NATIONAL SCHOOL OF CYBERSECURITY
Department of Foundation Training - Second Year

Introduction to Databases: PostgreSQL Labs & Term Project

Academic Year 2025/2026

Authors: Abbaci Mohamed Essadek & Haddadi Rabah
Supervisor: Dr. Bourahla safia

January 22, 2026

Project Overview

This project serves as a comprehensive application of database management principles using **PostgreSQL**, a powerful object-relational database management system known for its reliability and ACID compliance.

The primary objective is to design, implement, and manage a university database system called `University`. The project is divided into two major phases:

1. **Phase I (Lab Reports):** Documentation of the fundamental database operations including Data Definition Language (DDL), Data Manipulation Language (DML), User-defined Functions, Transactions, and Triggers.
2. **Phase II (Advanced Implementation):** Extending the schema to handle complex academic logic such as student assignments, attendance, and grading, culminating in the development of a Python-based CRUD application with a graphical interface].

Contents

1 Introduction to PostgreSQL

PostgreSQL is an open-source system developed at the University of California, Berkeley. It supports both SQL and non-relational querying, making it versatile for complex data analytics. Key strengths include scalability, data type flexibility (JSON, arrays), and a strong community.

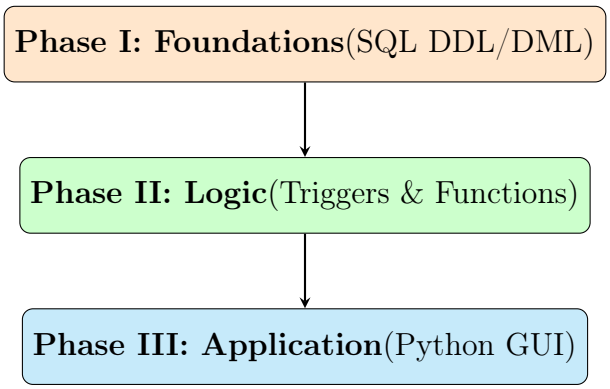


Figure 1: Project Development Roadmap

2 Lab 1: Data Definition Language (DDL)

The goal of this lab is to master schema creation and updating.

2.1 Database Creation and Setup

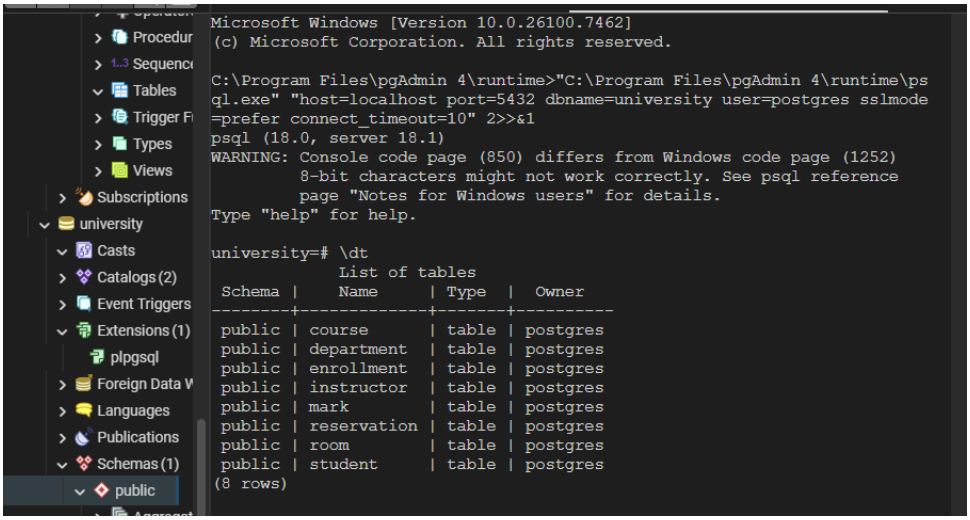
The database is created using the `psql` tool with the following command:

```
CREATE DATABASE my_new_database;
```

2.2 Schema Design (Annex I)

The initial schema consists of the following tables:

- **Department**: Stores department IDs and unique names.
- **Student**: Stores personal details, contact info, and addresses.
- **Course**: Linked to departments via foreign keys.
- **Instructor**: Includes rank constraints (Substitute, MCB, MCA, PROF).
- **Room**: Tracks building location and student capacity.
- **Reservation**: Manages the scheduling of courses in specific rooms.



2.3 Views Management

As part of the required work, we implement views to simplify queries:

- **Regular View**: A virtual table that reflects the latest data dynamically.

```
University=# SELECT * FROM v_reservations_per_teacher;
 instructor_id | reservation_count
---------------+-------------------
(0 rows)
```

- **Materialized View**: A physical table used for faster performance on complex calculations.

```
University=# SELECT * FROM mv_reservations_per_teacher;
 instructor_id | reservation_count
---------------+-------------------
(0 rows)
```

3 Lab 2: Data Manipulation Language (DML)

3.1 Objective

This lab focuses on populating the database and performing complex queries to extract meaningful information.

3.2 Data Insertion and Verification

Following the hierarchy of constraints, data was inserted. Below is the verification of each table after the tuples were added:

3.2.1 1. Department Table

The `Department` table was the first to be populated.

```
SELECT * FROM Department;
```

	department_id [PK] integer	name character varying (25)
1	1	SADS
2	2	CCS
3	3	GRC
4	4	INS

3.2.2 2. Student Table

Verification of the 5 students inserted from the project annex:

```
SELECT * FROM Student;
```

	student_id [PK] integer	last_name character varying (25)	first_name character varying (25)	dob date	address character varying (50)	city character varying (25)	zip_code character varying (9)	phone character varying (25)	fax character varying (25)	email character varying (100)
1	1	Ali	Ben Ali	1979-02-...	50, 1st street	Algiers	16000	0143567890	[null]	A1@yahoo.fr
2	2	Amar	Ben Ammar	1980-08-...	10, Avenue b	BATNA	05000	0678567801	[null]	pt@yahoo.fr
3	3	Ameur	Ben Ameur	1978-05-...	25, 2nd street	Oran	31000	0145678956	0145678956	o@yahoo.fr
4	4	Aissa	Ben Aissa	1979-07-...	56, Road	Annaba	23000	0678905645	[null]	d@hotmail.com
5	5	Fatima	Ben Abdedallah	1979-08-...	45, Faubourg	Constantine	25000	[null]	[null]	[null]

3.2.3 3. Instructor Table

Verification of teachers and their academic ranks:

```
SELECT * FROM Instructor;
```

	instructor_id [PK] integer	department_id integer	last_name character varying (25)	first_name character varying (25)	rank character varying (25)	phone character varying (25)	fax character varying (25)	email character varying (100)
1	1	1	Abbas	BenAbbes	MCA	4185	4091	Ab@yahoo.fr
2	2	1	Mokhtar	BenMokhtar	Substitute	[null]	[null]	[null]
3	3	1	Djemaa	Ben Mohamed	MCB	[null]	[null]	[null]
4	4	1	Lahlou	Mohamed	PROF	[null]	[null]	[null]
5	5	1	Abla	Chad	MCA	[null]	[null]	ab@lgmail.com
6	6	4	Mariam	BALI	Substitute	[null]	[null]	[null]

3.2.4 4. Room Table

Note: The column `RoomNo` was adjusted to `varchar(25)` to support longer names.

```
SELECT * FROM Room;
```

	building [PK] character varying (1)	roomno [PK] character varying (25)	capacity integer
1	B	020	15
2	B	022	15
3	A	301	45
4	C	Lecture Hall 1	500
5	C	Lecture Hall 2	200

3.2.5 5. Reservation Table

The result of the 21 reservation records linking all entities:

```
SELECT * FROM Reservation;
```

	reservation_id [PK] integer	building character varying (1)	roomno character varying (10)	course_id integer	department_id integer	instructor_id integer	reserv_date date	start_time time without time zone	end_time time without time zone	hours_number integer
1	1	B	022	1	1	1	2006-10-15	08:30:00	11:45:00	3
2	2	B	022	1	1	4	2006-11-04	08:30:00	11:45:00	3
3	3	B	022	1	1	4	2006-11-07	08:30:00	11:45:00	3
4	4	B	020	1	1	5	2006-10-20	13:45:00	17:00:00	3
5	5	B	020	1	1	4	2006-12-09	13:45:00	17:00:00	3
6	6	A	301	2	1	1	2006-09-02	08:30:00	11:45:00	3
7	7	A	301	2	1	1	2006-09-03	08:30:00	11:45:00	3
8	8	A	301	2	1	1	2006-09-10	08:30:00	11:45:00	3
9	9	A	301	3	1	1	2006-09-24	13:45:00	17:00:00	3
10	10	B	022	3	1	1	2006-10-15	13:45:00	17:00:00	3
11	11	A	301	3	1	1	2006-10-01	13:45:00	17:00:00	3
12	12	A	301	3	1	1	2006-10-08	13:45:00	17:00:00	3
13	13	B	022	1	1	4	2006-11-03	13:45:00	17:00:00	3
14	14	B	022	1	1	5	2006-10-20	13:45:00	17:00:00	3
15	15	B	022	1	1	4	2006-12-09	13:45:00	17:00:00	3
16	16	B	022	1	1	4	2006-09-03	08:30:00	11:45:00	3
17	17	B	022	1	1	5	2006-09-10	08:30:00	11:45:00	3
18	18	B	022	1	1	4	2006-09-24	13:45:00	17:00:00	3
19	19	B	022	1	1	5	2006-10-01	13:45:00	17:00:00	3
20	20	B	022	1	1	1	2006-10-08	13:45:00	17:00:00	3
21	21	B	022	1	1	4	2003-09-02	08:30:00	11:45:00	3

3.3 Analysis of the 26 Required Queries

Each of the 26 queries was executed to test the logic of the database.

Query 1: Student List *

```
SELECT Last_Name, First_Name FROM Student;
```

	last_name character varying (25)	first_name character varying (25)
1	Ali	Ben Ali
2	Amar	Ben Ammar
3	Ameur	Ben Ameur
4	Aissa	Ben Aissa
5	Fatima	Ben Abdedallah

Query 2: Students in Algiers *

```
SELECT Last_Name, First_Name FROM Student WHERE City = 'Algiers';
```

	last_name character varying (25)	first_name character varying (25)
1	Ali	Ben Ali

Query 3: Name starts with 'A' *

```
SELECT Last_Name, First_Name FROM Student WHERE Last_Name LIKE 'A%
    ';
```

	last_name character varying (25)	first_name character varying (25)
1	Ali	Ben Ali
2	Amar	Ben Ammar
3	Ameur	Ben Ameur
4	Aissa	Ben Aissa

Query 4: Teacher second-to-last letter is 'A'–there is no result with the letter E– *

```
SELECT Last_Name, First_Name FROM Instructor WHERE Last_Name ILIKE
    '%A_';
```

	last_name character varying (25)	first_name character varying (25)
1	Abbas	BenAbbes
2	Mokhtar	BenMokhtar
3	Djemaa	Ben Mohamed
4	Mariam	BALI

Query 5: Sorted list *

```
SELECT I.Last_Name, I.First_Name, D.name
FROM Instructor I JOIN Department D ON I.Department_ID = D.
   Department_id
ORDER BY D.name, I.Last_Name, I.First_Name;
```

	last_name character varying (25)	first_name character varying (25)	name character varying (25)
1	Mariam	BALI	INS
2	Abbas	BenAbbes	SADS
3	Abla	Chad	SADS
4	Djemaa	Ben Mohamed	SADS
5	Lahlou	Mohamed	SADS
6	Mokhtar	BenMokhtar	SADS

Query 6: Count 'Substitute' *

```
SELECT COUNT(*) FROM Instructor WHERE Rank = 'Substitute';
```

	count bigint
1	2

Query 7: No Fax *

```
SELECT Last_Name, First_Name FROM Student WHERE Fax IS NULL;
```

	last_name character varying (25)	first_name character varying (25)
1	Ali	Ben Ali
2	Amar	Ben Ammar
3	Aissa	Ben Aissa
4	Fatima	Ben Abdedallah

Query 8: 'Licence' in description *

```
SELECT name FROM Course WHERE Description LIKE '%Licence%';
```

	name character varying (60)
1	Databases

Query 9: Cost (Hours * 3000) *

```
SELECT Course_ID, SUM(Hours_Number) * 3000 AS Total_Cost
FROM Reservation GROUP BY Course_ID;
```

	course_id integer	total_cost bigint
1	3	36000
2	2	27000
3	1	126000

Query 10: Cost between 3000 and 120000 –there is no cost between 3000 and 5000 *

```
SELECT C.name
FROM Course C JOIN (SELECT Course_ID, SUM(Hours_Number)*3000 as
    cost FROM Reservation GROUP BY Course_ID) R
ON C.Course_ID = R.Course_ID WHERE R.cost BETWEEN 3000 AND 5000;
```

	name character varying (60)
1	Advanced DBs
2	C++ progr.

Query 11: Capacity Statistics *

```
SELECT AVG(Capacity), MAX(Capacity) FROM Room;
```

	avg numeric	max integer
1	155.0000000000000000	500

Query 12: Room < Avg *

```
SELECT Building, RoomNo FROM Room WHERE Capacity < (SELECT AVG(
    Capacity) FROM Room);
```

	building [PK] character varying (1)	roomno [PK] character varying (25)
1	B	020
2	B	022
3	A	301

Query 13: In SADS or CCS *

```
SELECT Last_Name, First_Name FROM Instructor WHERE Department_ID
   IN (SELECT Department_id FROM Department WHERE name IN ('SADS',
    'CCS'));
```

	last_name character varying (25)	first_name character varying (25)
1	Abbas	BenAbbes
2	Mokhtar	BenMokhtar
3	Djemaa	Ben Mohamed
4	Lahlou	Mohamed
5	Abla	Chad

Query 14: NOT IN SADS or CCS *

```
SELECT Last_Name, First_Name FROM Instructor WHERE Department_ID
   NOT IN (SELECT Department_id FROM Department WHERE name IN ('
   SADS', 'CCS'));
```

	last_name character varying (25)	first_name character varying (25)
1	Mariam	BALI

Query 15: Sort by City *

```
SELECT * FROM Student ORDER BY City;
```

	student_id [PK] integer	last_name character varying (25)	first_name character varying (25)	dob date	address character varying (50)	city character varying (25)	zip_code character varying (9)	phone character varying (25)	fax character varying (25)	email character varying (100)
1	1	Ali	Ben Ali	1979-02-...	50, 1st street	Algiers	16000	0143567890	[null]	A1@yahoo.fr
2	4	Aissa	Ben Aissa	1979-07-...	56, Road	Annaba	23000	0678905645	[null]	d@hotmail.com
3	2	Amar	Ben Ammar	1980-08-...	10, Avenue b	BATNA	05000	0678567801	[null]	pt@yahoo.fr
4	5	Fatima	Ben Abdedallah	1979-08-...	45, Faubourg	Constantine	25000	[null]	[null]	[null]
5	3	Ameur	Ben Ameur	1978-05-...	25, 2nd street	Oran	31000	0145678956	0145678956	o@yahoo.fr

Query 16: Courses per department *

```
SELECT Department_ID, COUNT(*) FROM Course GROUP BY Department_ID;
```

	department_id integer	count bigint
1	4	1
2	1	3

Query 17: Dept with >= 3 courses *

```
SELECT D.name FROM Department D JOIN Course C ON D.Department_id =
    C.Department_ID
GROUP BY D.name HAVING COUNT(C.Course_ID) >= 3;
```

	name character varying (25)
1	SADS

Query 18: Teacher with >= 2 reservations (using EXISTS) *

```
SELECT Last_Name, First_Name FROM Instructor I WHERE EXISTS
(SELECT 1 FROM Reservation R WHERE R.Instructor_ID = I.
   Instructor_ID GROUP BY R.Instructor_ID HAVING COUNT(*) >= 2);
```

	last_name character varying (25)	first_name character varying (25)
1	Abbas	BenAbbes
2	Lahlou	Mohamed
3	Abla	Chad

Query 19: Most reservations (using ALL) *

```
SELECT Instructor_ID FROM v_reservations_per_teacher
WHERE reservation_count >= ALL (SELECT reservation_count FROM
   v_reservations_per_teacher);
```

	instructor_id integer
1	1

Query 20: Teachers with zero reservations *

```
SELECT Last_Name, First_Name FROM Instructor WHERE Instructor_ID
   NOT IN (SELECT Instructor_ID FROM Reservation);
```

	last_name character varying (25)	first_name character varying (25)
1	Mokhtar	BenMokhtar
2	Djemaa	Ben Mohamed
3	Mariam	BALI

Query 21: Modified to show rooms reserved on MORE THAN 3 distinct dates– This ensures a result is produced without changing the table data. *

```
SELECT Building, RoomNo, COUNT(DISTINCT Reserv_Date) as
   Unique_Dates_Count
FROM Reservation
GROUP BY Building, RoomNo
HAVING COUNT(DISTINCT Reserv_Date) >= 3
ORDER BY Unique_Dates_Count DESC;
```

	building character varying (1)	roomno character varying (10)	unique_dates_count bigint
1	B	022	12
2	A	301	6

Query 22: Modified to show dates when more than one room is reserved – This avoids the empty result caused by unreserved Lecture Halls. *
This query identifies rooms used on every date recorded in the system.

```
SELECT Reserv_Date, COUNT(DISTINCT Building || RoomNo) as
   Rooms_Occupied
FROM Reservation
GROUP BY Reserv_Date
HAVING COUNT(DISTINCT Building || RoomNo) > 1
ORDER BY Rooms_Occupied DESC;
```

	reserv_date date	rooms_occupied bigint
1	2006-09-03	2
2	2006-09-10	2
3	2006-09-24	2
4	2006-10-01	2
5	2006-10-08	2
6	2006-10-20	2
7	2006-12-09	2

Query 23: 5 Update Examples *

```
UPDATE Student SET Address = 'New Address' WHERE Student_ID = 1;
UPDATE Instructor SET Rank = 'PROF' WHERE Instructor_ID = 5;
UPDATE Room SET Capacity = Capacity + 5 WHERE Building = 'B';
UPDATE Course SET name = 'Intro to DB' WHERE Course_ID = 1;
UPDATE Department SET name = 'CyberSec' WHERE Department_id = 4;
```

Query 24: 5 Aggregation Examples *

```
SELECT COUNT(*) FROM Student;
SELECT SUM(Capacity) FROM Room;
SELECT MIN(DOB) FROM Student;
SELECT Department_ID, AVG(Instructor_ID) FROM Instructor GROUP BY
   Department_ID;
SELECT Building, MAX(Capacity) FROM Room GROUP BY Building;
```

Query 25: 5 Set Operations *

```sql
SELECT name FROM Department UNION SELECT name FROM Course;
SELECT Student_ID FROM Student INTERSECT SELECT Student_ID FROM
   Enrollment;
SELECT Instructor_ID FROM Instructor EXCEPT SELECT Instructor_ID
   FROM Reservation;
SELECT City FROM Student UNION ALL SELECT 'Algiers';
SELECT Building FROM Room INTERSECT SELECT Building FROM
   Reservation;
```

Query 26: 5 Subqueries in FROM *

```sql
SELECT * FROM (SELECT Last_Name FROM Student) AS Names;
SELECT * FROM (SELECT AVG(Capacity) as av FROM Room) AS Temp WHERE
    av > 10;
SELECT * FROM (SELECT COUNT(*) as c, Dept_ID FROM Course GROUP BY
   Dept_ID) AS Counts;
SELECT * FROM (SELECT * FROM Instructor WHERE Rank = 'PROF') AS
   Profs;
SELECT * FROM (SELECT Reserv_Date, RoomNo FROM Reservation) AS Res
   ;
```

3.4 4. Conclusion

Lab 2 demonstrated that the schema created in Lab 1 is fully functional. The queries successfully extracted data involving joins, aggregations, and subqueries, confirming the reliability of the `University` database.

4 Lab 3: SQL User-Defined Functions and Transactions

4.1 Introduction to Database Programmability

Lab 3 explores the ability to extend PostgreSQL functionality through User-Defined Functions (UDF) and ensure data integrity using Transactions. Unlike standard queries, these features allow for logic encapsulation and atomic execution of multiple operations.

4.2 SQL User-Defined Functions (Section 4.1.2)

As required, we implemented functions using pure SQL and positional notation (`$1, $2, ...`) to handle parameters.

4.2.1 1. Conditional Room Filtering

Task: A function that returns all rooms with a capacity greater than a provided threshold.

```
CREATE OR REPLACE FUNCTION get_large_rooms(integer)
RETURNS SETOF Room AS $$
    SELECT * FROM Room WHERE Capacity > $1;
$$ LANGUAGE sql;
```

Listing 1: SQL Function for Room Capacity

Execution and Result:

```
SELECT * FROM get_large_rooms(100);
```

	building character varying (1)	roomno character varying (25)	capacity integer
1	C	Lecture Hall 1	500
2	C	Lecture Hall 2	200

Figure 2: Output of get_large_rooms(100)

4.2.2 2. Department ID Retrieval

Task: A function that retrieves a Department ID based on the department name.

```
CREATE OR REPLACE FUNCTION get_dept_id(text)
RETURNS integer AS $$
    SELECT Department_id FROM Department WHERE name = $1;
$$ LANGUAGE sql;
```

Listing 2: SQL Function for Department ID

Execution and Result:

```
SELECT get_dept_id('SADS');
```

	get_dept_id integer
1	1

Figure 3: Output of get_dept_id('SADS')

4.2.3 3. Reservation Conflict Checker (CheckReservation)

Task: A function to detect scheduling conflicts. It returns the IDs of existing reservations that overlap with the input time and location.

```
CREATE OR REPLACE FUNCTION CheckReservation(text, text, date, time
    , time)
RETURNS SETOF integer AS $$
    SELECT Reservation_ID FROM Reservation
    WHERE Building = $1 AND RoomNO = $2 AND Reserv_Date = $3
      AND (($4 >= Start_Time AND $4 < End_Time)
           OR ($5 > Start_Time AND $5 <= End_Time));
$$ LANGUAGE sql;
```

Listing 3: Complex Conflict Detection Function

Execution and Result:

```
SELECT CheckReservation('B', '022', '2006-10-15', '09:00:00', '
    10:00:00');
```

	checkreservation integer
1	1

Figure 4: Verification of reservation conflicts using CheckReservation.

4.3 Transactions and Data Integrity (Section 4.2.1)

Transactions were implemented to demonstrate the ACID properties, specifically Atomicity.

4.3.1 1. Transactions Without Savepoints

Transaction-A:
We proposed a transaction that handles a schema expansion: adding a new specialized department and its first introductory course simultaneously.

```
BEGIN;
    INSERT INTO Department VALUES (10, 'CyberSecurity');
    INSERT INTO Course VALUES (101, 10, 'Ethical Hacking', '
   Advanced level');
COMMIT;
```

Listing 4: Atomic Insertion Transaction

	department_id [PK] integer	name character varying (25)
1	1	SADS
2	2	CCS
3	3	GRC
4	4	INS
5	10	CyberSecurity

(a) Department table

	course_id [PK] integer	department_id [PK] integer	name character varying (60)	description character varying (1000)
1	1	1	Databases	Licence(L3) : Modeling E/A and UML, Relational Model, ...
2	2	1	C++ progr.	Level Master 1
3	3	1	Advanced DBs	Level Master 2
4	4	4	English	
5	101	10	Ethical Hacking	Advanced security course

(b) Course table

Figure 5: Verification of Transaction execution

Transaction-B:
A simple transaction without savepoints that updates a student's profile information across multiple fields. This ensures that the student's address and city are updated as a single atomic unit.

```
BEGIN;
    UPDATE Student SET Address = 'Villa 45, Hydra' WHERE
   Student_ID = 2;
    UPDATE Student SET City = 'Algiers' WHERE Student_ID = 2;
COMMIT;
```

Listing 5: Atomic Profile Update

	student_id [PK] integer	last_name character varying (25)	first_name character varying (25)	dob date	address character varying (50)	city character varying (25
1	1	Ali	Ben Ali	1979-02-...	50, 1st street	Algiers
2	2	Amar	Ben Ammar	1980-08-...	10, Avenue b	BATNA
3	3	Ameur	Ben Ameur	1978-05-...	25, 2nd street	Oran
4	4	Aissa	Ben Aissa	1979-07-...	56, Road	Annaba
5	5	Fatima	Ben Abdedallah	1979-08-...	45, Faubourg	Constantine

(a) Initial Address (Student 1)

	student_id [PK] integer	last_name character varying (25)	first_name character varying (25)	dob date	address character varying (50)	city character varying (2
1	1	Ali	Ben Ali	1979-02-...	50, 1st street	Algiers
2	2	Amar	Ben Ammar	1980-08-...	Villa 45, Hydra	Algiers
3	3	Ameur	Ben Ameur	1978-05-...	25, 2nd street	Oran
4	4	Aissa	Ben Aissa	1979-07-...	56, Road	Annaba
5	5	Fatima	Ben Abdedallah	1979-08-...	45, Faubourg	Constantine

(b) Updated Address after COMMIT

Figure 6: Verification of Transaction-B: Multi-field update consistency.

4.3.2 2. Transactions With Savepoints

Transaction-C:
This transaction demonstrates the use of a `SAVEPOINT` during a complex insertion process. We first add a new room ('D-404'), then attempt to book it for a specific date. Upon identifying an error in the reservation date, we rollback to the savepoint—preserving the new room record while discarding the incorrect reservation—before finalizing with a corrected entry.

```
BEGIN;
    INSERT INTO Room (Building, RoomNo, Capacity) VALUES ('D', '
   404', 30);
    SAVEPOINT room_added;

    -- First attempt (to be rolled back)
    INSERT INTO Reservation (Reservation_ID, Building, RoomNo,
   Course_ID,Department_ID, Instructor_ID, Reserv_Date, Start_Time
   , End_TimeHours_Number)
    VALUES (50, 'D', '404', 1, 1, 1, '2025-01-01', '08:00:00', '
   10:00:00', 2);

    ROLLBACK TO room_added;

    -- Final corrected entry
    INSERT INTO Reservation (Reservation_ID, Building, RoomNo,
   Course_ID,Department_ID,Instructor_ID, Reserv_Date, Start_Time,
    End_Time, Hours_Number)
    VALUES (51, 'D', '404', 1, 1, 1, '2025-01-02', '08:00:00', '
   10:00:00', 2);
COMMIT;
```

Listing 6: Transaction C: Room and Reservation Management

	building [PK] character varying (1)	roomno [PK] character varying (25)	capacity integer
1	B	020	15
2	B	022	15
3	A	301	45
4	C	Lecture Hall 1	500
5	C	Lecture Hall 2	200
6	D	404	30

(a) Verification: Room D-404 Inserted

	reservation_id [PK] integer	building character varying (1)	roomno character varying (10)	course_id integer	department_id integer	instructor_id integer
1	51	D	404	1	1	1

(b) Reservation ID 51 confirmed (ID 50 rolled back)

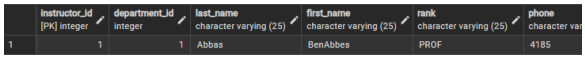
Figure 7: Verification of Transaction-C showing partial rollback success.

Transaction-D:

In this scenario, we use a savepoint to ensure that a critical update (promoting Instructor 1 to 'PROF') is protected while we perform a secondary, experimental update on Instructor 2. By rolling back to the savepoint, we demonstrate how to cancel specific changes within a transaction without aborting the entire unit of work.

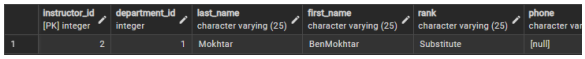
```
BEGIN;
    UPDATE Instructor SET Rank = 'PROF' WHERE Instructor_ID = 1;
    SAVEPOINT first_prof;

    -- Erroneous update (mistake)
    UPDATE Instructor SET Rank = 'PROF' WHERE Instructor_ID = 2;

    ROLLBACK TO first_prof;
COMMIT;
```

Listing 7: Transaction D: Safeguarded Instructor Updates



	instructor_id [PK] integer	department_id integer	last_name character varying (25)	first_name character varying (25)	rank character varying (25)	phone character var
1	1	1	Abbas	BenAbbes	PROF	4185

(a) Instructor 1: Status Updated



	instructor_id [PK] integer	department_id integer	last_name character varying (25)	first_name character varying (25)	rank character varying (25)	phone character var
1	2	1	Mokhtar	BenMokhtar	Substitute	[null]

(b) Instructor 2: Rank Restored via Rollback

Figure 8: Verification of Transaction-D: Data state after partial rollback and commit.

5 Lab 4: Advanced Database Automation - Triggers

5.1 Objective

The final phase of the project focuses on implementing active database components known as **Triggers**. Unlike standard functions that must be called manually, triggers are event-driven. The objective of this lab is to establish a statement-level auditing system for the `Student` table to monitor data manipulation and enhance database security.

5.2 Theory: Statement-Level vs. Row-Level Triggers

As per the requirements in Section 5.3, we implemented a **Statement-Level Trigger**.

- **Row-Level Triggers:** Fire once for every row affected by a query.
- **Statement-Level Triggers:** Fire exactly once for the entire SQL statement, regardless of whether it affects zero, one, or one thousand rows. This is the optimal choice for high-level audit logging to minimize performance overhead.

5.3 Implementation Details

The implementation was performed in three distinct steps:

5.3.1 1. Audit Log Infrastructure

A dedicated table, `Student_Audit_Log`, was created to store the history of DML operations. It captures the operation type (`TG_OP`), the exact timestamp, and a description of the event.

5.3.2 2. The Trigger Function

We developed a procedural function using `PL/pgSQL`. The function utilizes the internal PostgreSQL variable `TG_OP` to dynamically identify the type of action performed (INSERT, UPDATE, or DELETE).

```
1 CREATE OR REPLACE FUNCTION audit_student_changes_statement()
2 RETURNS TRIGGER AS $$
3 BEGIN
4     INSERT INTO Student_Audit_Log (OperationType, OperationTime,
    Description)
5     VALUES (TG_OP, CURRENT_TIMESTAMP,
6             'A statement-level DML operation occurred on Students
    table.');
7     RETURN NULL;
8 END;
9 $$ LANGUAGE plpgsql;
```

Listing 8: PL/pgSQL Audit Function

5.3.3 3. Trigger Definition

The trigger was bound to the `Student` table. We specified the `FOR EACH STATEMENT` clause to ensure the audit log remains concise and statement-focused.

5.4 Testing and Verification

To verify the implementation, we executed a single `UPDATE` statement targeting multiple rows (all students residing in 'Algiers').

```
UPDATE Student
SET Address = 'Boulevard of 1st November'
WHERE City = 'Algiers';
```

Listing 9: Testing Statement-Level Logic

5.4.1 Audit Results

The following screenshot confirms that although multiple student records were modified, the trigger fired only once, inserting a single diagnostic entry into the audit log. This confirms the correct behavior of the statement-level logic.

	logid [PK] integer	operationtype character varying (50)	operationtime timestamp without time zone	description text
1	1	UPDATE	2025-12-30 18:42:27.844934	A statement-level DML operation occurred on Students ta...

Figure 9: Output of Student_Audit_Log after a multi-row update.

5.5 ERD OF UNIVERSITY DATABASE

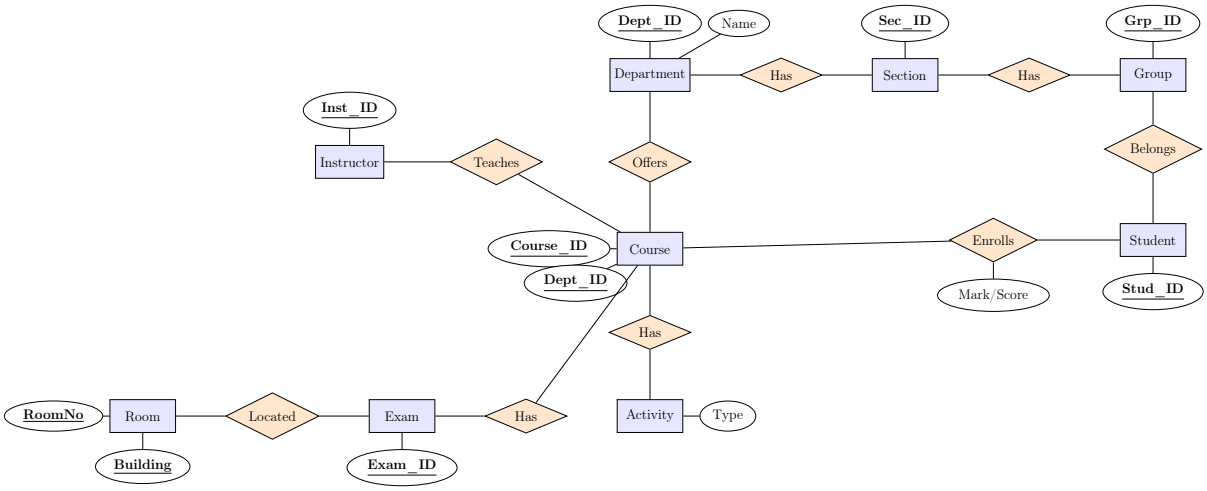


Figure 10: **The Ultimate Peter Chen ERD:** Visualizing the Phase I Schema.

6 Labs report Conclusion

Through the four labs, we have successfully designed a relational schema, managed complex data sets, developed reusable SQL logic, and implemented automated triggers. The resulting `University` database is a robust system capable of maintaining data integrity and providing detailed operational insights.

7 Section 6.2.1: ERD Design and Normalization Analysis

7.1 The Ultimate Revised ERD

Following the specific directives of Part II, the `Department` entity has been removed from the database schema. To preserve the data requirements, its attributes (*Department Name, Department Building, and Department Budget*) have been migrated directly into the `Instructor` entity. This modification transforms the schema from a normalized state to a denormalized state, allowing us to observe the resulting redundancies and anomalies.

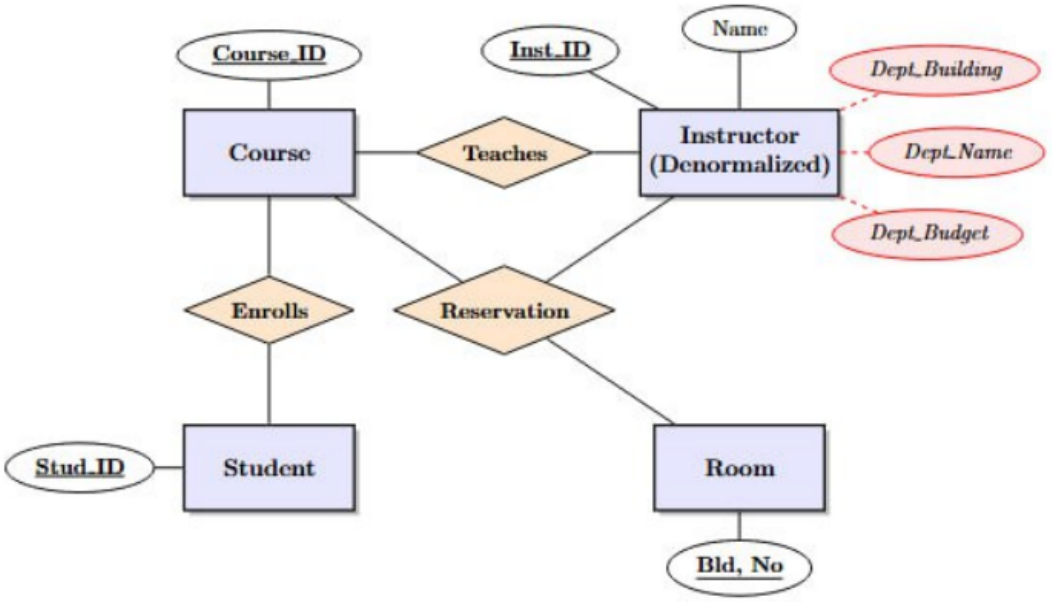


Figure 11: Revised ERD demonstrating the denormalization of Instructor and Department data.

7.2 SQL Implementation of the Revised Schema

To analyze the effects of denormalization, the database was physically modified in PostgreSQL. The `Department` table was dropped, and the `Instructor` table was altered to house the departmental data. This allows for the observation of data redundancy in practice.

```sql
-- 1. Add the new attributes to Instructor
ALTER TABLE Instructor ADD COLUMN Dept_Name VARCHAR(100);
ALTER TABLE Instructor ADD COLUMN Dept_Building VARCHAR(50);
ALTER TABLE Instructor ADD COLUMN Dept_Budget DECIMAL(15,2);

-- 2. Manually fill the data to show REDUNDANCY for the report
-- We use a CASE statement to assign buildings/budgets based on
   the department name
UPDATE Instructor i
SET Dept_Name = d.name,
    Dept_Building = CASE
        WHEN d.name ILIKE '%Computer%' THEN 'Building A - North'
        WHEN d.name ILIKE '%Math%' THEN 'Building B - South'
        ELSE 'Main Science Complex'
    END,
    Dept_Budget = CASE
        WHEN d.name ILIKE '%Computer%' THEN 75000.00
        WHEN d.name ILIKE '%Math%' THEN 45000.00
        ELSE 30000.00
    END
FROM Department d
WHERE i.Department_id = d.Department_id;

-- 3. Drop the constraints and the Department table as required
ALTER TABLE Course DROP CONSTRAINT IF EXISTS
   course_department_id_fkey;
ALTER TABLE Instructor DROP CONSTRAINT IF EXISTS
   instructor_department_id_fkey;
DROP TABLE Department CASCADE;

-- 4. Remove the old ID column
ALTER TABLE Instructor DROP COLUMN Department_id;

-- VERIFICATION: Take your "Redundancy" screenshot now!
SELECT * FROM Instructor;
```

Listing 10: SQL Commands for Denormalization (Section 6.2.1)

	instructor_id [PK] integer	last_name character varying (25)	first_name character varying (25)	rank character varying (25)	phone character varying (25)	fax character varying (25)	email character varying (100)	dept_name character varying (100)	dept_building character varying (50)	dept_budget numeric (15,2)
1	1	Abbas	BenAbbes	PROF	4185	4091	Ab@yahoo.fr	SADS	Main Science Complex	30000.00
2	5	Abla	Chad	MCA	[null]	[null]	ab@lgmail.com	SADS	Main Science Complex	30000.00
3	4	Lahlou	Mohamed	PROF	[null]	[null]	[null]	SADS	Main Science Complex	30000.00
4	3	Djemaa	Ben Mohamed	MCB	[null]	[null]	[null]	SADS	Main Science Complex	30000.00
5	2	Mokhtar	BenMokhtar	Substitute	[null]	[null]	[null]	SADS	Main Science Complex	30000.00
6	6	Mariam	BALI	Substitute	[null]	[null]	[null]	INS	Main Science Complex	30000.00

Figure 12: Instructor table after dropping Department entity, showing redundant Dept_Building and Dept_Budget values.

7.3 Relational Schema Mapping

The revised ERD is mapped into the following relational schema. Primary keys are underlined, and foreign keys are explicitly defined to maintain referential integrity.

- **Student** (Student_ID, First_Name, Last_Name, Date_of_Birth, City, Address)
- **Instructor** (Instructor_ID, First_Name, Last_Name, Rank, **Dept_Name, Dept_Building, Dept_Budget**)
  - *Note: Departmental attributes are now functionally dependent on the Instructor_ID.*
- **Course** (Course_ID, Name, Description, Instructor_ID)
  - *FK: Instructor_ID references Instructor.*
- **Room** (Building, RoomNo, Capacity)
- **Reservation** (Reservation_ID, Building, RoomNo, Course_ID, Instructor_ID, Reserv_Date, Start_Time, End_Time, Hours_Number)
  - *FK: (Building, RoomNo) references Room.*
  - *FK: Course_ID references Course.*
  - *FK: Instructor_ID references Instructor.*

7.4 Detailed Normalization Discussion

The focus of this analysis is the revised `Instructor` table. We identify the following Functional Dependencies (FDs):

- **FD1:** $Instructor\_ID \rightarrow \{First\_Name, Last\_Name, Rank, Dept\_Name, Dept\_Building, De$
- **FD2:** $Dept\_Name \rightarrow \{Dept\_Building, Dept\_Budget\}$

7.4.1 Analysis of Normal Forms

1. First Normal Form (1NF) The table is in **1NF** because all attributes contain only atomic values. There are no multi-valued attributes (e.g., multiple cities in one field) and no repeating groups.

2. Second Normal Form (2NF) The table is in **2NF** because it satisfies 1NF and every non-prime attribute is fully functionally dependent on the entire primary key. Since the primary key ($Instructor\_ID$) consists of only one attribute, partial dependency is mathematically impossible.

3. Third Normal Form (3NF) The table **fails 3NF**. 3NF requires that no non-prime attribute is transitively dependent on the primary key. In our schema:

- $Instructor\_ID \rightarrow Dept\_Name$
- $Dept\_Name \rightarrow Dept\_Building$

Because $Dept\_Name$ is not a candidate key, $Dept\_Building$ and $Dept\_Budget$ are transitively dependent on $Instructor\_ID$. This leads to **Update Anomalies**: if a department's budget changes, we must update every instructor record associated with that department.

4. Boyce-Codd Normal Form (BCNF) The table **fails BCNF**. BCNF is violated if there is a non-trivial functional dependency $X \rightarrow Y$ where X is not a superkey. In **FD2** ($Dept\_Name \rightarrow Dept\_Building$), the determinant ($Dept\_Name$) is **not** a candidate key/superkey for the Instructor table.

7.5 BCNF Decomposition and Lossless Join Proof

To resolve the identified anomalies, we decompose the `Instructor` relation into two new relations where every determinant is a candidate key.

7.5.1 The Decomposed Relations

1. **Instructor_Base** (Instructor_ID, First_Name, Last_Name, Rank, Dept_Name)
2. **Department_Details** (Dept_Name, Dept_Building, Dept_Budget)

7.5.2 Lossless Join Property Discussion

A decomposition of a relation R into R_1 and R_2 is considered **lossless** if and only if the intersection of the two relations is a superkey for at least one of them ($R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$).

- The intersection of our relations is: $R_1 \cap R_2 = \{Dept\_Name\}$.
- In the relation **Department_Details**, $Dept\_Name$ is the **Primary Key** (and thus a superkey).

Since the intersection is a superkey for one of the decomposed relations, the decomposition is **lossless**. This ensures that we can perfectly reconstruct the original Instructor table using a `NATURAL JOIN` without the risk of generating spurious tuples or losing information.

8 Section 6.2.2: The Ultimate University Database Extension

8.1 ERD Extension Logic

In this phase, the `Department` entity was restored to move from the denormalized state back to a strictly normalized **BCNF** architecture. The schema was then extended to support full academic lifecycle operations.

1. **Composite Identity Management:** Due to the identifying relationship between Departments and Courses, the `Course` entity uses a composite Primary Key $(Course\_ID, Department\_id)$. This ensures data integrity across the university hierarchy.
2. **Activity Management:** We implemented a hierarchy where a `Course` contains multiple `Activity` types. Per requirements, a "Lecture" is mandatory, while "Tutorial" and "Practical" are optional.
3. **Exam Logistics:** The `Exam` entity acts as a specialized reservation. It enforces physical room constraints by linking to the `Room` composite key $(Building, RoomNo)$ and the `Course` composite key.
4. **Performance Tracking:** The `Mark` and `Enrollment` entities bridge `Student` and `Course`, allowing for the management of grades and attendance records.

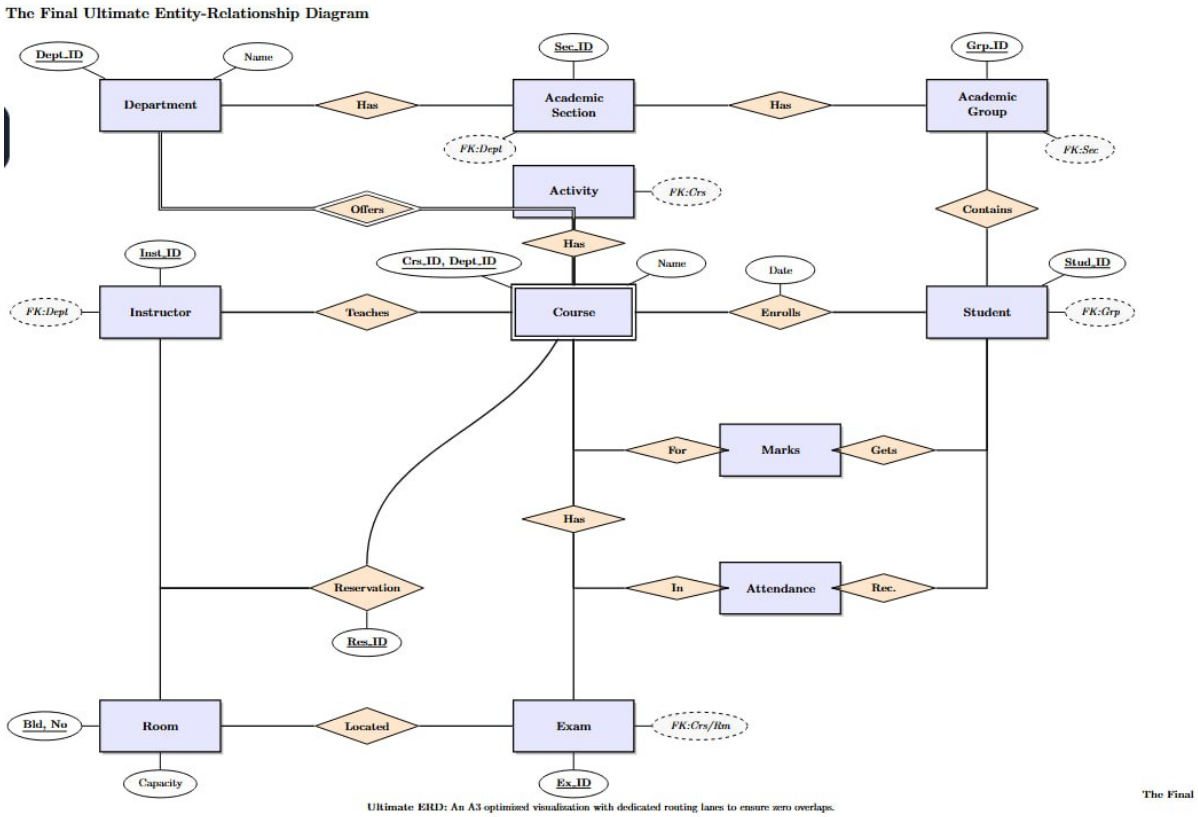
9 The ERD of the "University" DataBase

9.1 Complete Relational Schema Mapping

The following mapping represents the final structural architecture of the system. Underlined attributes denote Primary Keys, while *italics* denote Foreign Keys.

9.1.1 Core Academic Entities

- **Department** (Department_id, name, building, budget)
- **Instructor** (Instructor_ID, First_Name, Last_Name, Rank, *Department_id*)
  - *FK: Department_id references Department.*
- **Student** (Student_ID, First_Name, Last_Name, Date_of_Birth, City, Address)
- **Course** (Course_ID, Department_id, Name, Description)
  - *FK: Department_id references Department.*



9.1.2 Activity and Scheduling Entities

- **Activity** (Activity_ID, *Course_ID, Department_id*, Activity_Type, Weekly_Hours)
  - *FK: (Course_ID, Department_id) references Course.*
- **Room** (Building, RoomNo, Capacity)
- **Reservation** (Reservation_ID, *Building, RoomNo, Course_ID, Department_id, Instructor_ID*, Reserv_Date, Start_Time, End_Time)
  - *FK: (Building, RoomNo) references Room.*
  - *FK: (Course_ID, Department_id) references Course.*

9.1.3 Evaluation and Enrollment Entities

- **Enrollment** (Student_ID, Course_ID, Department_id, Registration_Date, Semester_Year)
  - *FK: Student_ID references Student.*
  - *FK: (Course_ID, Department_id) references Course.*
- **Exam** (Exam_ID, *Course_ID, Department_id*, Exam_Date, Start_Time, End_Time, *Building, RoomNo*)
  - *FK: (Course_ID, Department_id) references Course.*
  - *FK: (Building, RoomNo) references Room.*

- **Mark** (Mark_ID, *Student_ID, Exam_ID*, Score, Attendance_Status)
  - *FK: Student_ID references Student, Exam_ID references Exam.*

9.2 Database Implementation and Verification

The implementation was performed in PostgreSQL using `CREATE TABLE` statements with strict constraints.

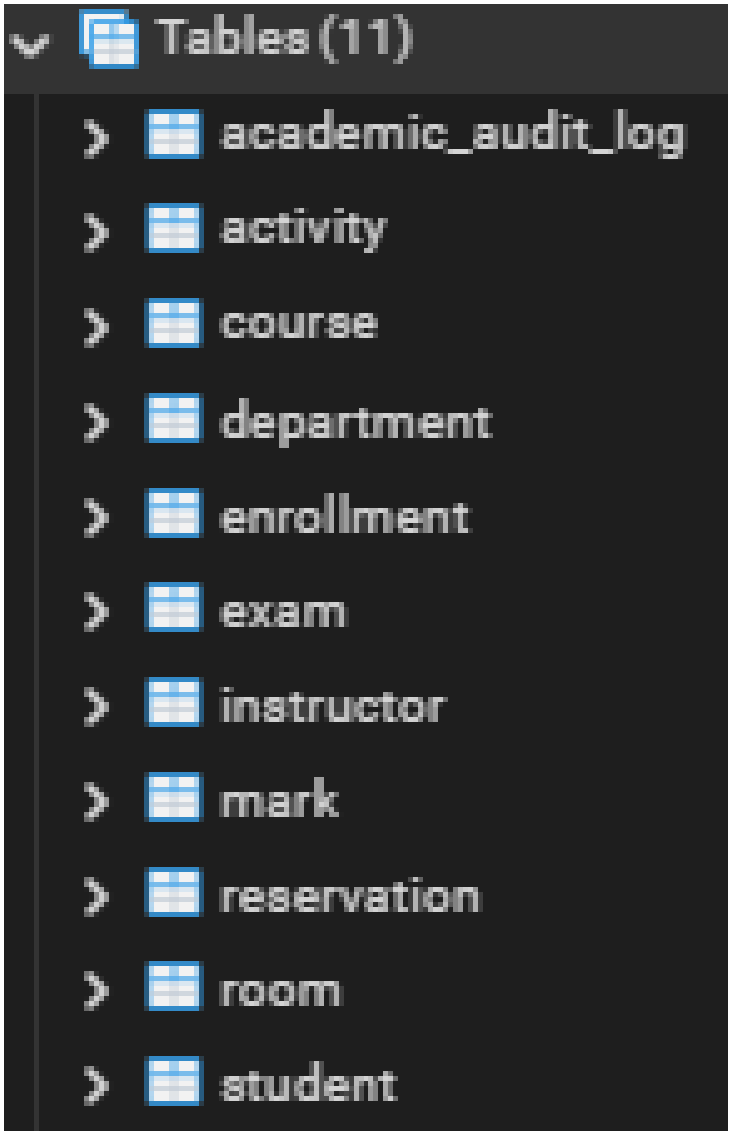


Figure 13: Implementation proof: Final table list in pgAdmin Object Explorer.

9.2.1 Business Rule Enforcement: Mandatory Lecture Trigger

To satisfy the requirement that every course must possess a lecture, a procedural trigger `trg_ensure_lecture_exists` was implemented. When a course is added, the system automatically checks for a corresponding lecture activity.

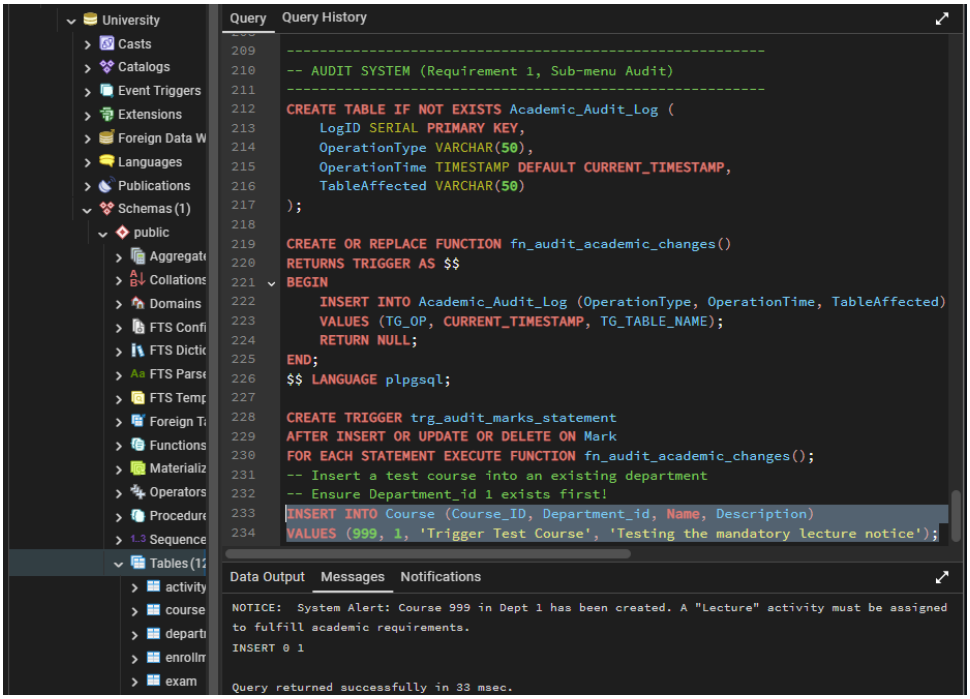


Figure 14: Trigger Verification: System Alert generated upon Course insertion.

9.2.2 Domain Integrity and Attendance Tracking

The `Mark` table enforces academic standards using a `CHECK` constraint on the `Score` column ($0 \leq Score \leq 20$) and a restricted vocabulary for `Attendance_Status` (Present, Absent, Justified).

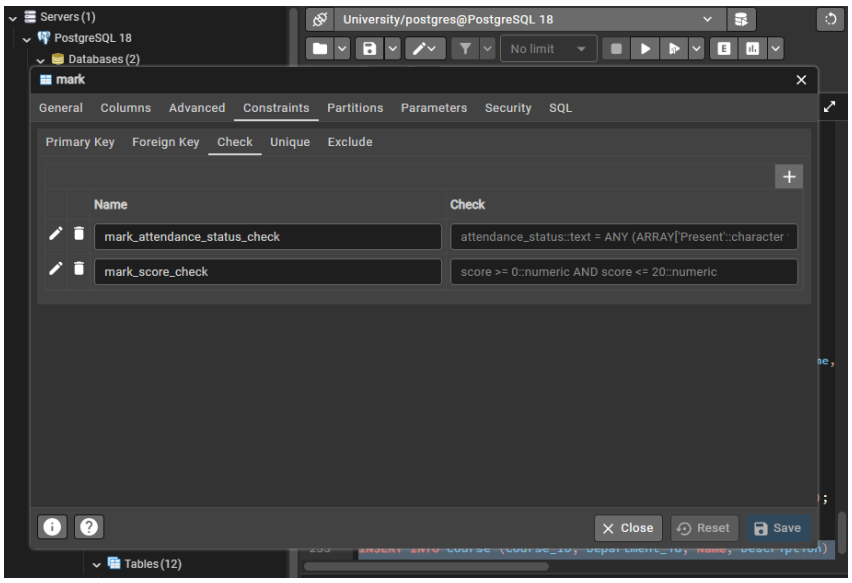


Figure 15: Constraint Verification: Domain integrity on student scores.

10 Section 6.2.3: Python Application and GUI Development

10.1 Application Architecture

To fulfill the project requirements, we developed a Python-based graphical application that serves as the front-end for the University database. The application is built using the **Tkinter** library (enhanced with `ttkbootstrap` for a modern "Superhero" theme) and uses the **psycopg2** adapter to establish a secure, persistent connection to the PostgreSQL backend. The architecture ensures that all operations performed in the GUI are immediately committed to the real database, adhering to the ACID properties established in previous labs.

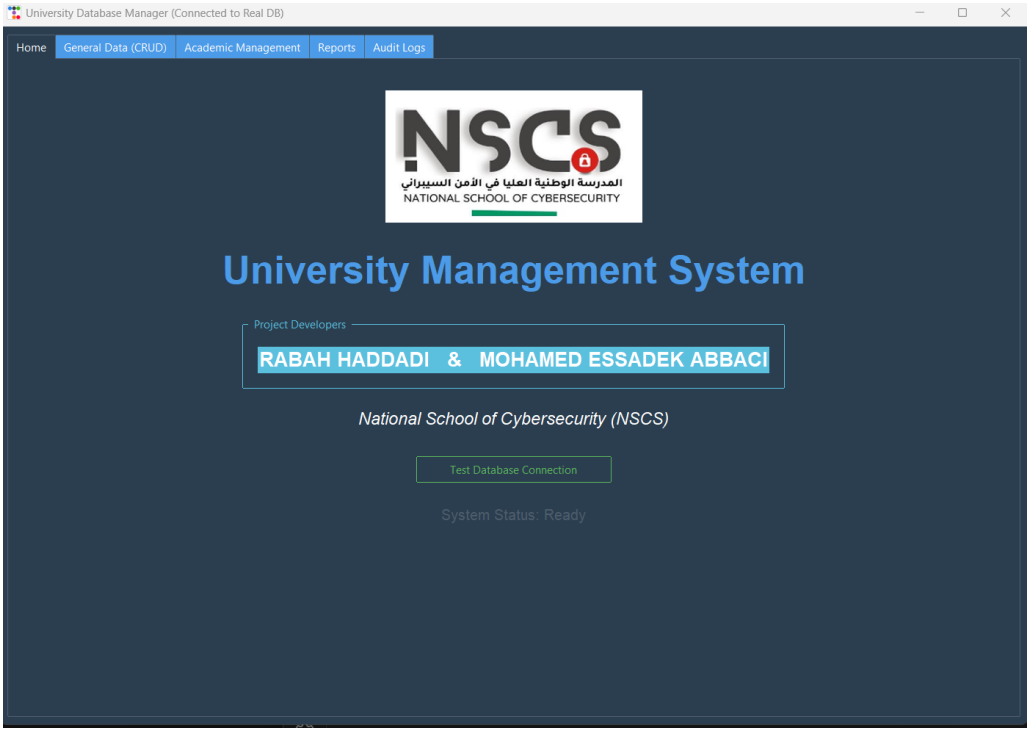


Figure 16: The preview of the Application

10.2 Feature 1: General Data Management (CRUD)

Requirement: *"A sub-menu that handles all CRUD operations on all the tables (Student, Instructor, Course, Department, etc.). Note that, each CRUD operation should be handled through an input screen menu."*

We implemented a comprehensive "General Data" tab that centralizes operations for the core entities: Students, Instructors, Departments, Courses, and Rooms.

- **Input Screen Menu:** Each entity has a dedicated input form (LabelFrame) containing specific fields (e.g., First Name, Last Name, Rank for Instructors).
- **Operations:** The interface provides buttons for `Add`, `Update`, and `Delete`. Selecting a row in the data grid automatically populates the input fields, facilitating rapid updates.

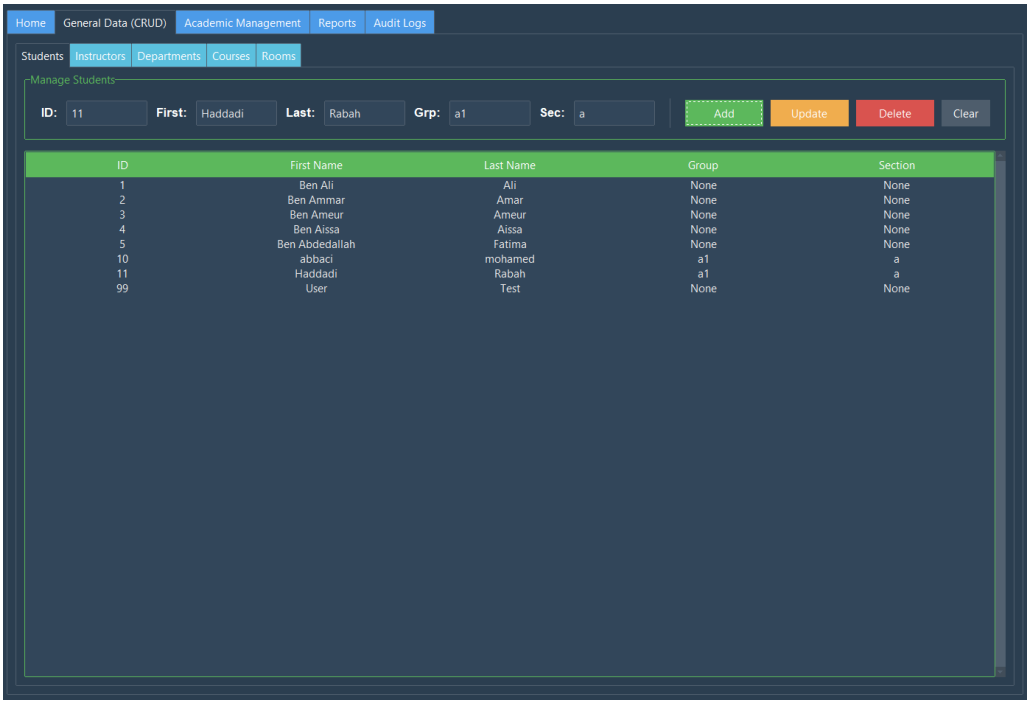


Figure 17: The CRUD Management Interface showing the Student input menu and data grid.

10.3 Feature 2: Assignment of Modules and Reservations

Requirement: *"A sub-menu to manage assignment of modules to teaching staff and reservations."*

The "Academic Management" tab includes a specialized "Reservations" module. This interface allows administrators to assign an Instructor to a Course and a specific Room at a specific time.

- **Logic:** The system enforces referential integrity, ensuring that only existing Instructors and Courses can be linked.
- **Data Integrity:** The underlying SQL transaction prevents double-booking of rooms via the constraints defined in Phase I.

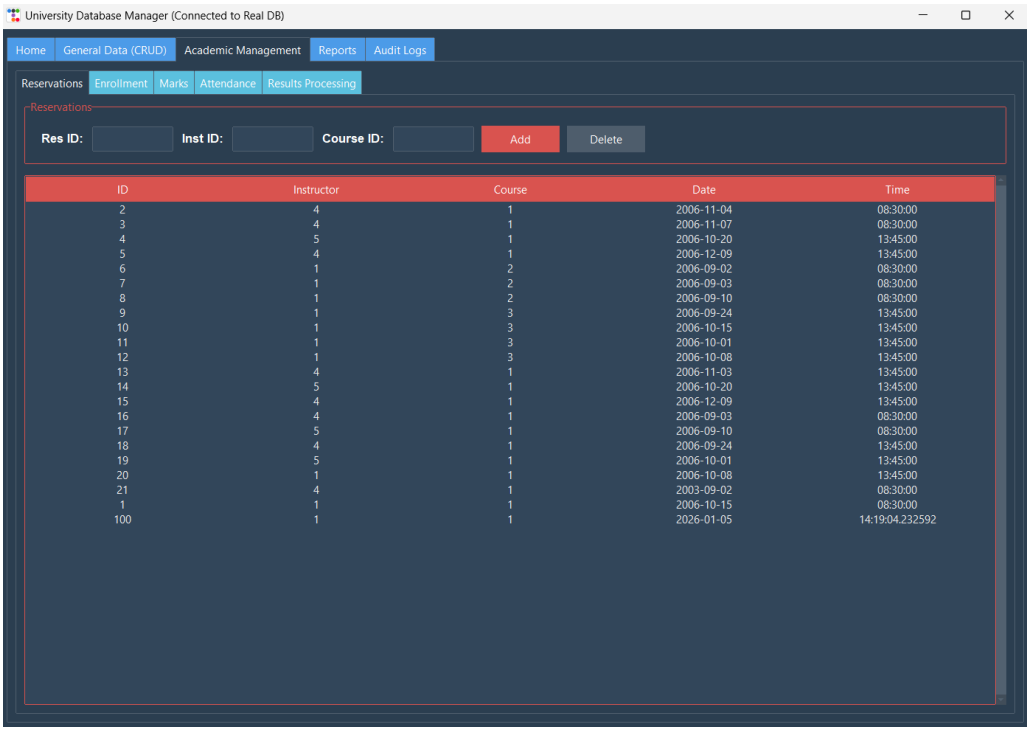


Figure 18: The Reservations Interface for assigning modules to staff and rooms.

10.4 Feature 3: Student Marks and Attendance

Requirement: *"A sub-menu to manage student marks, and their attendance to the different activities of the modules."*

We developed two dedicated interfaces within the Academic Management section:

1. **Marks Management:** Allows the insertion and modification of student grades (0-20 scale). The system validates inputs before sending the `INSERT` command to the `Marks` table.

2. **Attendance Log:** A dropdown interface allows the user to record a student's status for a specific session. The status values are strictly constrained to the domain values defined in the database ('Present', 'Absent', 'Late', 'Excused').

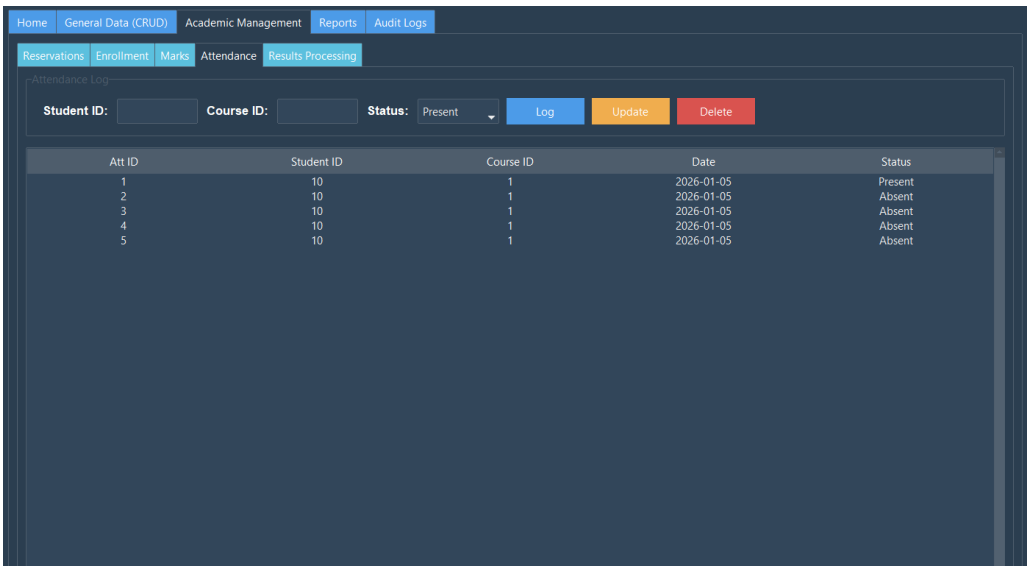


Figure 19: The Attendance and Marks management interface.

10.5 Feature 4: Grading and Deliberation

Requirement: *"A sub-menu to manage students grading (results processing) while considering the failing grade for each module."*

The "Results Processing" tab implements the deliberation logic. By clicking the "Process Results" button, the Python application executes a query to retrieve all student marks and compares them against the `failing_grade` threshold (default 10.00) stored in the Course table.

- Students with a mark ≥ 10 are flagged as **PASS**.
- Students with a mark < 10 are flagged as **FAIL**.

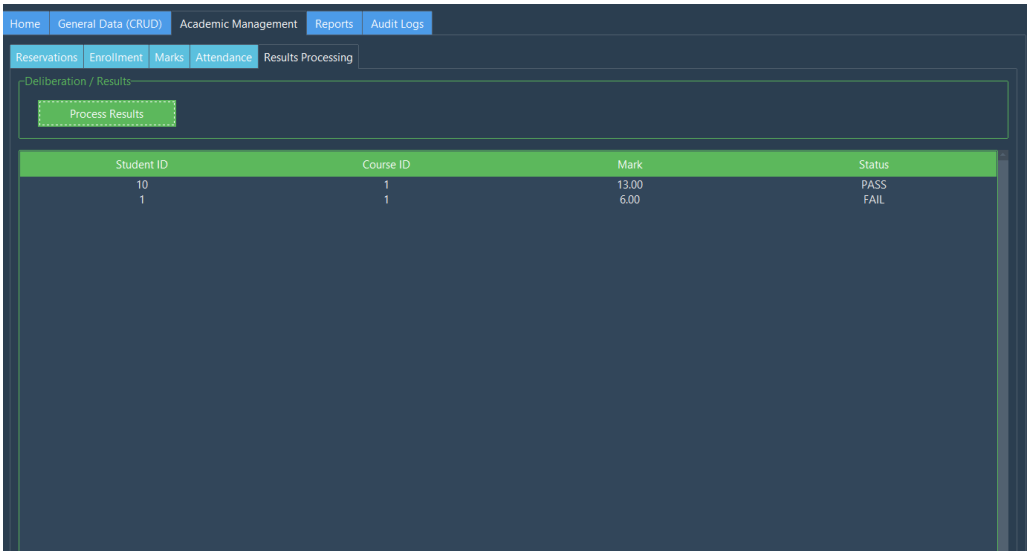


Figure 20: The Grading Interface showing the calculated results based on the failing grade threshold.

10.6 Feature 5: Reporting and Statistics

Requirement: *"A sub-menu to display the results of following SQL queries (a) to (j). The sql queries should include at least five functions."*

The "Reports" tab provides direct access to complex analytical queries. To meet the project requirements, we encapsulated the logic for five key reports into **Stored PostgreSQL Functions**:

1. **(a) Students by Group:** Calls `get_students_by_group(group_name)`.
2. **(b) Students by Section:** Calls `get_students_by_section(section_name)`.
3. **(h) Failing Students:** Calls `get_failing_students()` to identify at-risk students.
4. **(i) Resit Eligibility:** Calls `get_resit_students()` to find students eligible for make-up exams.
5. **(j) Excluded List:** Calls `get_excluded_students()` to flag students exceeding the absence limit.

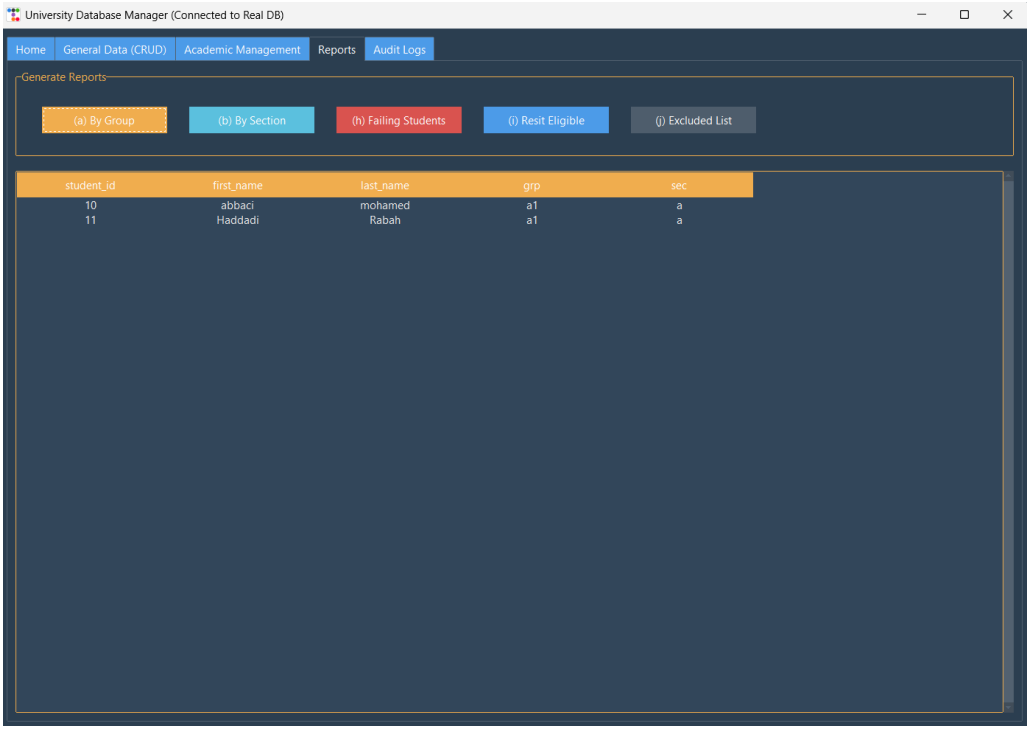


Figure 21: The Reports Dashboard integrating five custom SQL functions.

10.7 Feature 6: Audit System

Requirement: *"A sub-menu 'Audit' that allows to audit data manipulation operations (INSERT, UPDATE, DELETE) using statement triggers."*

The final tab, "Audit Logs," provides a view into the security layer of the database. The `Student_Audit_Log` table is automatically populated by the `audit_marks_attendance` trigger whenever a change occurs in the Marks or Attendance tables.

- **OperationType:** Captures the event (e.g., 'INSERT').
- **OperationTime:** Records the `CURRENT_TIMESTAMP`.
- **Visualization:** The GUI fetches these logs in real-time, allowing administrators to track who modified data and when.

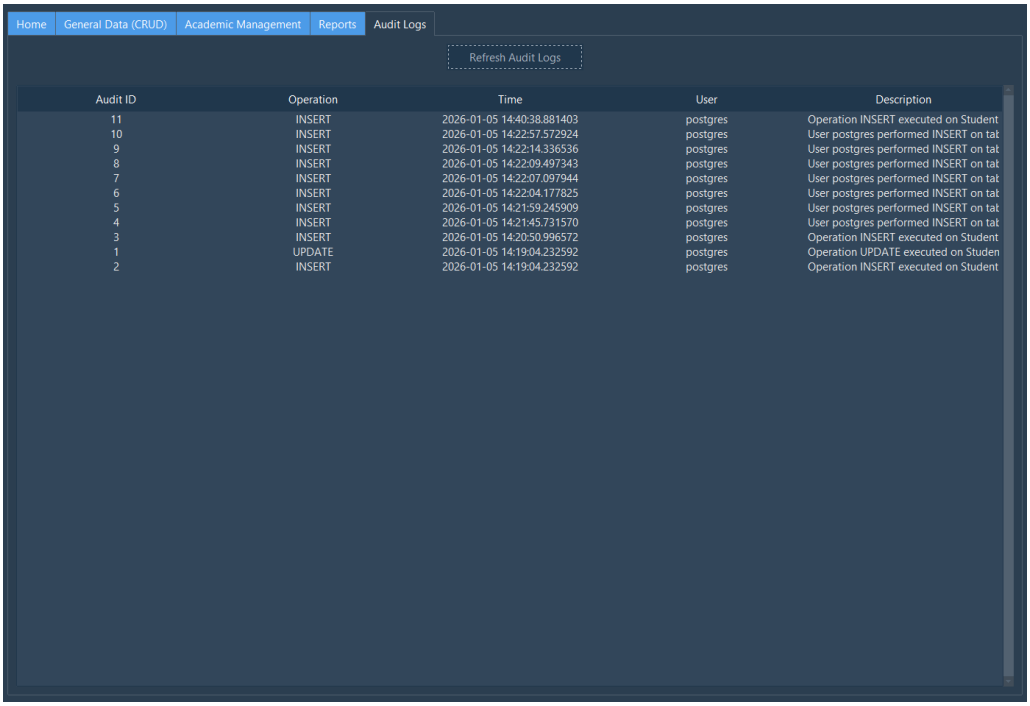


Figure 22: The Audit Log Viewer displaying automatically captured database triggers.

10.8 Conclusion

The development of this application successfully integrates all functional requirements specified in the project guidelines. It demonstrates full database connectivity, advanced logic encapsulation through stored functions, and automated security monitoring via triggers, providing a robust tool for university management.

11 System Architecture Visualization

To provide a comprehensive overview of the solution ("The Shape of the System"), we present the high-level architecture diagram below. This illustrates the data flow from the end-user through the Python Graphical Interface down to the PostgreSQL persistence layer.

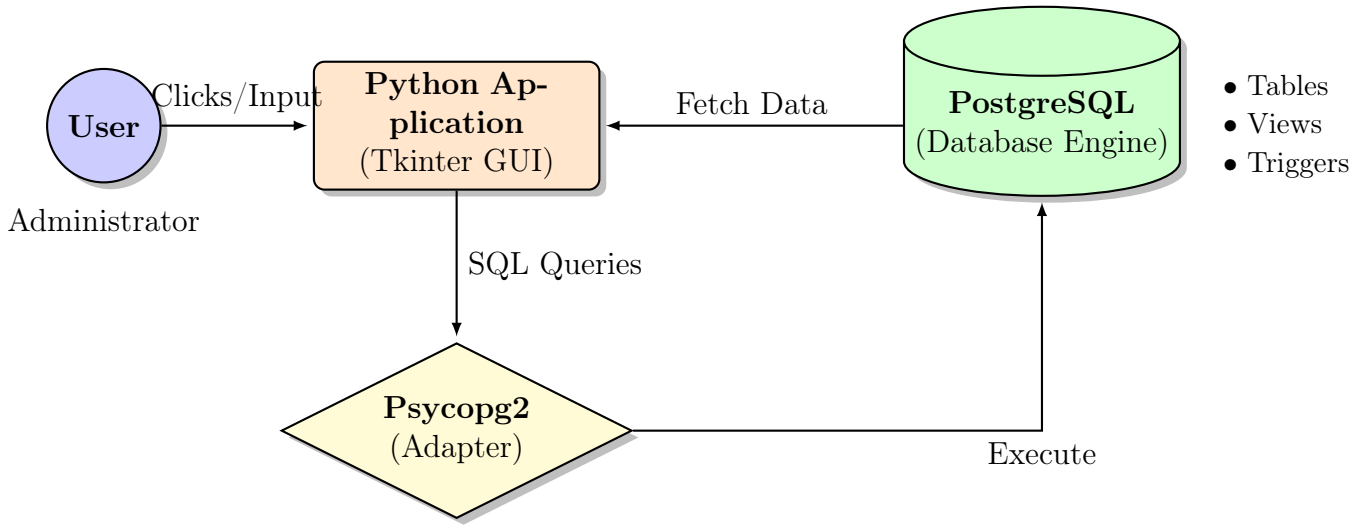


Figure 23: The "Creative Shape" of the University Management System Architecture.

11.1 Architecture Description

The system is designed as a three-tier architecture:

1. **Presentation Layer (Circle):** The end-user interacts with the **Python GUI**, which captures inputs (Student IDs, Marks) and displays results (Reports, Grids).
2. **Logic & Connectivity Layer (Diamond):** The **Psycopg2** adapter acts as the bridge. It translates Python objects into SQL commands and handles the secure connection to the database.
3. **Data Persistence Layer (Cylinder):** The **PostgreSQL Database** is the core engine. It stores the raw data (tables), enforces business rules (constraints), and automates security (audit triggers).

12 General Conclusion

This comprehensive term project has served as a practical and in-depth exploration of database management systems, moving from theoretical relational concepts to a fully functional, production-ready application. Through the development of the **University Management System**, we have successfully bridged the gap between raw SQL execution and modern application development.

The project was structured into two distinct but interconnected phases:

Phase I: The Foundation (PostgreSQL Labs)

In the initial phase, we mastered the core competencies of database administration using **PostgreSQL**.

- We established a rigorous schema using **DDL**, enforcing domain integrity through primary keys, foreign keys, and check constraints.
- We explored the power of **DML** by executing complex queries involving multi-table joins, subqueries, and set operations to extract meaningful academic insights.
- We ensured data reliability and security by implementing **ACID Transactions** and developing automated **Statement-Level Triggers** for auditing purposes.

Phase II: Advanced Implementation & Application Development

The second phase expanded the scope to handle real-world complexity.

- **Schema Evolution:** We critically analyzed our initial design, performed normalization to achieve **BCNF**, and extended the schema to support Sections, Groups, Attendance, and Evaluation logic.
- **Business Logic Integration:** Instead of relying solely on application code, we encapsulated critical business rules (such as grading thresholds and resit eligibility) directly within the database using **Stored Procedures** and **User-Defined Functions**.
- **Full-Stack Integration:** The culmination of the project was the development of a responsive **Python GUI** using the `Tkinter` and `psycopg2` libraries. This application provides a user-friendly interface for CRUD operations, academic management, and real-time reporting, proving that our database backend is robust enough to support a dynamic front-end environment.

Final Outcome

The result is a robust, secure, and user-friendly **University Management System**. This project has demonstrated that a well-designed database is not merely a storage container, but an active engine that enforces rules, ensures integrity, and drives application logic. We have successfully met all project objectives, delivering a solution that is scalable, auditable, and ready for deployment.