جامعة الإسكندرية

ALEXANDRIA
UNIVERSITY

Adv Database Project

Team Members

Name	ID	Task
حمزة حسين يوسف عمران	22011501	Shared in Tasks 1, 2 Did Task 3 Alone
معاذ مصطفى عبدالحميد مصطفى	22010263	2
احمد عطيه احمد عثمان	22010025	1
كريم محمد سامي أبوشادي	22010378	4

🎓 Alexandria University Database Schema Explanation

🗂️ Overview:

The database is designed to manage and organize academic data for Alexandria University. It includes **students, instructors, departments, courses, and enrollments**, with additional logic through **functions, views, and triggers** to automate and simplify common operations.

🧱 Tables and Their Roles

1. Department

- Stores information about university departments.
- Key Fields: DeptID (PK), DeptName, DeptPhone
- Connected to: Student, Instructor, and Course tables.

---

2. Student

- Contains personal and academic info for each student.
- Key Fields: StdID (PK), StdName, Gender, Birthdate, Phone, DeptID (FK)
- Linked to: Department (via DeptID), Enrollment (via StdID)

3. Instructor

- Contains details about instructors.
- Key Fields: InsID (PK), InsName, Salary, DeptID (FK)
- Linked to: Department (via DeptID), Course (via InsID)

---

4. Course

- Represents courses offered at the university.
- Key Fields: CourseID (PK), CourseName, Hours, DeptID (FK), InsID (FK)
- Linked to: Department, Instructor, and Enrollment

---

5. Enrollment

- Tracks which students are enrolled in which courses and their grades.
- Key Fields: StdID (FK), CourseID (FK), Grade
- Composite Primary Key: (StdID, CourseID)
- Links Student and Course tables.

---

6. Log

- Records events such as inserts, updates, and deletes.
- Key Fields: LogID (PK), ActionType, ActionTime, TableName, Details

- Automatically updated using triggers.

🔗 Relationships Between Tables

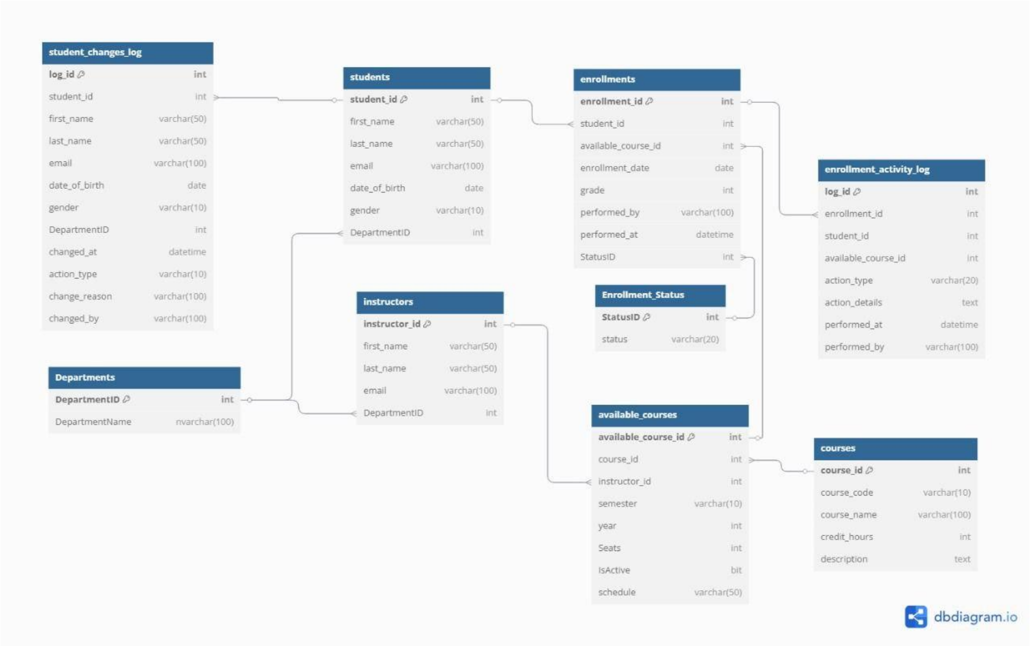
From Table	Field	To Table	Type
Student	DeptID	Department	Many-to-One
Instructor	DeptID	Department	Many-to-One
Course	DeptID	Department	Many-to-one
Course	InsID	Instructor	Many-to-One
Enrollment	StdID	Student	Many-to-One
Enrollment	CourseID	Course	Many-to-One

⚙ Functions

- **fn_GetStudentAge(@StdID)**: Returns the age of a student.
- **fn_GetInstructorSalaryAfterRaise(@InsID, @RaisePercent)**: Calculates instructor salary after applying a raise.

Views

- **vw_StudentDetails**: Combines student info with department name.
- **vw_CourseDetails**: Combines course info with department and instructor names.
- **vw_EnrollmentStatus**: Shows student-course enrollments and grades.



Member 2: Procedures, functions, and triggers

Functions Report

1. Scalar-Valued Function: GetCourseNameByID

1.1 Overview

This section demonstrates the implementation of a scalar-valued function named `GetCourseNameByID`, fulfilling the project requirement to create at least one scalar-valued function.

1.2 Function Purpose

The `GetCourseNameByID` function retrieves the course name from the `courses` table using the provided `course_id` as input.

This function improves query readability and supports data organization within the University Course Enrollment and Management System.

1.3 Function Definition

Written in T-SQL, this scalar-valued function accepts a course ID as input and returns the corresponding course name.

Input:

- - @course_id (INT): The Course ID.

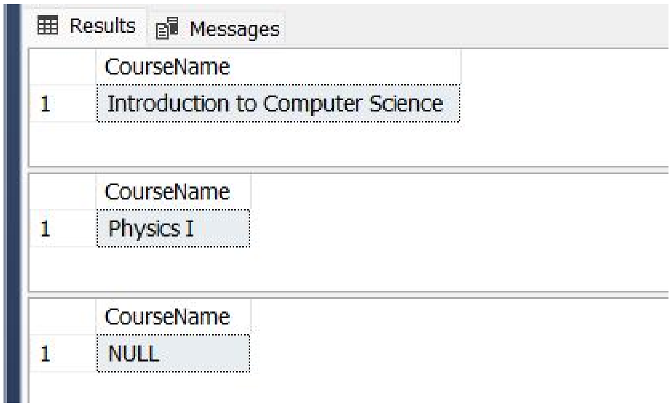
Output:

- - Course Name (VARCHAR(100))

1.4 Sample Usage

```sql
SELECT dbo.GetCourseNameByID(1) AS CourseName;
SELECT dbo.GetCourseNameByID(8) AS CourseName;
SELECT dbo.GetCourseNameByID(88) AS CourseName;
```

1.5 Function Execution



	CourseName
1	Introduction to Computer Science

	CourseName
1	Physics I

	CourseName
1	NULL

2. Scalar-Valued Function: GetCourseIDByEnrollment

2.1 Overview

This section presents the scalar-valued function named `GetCourseIDByEnrollment`, implemented as part of the scalar function requirement.

2.2 Function Purpose

The function retrieves the `course_id` from the `available_courses` table based on the provided `available_course_id`.

It simplifies mapping available courses to their corresponding course IDs, essential for managing enrollments.

2.3 Function Definition

Written in T-SQL, it accepts `available_course_id` as input and returns the corresponding `course_id`.

Input:

- - @available_course_id (INT): The ID from `available_courses`.

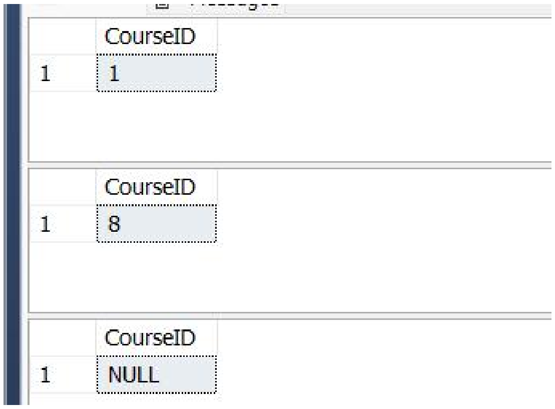
Output:

- - course_id (INT)

2.4 Sample Usage

```sql
SELECT dbo.GetCourseIDByEnrollment(1) AS CourseID;
SELECT dbo.GetCourseIDByEnrollment(8) AS CourseID;
SELECT dbo.GetCourseIDByEnrollment(45) AS CourseID;
```

2.5 Function Execution



	CourseID
1	1

	CourseID
1	8

	CourseID
1	NULL

Stored Procedure Report

2. Stored Procedure: RegisterStudentInCourse

2.1 Overview

This section presents the stored procedure `RegisterStudentInCourse`, which fulfills the project requirement to implement at least one stored procedure. It is designed to handle student registration in a selected course from the `available_courses` table.

2.2 Procedure Purpose

The `RegisterStudentInCourse` procedure enables student enrollment into a specific course offering while enforcing constraints such as preventing duplicate enrollments and ensuring the availability of the selected course.

It provides a reliable and reusable approach to manage course registrations, supporting administrative operations in the University Course Enrollment and Management System.

2.3 Procedure Definition

This procedure uses T-SQL and supports robust error handling through a `TRY...CATCH` block. It takes the student ID and available course ID as inputs and returns a message via an output parameter to confirm the result of the registration process.

Inputs:
- `@StudentID` (INT): The ID of the student attempting to enroll.
- `@AvailableCourseID` (INT): The ID of the course offering from the `available_courses` table.

Output:
- `@Message` (NVARCHAR(200)): A textual message indicating the outcome of the operation (e.g., success, duplicate, or error).

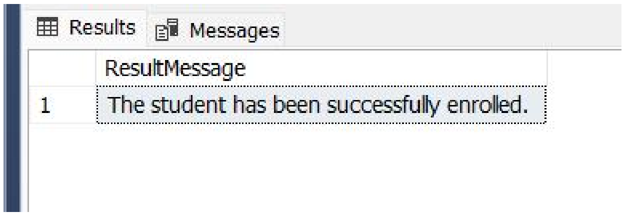
2.4 Key Logic

- Validates that the student is not already enrolled in the selected course.
- Confirms the existence of the specified available course.
- If validation passes, inserts a new enrollment record with the current timestamp, registration status, and system user as the performer.
- Captures and returns appropriate messages in all cases.

2.5 Sample Usage

```
DECLARE @msg NVARCHAR(200);

EXEC dbo.RegisterStudentInCourse
    @StudentID = 1,
    @AvailableCourseID = 2,
    @Message = @msg OUTPUT;

SELECT @msg AS ResultMessage;
```

2.6 Procedure Execution Screenshot



	ResultMessage
1	The student has been successfully enrolled.

Functions and Triggers Report

3. Function: GetInstructorNameByEnrollmentID

3.1 Overview

This section introduces a scalar-valued function named GetInstructorNameByEnrollmentID. It fulfills the requirement of creating an additional scalar function.

3.2 Function Purpose

The purpose of this function is to retrieve the full name of the instructor assigned to a specific enrollment by joining multiple related tables.

3.3 Function Definition

The function accepts an enrollment ID and returns the instructor’s full name (first name + last name).

- Input:
  - @EnrollmentID (INT): The enrollment record ID
- Output:
  - Full Instructor Name (VARCHAR(200))

```
CREATE FUNCTION GetInstructorNameByEnrollmentID (@EnrollmentID INT)
RETURNS VARCHAR(200)
AS
BEGIN
   DECLARE @InstructorName VARCHAR(200)
   SELECT @InstructorName = i.first_name + ' ' + i.last_name
   FROM enrollments e
   JOIN available_courses ac ON e.available_course_id = ac.available_course_id
   JOIN instructors i ON ac.instructor_id = i.instructor_id
   WHERE e.enrollment_id = @EnrollmentID
   RETURN @InstructorName
END
```

3.4 Sample Usage

```
SELECT dbo.GetInstructorNameByEnrollmentID(1) AS InstructorName;
SELECT dbo.GetInstructorNameByEnrollmentID(78) AS InstructorName;
SELECT dbo.GetInstructorNameByEnrollmentID(9) AS InstructorName;
SELECT dbo.GetInstructorNameByEnrollmentID(22) AS InstructorName;
```

4. INSTEAD OF Trigger: Prevent Duplicate Student Email

4.1 Overview

This trigger prevents inserting a student with an email that already exists in the system. It is created using the INSTEAD OF INSERT mechanism.

4.2 Trigger Definition

```
CREATE TRIGGER trg_PreventDuplicateEmail
ON students
INSTEAD OF INSERT
AS
BEGIN
  IF EXISTS (
    SELECT 1
    FROM students s
    JOIN inserted i ON s.email = i.email
  )
  BEGIN
    PRINT 'This email is already registered in the system. We cannot add the student.'
    RETURN
  END

  INSERT INTO students (
    first_name, last_name, email, date_of_birth, gender, DepartmentID
  )
  SELECT
    first_name, last_name, email, date_of_birth, gender, DepartmentID
  FROM inserted
END
```

4.3 Sample Usage

```
-- Valid Insertion
INSERT INTO students (first_name, last_name, email, date_of_birth, gender, DepartmentID)
VALUES ('Ali', 'Hassan', 'ali.hassan@example.com', '2001-05-20', 'M', 1);

-- Duplicate Email Insertion (will be blocked)
INSERT INTO students (first_name, last_name, email, date_of_birth, gender, DepartmentID)
```

```
VALUES ('Omar', 'Youssef', 'ali.hassan@example.com', '2002-03-15', 'M', 2);
```

5. AFTER UPDATE/DELETE Trigger: Log Student Changes

5.1 Overview

This trigger logs changes made to student records into a student_changes_log table upon update or delete operations.

5.2 Trigger Definition

```
CREATE TRIGGER trg_LogStudentChanges
ON students
AFTER UPDATE, DELETE
AS
BEGIN
  SET NOCOUNT ON;
  IF EXISTS (SELECT * FROM INSERTED)
  BEGIN
    INSERT INTO student_changes_log (
      student_id, first_name, last_name, email,
      date_of_birth, gender, DepartmentID,
      changed_at, action_type, change_reason, changed_by
    )
    SELECT
      d.student_id, d.first_name, d.last_name, d.email,
      d.date_of_birth, d.gender, d.DepartmentID,
      GETDATE(), 'UPDATE', 'Record updated', SYSTEM_USER
    FROM DELETED d
    JOIN INSERTED i ON d.student_id = i.student_id;
  END
  ELSE
  BEGIN
    INSERT INTO student_changes_log (
      student_id, first_name, last_name, email,
      date_of_birth, gender, DepartmentID,
      changed_at, action_type, change_reason, changed_by
    )
    SELECT
      student_id, first_name, last_name, email,
      date_of_birth, gender, DepartmentID,
      GETDATE(), 'DELETE', 'Record deleted', SYSTEM_USER
    FROM DELETED;
    END
END
```

5.3 Sample Usage

```
INSERT INTO students (first_name, last_name, email, date_of_birth, gender, DepartmentID)
VALUES ('Sara', 'Ibrahim', 'sara.ibrahim@example.com', '2000-12-12', 'F', 2);

UPDATE students
SET last_name = 'Mohamed'
WHERE email = 'sara.ibrahim@example.com';

SELECT * FROM student_changes_log WHERE email = 'sara.ibrahim@example.com';
```

6. AFTER Trigger: Log Enrollment Activities

6.1 Overview

This trigger logs INSERT, UPDATE, and DELETE operations on the enrollments table into the enrollment_activity_log.

6.2 Trigger Definition

```
CREATE TRIGGER trg_LogEnrollmentActivity
ON enrollments
AFTER INSERT, UPDATE, DELETE
AS
BEGIN
    SET NOCOUNT ON;
    IF EXISTS (SELECT * FROM INSERTED) AND NOT EXISTS (SELECT * FROM DELETED)
    BEGIN
        INSERT INTO enrollment_activity_log (
            enrollment_id, student_id, available_course_id,
            action_type, action_details, performed_at, performed_by
        )
        SELECT
            i.enrollment_id, i.student_id, i.available_course_id,
            'INSERT', 'New enrollment created', GETDATE(), i.performed_by
        FROM INSERTED i;
    END
    ELSE IF EXISTS (SELECT * FROM INSERTED) AND EXISTS (SELECT * FROM DELETED)
    BEGIN
```

```
        INSERT INTO enrollment_activity_log (
            enrollment_id, student_id, available_course_id,
            action_type, action_details, performed_at, performed_by
        )
        SELECT
            i.enrollment_id, i.student_id, i.available_course_id,
            'UPDATE',
            'Enrollment updated. Old grade: ' + ISNULL(CAST(d.grade AS VARCHAR), 'NULL') +
            ', New grade: ' + ISNULL(CAST(i.grade AS VARCHAR), 'NULL') +
            ', Old status: ' + ISNULL(CAST(d.StatusID AS VARCHAR), 'NULL') +
            ', New status: ' + ISNULL(CAST(i.StatusID AS VARCHAR), 'NULL'),
            GETDATE(),
            ISNULL(i.performed_by, SYSTEM_USER)
        FROM INSERTED i
        JOIN DELETED d ON i.enrollment_id = d.enrollment_id;
    END
    ELSE IF NOT EXISTS (SELECT * FROM INSERTED) AND EXISTS (SELECT * FROM
DELETED)
    BEGIN
        INSERT INTO enrollment_activity_log (
            enrollment_id, student_id, available_course_id,
            action_type, action_details, performed_at, performed_by
        )
        SELECT
            d.enrollment_id, d.student_id, d.available_course_id,
            'DELETE', 'Enrollment deleted', GETDATE(), SYSTEM_USER
        FROM DELETED d;
    END
END
```

6.3 Sample Usage

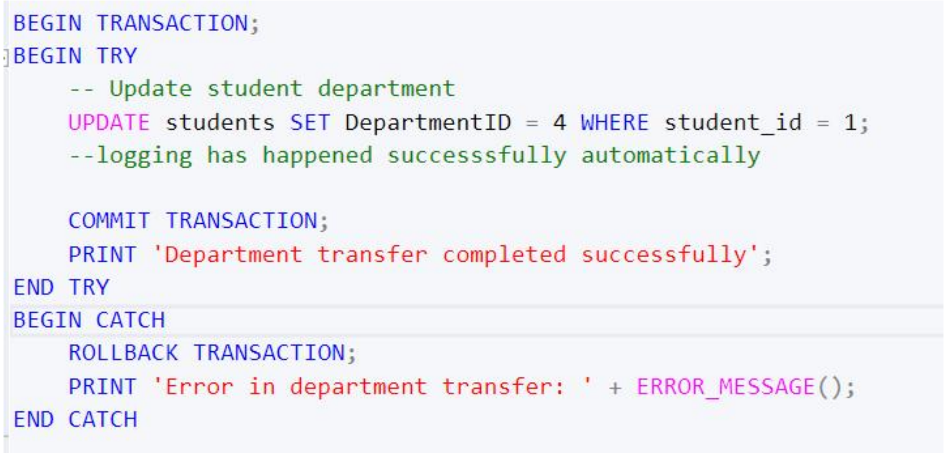
```
INSERT INTO enrollments (student_id, available_course_id, grade, StatusID, performed_by)
VALUES (1, 1, NULL, 1, SYSTEM_USER);

UPDATE enrollments
SET grade = 95, StatusID = 2, performed_by = SYSTEM_USER
WHERE enrollment_id = 1;

SELECT * FROM enrollment_activity_log WHERE enrollment_id = 1;
```

Member 3: Transactions and Concurrency

(حمزة حسين يوسف عمران 22011501)

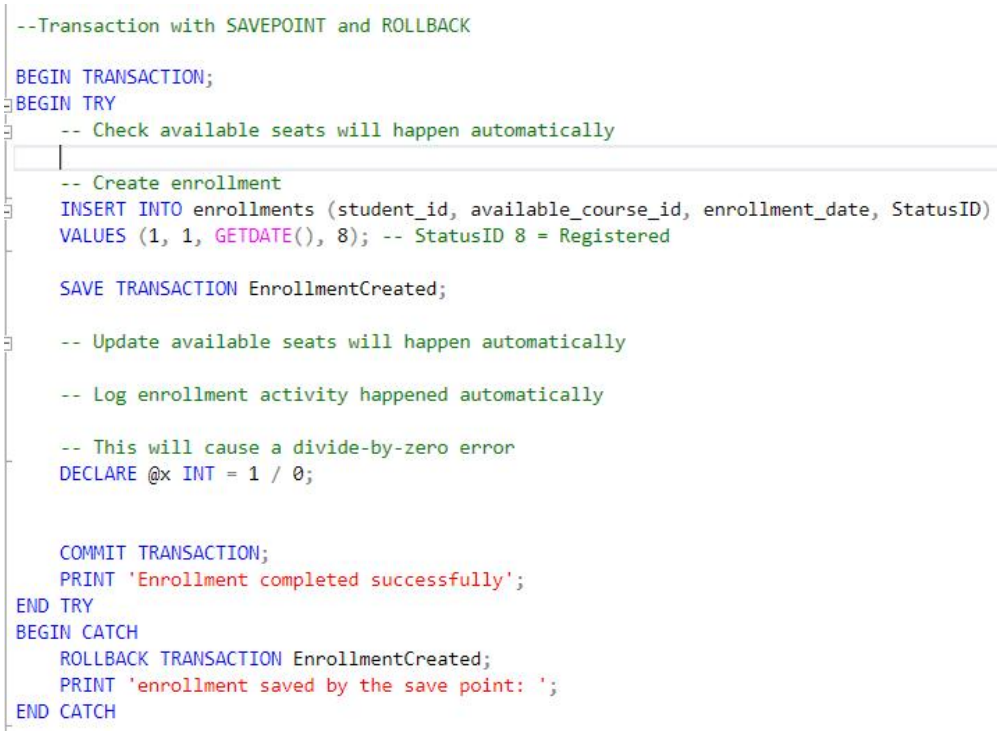
1. Transaction Management

- Simple transactions use TRY...CATCH blocks for commit/rollback safety.



```
BEGIN TRANSACTION;
BEGIN TRY
    -- Update student department
    UPDATE students SET DepartmentID = 4 WHERE student_id = 1;
    --logging has happened successsfully automatically

    COMMIT TRANSACTION;
    PRINT 'Department transfer completed successfully';
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT 'Error in department transfer: ' + ERROR_MESSAGE();
END CATCH
```

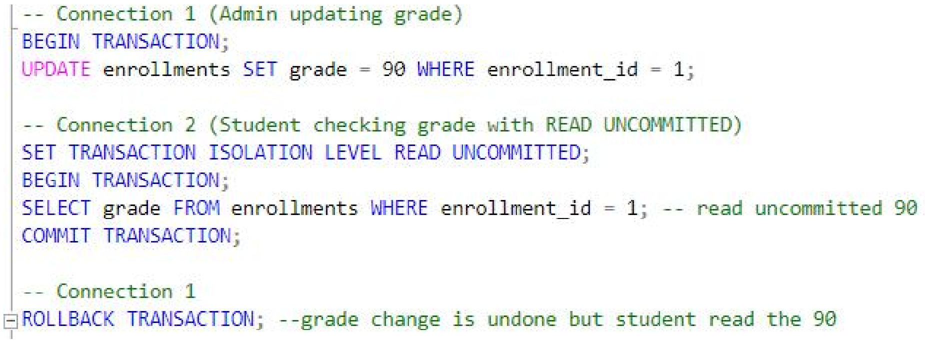
- Savepoints allow partial rollback; rollback after error preserves earlier inserts.



```
--Transaction with SAVEPOINT and ROLLBACK

BEGIN TRANSACTION;
BEGIN TRY
    -- Check available seats will happen automatically

    -- Create enrollment
    INSERT INTO enrollments (student_id, available_course_id, enrollment_date, StatusID)
    VALUES (1, 1, GETDATE(), 8); -- StatusID 8 = Registered

    SAVE TRANSACTION EnrollmentCreated;

    -- Update available seats will happen automatically

    -- Log enrollment activity happened automatically

    -- This will cause a divide-by-zero error
    DECLARE @x INT = 1 / 0;


    COMMIT TRANSACTION;
    PRINT 'Enrollment completed successfully';
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION EnrollmentCreated;
    PRINT 'enrollment saved by the save point: ';
END CATCH
```

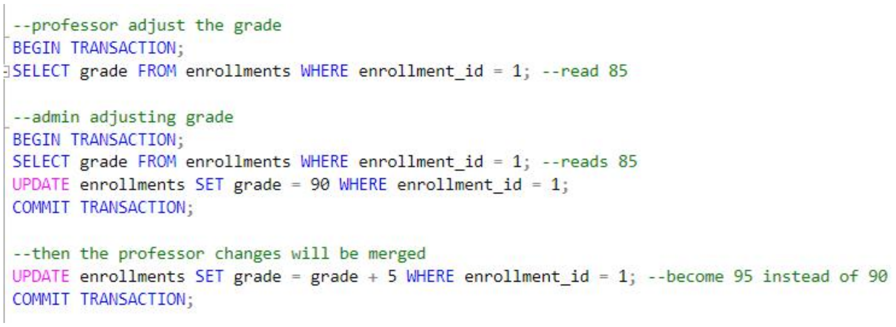
2. Concurrency Problem Demonstrations

- Dirty Read: Demonstrates READ UNCOMMITTED allowing access to uncommitted changes.



```
-- Connection 1 (Admin updating grade)
BEGIN TRANSACTION;
UPDATE enrollments SET grade = 90 WHERE enrollment_id = 1;

-- Connection 2 (Student checking grade with READ UNCOMMITTED)
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN TRANSACTION;
SELECT grade FROM enrollments WHERE enrollment_id = 1; -- read uncommitted 90
COMMIT TRANSACTION;

-- Connection 1
ROLLBACK TRANSACTION; --grade change is undone but student read the 90
```

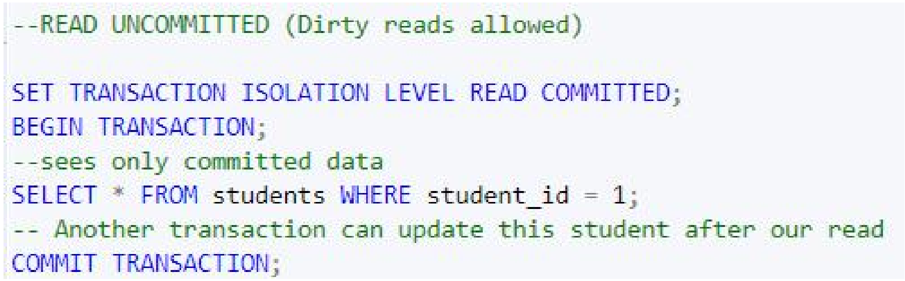
- Lost Update: Two sessions read and write the same row causing overwrite issues.



```
--professor adjust the grade
BEGIN TRANSACTION;
SELECT grade FROM enrollments WHERE enrollment_id = 1; --read 85

--admin adjusting grade
BEGIN TRANSACTION;
SELECT grade FROM enrollments WHERE enrollment_id = 1; --reads 85
UPDATE enrollments SET grade = 90 WHERE enrollment_id = 1;
COMMIT TRANSACTION;

--then the professor changes will be merged
UPDATE enrollments SET grade = grade + 5 WHERE enrollment_id = 1; --become 95 instead of 90
COMMIT TRANSACTION;
```

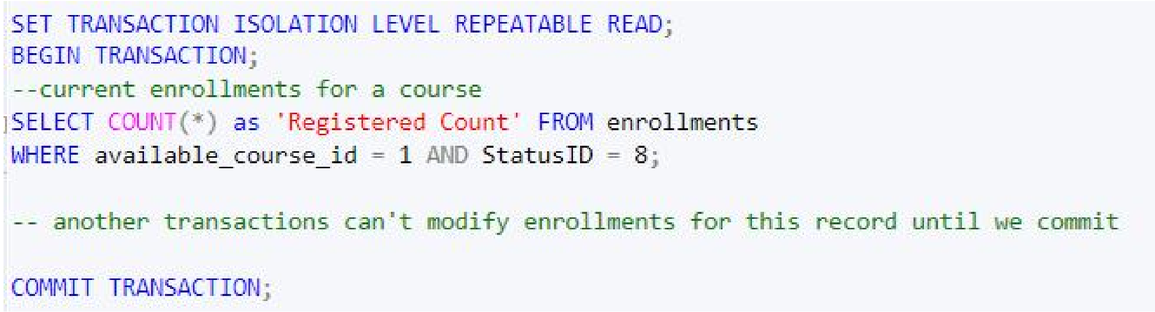
3. Isolation Levels

Read Uncomitted: Dirty read of uncommitted data



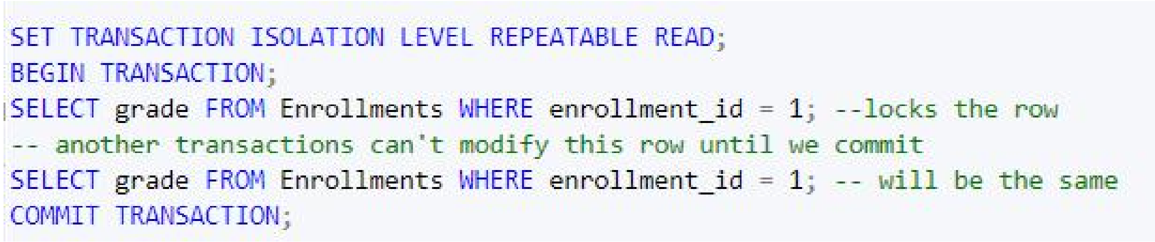
```
--READ UNCOMMITTED (Dirty reads allowed)

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRANSACTION;
--sees only committed data
SELECT * FROM students WHERE student_id = 1;
-- Another transaction can update this student after our read
COMMIT TRANSACTION;
```

READ COMMITTED: Reads only committed data.



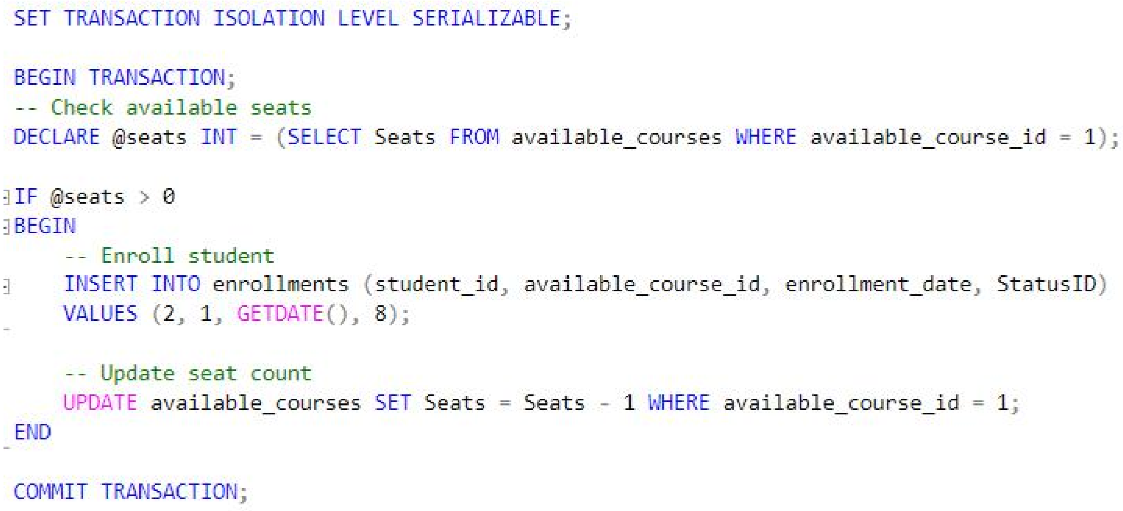
```sql
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION;
--current enrollments for a course
SELECT COUNT(*) as 'Registered Count' FROM enrollments
WHERE available_course_id = 1 AND StatusID = 8;

-- another transactions can't modify enrollments for this record until we commit

COMMIT TRANSACTION;
```

REPEATABLE READ: Prevents updates/deletes to read rows until transaction ends.



```sql
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION;
SELECT grade FROM Enrollments WHERE enrollment_id = 1; --locks the row
-- another transactions can't modify this row until we commit
SELECT grade FROM Enrollments WHERE enrollment_id = 1; -- will be the same
COMMIT TRANSACTION;
```

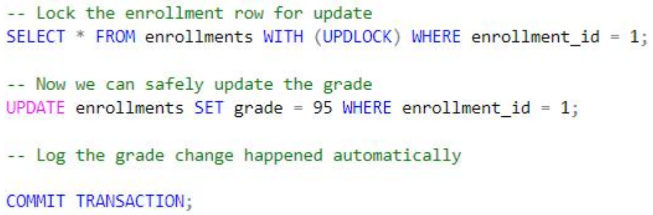
SERIALIZABLE: Prevents new rows from being inserted that affect existing reads.



```sql
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION;
-- Check available seats
DECLARE @seats INT = (SELECT Seats FROM available_courses WHERE available_course_id = 1);

IF @seats > 0
BEGIN
    -- Enroll student
    INSERT INTO enrollments (student_id, available_course_id, enrollment_date, StatusID)
    VALUES (2, 1, GETDATE(), 8);

    -- Update seat count
    UPDATE available_courses SET Seats = Seats - 1 WHERE available_course_id = 1;
END

COMMIT TRANSACTION;
```

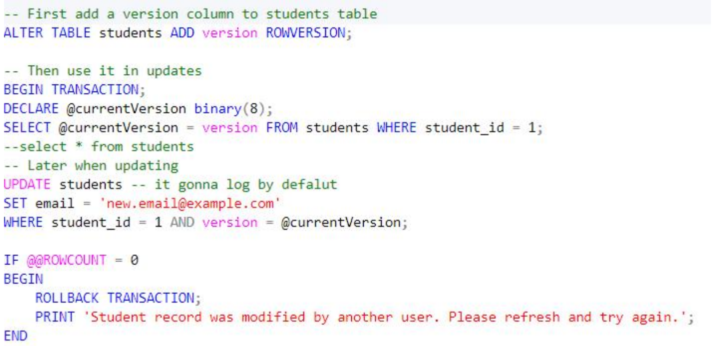
4. Concurrency Solutions

- Pessimistic Locking: Uses UPDLOCK to prevent concurrent updates.



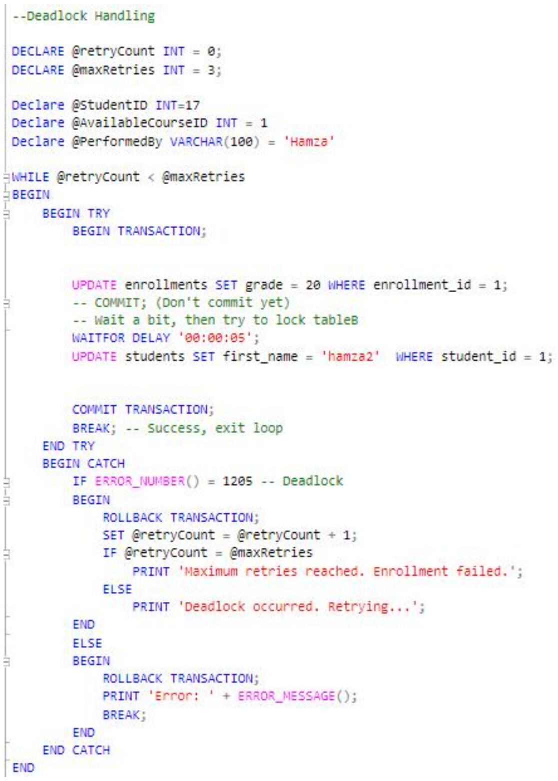
```sql
-- Lock the enrollment row for update
SELECT * FROM enrollments WITH (UPDLOCK) WHERE enrollment_id = 1;

-- Now we can safely update the grade
UPDATE enrollments SET grade = 95 WHERE enrollment_id = 1;

-- Log the grade change happened automatically

COMMIT TRANSACTION;
```

- Optimistic Concurrency: Uses ROWVERSION to detect update conflicts.



```sql
-- First add a version column to students table
ALTER TABLE students ADD version ROWVERSION;

-- Then use it in updates
BEGIN TRANSACTION;
DECLARE @currentVersion binary(8);
SELECT @currentVersion = version FROM students WHERE student_id = 1;
--select * from students
-- Later when updating
UPDATE students -- it gonna log by defalut
SET email = 'new.email@example.com'
WHERE student_id = 1 AND version = @currentVersion;

IF @@ROWCOUNT = 0
BEGIN
    ROLLBACK TRANSACTION;
    PRINT 'Student record was modified by another user. Please refresh and try again.';
END
```

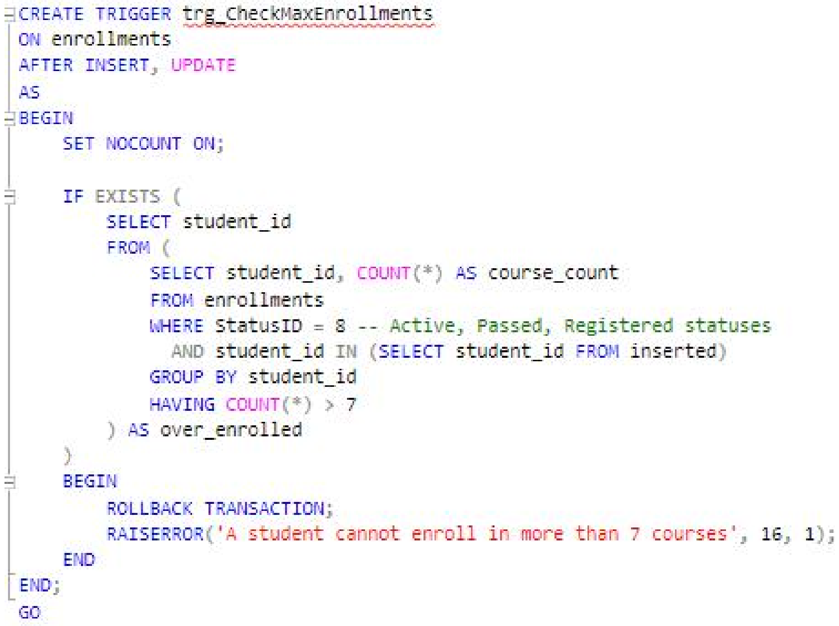
- Deadlock Handling: Retries transactions when error 1205 occurs.



```sql
--Deadlock Handling

DECLARE @retryCount INT = 0;
DECLARE @maxRetries INT = 3;

Declare @StudentID INT=17
Declare @AvailableCourseID INT = 1
Declare @PerformedBy VARCHAR(100) = 'Hamza'

WHILE @retryCount < @maxRetries
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION;


        UPDATE enrollments SET grade = 20 WHERE enrollment_id = 1;
        -- COMMIT; (Don't commit yet)
        -- Wait a bit, then try to lock tableB
        WAITFOR DELAY '00:00:05';
        UPDATE students SET first_name = 'hamza2'  WHERE student_id = 1;


        COMMIT TRANSACTION;
        BREAK; -- Success, exit loop
    END TRY
    BEGIN CATCH
        IF ERROR_NUMBER() = 1205 -- Deadlock
        BEGIN
            ROLLBACK TRANSACTION;
            SET @retryCount = @retryCount + 1;
            IF @retryCount = @maxRetries
                PRINT 'Maximum retries reached. Enrollment failed.';
            ELSE
                PRINT 'Deadlock occurred. Retrying...';
        END
        ELSE
        BEGIN
            ROLLBACK TRANSACTION;
            PRINT 'Error: ' + ERROR_MESSAGE();
            BREAK;
        END
    END CATCH
END
```

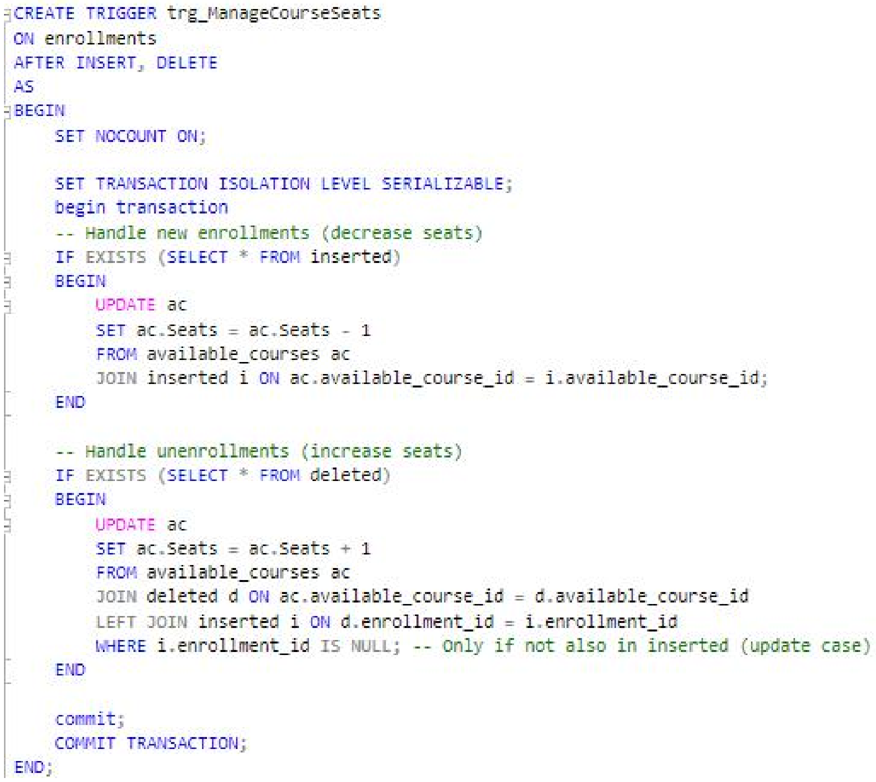
5. Integrity Constraints & Business Rules

- Trigger trg_CheckMaxEnrollments prevents more than 7 course registrations.



```sql
CREATE TRIGGER trg_CheckMaxEnrollments
ON enrollments
AFTER INSERT, UPDATE
AS
BEGIN
    SET NOCOUNT ON;

    IF EXISTS (
        SELECT student_id
        FROM (
            SELECT student_id, COUNT(*) AS course_count
            FROM enrollments
            WHERE StatusID = 8 -- Active, Passed, Registered statuses
              AND student_id IN (SELECT student_id FROM inserted)
            GROUP BY student_id
            HAVING COUNT(*) > 7
        ) AS over_enrolled
    )
    BEGIN
        ROLLBACK TRANSACTION;
        RAISERROR('A student cannot enroll in more than 7 courses', 16, 1);
    END
END;
GO
```

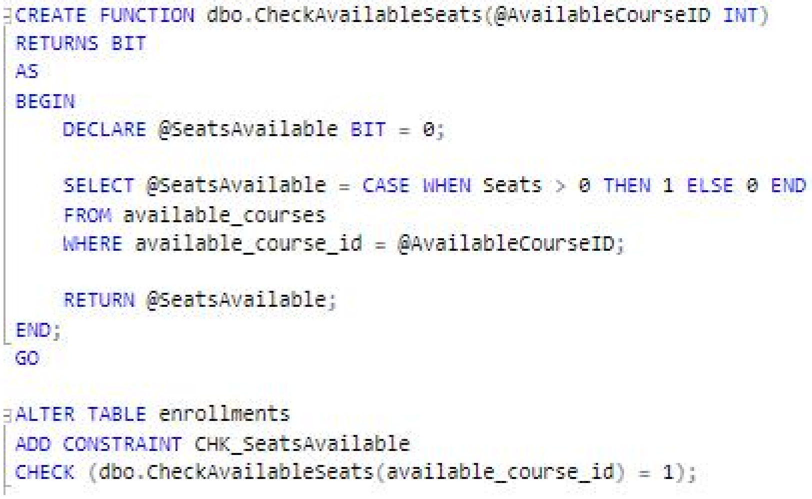
- Trigger trg_ManageCourseSeats adjusts seat count automatically on insert/delete.



```sql
CREATE TRIGGER trg_ManageCourseSeats
ON enrollments
AFTER INSERT, DELETE
AS
BEGIN
    SET NOCOUNT ON;

    SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
    begin transaction
    -- Handle new enrollments (decrease seats)
    IF EXISTS (SELECT * FROM inserted)
    BEGIN
        UPDATE ac
        SET ac.Seats = ac.Seats - 1
        FROM available_courses ac
        JOIN inserted i ON ac.available_course_id = i.available_course_id;
    END

    -- Handle unenrollments (increase seats)
    IF EXISTS (SELECT * FROM deleted)
    BEGIN
        UPDATE ac
        SET ac.Seats = ac.Seats + 1
        FROM available_courses ac
        JOIN deleted d ON ac.available_course_id = d.available_course_id
        LEFT JOIN inserted i ON d.enrollment_id = i.enrollment_id
        WHERE i.enrollment_id IS NULL; -- Only if not also in inserted (update case)
    END

    commit;
    COMMIT TRANSACTION;
END;
```

- Scalar function CheckAvailableSeats enforces course seat availability.

- Constraint CHK_SeatsAvailable uses the function to validate seats.



```sql
CREATE FUNCTION dbo.CheckAvailableSeats(@AvailableCourseID INT)
RETURNS BIT
AS
BEGIN
    DECLARE @SeatsAvailable BIT = 0;

    SELECT @SeatsAvailable = CASE WHEN Seats > 0 THEN 1 ELSE 0 END
    FROM available_courses
    WHERE available_course_id = @AvailableCourseID;

    RETURN @SeatsAvailable;
END;
GO

ALTER TABLE enrollments
ADD CONSTRAINT CHK_SeatsAvailable
CHECK (dbo.CheckAvailableSeats(available_course_id) = 1);
```

Kareem mohamed samy Aboshady

22010378

Member 4:

Technical Report: Indexing, Security, Roles, and Performance Optimization in the *AlexandriaUniversity* Database

1. Safe Index Creation

Tables and Columns Indexed:

- **students table:**
  - email → IX_students_email
  - DepartmentID → IX_students_department
  - last_name, first_name → IX_students_name
- **enrollments table:**
  - student_id → IX_enrollments_student
  - available_course_id → IX_enrollments_course
  - StatusID → IX_enrollments_status
- **available_courses table:**
  - course_id → IX_available_courses_course
  - instructor_id → IX_available_courses_instructor
  - semester, year → IX_available_courses_semester
- **courses table:**
  - course_code → IX_courses_code
  - course_name → IX_courses_name

Condition: Each index is only created if it doesn't already exist.

2. Safe Sequence Creation

Sequence: StudentIDSeq

- **Start Value:** 1000
- **Increment:** 1
- **Purpose:** Auto-generate student_id in sample_students table.

Table Created (if not exists):

sql
CopyEdit

```
sample_students (
   student_id INT PRIMARY KEY DEFAULT NEXT VALUE FOR StudentIDSeq,
   first_name VARCHAR(50),
   last_name VARCHAR(50)
)
```

---

3. Row-Level Security (RLS)

Schema: Security
Function: fn_securitypredicate(@student_id)

- Allows access **only** if:
  - SESSION_CONTEXT('student_id') matches the row student_id
  - OR the user is one of: 'admin1', 'registrar_jones'

Policy: Security.studentFilter

- Applied on: dbo.students
- **Enforcement:** Enabled

---

4. Dynamic Data Masking (DDM)

Masked Columns in students:

- email → MASKED WITH (FUNCTION = 'email()')
- date_of_birth → MASKED WITH (FUNCTION = 'default()')

Purpose: Protect sensitive data from unauthorized SELECTs.

---

5. Performance Testing

Queries Tested:

- Basic select on enrollments by student_id
- Same query **with** index hint (WITH(INDEX(...)))
- Join between students and enrollments

Timing Mechanism:

sql
CopyEdit

```
SET @StartTime = GETDATE();
-- Query
SET @EndTime = GETDATE();
PRINT 'Elapsed: ' + CAST(DATEDIFF(MILLISECOND, @StartTime, @EndTime) AS VARCHAR) + ' ms';
```

Goal: Demonstrate index performance impact.

---

6. Roles & Permissions (Safe)

Roles Created (if not exists):

Role	Purpose
StudentRole	Basic student access
InstructorRole	Access for instructors
AdminRole	Full DB control
RegistrarRole	Data entry roles

Users and Logins:

Login	User	Assigned Role
student1	student1	StudentRole
prof_smith	prof_smith	InstructorRole
registrar_jones	registrar_jones	RegistrarRole
admin1	admin1	AdminRole

Sample Grants:

- RegistrarRole can SELECT, INSERT, UPDATE on students and enrollments
- AdminRole gets full database control

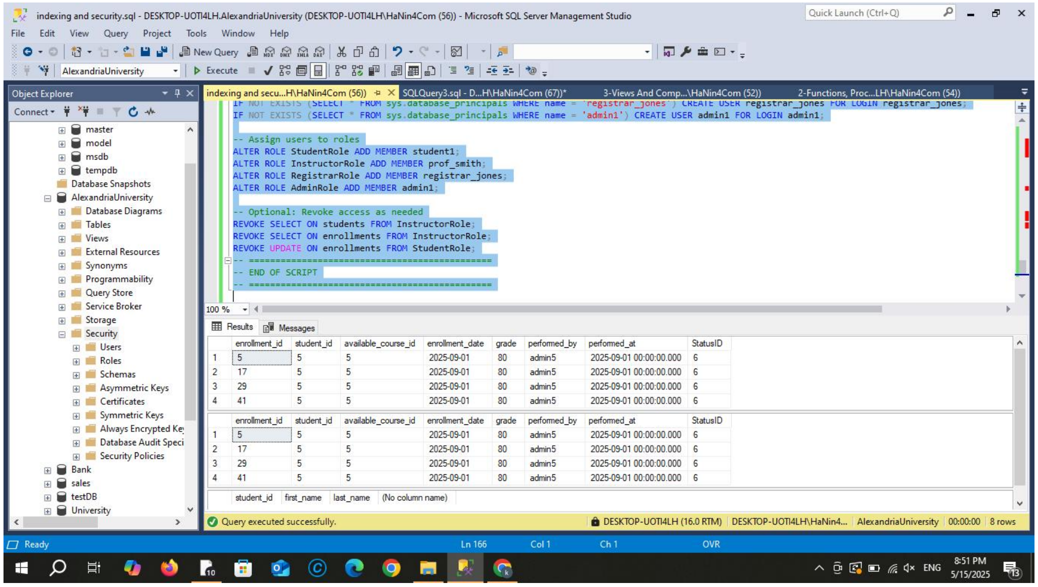
Revoked Access:

- InstructorRole → No direct access to students, enrollments

- StudentRole → Cannot UPDATE enrollments

Note: This script includes **safe checks** before creating or modifying objects. Use in production with confidence, but always test in a development environment first.

End of Report
Prepared for: Alexandria University Database Project
Date: 2025-05-15



```sql
IF NOT EXISTS (SELECT * FROM sys.database_principals WHERE name = 'registrar_jones') CREATE USER registrar_jones FOR LOGIN registrar_jones;
IF NOT EXISTS (SELECT * FROM sys.database_principals WHERE name = 'admin1') CREATE USER admin1 FOR LOGIN admin1;

-- Assign users to roles
ALTER ROLE StudentRole ADD MEMBER student1;
ALTER ROLE InstructorRole ADD MEMBER prof_smith;
ALTER ROLE RegistrarRole ADD MEMBER registrar_jones;
ALTER ROLE AdminRole ADD MEMBER admin1;

-- Optional: Revoke access as needed
REVOKE SELECT ON students FROM InstructorRole;
REVOKE SELECT ON enrollments FROM InstructorRole;
REVOKE UPDATE ON enrollments FROM StudentRole;
-- ============================================
-- END OF SCRIPT
-- ============================================
```

	enrollment_id	student_id	available_course_id	enrollment_date	grade	performed_by	performed_at	StatusID
1	5	5	5	2025-09-01	80	admin5	2025-09-01 00:00:00.000	6
2	17	5	5	2025-09-01	80	admin5	2025-09-01 00:00:00.000	6
3	29	5	5	2025-09-01	80	admin5	2025-09-01 00:00:00.000	6
4	41	5	5	2025-09-01	80	admin5	2025-09-01 00:00:00.000	6

	enrollment_id	student_id	available_course_id	enrollment_date	grade	performed_by	performed_at	StatusID
1	5	5	5	2025-09-01	80	admin5	2025-09-01 00:00:00.000	6
2	17	5	5	2025-09-01	80	admin5	2025-09-01 00:00:00.000	6
3	29	5	5	2025-09-01	80	admin5	2025-09-01 00:00:00.000	6
4	41	5	5	2025-09-01	80	admin5	2025-09-01 00:00:00.000	6

	student_id	first_name	last_name	(No column name)

Query executed successfully. DESKTOP-UOTI4LH (16.0 RTM) | DESKTOP-UOTI4LH\HaNin4... | AlexandriaUniversity | 00:00:00 | 8 rows