University Database Management System - Technical Documentation

Project Overview

This project implements a comprehensive **Educational Institution Management System** using PostgreSQL. The system is designed to manage various aspects of an academic environment, including students, instructors, courses, departments, classrooms, and course reservations. The database architecture follows relational database best practices with proper normalization, referential integrity, and constraint enforcement.

Purpose and Scope

The primary objective of this system is to provide a robust database solution for managing:

- Student enrollment and academic records
- Instructor assignments and departmental organization
- Course offerings and scheduling
- Room reservations and resource allocation
- Academic performance tracking through marks

This documentation provides a detailed analysis of the database schema, data population, querying capabilities, custom functions, transaction management, and automated triggers.

All this is to make sure we have a better understanding specially in the technical part for the University Database Management System.

Table of Contents

Database Schema and Structure

The database consists of **eight interconnected tables** that model the educational institution's data requirements. The schema demonstrates proper relational design with clear entity separation, foreign key relationships, and data integrity constraints.

Entity-Relationship Overview

Entities:

- Department (Core entity)
- Course (Core entity)
- Student (People entity)
- Instructor (People entity)
- Room (Resource entity)
- Enrollment (Associative entity - M:N between Student and Course)
- Reservation (Associative entity - links Course, Instructor, and Room)
- Mark (Associative entity - links Student and Course for grades)

Relationships:

- Department offers Course (1:M, TOTAL participation)
- Department employs Instructor (1:M, TOTAL participation)
- Student enrolls in Course via Enrollment (M:N)
- Student receives Mark (1:M, PARTIAL participation)
- Course grades Mark (1:M, PARTIAL participation)
- Course scheduled in Reservation (1:M, PARTIAL participation)
- Instructor teaches Reservation (1:M, PARTIAL participation)
- Room hosts Reservation (1:M, PARTIAL participation)

Table Definitions

1. Department Table

Stores information about academic departments within the institution.

```
CREATE TABLE Department (
    Department_id INTEGER,
    name VARCHAR(25) NOT NULL,
    CONSTRAINT UN_Department_Name UNIQUE (name),
    CONSTRAINT PK_Department PRIMARY KEY (Department_id)
);
```

Attributes:

- `Department_id` (INTEGER): Primary key uniquely identifying each department
- `name` (VARCHAR(25)): Department name with uniqueness constraint

Constraints:

- Primary key on `Department_id`
- Unique constraint on `name` to prevent duplicate department names

Business Logic: Each department must have a unique identifier and name. The NOT NULL constraint ensures no department exists without a name.

---

2. Student Table

Manages student information including personal details and contact information.

```
CREATE TABLE Student (
    Student_ID INTEGER,
    Last_Name VARCHAR(25) NOT NULL,
    First_Name VARCHAR(25) NOT NULL,
    DOB DATE NOT NULL,
    Address VARCHAR(50),
    City VARCHAR(25),
    Zip_Code VARCHAR(9),
    Phone VARCHAR(10),
    Fax VARCHAR(10),
    Email VARCHAR(100),
    CONSTRAINT PK_Student PRIMARY KEY (Student_ID)
);
```

Attributes:

- `Student_ID` (INTEGER): Primary key
- `Last_Name`, `First_Name` (VARCHAR(25)): Required name fields
- `DOB` (DATE): Date of birth (required)
- `Address`, `City`, `Zip_Code` : Optional location information
- `Phone`, `Fax`, `Email` : Optional contact details

Business Logic: Students must have basic identifying information (ID, name, DOB), while contact and address details are optional, allowing flexibility for incomplete records.

Table Structure:

- Primary Key: Student_ID [PK]
- Last_Name: VARCHAR(25) [NOT NULL]
- First_Name: VARCHAR(25) [NOT NULL]
- DOB: DATE [NOT NULL]
- Address: VARCHAR(50)
- City: VARCHAR(25)
- Zip_Code: VARCHAR(9)
- Phone: VARCHAR(10)
- Fax: VARCHAR(10)
- Email: VARCHAR(100)

---

3. Course Table

Defines courses offered by different departments.

```
CREATE TABLE Course (
    Course_ID INT4 NOT NULL,
    Department_ID INT4 NOT NULL,
    name VARCHAR(60) NOT NULL,
    Description VARCHAR(1000),
    CONSTRAINT PK_Course PRIMARY KEY (Course_ID, Department_ID),
    CONSTRAINT FK_Course_Department
        FOREIGN KEY (Department_ID)
        REFERENCES Department (Department_id)
        ON UPDATE RESTRICT
        ON DELETE RESTRICT
);
```

Attributes:

- `Course_ID`, `Department_ID` : Composite primary key
- `name` (VARCHAR(60)): Course title (required)
- `Description` (VARCHAR(1000)): Optional detailed course information

Relationships:

- **Many-to-One** with Department: Each course belongs to exactly one department
- Foreign key constraint with RESTRICT policy prevents deletion/update of departments with associated courses

Design Decision: The composite primary key (Course_ID, Department_ID) allows the same course number to exist in different departments while maintaining uniqueness within each department.

---

4. Instructor Table

Stores instructor information with their departmental affiliation and academic rank.

```
CREATE TABLE Instructor (
    Instructor_ID INTEGER,
    Department_ID INTEGER NOT NULL,
    Last_Name VARCHAR(25) NOT NULL,
    First_Name VARCHAR(25) NOT NULL,
    Rank VARCHAR(25),
    Phone VARCHAR(10),
    Fax VARCHAR(10),
    Email VARCHAR(100),
    CONSTRAINT CK_Instructor_Rank
        CHECK (Rank IN ('Substitute', 'MCB', 'MCA', 'PROF')),
    CONSTRAINT PK_Instructor PRIMARY KEY (Instructor_ID),
    CONSTRAINT FK_Instructor_Department
        FOREIGN KEY (Department_ID)
        REFERENCES Department (Department_id)
        ON UPDATE RESTRICT
        ON DELETE RESTRICT
);
```

Attributes:

- `Instructor_ID` (INTEGER): Primary key
- `Department_ID` (INTEGER): Required foreign key to Department
- `Last_Name`, `First_Name` : Required name fields
- `Rank` : Academic rank with CHECK constraint
- Contact information (Phone, Fax, Email): Optional

Business Rules:

- **Rank Constraint:** Only allows four specific values: 'Substitute', 'MCB', 'MCA', 'PROF'
- Every instructor must be assigned to a department
- Referential integrity prevents orphaned instructor records

Instructor-Department Relationship:

- Department employs Instructor (1:M)
- Department has Department_id (PK, UNIQUE, NOT NULL)
- Instructor has Instructor_ID (PK), Last_Name (NOT NULL), First_Name (NOT NULL), Rank (CHECK IN: Substitute, MCB, MCA, PROF), Phone, Fax, Email, Department_ID (FK, NOT NULL, references Department)

Rank Constraint Values: The Rank field can only contain:

- Substitute
- MCB
- MCA
- PROF

---

5. Room Table

Manages physical classroom and lecture hall information.

```
CREATE TABLE Room (
    Building VARCHAR(1),
    RoomNo VARCHAR(10),
    Capacity INTEGER CHECK (Capacity > 1),
    CONSTRAINT PK_Room PRIMARY KEY (Building, RoomNo)
);
```

Attributes:

- `Building`, `RoomNo` : Composite primary key
- `Capacity` (INTEGER): Number of seats with constraint ensuring at least 2

Design Rationale:

- The composite key allows room numbers to be reused across different buildings
- The capacity constraint ensures rooms are suitable for teaching (minimum 2 people)

---

6. Reservation Table

The central table managing room bookings for courses, linking instructors, rooms, and courses.

```
CREATE TABLE Reservation (
    Reservation_ID INTEGER,
    Building VARCHAR(1) NOT NULL,
    RoomNo VARCHAR(10) NOT NULL,
    Course_ID INTEGER NOT NULL,
    Department_ID INTEGER NOT NULL,
    Instructor_ID INTEGER NOT NULL,
    Reserv_Date DATE NOT NULL DEFAULT CURRENT_DATE,
    Start_Time TIME NOT NULL DEFAULT CURRENT_TIME,
    End_Time TIME NOT NULL DEFAULT '23:00:00',
    Hours_Number INTEGER NOT NULL,

    CONSTRAINT PK_Reservation PRIMARY KEY (Reservation_ID),
    CONSTRAINT FK_Reservation_Room
        FOREIGN KEY (Building, RoomNo)
        REFERENCES Room (Building, RoomNo)
        ON UPDATE RESTRICT ON DELETE RESTRICT,
    CONSTRAINT FK_Reservation_Course
        FOREIGN KEY (Course_ID, Department_ID)
        REFERENCES Course (Course_ID, Department_ID)
        ON UPDATE RESTRICT ON DELETE RESTRICT,
    CONSTRAINT FK_Reservation_Instructor
        FOREIGN KEY (Instructor_ID)
        REFERENCES Instructor (Instructor_ID)
        ON UPDATE RESTRICT ON DELETE RESTRICT,
    CONSTRAINT CK_Reservation_Hours
        CHECK (Hours_Number >= 1),
    CONSTRAINT CK_Reservation_StartEndTime
        CHECK (Start_Time < End_Time)
);
```

Attributes:

- `Reservation_ID` : Primary key
- `Building`, `RoomNo` : Foreign key to Room (composite)
- `Course_ID`, `Department_ID` : Foreign key to Course (composite)
- `Instructor_ID` : Foreign key to Instructor

- `Reserv_Date` : Reservation date with default to current date
- `Start_Time` , `End_Time` : Time boundaries with validation
- `Hours_Number` : Duration in hours

Complex Relationships: This table acts as a junction with three foreign key relationships:

1. **To Room:** Specifies which room is reserved
2. **To Course:** Identifies the course being taught
3. **To Instructor:** Designates who is teaching

Business Rules:

- Hours must be at least 1
- Start time must be before end time
- All references use RESTRICT policy to maintain data integrity

Reservation Relationships:

- Room hosts Reservation (Building PK, RoomNo PK, Capacity CHECK > 1)
- Reservation (Reservation_ID PK, Building FK, RoomNo FK, Course_ID FK, Department_ID FK, Instructor_ID FK, Reserv_Date DEFAULT CURRENT_DATE, Start_Time DEFAULT CURRENT_TIME, End_Time DEFAULT 23:00:00, Hours_Number CHECK >= 1)
- Course scheduled in Reservation (Course_ID PK, Department_ID PK, name, Description)
- Instructor teaches Reservation (Instructor_ID PK, Department_ID FK, Last_Name, First_Name, Rank)

Reservation Constraints:

1. Check if Hours_Number >= 1
    - Yes: Check if Start_Time < End_Time
        - Yes: Valid Reservation
        - No: Reject (Start must be before End)
    - No: Reject (Hours must be >= 1)

---

7. Enrollment Table

Tracks student enrollment in courses (many-to-many relationship).

```
CREATE TABLE Enrollment (
    Student_ID INT NOT NULL,
    Course_ID INT NOT NULL,
    Department_ID INT NOT NULL,
    Enrollment_Date DATE NOT NULL,

    CONSTRAINT PK_Enrollment
        PRIMARY KEY (Student_ID, Course_ID, Department_ID),
    CONSTRAINT FK_Enrollment_Student
        FOREIGN KEY (Student_ID)
        REFERENCES Student (Student_ID)
        ON UPDATE RESTRICT ON DELETE RESTRICT,
    CONSTRAINT FK_Enrollment_Course
        FOREIGN KEY (Course_ID, Department_ID)
        REFERENCES Course (Course_ID, Department_ID)
        ON UPDATE RESTRICT ON DELETE RESTRICT
);
```

Purpose: Resolves the many-to-many relationship between students and courses. A student can enroll in multiple courses, and a course can have multiple students.

Composite Primary Key: Ensures a student can enroll in a specific course only once.

---

8. Mark Table

Records student grades/marks for courses.

```
CREATE TABLE Mark (
    Mark_ID SERIAL,
    Student_ID INTEGER NOT NULL,
    Course_ID INTEGER NOT NULL,
    Department_ID INTEGER NOT NULL,
    Mark_Value NUMERIC(4,2) NOT NULL,
    Mark_Date DATE NOT NULL,

    CONSTRAINT PK_Mark PRIMARY KEY (Mark_ID),
    CONSTRAINT FK_Mark_Student
        FOREIGN KEY (Student_ID)
        REFERENCES Student (Student_ID)
        ON UPDATE RESTRICT ON DELETE RESTRICT,
    CONSTRAINT FK_Mark_Course
        FOREIGN KEY (Course_ID, Department_ID)
        REFERENCES Course (Course_ID, Department_ID)
        ON UPDATE RESTRICT ON DELETE RESTRICT,
    CONSTRAINT CK_Mark_Value
        CHECK (Mark_Value >= 0 AND Mark_Value <= 20)
);
```

Attributes:

- `Mark_ID` (SERIAL): Auto-incrementing primary key
- `Mark_Value` (NUMERIC(4,2)): Precise grade with 2 decimal places
- `Mark_Date`: When the grade was recorded

Business Rules:

- Marks must be between 0 and 20 (standard Algerian grading system)
- Foreign keys ensure marks are only recorded for valid student-course combinations

Complete Schema Relationships:

- Department offers Course (1:M)
- Department employs Instructor (1:M)
- Student enrolls via Enrollment (M:N with Course)
- Student receives Mark (1:M)
- Course grades for Mark (1:M)
- Course scheduled Reservation (1:M)
- Instructor teaches Reservation (1:M)
- Room hosts Reservation (1:M)

Data Population Strategy

The database is populated with realistic sample data representing an educational institution with 4 departments, 5 students, 6 instructors, 4 courses, 5 rooms, and numerous reservations.

Department Data

Four academic departments are established:

```
 INSERT INTO Department VALUES (1, 'SADS');
INSERT INTO Department VALUES (2, 'CCS');
INSERT INTO Department VALUES (3, 'GRC');
INSERT INTO Department VALUES (4, 'INS');
```

Student Population

Five students with diverse demographic information:

Student_ID	Name	City	Contact Available
1	Ali Ben Ali	Algiers	Full contact info
2	Amar Ben Ammar	BATNA	Phone only
3	Ameur Ben Ameur	Oran	Phone and Fax

Student_ID	Name	City	Contact Available
4	Aissa Ben Aissa	Annaba	Phone and Email
5	Fatima Ben Abdedallah	Constantine	No contact info

Design Note: Notice the intentional variation in contact information completeness, simulating real-world scenarios with incomplete data.

Student Distribution by City:

- Algiers: 1 student
- BATNA: 1 student
- Oran: 1 student
- Annaba: 1 student
- Constantine: 1 student

Distribution Summary:

- 5 students distributed across 5 different cities
- 1 student per city (even distribution)

Instructor Assignment

Six instructors distributed across departments with varying ranks:

- **SADS Department (ID: 1):** 5 instructors
  - 1 Professor (PROF)
  - 2 Associate Professors (MCA)
  - 1 Assistant Professor (MCB)
  - 1 Substitute
- **INS Department (ID: 4):** 1 Substitute instructor

Rank Distribution:

```
PROF: 1 instructor
MCA: 2 instructors
MCB: 1 instructor
Substitute: 2 instructors
```

Course Catalog

Four courses are defined:

1. **Databases** (SADS) - Licence L3 level covering E/A modeling, SQL, normalization
2. **C++ Programming** (SADS) - Master 1 level
3. **Advanced Databases** (SADS) - Master 2 level
4. **English** (INS) - General requirement

Course Distribution by Department:

- SADS: 3 courses (75%)
- INS: 1 course (25%)

Room Configuration

Five teaching spaces with varying capacities:

Building	Room Number	Capacity	Type
B	020	15	Small classroom
B	022	15	Small classroom
A	301	45	Medium classroom
C	Hall 1	500	Large lecture hall
C	Hall 2	200	Medium lecture hall

Note: There's a duplicate entry in the original data (Room B-020 inserted twice with slightly different formatting), which should be handled during actual implementation.

Reservation Patterns

Twenty-one reservations are created spanning from September to December 2006, with a few outlier dates. The data shows:

- **Most Active Room:** B-022 with 14 reservations
- **Most Active Instructor:** Instructor #1 and #4 (most reservations)

- **Peak Time Slots:** Morning (08:30-11:45) and Afternoon (13:45-17:00)
- **All sessions:** 3-hour blocks

Room Utilization Analysis:

- **B-022**: 14 reservations (Highest usage - Small classroom)
- **A-301**: 6 reservations (Medium usage - Medium classroom)
- **B-020**: 2 reservations (Low usage - Small classroom)
- **C-Hall 1**: 0 reservations (Unused - Large lecture hall)
- **C-Hall 2**: 0 reservations (Unused - Medium lecture hall)

Query Operations and Analysis

The `Qureying.sql` file demonstrates a comprehensive range of SQL operations from basic SELECT statements to complex queries involving subqueries, joins, aggregations, and set operations.

Basic Queries (Questions 1-4)

Simple Selection and Filtering

Q1: List all student names

```
SELECT last_name, first_name FROM Student;
```

Q2: Students from a specific city

```
SELECT last_name, first_name FROM Student WHERE city LIKE 'Annaba';
```

Result: Returns Aissa Ben Aissa

Q3: Pattern matching - Last names starting with 'A'

```
SELECT last_name, first_name FROM Student WHERE last_name LIKE 'A%';
```

Uses the wildcard operator for pattern matching

Q4: Advanced pattern - Second-to-last letter is 'E'

```
SELECT last_name, first_name FROM Instructor WHERE last_name LIKE '%e_';
```

The pattern '%e_' means: any characters, then 'e', then exactly one more character

Pattern Matching Guide:

- `%` = Any sequence of characters (including zero)
- `_` = Exactly one character
- `A%` = Starts with 'A'
- `%e_` = 'e' is second-to-last character
- `%text%` = Contains 'text' anywhere

Pattern Examples:

- **Pattern: 'A%'**: Ali (Match - Starts with A), Amar (Match - Starts with A), Ben (No Match - Doesn't start with A)
- **Pattern: '%e_'**: Abbes (Match - Second-to-last is 'e'), Chad (No Match - No 'e' before last char)
- **Pattern: '%Licence%'**: Licence L3: Modeling (Match - Contains 'Licence'), Master 1 (No Match - Doesn't contain 'Licence')

Joins and Ordering (Question 5)

Complex Multi-Table Join with Sorting

```
SELECT i.last_name, i.first_name
FROM Instructor i
JOIN Department d ON i.Department_ID = d.Department_ID
ORDER BY d.name, i.last_name, i.first_name;
```

Technical Analysis:

- Inner join connects instructors with their departments
- Three-level sorting: Department name → Last name → First name
- Demonstrates hierarchical organization of data

JOIN Process:

1. For each row in Instructor
2. Find matching rows in Department (where Department_IDs match)
3. Combine matched columns
4. Sort results by 3 levels

Steps:

1. Instructor Table and Department Table → Start JOIN
2. Match on Department_ID?
    - Yes → Combine Rows → Result Set → ORDER BY (1. Department.name → 2. Last_Name → 3. First_Name) → Final Sorted Result
    - No → Discard Row

Aggregation Functions (Questions 6, 11)

Q6: Counting instructors by rank

```
SELECT COUNT(rank) AS num_teachers_with_grade_Supleant
FROM Instructor
WHERE rank LIKE 'Substitute';
```

Result: 2 substitute instructors

Q11: Statistical analysis of room capacities

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

Calculates average (155) and maximum (500) room capacity

Key Concept: Aggregate functions (COUNT, AVG, MAX, MIN, SUM) operate on sets of rows to produce single values.

NULL Value Handling (Question 7)

```
SELECT last_name, first_name FROM Student WHERE phone IS NULL;
```

Important: Uses `IS NULL` rather than `= NULL` because NULL represents unknown/missing data and cannot be compared using standard equality operators.

Three-Valued Logic in SQL:

- `TRUE` : Condition is satisfied
- `FALSE` : Condition is not satisfied
- `UNKNOWN` : Cannot be determined (involves NULL)

Correct NULL Checking:

- ✓ `WHERE column IS NULL`
- ✓ `WHERE column IS NOT NULL`
- ✗ `WHERE column = NULL` (always UNKNOWN)
- ✗ `WHERE column != NULL` (always UNKNOWN)

NULL Value Logic:

- Check if phone IS NULL → Value Status:
    - NULL (unknown) → IS NULL returns TRUE
    - Has value → IS NULL returns FALSE
- WRONG: phone = NULL → Comparison:
    - NULL compared to NULL → UNKNOWN (not TRUE or FALSE)
    - Value compared to NULL → UNKNOWN

Advanced Text Search (Question 8)

```
SELECT * FROM Course WHERE description ILIKE '%Licence%';
```

Features:

- `ILIKE` : Case-insensitive pattern matching (PostgreSQL-specific)
- Searches for courses with "Licence" anywhere in the description

---

Complex Calculations (Questions 9,10,22)

Q9: Course Cost Calculation

```
SELECT c.Course_ID, SUM(hours_number)*3000 AS course_total_cost
FROM Reservation r
JOIN Course c ON c.Course_ID = r.Course_ID
GROUP BY c.Course_ID
ORDER BY course_total_cost ASC;
```

Logic Flow:

1. Join Reservation with Course
2. Group by Course_ID to aggregate hours per course
3. Calculate total cost: Sum of hours × 3000 DA per hour
4. Sort by cost ascending

Course Costs:

- Course 1: Databases - 189,000 DA (63 hours × 3000)
- Course 3: Advanced DB - 36,000 DA (12 hours × 3000)
- Course 2: C++ Programming - 27,000 DA (9 hours × 3000)

Cost Calculation Formula:

```
Total Cost = SUM(Hours per Course) × 3000 DA/hour
```

Example Breakdown:

- Course 1: 63 hours × 3000 = 189,000 DA
- Course 2: 9 hours × 3000 = 27,000 DA
- Course 3: 12 hours × 3000 = 36,000 DA

Q10: Filtered Cost Analysis

```
SELECT c.name, SUM(r.hours_number) * 3000 AS course_total_cost
FROM Course c
JOIN Reservation r ON r.Course_ID = c.Course_ID
GROUP BY c.Course_ID, c.name
HAVING SUM(r.hours_number) * 3000 BETWEEN 30000 AND 50000
ORDER BY course_total_cost ASC;
```

Key Features:

- Uses `HAVING` clause to filter aggregated results (not WHERE)
- BETWEEN operator for range checking
- Groups by both ID and name for complete information

Q22: Same Date Reserved Rooms

```
SELECT reserv_date
FROM reservation
WHERE roomno IN ('022','020','301')
GROUP BY reserv_date
HAVING COUNT(DISTINCT roomno) = 3;
```

WHERE vs HAVING:

Aspect	WHERE	HAVING
Filters	Individual rows	Grouped results
Timing	Before GROUP BY	After GROUP BY
Can use aggregates	✗ No	✓ Yes
Example	`WHERE city = 'Annaba'`	`HAVING COUNT(*) > 2`

Example:

```
-- Select departments with more than 2 instructors from Annaba
SELECT Department_ID, COUNT(*) AS total
FROM Instructor
WHERE city = 'Annaba'        -- Filter Rows before grouping
GROUP BY Department_ID
HAVING COUNT(*) > 2;           -- Filter Groups after grouping
```

Note: this example is used for documentation purposes (it's not included within the `Qureying.sql` file)

Process Flow:

- WHERE Clause (Filters BEFORE Grouping): All Rows → WHERE filters rows → Filtered Rows → GROUP BY → Grouped Results
- HAVING Clause (Filters AFTER Grouping): All Rows → GROUP BY → Grouped Results → HAVING filters groups → Final Filtered Groups

---

Subqueries (Questions 12)

Q12: Rooms Below Average Capacity

```
SELECT roomno, capacity
FROM Room
WHERE capacity < (
    SELECT AVG(capacity)
    FROM Room
);
```

Execution Flow:

1. Inner query calculates average capacity
2. Outer query compares each room's capacity to this average
3. Returns rooms below average

Result: Returns small classrooms (capacity 15 and 45) since average is 155.

Subquery Execution:

1. Query Execution → Execute Inner Query First
2. Inner Query: SELECT AVG(capacity) FROM Room → Result: 155
3. Use Result (155) in Outer Query
4. Outer Query: SELECT roomno, capacity FROM Room WHERE capacity < (...)
5. Final Results: Rooms with capacity < 155 (B-020: 15, B-022: 15, A-301: 45)

Multiple Solutions (Question 13)

Solution 1: Using IN operator

```
SELECT first_name, last_name, d.name
FROM Instructor i
JOIN department d ON d.department_id = i.department_id
WHERE d.name IN ('SADS','CSS');
```

Solution 2: Using OR operator

```
SELECT first_name, last_name, d.name
FROM Instructor i
JOIN department d ON d.department_id = i.department_id
WHERE d.name = 'SADS' OR d.name = 'CSS';
```

Comparison: Both achieve the same result; IN is more concise for multiple values.

Negation Queries (Questions 14, 20)

Q14: Instructors NOT in specific departments

```
SELECT first_name, last_name, d.name
FROM Instructor i
JOIN department d ON d.department_id = i.department_id
WHERE d.name NOT IN ('SADS','CSS');
```

Q20: Instructors without reservations

```
SELECT first_name, last_name FROM instructor
WHERE (first_name, last_name) NOT IN (
    SELECT first_name, last_name FROM instructor_reservation
);
```

Advanced Feature: Tuple comparison using `(first_name, last_name)` to check multiple columns simultaneously.

GROUP BY Operations (Questions 16-17)

Q16: Course count per department

```
SELECT d.name, COUNT(c.department_id)
FROM department d
LEFT JOIN course c ON d.department_id = c.department_id
GROUP BY d.name;
```

Technical Note: LEFT JOIN ensures all departments appear, even those without courses (count would be 0).

Q17: Departments with 3+ courses

```
SELECT * FROM
    (SELECT d.name, COUNT(c.department_id) AS course_number
     FROM department d
     LEFT JOIN course c ON d.department_id = c.department_id
     GROUP BY d.name)
WHERE course_number >= 3;
```

Nested Query Structure: Inner query performs grouping, outer query filters the result.

GROUP BY Process:

1. Original Table: Instructor
2. Step 1: GROUP BY Department_ID
    - Grouped Rows:
        - Dept 1: Instructor 1, 2, 3, 4, 5
        - Dept 4: Instructor 6
3. Step 2: Apply COUNT Aggregation
    - Aggregated Results:
        - Dept 1: COUNT = 5
        - Dept 4: COUNT = 1
4. Final Output

GROUP BY Explanation:

1. **Partition** rows into groups based on grouping column(s)
2. **Aggregate** each group using aggregate function (COUNT, SUM, AVG, etc.)
3. **Return** one row per group with aggregated values

Views and EXISTS (Questions 18-19)

Creating a View

```
CREATE VIEW instructor_reservation AS
    SELECT i.first_name, i.last_name, COUNT(r.instructor_id) AS reservations
    FROM instructor i
    JOIN reservation r ON r.instructor_id = i.instructor_id
    GROUP BY i.first_name, i.last_name;
```

View Benefits:

- Encapsulates complex logic
- Reusable in multiple queries
- Simplifies future queries

Q18: Instructors with 2+ reservations - Two solutions

Solution 1: Using the View

```
SELECT * FROM instructor_reservation WHERE reservations >= 2;
```

Solution 2: Using EXISTS

```
SELECT i.first_name, i.last_name
FROM instructor i
WHERE EXISTS (
    SELECT 1 FROM reservation r
    WHERE r.instructor_id = i.instructor_id
    OFFSET 1
);
```

EXISTS Logic: Returns true if the subquery returns any rows. The `OFFSET 1` skips the first row, so EXISTS succeeds only if there are 2+ rows.

Q19: Instructors with maximum reservations

```
SELECT * FROM instructor_reservation
WHERE reservations >= ALL (
    SELECT reservations FROM instructor_reservation
);
```

ALL Operator: Compares against every value in the subquery result. Greater than or equal to ALL means maximum value.

Comparison of EXISTS, IN, and ALL:

- **EXISTS**: Use for checking existence of rows; Execution stops at first matching row
- **IN**: Use for matching values from a list or subquery; Execution compares each row with collected values
- **ALL**: Use for comparing a value against all results of a subquery; Execution compares each row with all values returned

---

Update Operations (Question 23)

Five diverse UPDATE examples demonstrating different modification patterns:

1. Single row update

```
UPDATE Student SET fax = '0145678970' WHERE student_id = 5;
```

2. Calculated update

```
UPDATE Room SET capacity = capacity + 5 WHERE roomno = '301';
```

Increments existing value

3. Foreign key update

```
UPDATE Instructor SET department_id = 3 WHERE instructor_id = 5;
```

Reassigns instructor to different department

4. Batch update with condition

```
UPDATE Reservation SET end_time = '17:30:00' WHERE end_time >= '17:00:00';
```

Updates multiple rows matching criteria

5. Course reassignment

```
UPDATE course SET department_id = 3 WHERE course_id = 3;
```

UPDATE Process:

1. UPDATE Statement
2. Step 1: WHERE Clause - Filter rows to update
    - Filtering: All Rows in Table → Matches WHERE Condition?
        - Yes → Selected Rows
        - No → Unchanged Rows
3. Step 2: SET Clause - Modify column values
    - Modification: Apply new values to selected rows
    - Example: Old: capacity = 45, New: capacity = 50
4. Step 3: Constraint Validation
    - All Constraints Satisfied?
        - Yes → ✓ Commit Changes → Updated Table State
        - No → ✗ Rollback, Error

UPDATE Safety Checklist:

- ✓ Always use WHERE clause (unless updating all rows intentionally)
- ✓ Test with SELECT first: `SELECT * FROM table WHERE condition`
- ✓ Use transactions for safety: `BEGIN; UPDATE ...; ROLLBACK;` (test)
- ✓ Check constraint violations before executing

Aggregation Examples (Question 24)

Five distinct aggregation patterns:

1. Simple count

```
SELECT COUNT(*) AS total_reservations FROM Reservation;
```

2. Maximum value

```
SELECT MAX(Capacity) AS max_capacity FROM Room;
```

3. Grouped count

```
SELECT Department_ID, COUNT(Instructor_ID) AS num_instructors
FROM Instructor
GROUP BY Department_ID;
```

4. Grouped count with foreign key

```
SELECT Course_ID, COUNT(Reservation_ID) AS total_reservations_per_course
FROM Reservation
GROUP BY Course_ID;
```

5. Course count by department

```
SELECT Department_ID, COUNT(Course_ID) AS total_courses_per_department
FROM Course
GROUP BY Department_ID;
```

Aggregate Functions:

- **COUNT**: Count rows - Example: COUNT(*)
- **SUM**: Total values - Example: SUM(hours)
- **AVG**: Average value - Example: AVG(capacity)
- **MAX**: Maximum value - Example: MAX(mark_value)
- **MIN**: Minimum value - Example: MIN(reserv_date)

When to Use Each:

- **COUNT**: How many? (e.g., "How many students?")
- **SUM**: What's the total? (e.g., "Total hours taught?")
- **AVG**: What's the average? (e.g., "Average room capacity?")
- **MAX**: What's the highest? (e.g., "Highest grade?")
- **MIN**: What's the lowest? (e.g., "Earliest reservation date?")

Set Operations (Question 25)

Five examples demonstrating UNION, INTERSECT, and EXCEPT:

1. UNION - Combine unique results

```
SELECT RoomNo FROM Room WHERE Building = 'A'
UNION
SELECT RoomNo FROM Room WHERE Building = 'B';
```

Removes duplicates

2. UNION ALL - Include duplicates

```
SELECT RoomNo FROM Room WHERE Building = 'A'
UNION ALL
SELECT RoomNo FROM Room WHERE Building = 'B';
```

3. INTERSECT - Common values

```
SELECT RoomNo FROM Room WHERE Building = 'A'
INTERSECT
SELECT RoomNo FROM Room WHERE Capacity > 30;
```

Returns rooms in building A with capacity > 30

4. EXCEPT - Set difference

```
SELECT RoomNo FROM Room WHERE Building = 'A'
EXCEPT
SELECT RoomNo FROM Room WHERE Capacity < 20;
```

Rooms in A excluding those with capacity < 20

5. Course exclusion

```
SELECT Course_ID FROM Course WHERE Department_ID = 1
EXCEPT
SELECT Course_ID FROM Course WHERE Department_ID = 2;
```

Set Operations Explained:

- **UNION (A ∪ B)**: Combine Both Sets (Unique) - All unique values from both sets
- **INTERSECT (A ∩ B)**: Common Elements Only - Only values in BOTH sets
- **EXCEPT (A - B)**: Set Difference - Values in A but NOT in B

Visual Representation:

- **UNION**: All from A and B (no duplicates)
- **INTERSECT**: Only common elements
- **EXCEPT**: A minus B (left only)

Subqueries in FROM Clause (Question 26)

Five examples using derived tables (subqueries in FROM):

1. Room reservation statistics

```
SELECT RoomNo, num_reservations
FROM (
    SELECT RoomNo, COUNT(*) AS num_reservations
    FROM Reservation
    GROUP BY RoomNo
) AS room_counts;
```

2. Instructor reservation totals

```
SELECT Instructor_ID, total_reservations
FROM (
    SELECT Instructor_ID, COUNT(*) AS total_reservations
    FROM Reservation
    GROUP BY Instructor_ID
) AS inst_res;
```

Pattern: All five examples follow the same structure:

- Inner query performs aggregation
- Outer query selects from the aggregated result
- Alias (AS) is required for the derived table

Derived Table Process:

1. Main Query → FROM Clause
2. Subquery (Derived Table): SELECT RoomNo, COUNT(*) as num_reservations FROM Reservation GROUP BY RoomNo → Temporary Result: 'Virtual Table'
3. Alias: AS room_counts
4. Outer SELECT queries the derived table: SELECT RoomNo, num_reservations FROM room_counts
5. Final Result

Key Concept:
A subquery in the FROM clause creates a **temporary result set** (derived table) that acts like a regular table for the outer query.

Requirements:

- Must have an alias (AS table_name)
- Can be treated like any other table in the outer query
- Useful for multi-step aggregations

Custom SQL Functions

The `USING_SQL_FUNCTIONS.sql` file contains three custom PostgreSQL functions demonstrating different return types and complexity levels.

Function 1: BigCapacityRooms

Purpose: Returns all rooms with capacity greater than a specified value.

```
CREATE FUNCTION BigCapacityRooms(capa INTEGER)
RETURNS TABLE (roomno VARCHAR, capacity INT) AS $
    SELECT roomno, capacity FROM room WHERE capacity > $1
$ LANGUAGE SQL;
```

Technical Details:

- **Parameter:** `capa` (INTEGER) - minimum capacity threshold
- **Return Type:** TABLE with two columns
- **Language:** SQL (inline query)
- **Parameter Reference:** `$1` references the first parameter

Usage Example:

```
SELECT * FROM BigCapacityRooms(15);
```

Result: Returns rooms with capacity > 15 (room 301, Hall 1, Hall 2)

Function Execution Flow:

1. Function Call: BigCapacityRooms(15)
2. Function Receives Parameter: capa = 15
3. Execute Query: SELECT roomno, capacity FROM room WHERE capacity > $1
4. Substitute $1 with 15: WHERE capacity > 15
5. Filter Results → Matching Rooms: 301: 45, Hall 1: 500, Hall 2: 200
6. Return TABLE (roomno, capacity)

Function 2: Find_ID

Purpose: Returns the department ID given its name.

```
CREATE FUNCTION Find_ID(Name VARCHAR)
RETURNS INT AS $
    SELECT department_id FROM department WHERE name ILIKE $1;
$ LANGUAGE SQL;
```

Technical Details:

- **Parameter:** `Name` (VARCHAR) - department name
- **Return Type:** INT (single value)
- **Case Sensitivity:** Uses ILIKE for case-insensitive matching

Usage Example:

```
SELECT Find_ID('CCS');
```

Result: Returns 2 (the department ID for CCS)

Use Case: Useful for lookup operations when you need to convert user-friendly names to internal IDs.

Function 3: Reservation Conflict Detection

Purpose: Check if a room reservation is possible and identify conflicts.

This is implemented as two complementary functions:

CheckReservation (Boolean Check)

```
CREATE FUNCTION CheckReservation(
    Building VARCHAR,
    RoomNo VARCHAR,
    reservation_date DATE,
    start_t TIME,
    end_t TIME
) RETURNS BOOLEAN AS $
    SELECT NOT EXISTS (
        SELECT 1 FROM reservation
        WHERE building = $1
        AND roomno = $2
        AND reserv_date = $3
        AND NOT (
            end_time <= $4 OR start_time >= $5
        )
    );
$ LANGUAGE SQL;
```

Parameters:

1. `Building` - Building identifier
2. `RoomNo` - Room number
3. `reservation_date` - Date of reservation
4. `start_t` - Proposed start time
5. `end_t` - Proposed end time

Return Value:

- `TRUE` if reservation is possible (no conflicts)
- `FALSE` if conflicts exist

Conflict Detection Logic: A conflict exists when:

```
NOT (existing_end_time <= new_start_time OR existing_start_time >= new_end_time)
```

This means the times overlap if neither condition is true:

- Existing reservation doesn't end before new one starts
- Existing reservation doesn't start after new one ends

Time Conflict Scenarios:

- **Scenario 1: No Conflict** - Existing Reservation: 08:00-10:00, New Reservation (After): 11:00-13:00
- **Scenario 2: Overlap Conflict** - Existing Reservation: 08:00-12:00, New Reservation (Overlap): 10:00-14:00
- **Scenario 3: Full Conflict** - Existing Reservation: 08:00-17:00, New Reservation (Inside): 10:00-12:00

Conflict Detection Steps:

1. Check Reservation
2. Same Room? Same Date?
    - No → ✓ No Conflict (Different room/date)
    - Yes → Time Overlap?
        - Existing ends <= New starts?
            - Yes → ✓ No Conflict (Before)
            - No → Existing starts >= New ends?
                - Yes → ✓ No Conflict (After)
                - No → ✗ CONFLICT! (Times overlap)

Formula:

```
No Conflict IF:
  existing_end_time <= new_start_time  OR
  existing_start_time >= new_end_time

Conflict = NOT (No Conflict)
```

ReservationConflicts (Detailed Conflicts)

```
CREATE FUNCTION ReservationConflicts(
    Building VARCHAR,
    RoomNo VARCHAR,
    reservation_date DATE,
    start_t TIME,
    end_t TIME
) RETURNS TABLE (reservation_id INTEGER) AS $
    SELECT reservation_id
    FROM reservation
    WHERE building = $1
    AND roomno = $2
    AND reserv_date = $3
    AND NOT (
        end_time <= $4 OR start_time >= $5
    );
$ LANGUAGE SQL;
```

Purpose: Returns the specific reservation IDs that conflict with the proposed booking.

Usage Example:

```
SELECT ReservationConflicts('A','301','2006-09-24','13:45:00','17:00:00');
```

Difference from CheckReservation:

- `CheckReservation` → Simple yes/no answer
- `ReservationConflicts` → Detailed list of conflicting reservations

Practical Application: The boolean check is useful for quick validation, while the detailed function helps with user feedback ("Your reservation conflicts with reservations #9 and #15").

Function Comparison:

- **CheckReservation()**: Returns BOOLEAN (TRUE = Available, FALSE = Conflict)
- **ReservationConflicts()**: Returns TABLE (List of conflicting reservation IDs, Empty = No conflicts)

Example Usage:

- User Input: Room A-301, Date: 2006-09-24, Time: 13:45-17:00
- CheckReservation() Output: FALSE
- ReservationConflicts() Output: Reservation #9

Transaction Management

The `USING_TRANSACTION.sql` file demonstrates PostgreSQL transaction control with and without savepoints, ensuring data consistency through atomic operations.

Transaction Fundamentals

ACID Properties:

- **Atomicity:** All operations succeed or all fail
- **Consistency:** Database moves from one valid state to another
- **Isolation:** Concurrent transactions don't interfere
- **Durability:** Committed changes persist

Transaction States:

- **Idle** → BEGIN → **Active** (Execute Operations)
- **Active** → COMMIT → **Committed** (All changes saved, Permanent and durable)
- **Active** → ROLLBACK → **RolledBack** (All changes discarded, Database unchanged)

ACID Properties Breakdown:

- **Atomicity**: All operations succeed OR all fail; No partial completion

- **Consistency**: Valid state before → Valid state after
- **Isolation**: Transaction A and Transaction B don't interfere with each other
- **Durability**: COMMIT → Changes persist even after crash

Example 1: Simple Transaction (Without Savepoints)

```
BEGIN;

INSERT INTO Instructor VALUES
(7, 3, 'Chouaib', 'Reffas', 'MCA', NULL, NULL, 'CR@yahoo.en');

INSERT INTO Instructor VALUES
(8, 3, 'Sadek', 'Fehis', 'MCA', '4185', '4091', 'FE@yahoo.en');

COMMIT;
```

Execution Flow:

1. `BEGIN` : Starts transaction
2. Two INSERT operations executed
3. `COMMIT` : Permanently saves both records

Atomicity Guarantee: If the second INSERT fails (e.g., constraint violation), neither instructor is added to the database.

Alternative Outcomes:

- If `COMMIT` → Both instructors are added
- If `ROLLBACK` → Both insertions are cancelled
- If error occurs → Automatic rollback

Transaction Flow:

1. BEGIN
2. INSERT instructor #7 → Operation1
3. INSERT instructor #8 → Operation2
4. Decision:
    - Success → COMMIT → Committed (Both instructors permanently added)
    - Error/Cancel → ROLLBACK → RolledBack (Neither instructor added)

Note: Temporary changes not yet visible to other transactions during Active state

---

Example 2: Transaction with Savepoints

```
BEGIN;

UPDATE instructor SET department_id=2 WHERE instructor_id=7;

SAVEPOINT step1;

UPDATE instructor SET department_id=2 WHERE instructor_id=8;

ROLLBACK TO step1;

COMMIT;
```

Execution Flow:

1. **BEGIN:** Start transaction
2. **First UPDATE:** Move instructor #7 to department 2
3. **SAVEPOINT step1:** Create restoration point
4. **Second UPDATE:** Move instructor #8 to department 2
5. **ROLLBACK TO step1:** Undo only the second UPDATE
6. **COMMIT:** Save the transaction state (only first UPDATE is committed)

Final Result:

- Instructor #7 → Department 2 ✓
- Instructor #8 → Unchanged (rollback to savepoint)

Transaction with Savepoints Flow:

1. BEGIN
2. UPDATE instructor #7 to dept 2 (Update1)
3. Create savepoint (SAVEPOINT_step1) - Restoration point created, Can rollback to here

4. UPDATE instructor #8 to dept 2 (Update2)
5. ROLLBACK TO step1 - Update2 is undone, Update1 is preserved
6. At Savepoint: Instructor #8 unchanged, Instructor #7 still updated
7. COMMIT → Transaction Complete (Only Update1 persists)

Savepoint Workflow:

1. BEGIN Transaction
2. Operation 1 (Kept)
3. SAVEPOINT sp1
4. Operation 2 (Will be undone)
5. Problem with Operation 2?
    - Yes → ROLLBACK TO sp1 → Operation 2 undone, Operation 1 intact → COMMIT
    - No → Continue... → COMMIT
6. Transaction Complete

Savepoint Use Cases

When to use savepoints:

1. **Batch operations with partial rollback needs:** Process 100 records but undo last 10 if issues arise
2. **Try-catch patterns:** Attempt risky operation, rollback if it fails, continue transaction
3. **Complex multi-step operations:** Allow selective undo without canceling entire transaction
4. **Error recovery:** Handle errors gracefully within a transaction

Nested Savepoints Example (Conceptual):

```
BEGIN;
  INSERT INTO table1 VALUES (...);
  SAVEPOINT sp1;
    UPDATE table2 SET ...;
    SAVEPOINT sp2;
      DELETE FROM table3 WHERE ...;
    ROLLBACK TO sp2;  -- Undo DELETE only
  -- UPDATE is still intact
COMMIT;
```

Nested Savepoint Levels:

- Transaction BEGIN → SAVEPOINT sp1 → SAVEPOINT sp2 → SAVEPOINT sp3

Rollback Targets:

- ROLLBACK TO sp1: Undoes sp2, sp3, and all after sp1
- ROLLBACK TO sp2: Undoes sp3 and all after sp2
- ROLLBACK TO sp3: Undoes only after sp3

Trigger Implementation

The `USING_TRIGGERS.sql` file implements a statement-level trigger for auditing DML operations on the Student table.

Audit Table Structure

```
CREATE TABLE Student_Audit_Log (
    LogID SERIAL PRIMARY KEY,
    OperationType VARCHAR(50) NOT NULL,
    OperationTime TIMESTAMP NOT NULL,
    Description TEXT
);
```

Purpose: Records all modifications to the Student table for compliance, debugging, or data lineage tracking.

Attributes:

- `LogID` : Auto-incrementing unique identifier
- `OperationType` : Type of operation (INSERT, UPDATE, DELETE)
- `OperationTime` : Timestamp of the operation
- `Description` : Human-readable description

Trigger Function

```
CREATE OR REPLACE FUNCTION audit_student_changes_statement()
RETURNS TRIGGER AS $
    DECLARE msg_description TEXT;
    DECLARE time TIMESTAMP;
    BEGIN
        msg_description := 'A statement-level DML operation ' || TG_OP ||
                           ' occurred on Students table.';
        time := CURRENT_TIMESTAMP;

        INSERT INTO student_audit_log
        (operationtype, operationtime, description)
        VALUES (TG_OP::VARCHAR, time::TIMESTAMP(0), msg_description);

        RETURN NULL;
    END;
$ LANGUAGE plpgsql;
```

Technical Components:

1. **Return Type:** `RETURNS TRIGGER` indicates this is a trigger function
2. **Language:** `plpgsql` (procedural language) supports variables and control structures
3. **Special Variables:**
    - `TG_OP` : Contains operation type ('INSERT', 'UPDATE', or 'DELETE')
4. **Type Casting:** `::VARCHAR` and `::TIMESTAMP(0)` for explicit type conversion
5. **RETURN NULL:** Required for AFTER triggers at statement level

Trigger Execution Flow:

1. DML Operation on Student (INSERT/UPDATE/DELETE)
2. Operation Completes
3. Trigger Fires: trg_audit_students_statement
4. Execute Function: audit_student_changes_statement()
5. Function Execution:
    - Declare Variables: msg_description, time
    - Build message using TG_OP (INSERT/UPDATE/DELETE)
    - Get CURRENT_TIMESTAMP
    - INSERT into student_audit_log
6. Return NULL (Required for AFTER trigger)
7. Audit Log Updated
8. Transaction Continues

Trigger Definition

```
CREATE TRIGGER trg_audit_students_statement
    AFTER INSERT OR UPDATE OR DELETE ON Student
    FOR EACH STATEMENT
    EXECUTE FUNCTION audit_student_changes_statement();
```

Configuration:

- **Timing:** `AFTER` - fires after the DML operation completes
- **Events:** `INSERT OR UPDATE OR DELETE` - monitors all data changes
- **Level:** `FOR EACH STATEMENT` - fires once per SQL statement (not per row)
- **Action:** Executes the audit function

Statement-Level vs. Row-Level:

- **Statement-level:** One trigger execution per statement (even if 1000 rows affected)
- **Row-level:** Separate trigger execution for each affected row

Comparison:

- **Statement-Level Trigger**: UPDATE affects 5 rows → Trigger fires ONCE → One audit log entry
- **Row-Level Trigger (Hypothetical)**: UPDATE affects 5 rows → Trigger fires 5 TIMES → Five audit log entries

Comparison Table:

Aspect	Statement-Level	Row-Level
Trigger keyword	`FOR EACH STATEMENT`	`FOR EACH ROW`
Execution frequency	Once per SQL statement	Once per affected row

Aspect	Statement-Level	Row-Level
Access to row data	No (OLD/NEW unavailable)	Yes (OLD/NEW available)
Performance	Faster for bulk operations	Slower for bulk operations
Use case	General audit logging	Detailed row-level tracking

Practical Demonstration

Test Operations:

```
UPDATE student SET city = 'Annaba' WHERE student_id=1;

INSERT INTO student VALUES
(6,'Chouaib','Reffas','2007-02-23','14, did','Annaba','23000',
 '0657633996',NULL,'ref@gmail.com');

DELETE FROM student WHERE student_id=2;
```

Expected Audit Log Entries:

LogID	OperationType	OperationTime	Description
1	UPDATE	2025-12-20 14:30:15	A statement-level DML operation UPDATE occurred on Students table.
2	INSERT	2025-12-20 14:30:45	A statement-level DML operation INSERT occurred on Students table.
3	DELETE	2025-12-20 14:31:10	A statement-level DML operation DELETE occurred on Students table.

Sequence of Events:

1. User → Application: Execute UPDATE statement
2. Application → Database: UPDATE student SET city='Annaba'
3. Database → Student Table: Update 1 row (ID=1)
4. Student Table → Trigger System: Operation Complete (Statement-level trigger fires)
5. Trigger System executes audit_student_changes_statement()
6. TG_OP = 'UPDATE', Build description message, Get CURRENT_TIMESTAMP
7. Trigger System → Audit Log Table: INSERT audit entry
8. Audit Log Table → Trigger System: Confirmation
9. Trigger System → Database: RETURN NULL
10. Database → Application: UPDATE successful
11. Application → User: Confirmation

Audit Log Entry Created:

- Type: UPDATE
- Time: 21:30:15
- Description: "DML operation UPDATE occurred"

Trigger Use Cases in Education Systems

1. **Compliance Auditing:** Track who modified student records and when
2. **Data Recovery:** Reconstruct historical data states
3. **Change Notification:** Alert administrators of critical changes
4. **Business Rule Enforcement:** Validate complex constraints across tables
5. **Automated Workflow:** Trigger downstream processes when data changes

Advanced Trigger Patterns (Not Implemented Here):

- Row-level triggers for detailed per-record auditing
- BEFORE triggers for validation and data transformation
- Conditional triggers (WHEN clause)
- Multiple triggers on same table (execution order matters)

Technical Considerations

Database Design Patterns

1. Composite Keys

The schema uses composite primary keys in several tables (Course, Room, Reservation FK references, Enrollment). This design:

Advantages:

- Natural relationship representation
- Reduces need for artificial surrogate keys
- Enforces business rules at the database level

Challenges:

- More complex JOIN conditions
- Larger index sizes
- Harder to reference in foreign keys

Composite Key Relationships:

- Course (Course_ID PK1, Department_ID PK2, name) referenced by Reservation
- Reservation (Reservation_ID PK, Course_ID FK1, Department_ID FK2, Reserv_Date)

Composite Key Explanation:

1. Course Table → Composite Primary Key → Course_ID + Department_ID
2. Why Composite Key?
    - Course #1 in Dept 1 = Databases
    - Course #1 in Dept 2 = Different Course
    - Same Course_ID, Different Meaning
3. Referencing Table: Reservation
4. Foreign Key Must Match → FK = (Course_ID, Department_ID) → BOTH columns required

---

2. Constraint Strategy

Types Used:

- **PRIMARY KEY:** Uniqueness and entity identification
- **FOREIGN KEY:** Referential integrity
- **UNIQUE:** Prevent duplicate business entities
- **CHECK:** Domain constraints (rank values, mark range, time validation)
- **NOT NULL:** Required fields

RESTRICT Policy: All foreign keys use `ON UPDATE RESTRICT` and `ON DELETE RESTRICT`, preventing:

- Orphaned records
- Cascading deletions
- Data inconsistencies

Alternative Policies (Not Used):

- `CASCADE` : Automatically propagate changes/deletions
- `SET NULL` : Set FK to NULL on parent deletion
- `SET DEFAULT` : Use default value

Referential Actions:

1. Parent Table Operation (UPDATE/DELETE) → Referential Action?
    - **RESTRICT**: ✗ Prevent operation if children exist
    - **CASCADE**: Propagate changes to child rows
    - **SET NULL**: ∅ Set FK to NULL in child rows
    - **SET DEFAULT**: Set FK to DEFAULT in child rows

Decision Tree for Choosing Referential Action:

1. Choose Referential Integrity Action
2. Should deleting parent delete children?
    - Yes → Use CASCADE (Example: Delete order → delete order items)
    - No → Can children exist without parent?
        - No → Use RESTRICT (Example: Can't delete dept with instructors)
        - Yes → Should FK become NULL?
            - Yes → Use SET NULL (Example: Delete manager → employees.manager_id = NULL)
            - No → Use SET DEFAULT (Example: Assign to default department)

---

3. Data Type Choices

SERIAL vs. INTEGER:

- `Mark_ID SERIAL` : Auto-incrementing for system-generated IDs
- Other IDs use `INTEGER` : Allows manual control or external ID assignment

VARCHAR Lengths:

- Strategic sizing based on data characteristics

- `name VARCHAR(25)` : Personal names
- `email VARCHAR(100)` : Email addresses
- `description VARCHAR(1000)` : Extended text

NUMERIC for Financial/Grades:

- `Mark_Value NUMERIC(4,2)` : Precise decimal storage (up to 99.99)
- Avoids floating-point rounding errors

TIME and DATE Separation:

- Better for queries targeting specific time ranges
- Allows independent validation (start < end time)

Normalization Analysis

The schema appears to be in **3NF (Third Normal Form)**:

1NF (First Normal Form):

- ✓ All attributes are atomic (no repeating groups)
- ✓ Each table has a primary key

2NF (Second Normal Form):

- ✓ No partial dependencies (all non-key attributes depend on full PK)
- Example: In Enrollment, `Enrollment_Date` depends on the full composite key `(Student_ID, Course_ID, Department_ID)`, not just part of it

3NF (Third Normal Form):

- ✓ No transitive dependencies
- All non-key attributes depend directly on PK

Potential Denormalization (If Performance Needed):

- Store instructor name in Reservation (avoid join for reports)
- Cache course count in Department table
- Trade-off: Faster reads vs. update complexity

Normalization Progression:

1. Unnormalized Data (✗ Repeating groups, ✗ No primary key)
2. 1NF: First Normal Form (✓ Atomic values only, ✓ Primary key defined, ✓ No repeating groups)
3. 2NF: Second Normal Form (✓ All of 1NF, ✓ No partial dependencies, ✓ Non-key attributes depend on FULL PK)
4. 3NF: Third Normal Form (✓ All of 2NF, ✓ No transitive dependencies, ✓ Non-key attributes depend ONLY on PK)
5. BCNF: Boyce-Codd NF (✓ All of 3NF, ✓ Every determinant is a candidate key)

Normalization Example:

- **Violation**: Student Table with Name and CourseList = 'John, Math,Physics,Chem'
- **1NF Fixed**:
  - Student Table: Name: John, Course: Math
  - Student Table: Name: John, Course: Physics

Query Optimization Insights

Indexing Recommendations:

```
 -- Frequently joined columns
CREATE INDEX idx_course_dept ON Course(Department_ID);
CREATE INDEX idx_instructor_dept ON Instructor(Department_ID);
CREATE INDEX idx_reservation_date ON Reservation(Reserv_Date);

-- Composite index for reservation lookups
CREATE INDEX idx_reservation_room_date
ON Reservation(Building, RoomNo, Reserv_Date);
```

Query Performance Tips:

1. **Use EXISTS instead of IN for large subqueries** - stops when match found
2. **LEFT JOIN vs. NOT IN** - Better NULL handling with LEFT JOIN
3. **LIMIT results** - Especially for large tables
4. **Avoid SELECT *** - Specify only needed columns
5. **Use EXPLAIN ANALYZE** - Understand query execution plans

Index Performance:

- **Without Index**: Query finds student ID=42 → Sequential Scan → Check row 1 ✗ → Check row 2 ✗ → Check row 3 ✗ → ... → Check row 42 ✓ → Result: Found after 42 checks
- **With Index on Student_ID**: Query finds student ID=42 → Index Lookup → B-Tree traversal (logarithmic time) → Direct jump to row 42 ✓ → Result: Found in ~3

checks ($\log_2 42 \approx 5.4$)

Performance Impact:

- **Without Index**: O(n) - Linear scan through all rows; 1000 rows checked - SLOW
- **With Index**: O(log n) - Logarithmic search; ~10 rows checked - FAST
- **Speed Improvement**: ~100x faster for 1000 rows

PostgreSQL-Specific Features Used

1. **SERIAL datatype** - Auto-increment
2. **ILIKE operator** - Case-insensitive matching
3. **OFFSET in subquery** - Skip rows in result set
4. **$1, $2 parameters** - Function parameter references
5. **PL/pgSQL** - Procedural language for triggers
6. **TG_OP variable** - Trigger operation type
7. **DATE and TIME types** - Separate temporal components

Best Practices

- Use transactions for multi-step operations
- Validate data before insertion
- Handle NULL values explicitly
- Test constraint violations
- Use savepoints for complex operations

Security Considerations

Not Implemented (But Recommended):

1. **User Roles and Permissions:**

```
CREATE ROLE student_user;
GRANT SELECT ON Student, Course, Enrollment TO student_user;
GRANT INSERT, UPDATE ON Enrollment TO student_user;
```

2. **Row-Level Security:**
    - Students can only see their own records
    - Instructors can only access their courses
3. **Input Validation:**
    - Sanitize user inputs in application layer
    - Use parameterized queries (preventing SQL injection)
4. **Audit Triggers:**
    - Track who (user) performed operations
    - Currently only tracks what and when

Security Layers:

1. **Network Security**: Firewall, VPN/SSL
2. **Authentication**: Username/Password, 2FA/MFA, Certificate-based
3. **Authorization (RBAC)**: User Roles, GRANT/REVOKE, Role Hierarchy
4. **Row-Level Security**: Policies, Students see only their data
5. **Column Encryption**: Sensitive data encrypted, Email and Phone encrypted
6. **Audit Logging**: Track all operations, WHO, WHAT, WHEN
7. Protected Data in Database

Scalability Considerations

Current Design Limitations:

1. **No Partitioning:** Large tables (especially audit logs) could benefit from partitioning by date
2. **CASCADE Restrictions:** All foreign keys use RESTRICT, making bulk updates challenging

Scaling Strategies:

1. **Read Replicas:** Distribute read queries across multiple servers
2. **Table Partitioning:**

```
CREATE TABLE Student_Audit_Log_2024 PARTITION OF Student_Audit_Log
FOR VALUES FROM ('2024-01-01') TO ('2024-12-31');
```

3. **Archival Strategy:** Move old reservations to historical tables

Scaling Approaches:

1. Read Replicas (Horizontal Scaling):

- Master DB (Writes) → Replica 1 (Reads), Replica 2 (Reads), Replica 3 (Reads)
- Load Balancer distributes read traffic across replicas

2. Horizontal Partitioning (Sharding):

- Application → Hash/Range Function
- Routes to: Shard 1 (Users A-M), Shard 2 (Users N-Z), Shard 3 (Archive)

Scaling Decision Matrix:

Strategy	Pros	Cons	When to Use
Read Replicas	Scales reads, High availability	Complex writes, Replication lag	Read-heavy workloads
Sharding	Unlimited scale	Very complex, Cross-shard queries hard	Very large datasets
Partitioning	Simple queries, Easy maintenance	Limited scalability	Time-based data
Caching	Fastest reads	Stale data risk	Frequently accessed data

---

Conclusion

The schema demonstrates:

- ✓ **Proper relational design** with normalized tables
- ✓ **Data integrity** through comprehensive constraints
- ✓ **Flexibility** with optional contact information
- ✓ **Extensibility** through views and custom functions
- ✓ **Auditability** via trigger-based logging
- ✓ **Transaction safety** with ACID compliance

Key Strengths:

- Clear separation of concerns across entities
- Strong referential integrity
- Flexible querying capabilities
- Automated auditing mechanisms

University Database ER Diagram (Simplified Chen Notation)

Entity Relationships:

- Department offers Course (1:M)
- Department employs Instructor (1:M)
- Student enrolls via Enrollment (M:N)
- Student receives Mark (1:M)
- Course awarded for Mark (1:M)
- Course scheduled as Reservation (1:M)
- Instructor teaches Reservation (1:M)
- Room hosts Reservation (1:M)

Entity Attributes:

Department:

- Department_id (INTEGER, PK)
- name (VARCHAR)

Student:

- Student_ID (INTEGER, PK)
- Last_Name (VARCHAR)
- First_Name (VARCHAR)
- DOB (DATE)
- Address (VARCHAR)
- City (VARCHAR)
- Zip_Code (VARCHAR)
- Phone (VARCHAR)
- Fax (VARCHAR)
- Email (VARCHAR)

Course:

- Course_ID (INTEGER, PK)
- Department_ID (INTEGER, FK)
- name (VARCHAR)
- Description (VARCHAR)

Instructor:

- Instructor_ID (INTEGER, PK)
- Department_ID (INTEGER, FK)
- Last_Name (VARCHAR)
- First_Name (VARCHAR)
- Rank (VARCHAR)
- Phone (VARCHAR)
- Fax (VARCHAR)
- Email (VARCHAR)

Room:

- Building (VARCHAR, PK)
- RoomNo (VARCHAR, PK)
- Capacity (INTEGER)

Reservation:

- Reservation_ID (INTEGER, PK)
- Building (VARCHAR, FK)
- RoomNo (VARCHAR, FK)
- Course_ID (INTEGER, FK)
- Department_ID (INTEGER, FK)
- Instructor_ID (INTEGER, FK)
- Reserv_Date (DATE)
- Start_Time (TIME)
- End_Time (TIME)
- Hours_Number (INTEGER)

Enrollment:

- Student_ID (INTEGER, FK)
- Course_ID (INTEGER, FK)
- Department_ID (INTEGER, FK)
- Enrollment_Date (DATE, PK)

Mark:

- Mark_ID (SERIAL, PK)
- Student_ID (INTEGER, FK)
- Course_ID (INTEGER, FK)
- Department_ID (INTEGER, FK)
- Mark_Value (NUMERIC)
- Mark_Date (DATE)

Appendix: Quick Reference

Table Relationship Summary:

- Department → Instructor (1:N)
- Department → Course (1:N)
- Instructor → Reservation (1:N)
- Course → Reservation (1:N)
- Room → Reservation (1:N)
- Student ↔ Course (N:M via Enrollment)
- Student → Mark (1:N)
- Course → Mark (1:N)

Constraint Count:

- Primary Keys: 8
- Foreign Keys: 9
- Unique Constraints: 1
- Check Constraints: 5

Function Summary:

- `BigCapacityRooms(capacity)` : Filter rooms by capacity
- `Find_ID(name)` : Get department ID from name
- `CheckReservation(...)` : Check if reservation possible
- `ReservationConflicts(...)` : List conflicting reservations

Trigger Summary:

- `trg_audit_students_statement` : Audit all Student table changes

Table Relationships (Simplified):

- Course (1:N) with Department
- Instructor (1:N) with Department
- Enrollment (M:N) between Student and Course

- Mark (1:N) with Student
- Mark (1:N) with Course
- Reservation (1:N) with Instructor
- Reservation (1:N) with Course
- Reservation (1:N) with Room

Entity Details:

Department:

- id (PK)

Student:

- id (PK)

Instructor:

- id (PK)
- dept_id (FK)

Course:

- id (PK)
- dept_id (FK)

Room:

- building (PK)
- roomno (PK)

Reservation:

- id (PK)
- course_id (FK)
- instructor_id (FK)
- building (FK)
- roomno (FK)

Enrollment:

- student_id (FK)
- course_id (FK)
- dept_id (FK)

Mark:

- id (PK)
- student_id (FK)
- course_id (FK)

Document Version: 1.0
Last Updated: December 21, 2025
Database System: PostgreSQL
