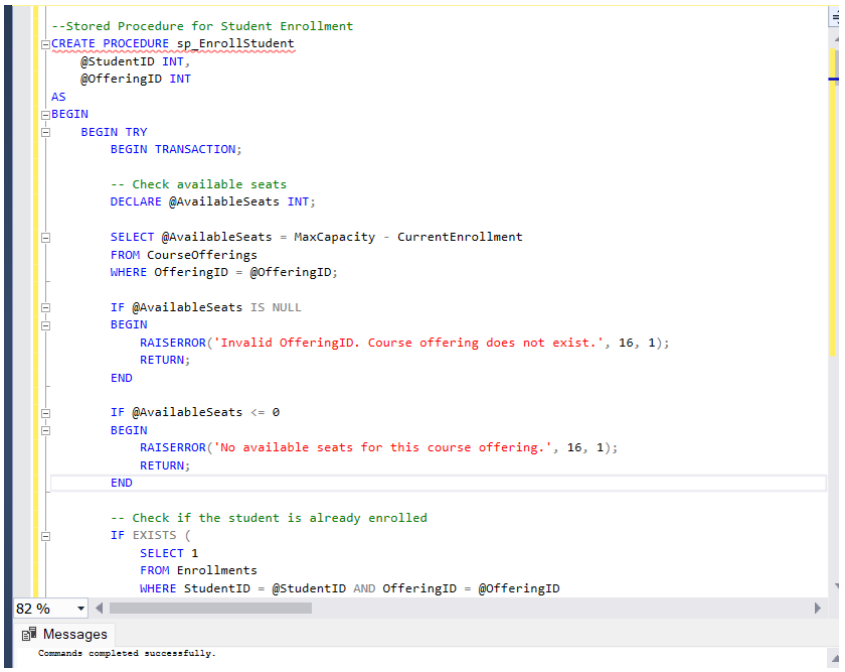
2. Stored Procedure for Enrollment Management “Rollback example”

A stored procedure sp_EnrollStudent was developed to manage student enrollments:

- Validates course offering availability and student enrollment status.
- Executes enrollment and updates seat count atomically within a transaction.
- Implements error handling using TRY...CATCH blocks to maintain data integrity.

Expected Behavior:

- Enrollment is successful only if seats are available and the student is not already enrolled.
- Rolls back and raises errors when constraints are violated.

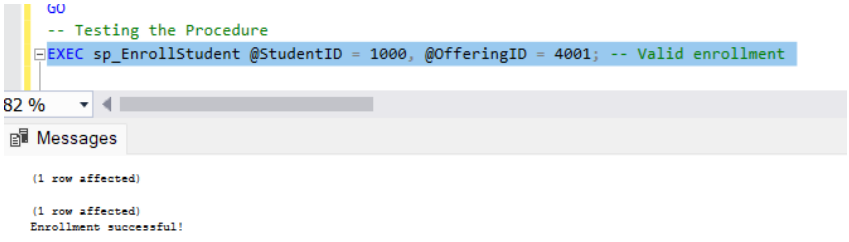


```
--Stored Procedure for Student Enrollment
CREATE PROCEDURE sp_EnrollStudent
    @StudentID INT,
    @OfferingID INT
AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION;

        -- Check available seats
        DECLARE @AvailableSeats INT;

        SELECT @AvailableSeats = MaxCapacity - CurrentEnrollment
        FROM CourseOfferings
        WHERE OfferingID = @OfferingID;

        IF @AvailableSeats IS NULL
        BEGIN
            RAISERROR('Invalid OfferingID. Course offering does not exist.', 16, 1);
            RETURN;
        END

        IF @AvailableSeats <= 0
        BEGIN
            RAISERROR('No available seats for this course offering.', 16, 1);
            RETURN;
        END

        -- Check if the student is already enrolled
        IF EXISTS (
            SELECT 1
            FROM Enrollments
            WHERE StudentID = @StudentID AND OfferingID = @OfferingID
```

82 %

Messages

Commands completed successfully.



```
GO
-- Testing the Procedure
EXEC sp_EnrollStudent @StudentID = 1000, @OfferingID = 4001; -- Valid enrollment
```

82 %

Messages

(1 row affected)

(1 row affected)
Enrollment successful!

```
UPDATE CourseOfferings SET CurrentEnrollment = MaxCapacity WHERE OfferingID = 4001;
EXEC sp_EnrollStudent @StudentID = 1000, @OfferingID = 4001; -- No seats available
EXEC sp_EnrollStudent @StudentID = 1000, @OfferingID = 9999; -- Invalid OfferingID
```

82 %

Messages

```
(1 row affected)
Enrollment failed: No available seats for this course offering.
```

```
EXEC sp_EnrollStudent @StudentID = 1000, @OfferingID = 9999; -- Invalid OfferingID
```

82 %

Messages

```
Enrollment failed: Invalid OfferingID. Course offering does not exist.
```

---

3. Task 1: Transaction Simulation with Savepoints

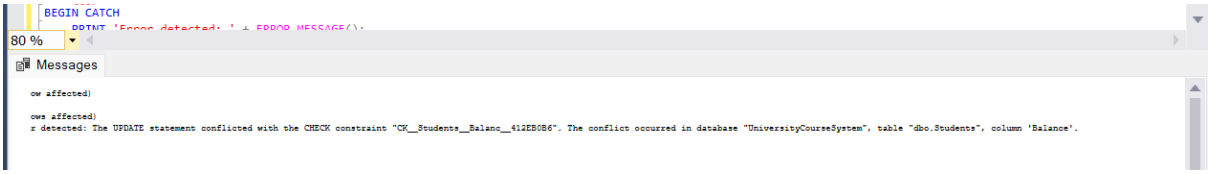
This task simulates a transaction that adds a student and attempts an invalid update:

- Uses SAVEPOINT after a successful insert.
- Triggers a rollback due to a CHECK constraint violation on negative balance.

Expected Behavior:

- The entire transaction, including the insert, is rolled back.
- Demonstrates how atomicity is preserved using BEGIN TRANSACTION and ROLLBACK.

```
--------------------------------------------------------------------------------------------------------------
--Task1: Simulate Transactions Using BEGIN TRANSACTION, SAVEPOINT, COMMIT, and ROLLBACK

BEGIN TRANSACTION;

BEGIN TRY
    -- Check if email already exists to avoid unique constraint error
    IF EXISTS (SELECT 1 FROM Students WHERE Email = 'new.student.email@university.edu')
    BEGIN
        THROW 50001, 'A student with this email already exists.', 1;
    END

    -- Insert new student with updated data
    INSERT INTO Students (FirstName, LastName, Email, DateOfBirth, EnrollmentDate, DepartmentID, ProgramI
    VALUES ('John', 'Doe', 'new.student.email@university.edu', '2002-07-22', '2025-03-01', 2, 3, 3.5, 150

    -- Savepoint after successful insert
    SAVE TRANSACTION SavePoint_AddStudent;

    -- Attempt invalid update (this will fail due to CHECK constraint)
    UPDATE Students SET Balance = -1000 WHERE StudentID = SCOPE_IDENTITY();

    -- If update succeeds (unexpected), commit transaction
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    PRINT 'Error detected: ' + ERROR_MESSAGE();

    -- Rollback entire transaction (savepoint may not exist if error before it)
    ROLLBACK TRANSACTION;
END CATCH;
```

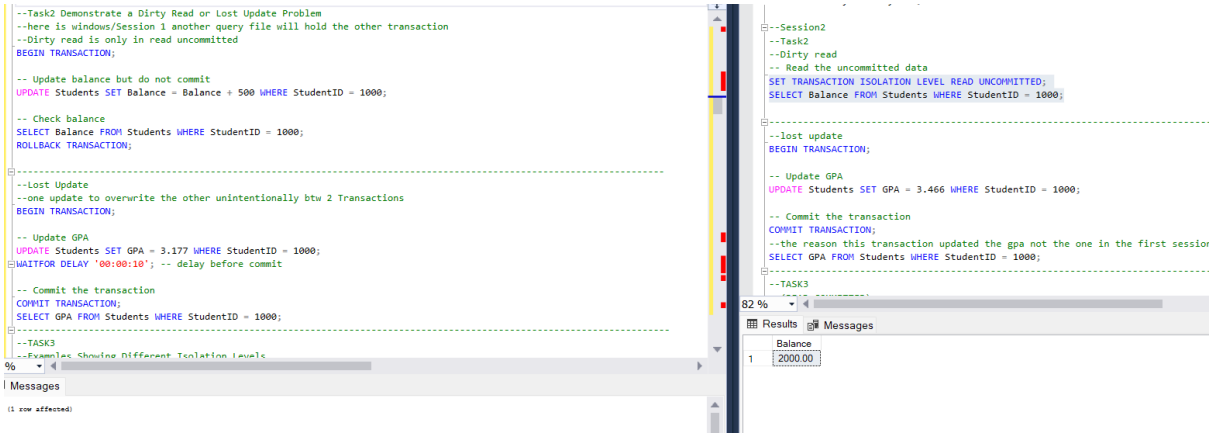


4. Task 2: Concurrency Issues

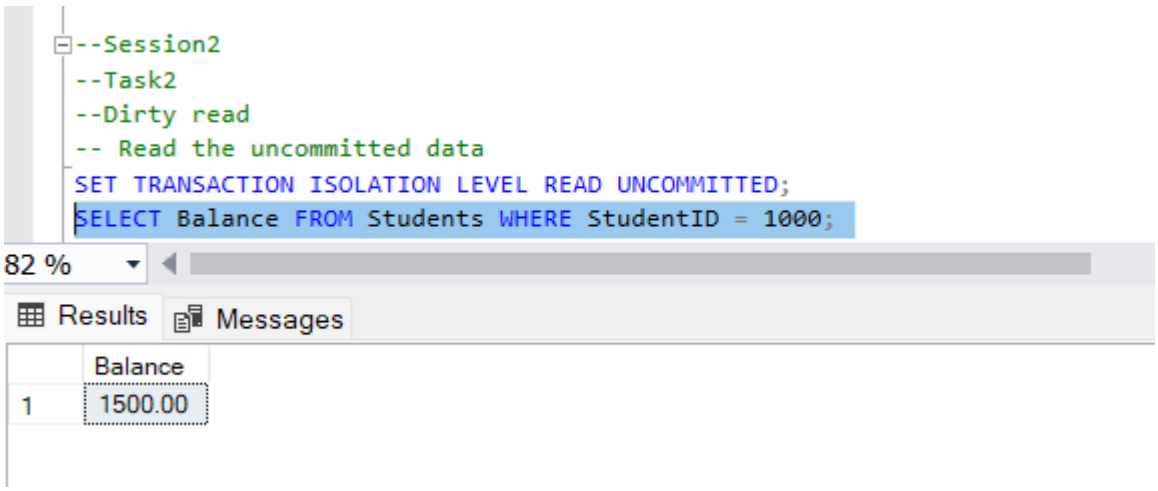
Dirty Read:

- Shows how uncommitted updates can be read using READ UNCOMMITTED.
- Session 2 reads temporary data that may be rolled back.

Session1 **Session2**



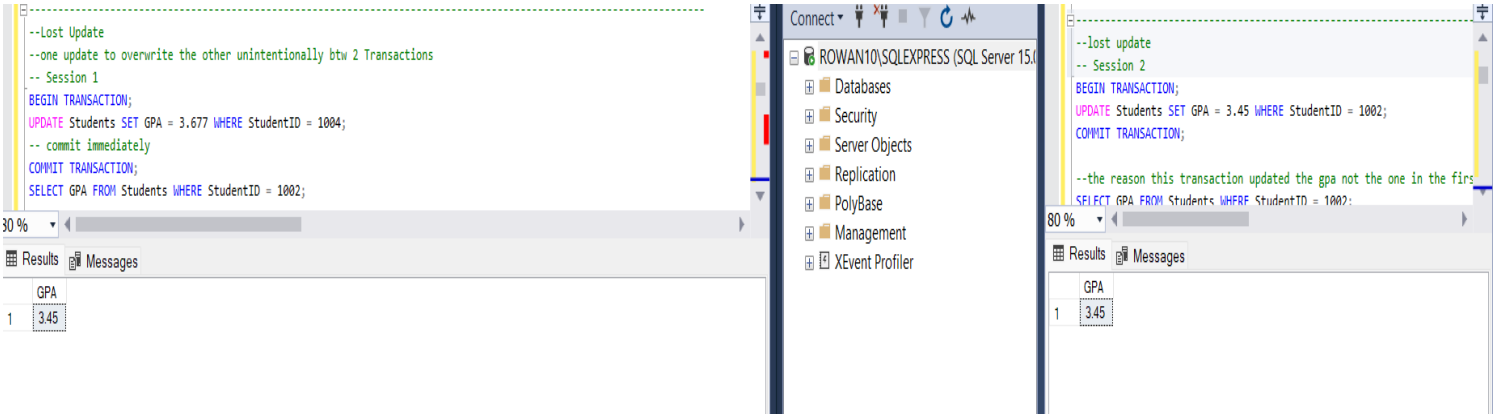
Rollback in session 1:



Back to normal 1500 Balance

Lost Update:

- Demonstrates how concurrent transactions can overwrite each other's changes.
- Last committed update is retained, earlier changes are lost.



5. Task 3: Isolation Levels

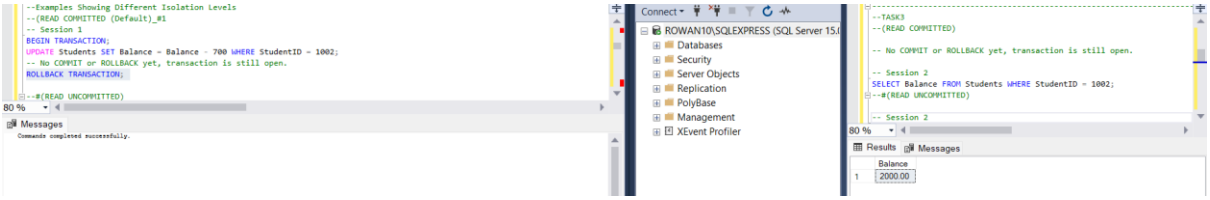
This task illustrates various SQL Server isolation levels:

- **READ COMMITTED**: Prevents dirty reads but allows non-repeatable and phantom reads.
- **READ UNCOMMITTED**: Allows dirty reads.
- **REPEATABLE READ**: Prevents modifications to selected rows during a transaction.
- **SERIALIZABLE**: Prevents new inserts in the scanned range, ensuring strict serialization.
- **SNAPSHOT**: Uses row versioning to provide a consistent view without locking.

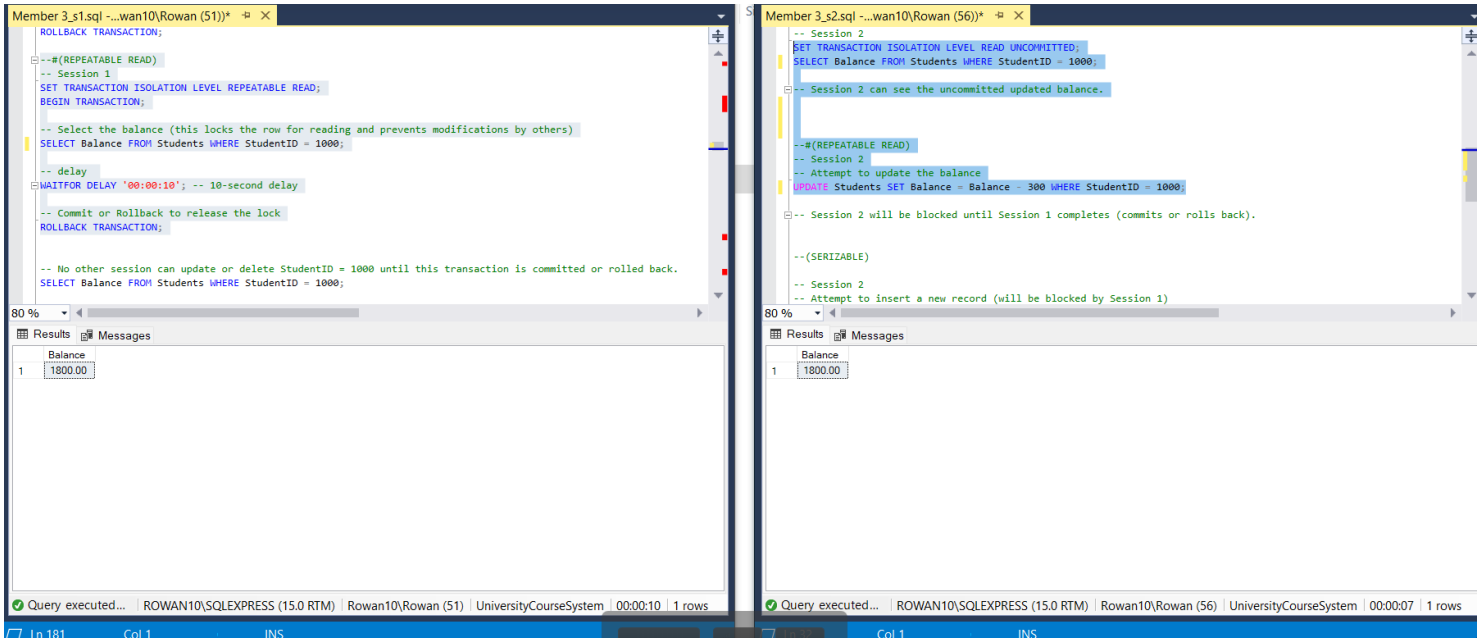
Expected Behavior:

- Each level demonstrates specific blocking, locking, or versioning behaviors.
- Emphasizes trade-offs between consistency and concurrency.

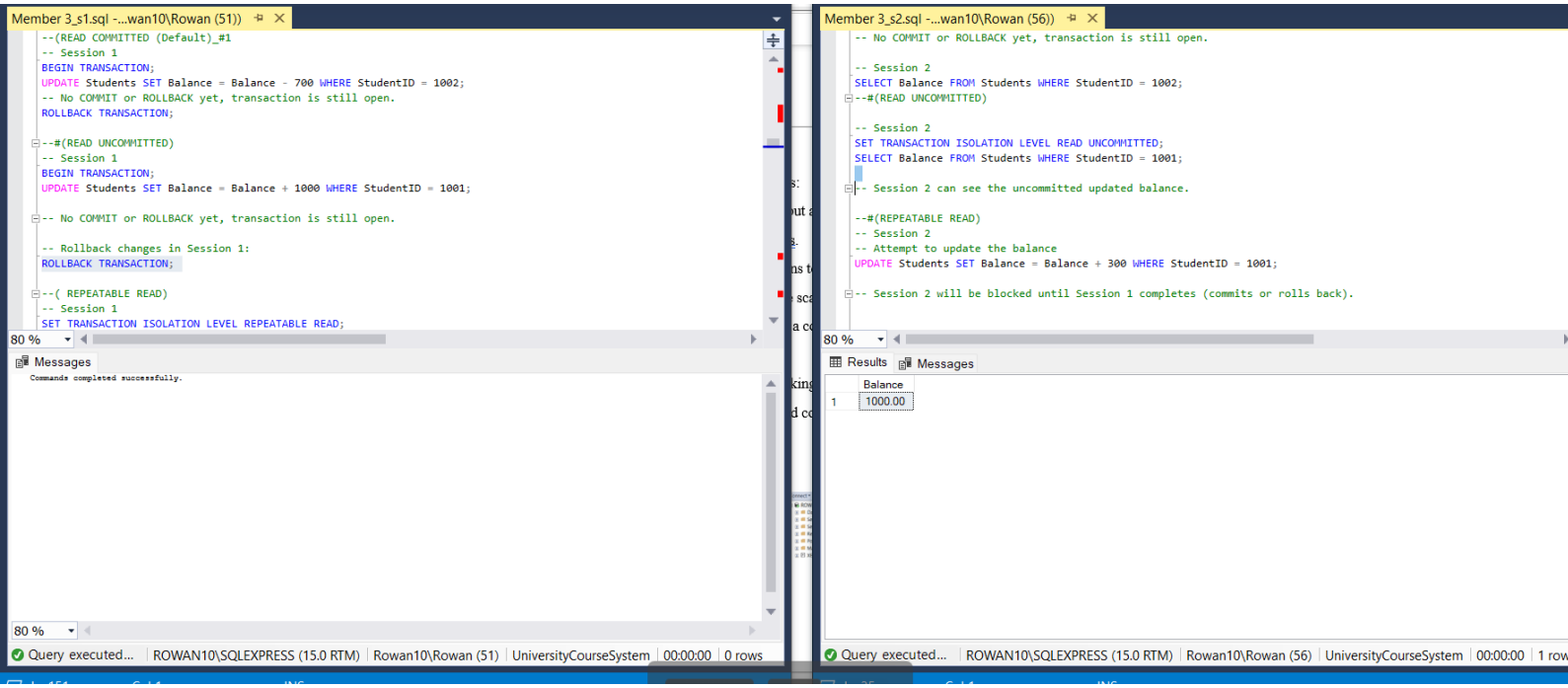
Read Committed:



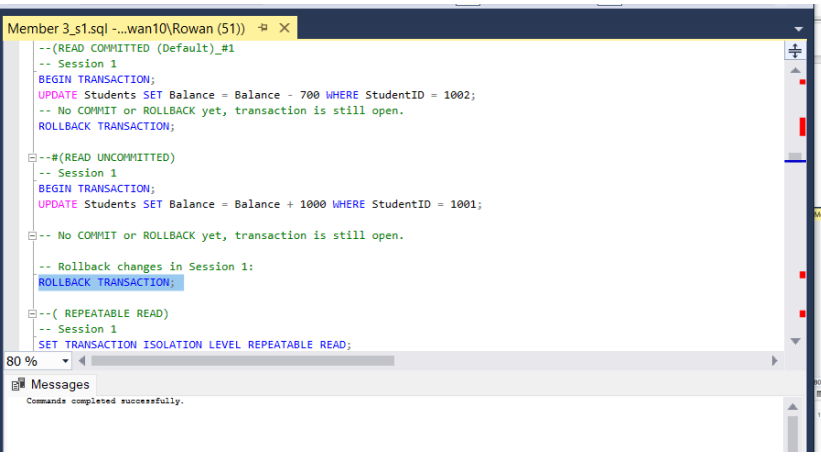
Read Uncommitted:



After Rollback



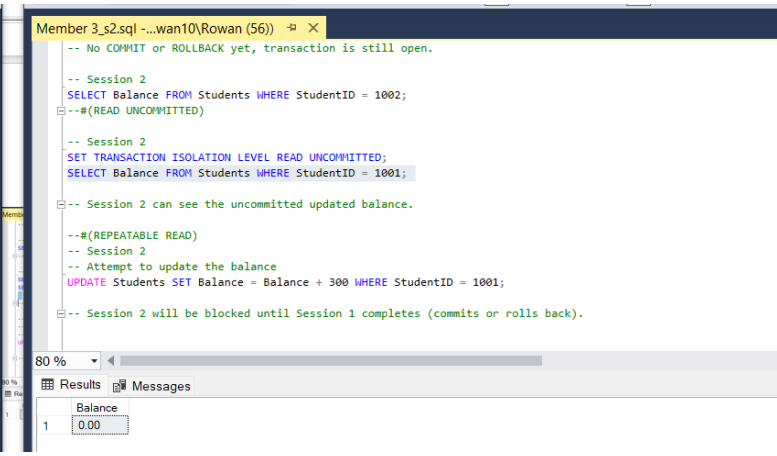
Reapetable Read:



```
--(READ COMMITTED (Default)_#1
-- Session 1
BEGIN TRANSACTION;
UPDATE Students SET Balance = Balance - 700 WHERE StudentID = 1002;
-- No COMMIT or ROLLBACK yet, transaction is still open.
ROLLBACK TRANSACTION;

--#(READ UNCOMMITTED)
-- Session 1
BEGIN TRANSACTION;
UPDATE Students SET Balance = Balance + 1000 WHERE StudentID = 1001;

-- No COMMIT or ROLLBACK yet, transaction is still open.

-- Rollback changes in Session 1:
ROLLBACK TRANSACTION;

--( REPEATABLE READ)
-- Session 1
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Messages

Commands completed successfully.

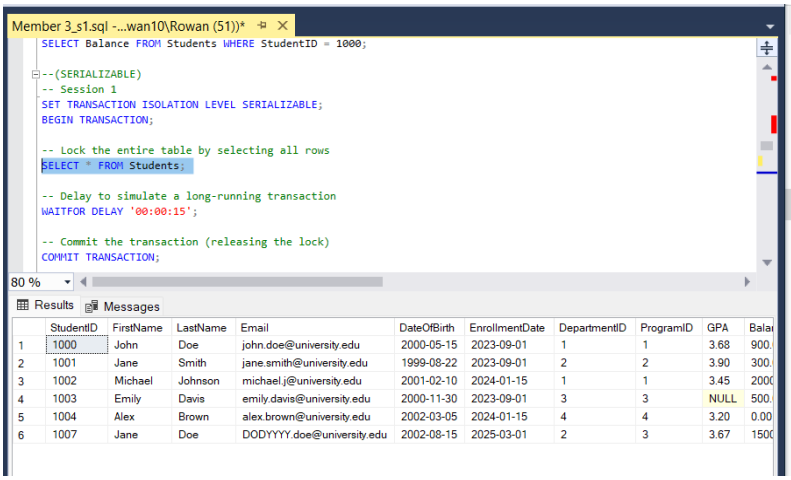


```
-- No COMMIT or ROLLBACK yet, transaction is still open.

-- Session 2
SELECT Balance FROM Students WHERE StudentID = 1002;
--#(READ UNCOMMITTED)

-- Session 2
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT Balance FROM Students WHERE StudentID = 1001;

-- Session 2 can see the uncommitted updated balance.

--#(REPEATABLE READ)
-- Session 2
-- Attempt to update the balance
UPDATE Students SET Balance = Balance + 300 WHERE StudentID = 1001;

-- Session 2 will be blocked until Session 1 completes (commits or rolls back).
```

Results Messages

	Balance
1	0.00

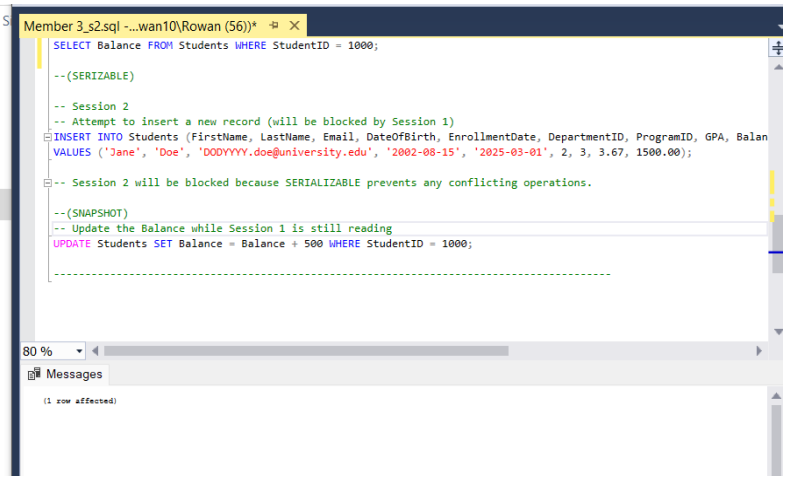
SERIALIZABLE :



```
SELECT Balance FROM Students WHERE StudentID = 1000;

--(SERIALIZABLE)
-- Session 1
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION;

-- Lock the entire table by selecting all rows
SELECT * FROM Students;

-- Delay to simulate a long-running transaction
WAITFOR DELAY '00:00:15';

-- Commit the transaction (releasing the lock)
COMMIT TRANSACTION;
```

Results Messages

	StudentID	FirstName	LastName	Email	DateOfBirth	EnrollmentDate	DepartmentID	ProgramID	GPA	Balan
1	1000	John	Doe	john.doe@university.edu	2000-05-15	2023-09-01	1	1	3.68	900.
2	1001	Jane	Smith	jane.smith@university.edu	1999-08-22	2023-09-01	2	2	3.90	300.
3	1002	Michael	Johnson	michael.j@university.edu	2001-02-10	2024-01-15	1	1	3.45	200(
4	1003	Emily	Davis	emily.davis@university.edu	2000-11-30	2023-09-01	3	3	NULL	500.
5	1004	Alex	Brown	alex.brown@university.edu	2002-03-05	2024-01-15	4	4	3.20	0.00
6	1007	Jane	Doe	DODYYYY.doe@university.edu	2002-08-15	2025-03-01	2	3	3.67	150(

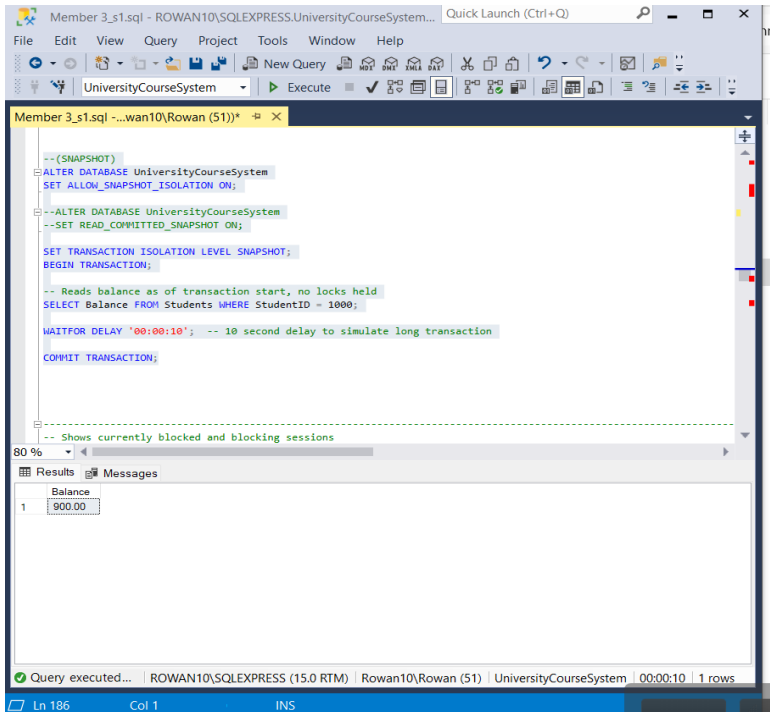


```
SELECT Balance FROM Students WHERE StudentID = 1000;

--(SERIZABLE)

-- Session 2
-- Attempt to insert a new record (will be blocked by Session 1)
INSERT INTO Students (FirstName, LastName, Email, DateOfBirth, EnrollmentDate, DepartmentID, ProgramID, GPA, Balan
VALUES ('Jane', 'Doe', 'DODYYYY.doe@university.edu', '2002-08-15', '2025-03-01', 2, 3, 3.67, 1500.00);

-- Session 2 will be blocked because SERIALIZABLE prevents any conflicting operations.

--(SNAPSHOT)
-- Update the Balance while Session 1 is still reading
UPDATE Students SET Balance = Balance + 500 WHERE StudentID = 1000;
```

Messages

(1 row affected)

SNAPSHOT:



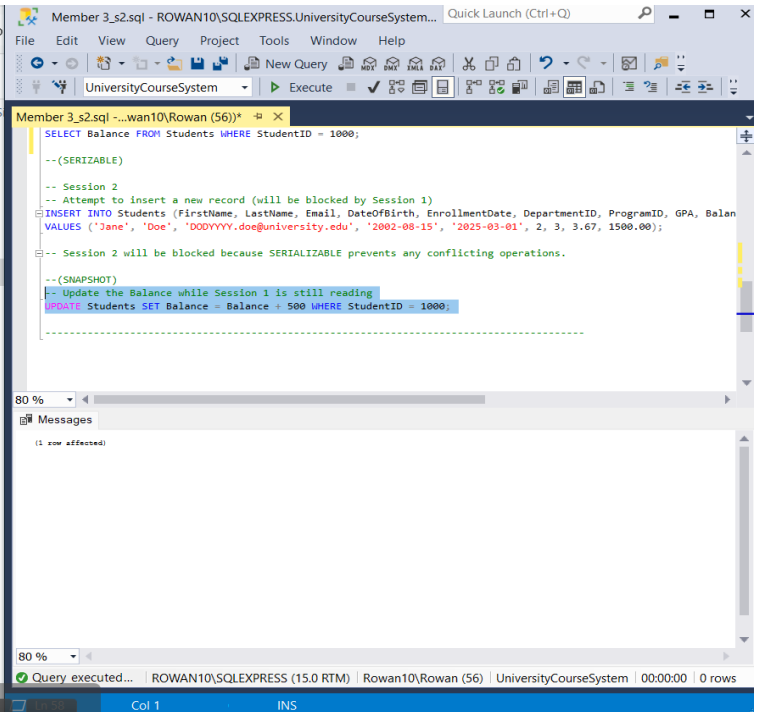
```
--(SNAPSHOT)
ALTER DATABASE UniversityCourseSystem
SET ALLOW_SNAPSHOT_ISOLATION ON;

--ALTER DATABASE UniversityCourseSystem
--SET READ_COMMITTED_SNAPSHOT ON;

SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;

-- Reads balance as of transaction start, no locks held
SELECT Balance FROM Students WHERE StudentID = 1000;

WAITFOR DELAY '00:00:10';  -- 10 second delay to simulate long transaction

COMMIT TRANSACTION;

-- Shows currently blocked and blocking sessions
```

Results Messages

	Balance
1	900.00

Query executed... ROWAN10\SQLEXPRESS (15.0 RTM) Rowan10\Rowan (51) UniversityCourseSystem 00:00:10 1 rows

Ln 186 Col 1 INS



```
SELECT Balance FROM Students WHERE StudentID = 1000;

--(SERIZABLE)

-- Session 2
-- Attempt to insert a new record (will be blocked by Session 1)
INSERT INTO Students (FirstName, LastName, Email, DateOfBirth, EnrollmentDate, DepartmentID, ProgramID, GPA, Balan
VALUES ('Jane', 'Doe', 'DODYYYY.doe@university.edu', '2002-08-15', '2025-03-01', 2, 3, 3.67, 1500.00);

-- Session 2 will be blocked because SERIALIZABLE prevents any conflicting operations.

--(SNAPSHOT)
-- Update the Balance while Session 1 is still reading
UPDATE Students SET Balance = Balance + 500 WHERE StudentID = 1000;
```

Messages

(1 row affected)

Query executed... ROWAN10\SQLEXPRESS (15.0 RTM) Rowan10\Rowan (56) UniversityCourseSystem 00:00:00 0 rows

Col 1 INS

6. Task 4: Concurrency Control Techniques

Shared Locks for Read Consistency:

- Uses HOLDLOCK and ROWLOCK to prevent modifications during read.

Member 3_s1.sql -...wan10\Rowan (51))*

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

BEGIN TRANSACTION;

-- Select courses the student is enrolled in with shared locks to prevent modifications during read
SELECT
    c.CourseID,
    c.CourseCode,
    c.CourseName,
    co.Semester,
    co.Year,
    co.Schedule
FROM Enrollments e WITH (HOLDLOCK, ROWLOCK)
JOIN CourseOfferings co ON e.OfferingID = co.OfferingID
JOIN Courses c ON co.CourseID = c.CourseID
WHERE e.StudentID = 1001 AND e.Status = 'Active';

-- Keep transaction open to hold locks while student reads the data
WAITFOR DELAY '00:00:10';  -- simulate 10 seconds reading time

COMMIT TRANSACTION;


--Uses tables: Enrollments, CourseOfferings, and Courses in a university system context.
```

80 %

Results | Messages

	CourseID	CourseCode	CourseName	Semester	Year	Schedule
1	3002	PHYS101	Mechanics	Spring	2024	MWF 09:00-10:00

Optimistic Locking for Course Enrollment:

- Applies UPDLOCK and conditional updates to safely modify capacity.

```sql
--Solving Concurrency Issues with Optimistic Locking
BEGIN TRANSACTION;

DECLARE @CurrentEnrollment INT, @MaxCapacity INT;

-- Acquire an update lock and read values
SELECT
    @CurrentEnrollment = CurrentEnrollment,
    @MaxCapacity = MaxCapacity
FROM CourseOfferings WITH (UPDLOCK, ROWLOCK)
WHERE OfferingID = 4002;

-- Check and update
IF @CurrentEnrollment < @MaxCapacity
BEGIN
    UPDATE CourseOfferings
    SET CurrentEnrollment = CurrentEnrollment + 1
    WHERE OfferingID = 4002;
```

80 %

Messages

```
(1 row affected)
Seat successfully reserved.
```

ROWVERSION for Instructor Salary:

- Ensures updates occur only if no other transactions modified the record.
- Combines security (authorized users) with concurrency control.

Member 3_s1.sql -...wan10\Rowan (51))*

```sql
GO

-- Step 2: Simulated current user and salary update with optimistic concurrency
DECLARE @InstructorID INT = 2000;
DECLARE @OldRowVer BINARY(8);
DECLARE @UserName NVARCHAR(100) = 'admin1';  -- Simulate login

-- Only allow certain users to perform the update
IF @UserName IN ('admin1', 'hr_manager')
BEGIN
    -- Capture current RowVersion
    SELECT @OldRowVer = RowVer
    FROM Instructors
    WHERE InstructorID = @InstructorID;

    -- Perform the update if RowVersion matches
    UPDATE Instructors
    SET Salary = Salary + 1000
    WHERE InstructorID = @InstructorID AND RowVer = @OldRowVer;

    IF @@ROWCOUNT = 0
        PRINT 'Salary update failed due to concurrent modification.';
    ELSE
        PRINT 'Salary updated successfully.';
END
ELSE
```

80 %

Messages

```
(1 row affected)
Salary updated successfully.
```

Trigger for Automated Tuition Deduction:

- Trigger trg_PayForAllActiveCourses deducts tuition upon activation.
- Checks if balance covers all active enrollments.
- Rolls back transaction if funds are insufficient.

```sql
ALTER TABLE CourseOfferings
ADD Price DECIMAL(10, 2) NOT NULL DEFAULT 0;
```

\

Member 3_s1.sql -...wan10\Rowan (51))*

```sql
CREATE TRIGGER trg_PayForAllActiveCourses
ON Enrollments
AFTER UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    -- Only proceed if any enrollment status changed to 'Active'
    IF EXISTS (
        SELECT 1
        FROM inserted i
        JOIN deleted d ON i.EnrollmentID = d.EnrollmentID
        WHERE i.Status = 'Active' AND d.Status <> 'Active'
    )
    BEGIN
        DECLARE @StudentID INT;

        -- Get distinct StudentIDs with activation in this update
        DECLARE activatedStudents CURSOR FOR
        SELECT DISTINCT i.StudentID
        FROM inserted i
        JOIN deleted d ON i.EnrollmentID = d.EnrollmentID
        WHERE i.Status = 'Active' AND d.Status <> 'Active';

        OPEN activatedStudents;
        FETCH NEXT FROM activatedStudents INTO @StudentID;

        WHILE @@FETCH_STATUS = 0
        BEGIN
            DECLARE @TotalFee DECIMAL(10,2);
            DECLARE @Balance DECIMAL(10,2);

            -- Calculate total fee for all active enrollments for this student
```

80 %

Messages

```
Commands completed successfully.
```

Member 3_s1.sql -...wan10\Rowan (51))*

```
        WHILE @@FETCH_STATUS = 0
        BEGIN
            DECLARE @TotalFee DECIMAL(10,2);
            DECLARE @Balance DECIMAL(10,2);

            -- Calculate total fee for all active enrollments for this student
            SELECT @TotalFee = SUM(co.Price)
            FROM Enrollments e WITH (UPDLOCK, HOLDLOCK)
            JOIN CourseOfferings co ON e.OfferingID = co.OfferingID
            WHERE e.StudentID = @StudentID AND e.Status = 'Active';

            -- Get current balance with lock
            SELECT @Balance = Balance
            FROM Students WITH (UPDLOCK, HOLDLOCK)
            WHERE StudentID = @StudentID;

            IF @Balance >= @TotalFee
            BEGIN
                -- Deduct total fee once
                UPDATE Students
                SET Balance = Balance - @TotalFee
                WHERE StudentID = @StudentID;
            END
            ELSE
            BEGIN
                DECLARE @BalanceStr VARCHAR(20);
                DECLARE @TotalFeeStr VARCHAR(20);

                SET @BalanceStr = CONVERT(VARCHAR(20), @Balance);
                SET @TotalFeeStr = CONVERT(VARCHAR(20), @TotalFee);

                RAISERROR('Student %d has insufficient balance (%s) to pay total active courses fee (%s).',
                          16, 1,
```

80 %

Messages

```
Commands completed successfully.
```

Member 3_s1.sql -...wan10\Rowan (51))*

```
            BEGIN
                -- Deduct total fee once
                UPDATE Students
                SET Balance = Balance - @TotalFee
                WHERE StudentID = @StudentID;
            END
            ELSE
            BEGIN
                DECLARE @BalanceStr VARCHAR(20);
                DECLARE @TotalFeeStr VARCHAR(20);

                SET @BalanceStr = CONVERT(VARCHAR(20), @Balance);
                SET @TotalFeeStr = CONVERT(VARCHAR(20), @TotalFee);

                RAISERROR('Student %d has insufficient balance (%s) to pay total active courses fee (%s).',
                          16, 1,
                          @StudentID,
                          @BalanceStr,
                          @TotalFeeStr);
                ROLLBACK TRANSACTION;
                CLOSE activatedStudents;
                DEALLOCATE activatedStudents;
                RETURN;
            END

            FETCH NEXT FROM activatedStudents INTO @StudentID;
        END

        CLOSE activatedStudents;
        DEALLOCATE activatedStudents;
    END
END;
GO
```

80 %

Messages

```
Commands completed successfully.
```

80 %

Query executed successfully. | ROWAN10\SQLEXPRESS (15.0 RTM) | Rowan10

```
--TEST
UPDATE Enrollments
SET Status = 'Active'
WHERE EnrollmentID = 5000;  --it was Completed


SELECT StudentID, Balance
FROM Students
WHERE StudentID = (SELECT StudentID FROM Enrollments WHERE EnrollmentID = 5000); --done
```

80 %

Messages

```
(1 row affected)
```

```
SELECT StudentID, Balance
FROM Students
WHERE StudentID = (SELECT StudentID FROM Enrollments WHERE EnrollmentID = 5000); --done
```

80 %

Results | Messages

	StudentID	Balance
1	1000	1400.00

---

7. System Monitoring

A diagnostic query on sys.dm_exec_requests tracks blocked and blocking sessions.

Expected Behavior:

- Identifies session wait types, blocked resources, and active conflicts.

```
-------------------------------------------------------------------
-- Shows currently blocked and blocking sessions
SELECT
    blocking_session_id,
    session_id,
    wait_type,
    wait_time,
    wait_resource,
    last_wait_type
FROM sys.dm_exec_requests
WHERE blocking_session_id <> 0;
-------------------------------------------------------------------
```

---

8. Conclusion

This implementation demonstrates comprehensive handling of transactional safety, isolation, and concurrency challenges within a university course management context. By combining procedural logic, error handling, and isolation strategies, the system ensures robust and predictable multi-user behavior in SQL Server.

Keywords: SQL Server, Transactions, Concurrency, Isolation Levels, Triggers, Locking, Rowversion, University Database.

---

4. Indexing and Security

1. Objective

Now we outline the implementation of key database performance, indexing, and security enhancements applied to the UniversityCourseSystem database. The goal is to improve query efficiency, enforce fine-grained access control, and log critical system changes for auditing purposes.

2. Indexing Strategy

Overview

Indexes were strategically created on frequently queried columns to enhance SELECT performance, especially in large-scale student/course datasets.

Implemented Indexes

Table	Index Name	Indexed Columns
Students	IX_Students_Email	Email
Courses	IX_Courses_CourseCode	CourseCode
Enrollments	IX_Enrollments_StudentID	StudentID
Enrollments	IX_Enrollments_OfferingID	OfferingID
Enrollments	IX_Enrollments_Status	Status
Waitlists	IX_Waitlists_OfferingStudent	OfferingID, StudentID
StudentStatus	IX_StudentStatus_StudentID	StudentID
CourseOfferings	IX_CourseOfferings_Semester_Year	Semester, Year
CourseOfferings	IX_CourseOfferings_CourseID	CourseID
CourseOfferings	IX_CourseOfferings_InstructorID	InstructorID
CourseOfferings	IX_CourseOfferings_Year	Year (INCLUDE CourseID, etc.)
CoursePrerequisites	IX_CoursePrerequisites_CourseID	CourseID

PaymentTransactions	IX_PaymentTransactions_StudentID	StudentID
Students	IX_Students_ProgramID	ProgramID

Impact

These indexes reduce table scan operations and significantly improve read efficiency in transactional and reporting workloads.

3. Role-Based Access Control (RBAC)

Created Roles

- StudentRole
- InstructorRole
- RegistrarRole
- AdminRole

Permissions Summary

Role	Granted Permissions
StudentRole	Read-only on courses, enrollments, financials; insert on payments
InstructorRole	Read student names; update grades/status
RegistrarRole	Full DML access on core academic/financial tables
AdminRole	Full control over the entire database

Revoked/DENIED Permissions

Sensitive data fields such as GPA, Date of Birth, and Balance are explicitly denied from students and instructors. DML rights are restricted for StudentRole.

---

4. Row-Level Security (RLS)

Purpose

To enforce data isolation by user identity, ensuring users only access records they are entitled to.

Implementation

Two inline table-valued functions were created:

- fn_SecurityPredicate(@InstructorID) for instructors.
- fn_StudentSecurityPredicate(@StudentID) for students.

These are applied via SQL Server SECURITY POLICY objects:

- InstructorsSecurityPolicy on the Instructors table.
- StudentsSecurityPolicy on the Students table.

User Mapping Tables

- UserStudentMapping: Maps LoginName to StudentID.
- UserInstructorMapping: Maps LoginName to InstructorID.

---

5. Audit Log & Sequence

Audit Table

AuditLog captures system events such as role changes or data updates.

Sequence

AuditSequence provides consistent, incrementing values for AuditID.

Sample Entry

Upon first run, the setup inserts an initial log:
'security setup' with the executing username.

---

6. Performance Benchmarking

Test Methodology

A temporary table #PerformanceResults is used to log execution times (ms) of specific queries before and after relevant indexes are applied.

Tested Scenarios

1. **Search by Email** (Students.Email)
2. **Course by CourseCode** (Courses.CourseCode)
3. **Enrollment by StudentID** (Enrollments.StudentID)

	QueryName	ExecutionTimeMS	IndexStatus
1	CourseByCourseCode	0	With Index
2	CourseByCourseCode	1	Without Index
3	EnrollmentByStuden...	0	With Index
4	EnrollmentByStuden...	0	Without Index
5	StudentByEmail	0	With Index
6	StudentByEmail	2	Without Index