Xi'an Jiaotong-Liverpool University
西交利物浦大學

DTS207TC - Database Development and Design: Lab 2 Intermediate SQL

Course Leader: Dr.Zhang Di, Co-Teacher: Dr. Affan Yasin

September 16, 2025

Contents

Instructions

1. First create the database, tables, and import the data, or run the queries given to create and import the data.
2. The table creation and data import are already performed in the first week. If you have issues, you can contact the teacher in the lab or inother case generate your own data and run the queries for revision.

1 Part A - Nested Queries (Subqueries)

A nested query, or subquery, is a SQL query placed inside another query. It allows you to use the result of one query as input for another. Nested queries are useful for filtering, comparing, or aggregating data in complex scenarios where a single query is not sufficient.

Objectives

By the end of this part, students will be able to:

- Understand what **nested queries** (subqueries) are.
- Use a query inside another query to filter or calculate data.
- Apply nested queries with `WHERE`, `IN`, and aggregate functions.

1.1 Nested Queries

1.1.1 Find students older than the average age

```
SELECT name, age
FROM students
WHERE age > (SELECT AVG(age) FROM students);
```

The subquery ‘(SELECT AVG(age) FROM students)‘ calculates the average age. The outer query lists students older than that average.

1.1.2 List students enrolled in the course ’Databases’

```
SELECT name
FROM students
WHERE student_id IN (
    SELECT student_id
    FROM enrollments
    WHERE course_id = (
        SELECT course_id
        FROM courses
        WHERE course_name = 'Databases'
    )
);
```

The innermost subquery finds the course ID for ”Databases”. The middle query finds students enrolled in that course. The outer query displays their names.

1.1.3 Find courses with more credits than the average course credits

```
SELECT course_name, credits
FROM courses
WHERE credits > (SELECT AVG(credits) FROM courses);
```

The subquery computes the average course credits. The outer query lists courses with higher credits.

1.1.4 Show students who got grade 'A' in at least one course

```
SELECT name
FROM students
WHERE student_id IN (
    SELECT student_id
    FROM enrollments
    WHERE grade = 'A'
);
```

The subquery finds all students who scored 'A'. The outer query shows their names.

1.1.5 Finds students enrolled in course 101.

```
SELECT name, age
FROM students
WHERE student_id IN (
    SELECT student_id
    FROM enrollments
    WHERE course_id = 101
);
```

1.1.6 Finds students older than the average age.

```
SELECT name, age
FROM students
WHERE age > (
    SELECT AVG(age)
    FROM students
);
```

1.2 Practice / Fill-in-the-Blanks

1. Find students enrolled in course 102:

```
SELECT name
FROM students
WHERE student_id ___ (
    SELECT student_id
    FROM enrollments
```

```
        WHERE course_id = 102
    );
```

2. List courses with more than 2 students:

```
    SELECT course_name
    FROM courses
    WHERE course_id IN (
        SELECT course_id
        FROM enrollments
        GROUP BY course_id
        HAVING COUNT(*) > ___
    );
```

3. Display students older than the average age:

```
    SELECT name, age
    FROM students
    WHERE age > (
        SELECT ___
        FROM students
    );
```

4. Find the youngest student:

```
    SELECT name, age
    FROM students
    WHERE age = (SELECT ___(age) FROM students);
```

5. Students not enrolled in course 102:

```
SELECT name
FROM students
WHERE student_id NOT IN (
    SELECT student_id
    FROM enrollments
    WHERE course_id = ___
);
```

XJTLU | SCHOOL OF AI AND ADVANCED COMPUTING

2 Part B - Joining Tables

Joining tables in SQL allows us to combine data from two or more related tables based on a common column. It helps retrieve meaningful information that cannot be obtained from a single table alone. Common types of joins include INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN, each controlling which rows are included in the results.

Objectives

By the end of this part, students will be able to:

- Understand the concept of joining tables in relational databases.
- Learn different types of joins: `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `FULL OUTER JOIN`.
- Retrieve related data from multiple tables in a single query.

2.1 Example of Different Types of JOINs

We will use two example tables:

Students

student_id	name	dept_id
1	Alice	10
2	Bob	20
3	Carol	30
4	David	40
5	Emma	50

Departments

dept_id	dept_name
10	Computer Sci
20	Data Science
30	AI
60	Math
70	Physics

2.1.1 INNER JOIN

Definition: Returns only the rows where there is a match in both tables.
Effect: Records without a match are excluded.
Example: Students whose `dept_id` exists in the Departments table.
Result: Intersection of both tables.

```
SELECT s.name, d.dept_name
FROM Students s
INNER JOIN Departments d
ON s.dept_id = d.dept_id;
```

name	dept_name
Alice	Computer Sci
Bob	Data Science
Carol	AI

2.1.2 LEFT JOIN (LEFT OUTER JOIN)

Definition: Returns all rows from the left table, and the matching rows from the right table.
Effect: If no match is found, values from the right table will be `NULL`.
Example: All students will be shown, but if a student's `dept_id` is not found in Departments, the `dept_name` will be `NULL`.

Result: Everything from the left table + matches from the right.

```
SELECT s.name, d.dept_name
FROM Students s
LEFT JOIN Departments d
ON s.dept_id = d.dept_id;
```

name	dept_name
Alice	Computer Sci
Bob	Data Science
Carol	AI
David	NULL
Emma	NULL

2.1.3 RIGHT JOIN (RIGHT OUTER JOIN)

Definition: Returns all rows from the right table, and the matching rows from the left table.
Effect: If no match is found, values from the left table will be `NULL`.
Example: All departments will be shown, but if no student belongs to a department, the student name will be `NULL`.
Result: Everything from the right table + matches from the left.

```
SELECT s.name, d.dept_name
FROM Students s
RIGHT JOIN Departments d
ON s.dept_id = d.dept_id;
```

name	dept_name
Alice	Computer Sci
Bob	Data Science
Carol	AI
NULL	Math
NULL	Physics

2.1.4 FULL OUTER JOIN

Definition: Returns all rows when there is a match in either left or right table.
Effect: If a row is missing on one side, the missing values are shown as `NULL`.
Example: Shows all students and all departments, even if they do not match.
Result: Union of both tables.

```
SELECT s.name, d.dept_name
FROM Students s
FULL OUTER JOIN Departments d
ON s.dept_id = d.dept_id;
```

name	dept_name
Alice	Computer Sci
Bob	Data Science
Carol	AI
David	NULL
Emma	NULL
NULL	Math
NULL	Physics

JOIN Type	Result Returned
INNER JOIN	Only matching rows (intersection)
LEFT JOIN	All rows from left + matches from right
RIGHT JOIN	All rows from right + matches from left
FULL OUTER JOIN	All rows from both, unmatched filled with NULL

2.2 More Examples to Understand

a) INNER JOIN

```
SELECT s.name, s.major, c.course_name, e.grade
FROM students s
INNER JOIN enrollments e ON s.student_id = e.student_id
INNER JOIN courses c ON e.course_id = c.course_id;
```

Displays students with their enrolled courses and grades. Only includes matching records.

b) LEFT JOIN

```
SELECT s.name, s.major, c.course_name, e.grade
FROM students s
LEFT JOIN enrollments e ON s.student_id = e.student_id
LEFT JOIN courses c ON e.course_id = c.course_id;
```

Displays all students, including those who are not enrolled in any course.

c) RIGHT JOIN

```
SELECT s.name, s.major, c.course_name, e.grade
FROM students s
RIGHT JOIN enrollments e ON s.student_id = e.student_id
RIGHT JOIN courses c ON e.course_id = c.course_id;
```

Displays all enrollments, including those without corresponding student information.

d) FULL OUTER JOIN

```
SELECT s.name, s.major, c.course_name, e.grade
FROM students s
FULL OUTER JOIN enrollments e ON s.student_id = e.student_id
FULL OUTER JOIN courses c ON e.course_id = c.course_id;
```

Displays all students and all enrollments, including unmatched records.

2.3 Practice / Fill-in-the-Blanks

1. Show students and their courses using INNER JOIN:

```
SELECT s.name, c.course_name
FROM students s
___ JOIN enrollments e ON s.student_id = e.student_id
___ JOIN courses c ON e.course_id = c.course_id;
```

2. List all students, including those not enrolled in any course (LEFT JOIN):

```
SELECT s.name, c.course_name
FROM students s
___ JOIN enrollments e ON s.student_id = e.student_id
___ JOIN courses c ON e.course_id = c.course_id;
```

3. Display all enrollments, even if student information is missing (RIGHT JOIN):

```
SELECT s.name, c.course_name, e.grade
FROM students s
___ JOIN enrollments e ON s.student_id = e.student_id;
```

2.4 Challenge / Practice Questions

1. Find students who are not enrolled in any course.
2. List courses with no students enrolled.
3. Display each student along with all courses, showing NULL for missing enrollments.

XJTLU | SCHOOL OF AI AND ADVANCED COMPUTING

3 Part C - Views

A VIEW is a virtual table in SQL that is created using a query. It does not store data itself but displays data from one or more tables. Views simplify complex queries, improve readability, and can enhance security by restricting access to specific columns or rows.

Objectives

By the end of this part, students will be able to:
- Understand what a **view** is in SQL.
- Learn to create, query, and drop views.
- Use views to simplify complex queries and enhance security.

a) Creating a simple view

```
CREATE VIEW student_grades AS
SELECT s.name, c.course_name, e.grade
FROM students s
JOIN enrollments e ON s.student_id = e.student_id
JOIN courses c ON e.course_id = c.course_id;
```

Creates a view that shows student names, courses, and grades.

b) Querying a view

```
SELECT *
FROM student_grades
WHERE grade = 'A';
```

Retrieves students with grade 'A' from the view.

c) Dropping a view

```
DROP VIEW student_grades;
```

Deletes the view from the database.

3.1 Practice / Fill-in-the-Blanks

1. Create a view showing students and their majors:

```
CREATE VIEW ___ AS
SELECT name, major
FROM students;
```

2. Retrieve all students from the view where the major is 'AI':

```
SELECT *
FROM ___
WHERE major = 'AI';
```

3. Drop the view you created:

```
DROP VIEW ___;
```

3.2 Challenge / Practice Questions

1. Create a view that shows students with their courses and grades only for courses with more than 3 credits.
2. Query the view to find students who scored 'B' or higher.
3. Modify the view to include student ages.

4 Part D - Integrity Constraints

Integrity constraints are rules applied to database tables to ensure the accuracy, consistency, and reliability of data. They prevent invalid data entry and maintain meaningful relationships between tables. Common types include Primary Key (uniqueness of rows), Foreign Key (valid references between tables), NOT NULL (no empty values), UNIQUE (no duplicates in a column), and CHECK (custom conditions).

Objectives

By the end of this part, students will be able to:

- Understand what **integrity constraints** are in a database.
- Learn different types of constraints: `PRIMARY KEY`, `FOREIGN KEY`, `UNIQUE`, `NOT NULL`, `CHECK`, `DEFAULT`.
- Ensure data accuracy and consistency in tables.

a) NOT NULL Constraint

```
CREATE TABLE students (
    student_id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    age INT,
    major VARCHAR(50)
);
```

Ensures that the `name` column cannot have NULL values.

b) UNIQUE Constraint

```
CREATE TABLE courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(100) UNIQUE,
    credits INT
);
```

Ensures that no two courses have the same course_name.

c) PRIMARY KEY and FOREIGN KEY

```
CREATE TABLE enrollments (
    enrollment_id SERIAL PRIMARY KEY,
    student_id INT REFERENCES students(student_id),
    course_id INT REFERENCES courses(course_id),
    grade CHAR(1)
);
```

Ensures each enrollment has a unique ID and references valid students and courses.

d) CHECK Constraint

```
CREATE TABLE students (
    student_id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    age INT CHECK (age >= 18)
);
```

Ensures that the `age` of students is at least 18.

e) DEFAULT Constraint

```
CREATE TABLE students (
    student_id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    major VARCHAR(50) DEFAULT 'Undeclared'
);
```

Provides a default value for `major` if none is specified.

4.1 Practice / Fill-in-the-Blanks

1. Create a table where `email` cannot be NULL:

```
CREATE TABLE students (
    student_id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    email ___
);
```

2. Ensure `course_name` is unique:

```
CREATE TABLE courses (
    course_id INT PRIMARY KEY,
    course_name ___,
    credits INT
);
```

3. Add a CHECK constraint so that `credits` are at least 1:

```
CREATE TABLE courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(100),
    credits INT ___
);
```

HINT: credits INT CHECK (credits >= 1)

4.2 Challenge / Practice Questions

1. Modify the `enrollments` table to ensure `grade` can only be 'A', 'B', 'C', 'D', or 'F'.
2. Add a default value of 'Math' for the `major` column in the `students` table.

3. Create a table where `student_id` is PRIMARY KEY, `email` is UNIQUE, and `age` is at least 18.

XJTLU | SCHOOL OF AI AND ADVANCED COMPUTING

5 Point to Ponder / Think : Execution of Join and Computational Cost ?

Join Execution Analysis

This code (uploaded on the Learning Mall) demonstrates a **join between the fact table** `big` **and dimension table** `small` using a function-based condition:

- The join condition is:

$$s.dept\_id = \texttt{get\_dept}(b.dept\_id)$$

- PostgreSQL performs a **sequential scan** on the entire `big` table, checking each row against the join condition.

Why this is computationally expensive:

1. **Sequential scan of many rows:** Every row of the `big` table must be processed to evaluate the function.
2. **Function evaluation per row:** Each row calls `get_dept()`, adding even a tiny delay multiplied by the number of rows.
3. **Nested loop join:** PostgreSQL may use a nested loop to combine `big` and `small`. For each row in `small`, the system scans matching rows in `big`, further increasing computation.
4. **High I/O and memory usage:** Large tables and repeated function calls increase buffer usage and CPU cycles.

Key Takeaways:

- Joins are fast when indexes can be used, but function-based conditions can make them expensive.
- Pre-filtering large tables before joins or avoiding non-sargable functions in join conditions improves performance.

5.1 Explanation of Code shared on Learning Mall - Output

Query Analysis of Output on our system

1. Buffers:
- `Buffers: shared hit=35` → PostgreSQL read 35 shared buffers from memory during execution.

2. Index Only Scan on `small`:
- Used the primary key index on `small` to find `dept_id = 123`.
- **Index Only Scan** means it could mostly read data from the index without touching the table heap.
- `Heap Fetches: 1` → only one table row needed to be fetched.
- Very fast: actual time `0.009..0.012 ms`.

3. Sequential Scan on `big`:
- PostgreSQL scanned the whole `big` table to find rows where `get_dept(dept_id) = 123`.
- `Rows Removed by Filter: 999` → almost all rows filtered out.
- Slow (actual time `6216..12873 ms`) because:
  1. It is a sequential scan (not using an index)
  2. The filter uses the `get_dept()` function, preventing index use.

4. Planning and Execution Time:
- Planning: `0.079 ms` (almost instantaneous)
- Execution: `12874 ms` → dominated by sequential scan on `big`.

Summary:
- Outer table `small` used an index → very fast.
- Inner table `big` performed sequential scan with a function filter → slow.
- Most rows in `big` were filtered, but PostgreSQL still scanned all of them.

Quote

"Behind every dataset lies a story waiting to be told with SQL."

XJTLU | SCHOOL OF AI AND ADVANCED COMPUTING