

DATABASE

LABORATORY GUIDE

Relational Database • SQL • SQLite • PostgreSQL • Python

Bogdan Nicușor BINDEA

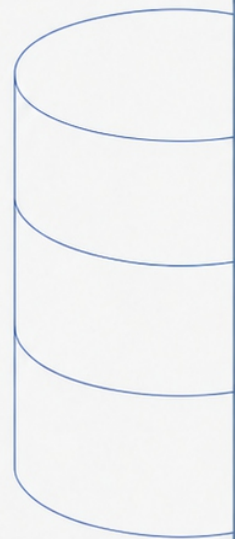
Călin Ovidiu CENAN



U.T.PRESS

Cluj-Napoca, 2026

ISBN 978-606-737-856-6



Bogdan Nicușor BINDEA

Călin Ovidiu CENAN

Database

Laboratory Guide



U.T. PRESS

Cluj-Napoca, 2026

ISBN 978-606-737-856-6



Editura U.T.PRESS
Str. Observatorului nr. 34
400775 Cluj-Napoca
Tel.: 0264-401.999
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Recenzia: Prof.dr.ing. Mihaela Dînșoreanu
Conf.dr.ing. Emil Chifu

Pregătire format electronic on-line: Gabriela Groza

Copyright © 2026 Editura U.T.PRESS
Reproducerea integrală sau parțială a textului sau ilustrațiilor din
această carte este posibilă numai cu acordul prealabil scris al editurii
U.T.PRESS.

ISBN 978-606-737-856-6

Laboratory no. 1

Introduction to SQL and Relational Databases

1.1. Objectives

The primary objective of this laboratory is to introduce students to the fundamental concepts of relational databases and Structured Query Language (SQL). By the end of the session, students should be able to:

- Understand the basic structure of relational databases, including tables, attributes, keys, and relationships.
- Create and populate database tables using SQL commands.
- Formulate and execute basic SQL queries to retrieve, filter, and manipulate data.
- Apply relational algebra concepts such as selection, projection, and joins through SQL statements.
- Gain hands-on experience with SQLite as a lightweight relational database management system.

1.2. Theoretical Foundation

1.2.1. Introduction to Databases

A database is an organized collection of data that can be easily accessed, managed, and updated. In most modern systems, databases are structured according to the relational model, which organizes data into tables (also called relations). Each table consists of rows (tuples) and columns (attributes).

For example, a database for a sailing club might include:

- A table of Sailors (each row representing one sailor)
- A table of Boats (each row representing one boat)
- A table of Reservations (each row representing a booking of a boat by a sailor on a given day)

This structure allows efficient storage, retrieval, and manipulation of information.

1.2.2. The relational model

The relational model was first introduced by E. F. Codd in 1970 in his landmark paper “A Relational Model of Data for Large Shared Data Banks”. His work provided the theoretical foundation for relational database management systems (RDBMS).

In the relational model:

- Tables (Relations): Organize data into rows and columns.
- Keys:
 - A primary key uniquely identifies each row (e.g., sid for Sailors).
 - A foreign key establishes relationships between tables (e.g., sid in Reservations references Sailors).
- Constraints: Rules that maintain data integrity (e.g., ensuring that no two sailors have the same sid).

Relational databases are widely used because they are flexible, scalable, and based on a solid mathematical foundation.

1.2.3. Structured Query Language (SQL)

To interact with relational databases, we use Structured Query Language (SQL). SQL was originally developed at IBM in the 1970s and has since become the standard language for relational databases, supported by systems such as Oracle, MySQL, PostgreSQL, SQL Server, and SQLite.

SQL consists of several components:

- Data Definition Language (DDL): Used to define and modify database structures (e.g., CREATE TABLE, ALTER TABLE).
- Data Manipulation Language (DML): Used to insert, update, delete, and query data (e.g., INSERT, UPDATE, DELETE, SELECT).
- Data Control Language (DCL): Used to manage permissions and access control (e.g., GRANT, REVOKE).
- Transaction Control Language (TCL): Used to manage transactions and ensure data consistency (e.g., COMMIT, ROLLBACK).

The most frequently used SQL command is SELECT, which retrieves information from one or more tables based on specified conditions.

1.2.4. Example Database Schema

In this laboratory, we will use a simple Sailing Database consisting of three tables:

- Sailors (sid, sname, rating, age)
- Boats (bid, bname, color)
- Reserves (sid, bid, day)

Here:

- sid is the primary key for Sailors.
- bid is the primary key for Boats.
- The primary key of Reserves is the combination of (sid, bid, day).
- Foreign keys ensure that reservations only refer to existing sailors and boats.

This schema allows us to explore fundamental SQL operations such as creating tables, inserting data, and running queries to answer questions like:

- Which sailors reserved a red boat?

- What boats were reserved by sailor “Lubber”?
- Which sailors have reserved both a red and a green boat?

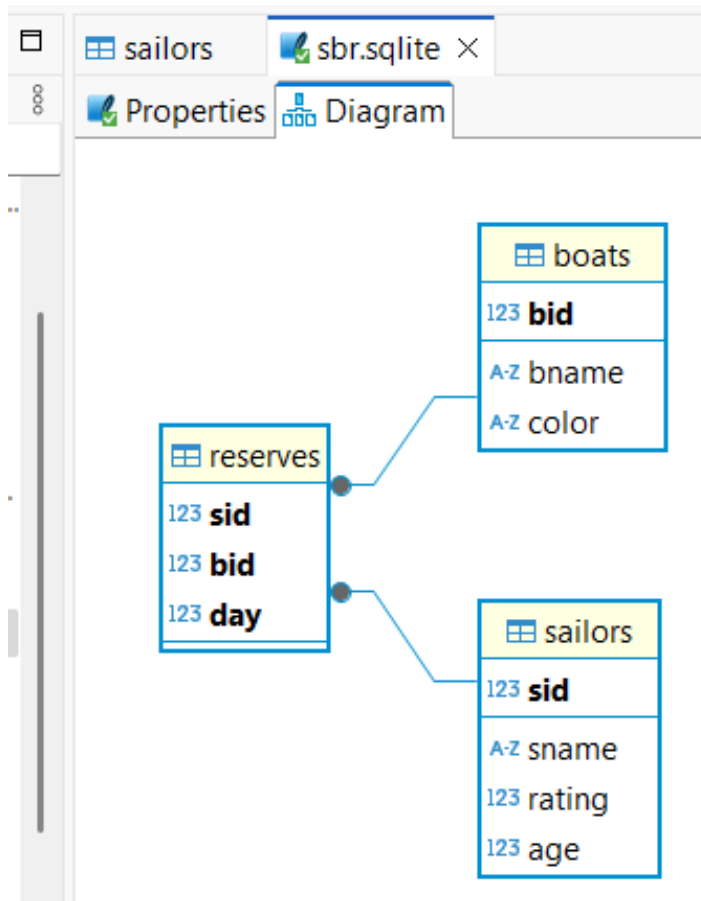


Fig. 1.1. Database Diagram

1.2.5. Why learn SQL?

SQL is essential because it:

- Provides a standardized way to interact with data across different systems.
- Allows both simple queries (e.g., listing all sailors) and complex analysis (e.g., finding sailors who reserved multiple boats on the same day).
- Bridges theory (relational algebra and set operations) with practical applications.
 - Is widely used in both academia and industry, making it a key skill for computer scientists, data analysts, and engineers.

1.3. Lab Exercises

1.3.1. Install the Tools

- Install DBeaver Community (GUI client)
 - Go to the official download page and get the installer for your OS (Windows/macOS/Linux). Then install with the default options.
- (Optional but useful) Install the SQLite command-line tools
 - Download the precompiled binaries for your OS from the official SQLite website and unzip them somewhere on your machine (e.g., C:\sqlite\ or /usr/local/bin).

Tip: You can do the entire lab inside DBeaver; the SQLite CLI is only if you want to test commands in a terminal.

1.3.2. Create a new SQLite Database in DBeaver

- Open DBeaver → Database → New Database Connection → search SQLite → Next.
- In the Database file field, click Browse... and choose where to create your database file (e.g., sailing.db).
- If prompted to download the SQLite driver, click Download. Then click Finish to open the connection.

1.3.3. Create the schema and load sample data

- Right-click your new SQLite connection → SQL Editor → New SQL Script.
- Copy the script below and run it (► Run script).

-- Create tables

```
CREATE TABLE sailors (
  sid    INTEGER NOT NULL PRIMARY KEY,
  sname  VARCHAR(50) NOT NULL,
  rating INTEGER NOT NULL,
  age    DOUBLE NOT NULL
);
```

```
CREATE TABLE boats (
  bid    INTEGER NOT NULL PRIMARY KEY,
  bname  VARCHAR(50) NOT NULL,
  color  VARCHAR(50) NOT NULL
);
```

```
CREATE TABLE reserves (
```

```

    sid INTEGER NOT NULL,
    bid INTEGER NOT NULL,
    day INTEGER NOT NULL,
    FOREIGN KEY (sid) REFERENCES sailors(sid),
    FOREIGN KEY (bid) REFERENCES boats(bid),
    PRIMARY KEY (sid, bid, day)
);

-- Seed data
INSERT INTO sailors (sid, sname, rating, age) VALUES
(22, 'Dustin', 7, 45.0), (29, 'Brutus', 1, 33.0), (31, 'Lubber', 8, 55.5),
(32, 'Andy', 8, 25.5), (58, 'Rusty', 10, 35.0), (64, 'Horatio', 7, 35.0),
(71, 'Zorba', 10, 16.0), (74, 'Horatio', 9, 35.0), (85, 'Art', 3, 25.5),
(95, 'Bob', 3, 63.5);

INSERT INTO boats (bid, bname, color) VALUES
(101, 'Interlake', 'blue'), (102, 'Interlake', 'red'),
(103, 'Clipper', 'green'), (104, 'Marine', 'red');

INSERT INTO reserves (sid, bid, day) VALUES
(22, 101, 1), (22, 102, 1), (22, 103, 2), (22, 104, 3),
(31, 102, 4), (31, 103, 5), (31, 104, 6),
(64, 101, 7), (64, 102, 8), (74, 103, 8);

```

- Verify: in the Database Navigator, expand Tables and check that sailors, boats, and reserves exist. You can also right-click a table → View data.
- 1.3.4. Your first query
- List all sailors (quick sanity check)

```
SELECT * FROM sailors ORDER BY sid;
```

- All sailors with rating > 7 (projection + selection):

```
SELECT S.sid, S.sname, S.rating, S.age  
FROM sailors AS S  
WHERE S.rating > 7  
ORDER BY S.sid;
```

- Names of sailors who reserved a red boat (join over 3 tables):

```
SELECT DISTINCT S.sname  
FROM boats B, reserves R, sailors S  
WHERE B.bid = R.bid AND S.sid = R.sid AND B.color = 'red'  
ORDER BY S.sname;
```

Note: Because we use DISTINCT on sname, if two different sailors share the same name (e.g., “Horatio”), the result will show the name only once. For precise identification, it is recommended to include the sailor ID (sid) or both sid and sname in the SELECT list.

- Sailors who reserved a red or a green boat (set logic with OR):

```
SELECT DISTINCT S.sname  
FROM boats B, reserves R, sailors S  
WHERE B.bid = R.bid AND S.sid = R.sid AND  
(B.color = 'red' OR B.color = 'green')  
ORDER BY S.sname;
```

- Sailors who reserved a red and a green boat

```
SELECT DISTINCT S1.sname  
FROM sailors S1
```

```

JOIN reserves R1 ON S1.sid = R1.sid
JOIN boats B1    ON R1.bid = B1.bid
WHERE B1.color = 'red'
INTERSECT
SELECT DISTINCT S2.sname
FROM sailors S2
JOIN reserves R2 ON S2.sid = R2.sid
JOIN boats B2    ON R2.bid = B2.bid
WHERE B2.color = 'green';

```

1.4. Troubleshooting

“Driver not found / No suitable driver” in DBeaver

When creating the SQLite connection, click Download if prompted to fetch the SQLite JDBC driver; DBeaver will install it for you.

Can't create/open the database file

Ensure the path you chose for sailing.db exists and you have write permissions. You can always pick a folder inside your user profile (e.g., Documents). (Connection wizard steps: Database → New Database Connection → SQLite → choose file → Finish.)

Prefer a terminal?

Use the official SQLite CLI (sqlite3 sailing.db) from the download page to run the same SQL statements.

1.5. Proposed Exercises

The following exercises are designed to reinforce the concepts introduced in Laboratory 1. You should solve them using the Sailing Database (sailors, boats, reserves) created during the lab session. Submit your SQL queries and, where possible, the corresponding results.

1.5.1. SELECT Basics

- Display all information about sailors.
- List only the names of all sailors.
- Retrieve the distinct ratings that exist in the sailors table.

- Show the name and age of each sailor.
- Display all the sailors, ordered by age (ascending).
- 1.5.2. Filtering with WHERE
 - Find all sailors whose age is less than 30.
 - List the sailors with a rating of 10.
 - Find all sailors with an age greater than or equal to 40.
 - Retrieve the sailors whose name is exactly 'Horatio'.
 - Find all sailors whose rating is between 5 and 8.
- 1.5.3. Pattern Matching (LIKE)
 - Find all sailors whose name begins with "B".
 - Display the sailors whose name ends with "o".
 - Find the sailors whose name contains the letter "r".
 - Retrieve the ages of sailors whose names are exactly 3 characters long.
- 1.5.4. First Steps with JOINS
 - List the names of sailors who have made a reservation (hint: join sailors and reserves).
 - Find the colors of boats reserved by Dustin.

Laboratory no. 2

Working with Nested Queries and Aggregation in SQL

2.1. Objectives

The objectives of this laboratory are to deepen students' understanding of SQL by introducing advanced query constructs and data summarization techniques. By the end of this session, students should be able to:

- Understand the concept of nested queries (subqueries) and how they can be used inside WHERE, FROM, and HAVING clauses.
- Differentiate between independent subqueries and correlated subqueries.
- Apply set comparison operators (IN, NOT IN, EXISTS, ANY, ALL) in practical scenarios.
- Use aggregate functions (COUNT, SUM, AVG, MIN, MAX) to summarize data.
- Write queries with GROUP BY and HAVING clauses to perform grouped analysis and apply conditions to groups.
- Combine theoretical relational algebra concepts with SQL implementations for real-world data analysis tasks.

2.2. Theoretical Foundation

2.2.1. Nested Queries (Subqueries)

A nested query, or subquery, is a query placed inside another SQL statement. Subqueries allow us to build conditions based on the results of other queries. They are commonly used in the WHERE clause, but can also appear in FROM or HAVING.

- Independent subquery: The inner query can be executed on its own (e.g., find all sailors who reserved boat 103).
- Correlated subquery: The inner query depends on a value from the outer query (e.g., for each sailor, check if they reserved a red boat).

Operators frequently used with subqueries:

- IN / NOT IN: check if a value belongs (or not) to a set.
- EXISTS / NOT EXISTS: check if a subquery returns any rows.
- ANY and ALL: compare values against one or all results from a subquery.

2.2.2. Aggregate Functions

SQL provides built-in functions to summarize data across multiple rows:

- COUNT: number of rows or distinct values.
- SUM: total of numeric values
- AVG: average value.

- MIN: smallest value.
- MAX: largest value.

Note: DISTINCT can be used with COUNT, SUM, and AVG, but not with MIN or MAX.

2.2.3. Grouping Data

Sometimes, we need results per group instead of the entire table. The GROUP BY clause divides rows into groups that share the same values in specified columns.

- Example: finding the average age of sailors per rating level.

The HAVING clause is applied after grouping, allowing us to filter groups based on conditions (e.g., show only rating levels with at least 2 sailors).

2.2.4. Relational Algebra Link

- Selection and projection extend into nested queries, where conditions can be based on intermediate results.
- Aggregation and grouping reflect set-based operations, such as partitioning a relation into subsets and computing summary values for each subset.

SQL therefore bridges theoretical relational algebra concepts with practical analysis, enabling both simple checks (e.g., sailors who reserved boat 103) and complex data summaries (e.g., average age per rating, restricted by group size).

2.3. Lab Exercises

Use the same Sailing Database (tables sailors, boats, reserves) and seed data from Lab 1. If needed, recreate the schema and inserts from your previous lab.

This lab focuses on nested queries, correlated subqueries, ANY/ALL, aggregates, GROUP BY, and HAVING.

2.3.1. Subqueries with IN / NOT IN

-- Ex. 1 Names of sailors who reserved boat 103 (subquery in WHERE + IN).

```
SELECT S.sname
FROM sailors AS S
WHERE S.sid IN (
    SELECT R.sid
    FROM reserves AS R
    WHERE R.bid = 103
)
ORDER BY S.sname;
```

-- With the sample data, the names are Dustin, Horatio, Lubber (two different Horatios exist).

-- Tip: using sid avoids ambiguity when names repeat.

-- Ex. 2 Sailors (by sid) who did not reserve boat 103 (NOT IN).

```
SELECT S.sid, S.sname
FROM sailors AS S
WHERE S.sid NOT IN (
    SELECT R.sid
    FROM reserves AS R
    WHERE R.bid = 103
)
```

ORDER BY S.sid;

-- Result shows the people (unique IDs) who never reserved 103 – not just names. This avoids the duplicate-name pitfall discussed in Lab 1.

2.3.2. Correlated subqueries with EXISTS / NOT EXISTS

-- Ex. 3 Names of sailors who reserved at least one red boat (correlated EXISTS).

```
SELECT S.sname
FROM sailors AS S
WHERE EXISTS (
    SELECT 1
    FROM reserves AS R
    JOIN boats AS B ON B.bid = R.bid
    WHERE R.sid = S.sid
    AND B.color = 'red'
)
```

ORDER BY S.sname;

-- With the seed data, this returns Dustin, Horatio (sid=64), Lubber. Correlated subquery: the inner query uses S.sid.

-- Ex. 4 Names of sailors who have not reserved any red boat (NOT EXISTS).

```
SELECT S.sname
FROM sailors AS S
WHERE NOT EXISTS (
    SELECT 1
    FROM reserves AS R
    JOIN boats AS B ON B.bid = R.bid
    WHERE R.sid = S.sid AND B.color = 'red'
)
ORDER BY S.sname;
```

2.3.3. SET comparisons: ANY / ALL

-- Ex. 5 Sailors whose rating is higher than any Horatio's rating (> ANY).

```
SELECT S.sid, S.sname, S.rating
FROM sailors AS S
WHERE S.rating > ANY (
    SELECT S2.rating FROM sailors AS S2 WHERE S2.sname =
    'Horatio'
)
ORDER BY S.rating DESC, S.sid;
```

-- ANY means “greater than at least one value in the subquery set.”

2.3.4. Aggregates, GROUP BY and HAVING

-- Ex. 6 Average age of all sailors (AVG).

```
SELECT AVG(S.age) AS avg_age
FROM sailors AS S;
```

-- With seed data, avg_age = 36.9. Remember: you can use DISTINCT with AVG, but it's not meaningful here.

-- Ex. 7 Name and age of the oldest sailor (aggregate in subquery).

```
SELECT S.sname, S.age
```

```

FROM sailors AS S
WHERE S.age = (
    SELECT MAX(S2.age) FROM sailors AS S2
);
-- Directly selecting S.sname, MAX(S.age) without GROUP BY is
illegal – use a subquery to compare against the scalar max.
-- Ex. 8 For each red boat, show the number of reservations
(GROUP BY).
SELECT B.bid, COUNT(*) AS reservation_count
FROM boats AS B
JOIN reserves AS R ON R.bid = B.bid
WHERE B.color = 'red'
GROUP BY B.bid
ORDER BY B.bid;
-- With sample data: bid=102 → 3, bid=104 → 2.
-- Ex. 9 Average age per rating for ratings that have at least
2 sailors (HAVING).
SELECT S.rating, AVG(S.age) AS avg_age
FROM sailors AS S
GROUP BY S.rating
HAVING COUNT(*) >= 2
ORDER BY S.rating;
-- With seed data (ratings 3, 7, 8, 10 qualify). HAVING filters
groups, not rows.

```

2.4. Troubleshooting

If grouping: every selected column must be aggregated or appear in the GROUP BY list; otherwise SQL can't determine which value to show per group.

Use EXISTS/NOT EXISTS for correlated checks (“does this sailor have...?”) and IN/NOT IN for set membership.

Prefer sid over sname when correctness matters (duplicate names).

2.5. Proposed Exercises

The following exercises are designed to strengthen your understanding of nested queries, set comparisons, and aggregate functions in SQL. Use the Sailing Database (sailors, boats, reserves) created in Lab 1 as your working dataset. Submit both the SQL queries and, where possible, their results.

2.5.1. Subqueries with IN / NOT IN

- List the names of sailors who reserved a green boat.
- Find the IDs and names of sailors who did not reserve any red boat.
- Show the sailors who reserved both boat 101 and boat 104 (use subqueries).

2.5.2. Correlated Subqueries (EXISTS / NOT EXISTS)

- Display the names of sailors who have made at least one reservation (join via subquery).
- List the sailors who have reserved every boat in the database.
- Find the sailors who reserved all red boats.
- Show the sailors who have never reserved a boat with color = 'green'.

2.5.3. Set Comparisons (ANY / ALL)

- Find sailors whose rating is higher than any Horatio's rating.
- Find sailors whose rating is higher than every Horatio's rating.
- Retrieve the oldest sailor(s) using $\geq$ ALL.
- Show the sailors whose age is less than all sailors with rating 7.

2.5.4. Aggregates and Grouping

- List the average age of sailors for each rating level.
- Find the youngest sailor for each rating.
- For each red boat, display how many times it was reserved.
- Show the average age of sailors 18+ per rating, only for ratings with at least two such sailors.
- Count how many distinct sailors exist per boat color.
- Find the rating level(s) with the highest average sailor age.

2.5.5. Mixed Exercises

- Find sailors older than the oldest sailor with rating 10 (two different solutions: subquery with MAX, and $>$ ALL).
- For each sailor, count how many different boats they have reserved. Return only sailors with at least 2 boats.
- List the sailors who reserved boats of at least two different colors.
- Find the sailors who reserved on the same day two different boats (hint: self-join on reserves).
- For each boat, show the number of distinct days it was reserved.

Laboratory no. 3

Working with NULL Values and Joins in SQL

3.1. Objectives

The main objective of this laboratory is to introduce students to handling missing values in SQL and to explore different types of joins for combining data across multiple tables. By the end of the session, students should be able to:

- Understand the role of NULL values in relational databases and how they affect comparisons, logic, and aggregate functions.
- Use the operators IS NULL and IS NOT NULL to identify and filter missing values.
- Recognize the behavior of NULLs in arithmetic operations and aggregation (COUNT, SUM, AVG, MIN, MAX).
- Write queries using different types of joins:
 - Inner Join (default join type).
 - Natural Join (automatic matching on common attributes).
 - Explicit Join with ON clause (custom join conditions).
 - Outer Joins (LEFT, RIGHT, FULL) to preserve unmatched rows.
- Distinguish between join conditions and filtering conditions for better readability and correctness.
- Apply joins and NULL handling to answer practical questions using the Sailing Database.
- Bridge relational algebra concepts with SQL implementations by understanding joins as extended selection and Cartesian product operations with filtering.

3.2. Theoretical Foundation

3.2.1. NULL Values

In relational databases, a column value can sometimes be unknown or not applicable. To represent such cases, SQL uses a special marker: NULL.

Examples:

- A new sailor may not yet have a rating → rating = NULL.
- A “maiden_name” column may only apply to certain rows; for others it is irrelevant → store NULL.

3.2.2. Properties of NULL

- Comparisons: Any expression like value = NULL or value <> NULL returns UNKNOWN, not TRUE or FALSE.
 - Correct syntax: IS NULL or IS NOT NULL.
- Logical operators: AND, OR, NOT operate with three-valued logic (TRUE, FALSE, UNKNOWN).
- Arithmetic: Operations like age + NULL → result is NULL.

- Aggregates:
 - COUNT(*) includes NULLs.
 - SUM, AVG, MIN, MAX ignore NULLs.

Example: This query finds all sailors whose rating is not yet assigned.

```
SELECT sname
FROM sailors
WHERE rating IS NULL;
```

3.2.3. JOINS in SQL

Joins combine rows from two or more tables based on related columns. They allow queries to integrate data across relations like sailors, reserves, and boats.

3.2.3.1. INNER JOIN (default)

- Returns only rows where there is a match between both tables.
- Equivalent to a Cartesian product + filtering condition.

```
SELECT S.sname, R.bid, R.day
FROM sailors AS S
JOIN reserves AS R ON S.sid = R.sid;
```

- Here, sailors is the left table and reserves the right table
- Only sailors who made at least one reservation appear

3.2.3.2. NATURAL JOIN

- Automatically joins on all columns with the same name in both tables.
- Risk: it may join on unintended attributes if column names overlap.

```
SELECT S.sid, S.sname, R.bid, R.day
FROM sailors NATURAL JOIN reserves;
```

- Automatically uses sid as the join key (since both tables have it).

3.2.3.3. Explicit JOIN with ON

- Safer than natural join, because we explicitly state the join condition.
- Both join keys remain visible in the result (e.g., sailors.sid and reserves.sid)

```
SELECT S.sid, S.sname, B.bname, R.day
FROM sailors AS S
JOIN reserves AS R ON S.sid = R.sid
JOIN boats AS B ON R.bid = B.bid;
```

- Step 1: Match sailors with reserves (sid).
- Step 2: Match reservations with boats (bid).
- Clearer and avoids accidental matches.

3.2.3.4. OUTER JOINS

Outer joins preserve rows from one or both tables, even if there is no match in the other table. Missing values are filled with NULL.

3.2.3.4.1. LEFT OUTER JOIN

- Keeps all rows from the left table.

```
SELECT S.sname, B.bname, R.day
```

```
FROM sailors AS S
```

```
LEFT OUTER JOIN reserves AS R ON S.sid = R.sid
```

```
LEFT OUTER JOIN boats AS B ON R.bid = B.bid;
```

- All sailors appear, even those without reservations.
- For sailors without matches, B.bname and R.day are NULL.

3.2.3.4.2. RIGHT OUTER JOIN

- Keeps all rows from the right table.

```
SELECT S.sname, B.bname, R.day
```

```
FROM sailors AS S
```

```
RIGHT OUTER JOIN reserves AS R ON S.sid = R.sid;
```

- All reservations appear, even if they reference a sailor not in the sailors
- table.
- Missing sailor info will be NULL.

3.2.3.4.3. FULL OUTER JOIN

- Keeps all rows from both tables.

```
SELECT S.sname, B.bname, R.day
```

```
FROM sailors AS S
```

```
FULL OUTER JOIN reserves AS R ON S.sid = R.sid;
```

- Includes all sailors and all reservations.
- Missing matches on either side filled with NULL.

3.2.3.5. Summary: LEFT vs. RIGHT Table in Joins

- In LEFT JOIN, the left table is fully preserved.
- In RIGHT JOIN, the right table is fully preserved.
- In FULL OUTER JOIN, both are preserved.
- In an INNER JOIN, only matching rows are kept.

This distinction is critical when we want to ensure that “unmatched” rows (e.g., sailors with no reservations) are not lost from the results.

3.3. Lab Exercises

Use the same Sailing Database (tables sailors, boats, reserves) and seed data from Lab 1. If needed, recreate the schema and inserts from your previous lab.

This lab focuses on NULL values, INNER JOIN, NATURAL JOIN, JOIN with ON, and OUTER JOIN.

3.3.1. Working with NULL Values

-- Ex. 1 Find all sailors without a rating.

```
SELECT sid, sname
```

```
FROM sailors
```

```
WHERE rating IS NULL;
```

-- Returns sailors where the rating is missing. = NULL does not work - you must use IS NULL.

-- Ex. 2 Count how many sailors have an assigned rating.

```
SELECT COUNT(rating) AS rated_sailors
```

```
FROM sailors;
```

-- Unlike COUNT(*), this ignores NULL values.

3.3.2. JOINS

-- Ex. 3 List the names of sailors who made reservations.

```
SELECT S.sname, R.bid, R.day
```

```
FROM sailors AS S
```

```
JOIN reserves AS R ON S.sid = R.sid
```

```
ORDER BY S.sname;
```

-- Only sailors with at least one reservation appear. Equivalent (conceptually) to CROSS JOIN + filter, but executed efficiently.

-- Ex. 4 List sailor IDs and reservation IDs using NATURAL JOIN.

```
SELECT sid, bid, day
```

```
FROM sailors NATURAL JOIN reserves;
```

-- NATURAL JOIN automatically matches columns with the same name (sid). Use with care: if multiple columns have the same name, it will join on all of them.

-- Ex. 5 Show sailor names, boat names, and reservation days.

```
SELECT S.sname, B.bname, R.day
```

```
FROM sailors AS S
```

```
JOIN reserves AS R ON S.sid = R.sid
```

```
JOIN boats AS B ON R.bid = B.bid
```

ORDER BY S.sname, R.day;

-- Safer than NATURAL JOIN, because we explicitly define join conditions.

-- Ex. 6 List all sailors and the boats they reserved (include sailors with no reservations).

```
SELECT S.sname, B.bname, B.color, R.day
```

```
FROM sailors AS S
```

```
LEFT OUTER JOIN reserves AS R ON S.sid = R.sid
```

```
LEFT OUTER JOIN boats AS B ON R.bid = B.bid
```

```
ORDER BY S.sname;
```

-- Sailors without reservations (e.g., Andy, Rusty) appear with NULL values for boat and reservation columns.

-- Ex. 7 List all boats and the sailors who reserved them (include boats with no reservations)

```
SELECT B.bname, B.color, S.sname
```

```
FROM boats AS B
```

```
RIGHT OUTER JOIN reserves AS R ON B.bid = R.bid
```

```
RIGHT OUTER JOIN sailors AS S ON R.sid = S.sid
```

```
ORDER BY B.bname;
```

-- If supported, RIGHT OUTER JOIN preserves all boats; unmatched boats will still appear with NULL for sailor details.

-- Ex. 8 List sailors together with the number of boats they reserved (include sailors with 0).

```
SELECT S.sname, COUNT(R.bid) AS reservations
```

```
FROM sailors AS S
```

```
LEFT JOIN reserves AS R ON S.sid = R.sid
```

```
GROUP BY S.sname
```

```
ORDER BY reservations DESC;
```

-- LEFT JOIN ensures all sailors are listed, even those with no reservations (count = 0).

3.4. Troubleshooting

➤ NULL handling

- Do not use = NULL or <> NULL; always use IS NULL / IS NOT NULL.

- Remember that COUNT(column) ignores NULL values, while COUNT(*) includes them.
- Arithmetic with NULL (5 + NULL) → result is always NULL.
- INNER vs. OUTER JOIN
 - INNER JOIN keeps only matches. Rows without matches are lost.
 - LEFT OUTER JOIN preserves all rows from the left table; missing values from the right are shown as NULL.
 - RIGHT OUTER JOIN preserves all rows from the right table.
 - FULL OUTER JOIN keeps all rows from both tables.
- Natural Join risks
 - NATURAL JOIN uses all columns with the same name. If more than one column overlaps, it may produce unintended matches. Safer alternative: JOIN ... ON.
- Join condition placement
 - Conditions in the ON clause define how tables are matched.
 - Conditions in the WHERE clause filter the result after the join.
 - Mixing these can change results, especially with outer joins.
- Detecting sailors with no reservations
 - Use LEFT JOIN + WHERE R.sid IS NULL.
 - Common mistake: putting the condition in the WHERE clause incorrectly can turn a left join into an inner join.

3.5. Proposed Exercises

The following exercises are designed to practice handling NULL values and using different types of joins in SQL. Use the Sailing Database (sailors, boats, reserves) created in Lab 1 as your working dataset. Submit both the SQL queries and, where possible, their results.

3.5.1. Working with NULL values

- Find all sailors whose rating is unknown.
- List all sailors whose age is known.
- Compute the average rating of sailors (ignoring NULL automatically).
- Count how many sailors do not have a rating assigned.
- Show the names of sailors who have either NULL rating or NULL age.

3.5.2. INNER and NATURAL JOINS

- Find the names of sailors and the colors of the boats they reserved.
- List all reservations using NATURAL JOIN between sailors and reserves.
- Retrieve the IDs of sailors who reserved a red boat (use an inner join).
- Show all sailor names together with the IDs of boats they reserved, allowing duplicates.

3.5.3. Explicit JOINS with ON

- Display sailor names, boat names, and reservation dates (explicit join with ON).
- List all sailors together with the number of boats they reserved.

- For each reservation, show the sailor name and boat name, ordered by reservation day.
 - Retrieve all sailors who reserved both boat 101 and boat 103.
- 3.5.4. OUTER JOINs
- List all sailors, including those who have never reserved a boat.
 - List all boats, including those that were never reserved.
 - Use a FULL OUTER JOIN to show all sailors and all boats, matched when possible.
 - Find sailors who never reserved any boat (hint: LEFT JOIN + IS NULL).
 - For each boat, count how many distinct sailors reserved it; include boats with 0 reservations.
- 3.5.5. Mixed JOIN Scenarios
- Show all sailors and the total number of reservations they made (including those with 0).
 - For each sailor, list the distinct colors of boats they reserved.
 - Find sailors who reserved boats of at least two different colors
 - For each sailor, list the earliest reservation date and the boat reserved.
 - Find all pairs of sailors who reserved the same boat (self-join on reserves).
 - Show all sailors who reserved boats on the same day as sailor Horatio.

Laboratory no. 4

Defining and Manipulating Database Schemas in SQLite

4.1. Objectives

The aim of this laboratory is to gain practical experience with schema definition, constraints, and data manipulation in SQL using SQLite and DBeaver. By the end of this session, students should be able to:

- Create a new SQLite database and establish a connection in DBeaver.
- Define tables using the CREATE TABLE statement, specifying appropriate data types.
- Understand and apply primary keys, candidate keys, superkeys, and foreign keys.
- Implement integrity constraints to maintain consistency between related tables.
- Modify table definitions using ALTER TABLE (add, rename, drop columns, change types).
- Remove tables using DROP TABLE while considering dependencies.
- Insert, update, and delete data using DML commands (INSERT, UPDATE, DELETE).
- Explore the behavior of auto-increment fields and ROWID in SQLite.
- Understand and enable foreign key constraints in SQLite (via PRAGMA foreign_keys).
- Apply CRUD operations on a simple database schema (e.g., sailor, boat, reserve).

4.2. Theoretical Foundation

4.2.1. SQL and Database Schema

SQL (Structured Query Language) is the standard language for defining, manipulating, and querying data in relational database systems.

A schema describes the structure of the database: the tables, their attributes, and the constraints that enforce correctness.

SQL has two important subsets for this laboratory:

- Data Definition Language (DDL): commands to define or modify database structure (CREATE, ALTER, DROP).
- Data Manipulation Language (DML): commands to manipulate table content (INSERT, UPDATE, DELETE, SELECT).

4.2.1.1. Data Definition Language DDL

4.2.1.1.1. CREATE TABLE

Used to define new tables with specified attributes and data types.

Example:

```
CREATE TABLE Students (
    sid CHAR(20),
    name CHAR(30),
    login CHAR(20),
    age INTEGER,
    gpa REAL
);
```

Keys and Constraints

- Primary Key: uniquely identifies each row. At most one primary key per table.
- Candidate Key: a minimal set of attributes that uniquely identify rows.
- Superkey: a set of attributes that includes a candidate key; may contain extra attributes.
- Foreign Key: an attribute in one table that references a primary key in another table, ensuring referential integrity.

Example with constraints:

```
CREATE TABLE Enrolled (
    studid CHAR(20),
    cid CHAR(20),
    grade CHAR(10),
    PRIMARY KEY (studid, cid),
    FOREIGN KEY (studid) REFERENCES Students(sid)
);
```

4.2.1.1.2. ALTER TABLE

Used to modify existing tables (add, drop, or rename columns, change constraints).

```
ALTER TABLE sailor ADD Email VARCHAR(255);
```

```
ALTER TABLE sailor DROP COLUMN Email;
```

4.2.1.1.3. DROP TABLE

Removes a table and its data permanently.

```
DROP TABLE reserve;
```

Keys in Relational Databases

- Every relation must have at least one key (since tuples are unique).
- A table may have multiple candidate keys; one is chosen as the primary key.
- Foreign keys enforce relationships:
 - e.g., `reserve.sid` must reference `sailor.sid`.
 - Violations occur if a reservation references a sailor that does not exist.

Tables in SQLite have a hidden 64-bit signed integer column called ROWID if no explicit primary key is defined.

- Declaring a column as `INTEGER PRIMARY KEY` makes it an alias for ROWID.
- Using `AUTOINCREMENT` ensures that IDs are never reused, even after deletions.
- Date and time values can be stored as:
- `TEXT` (ISO8601 string),
- `REAL` (Julian day number),
- `INTEGER` (Unix timestamp).
- Foreign keys are disabled by default in SQLite. They must be explicitly enabled:

```
PRAGMA foreign_keys = ON;
```

4.2.1.2. Data Manipulation Language DML

CRUD operations:

- `INSERT`: add rows

```
INSERT INTO sailor (sname, age, rating) VALUES ('Dustin', 45, 7);
```

- `SELECT`: read rows

```
SELECT sname, rating FROM sailor WHERE age > 30;
```

- `UPDATE`: modify rows

```
UPDATE sailor SET rating = rating + 1 WHERE sid = 1;
```

- `DELETE`: remove rows

```
DELETE FROM sailor WHERE sname = 'Klaus';
```

These operations ensure that the database content can evolve while respecting the defined schema and constraints.

4.3. Lab Exercises

In this laboratory, you will use DBeaver with SQLite to define tables, enforce constraints, and manipulate data.

4.3.1. Creating Tables and Keys

-- Ex. 1 Create a table sailor with a primary key and attributes.

```
CREATE TABLE sailor (  
    sid INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
    sname TEXT,  
    age REAL,  
    rating INTEGER  
);
```

-- sid is the primary key. The AUTOINCREMENT option ensures unique, automatically generated IDs.

-- Ex. 2 Create a table boat with a primary key.

```
CREATE TABLE boat (  
    bid INTEGER NOT NULL,  
    bname TEXT,  
    color TEXT,  
    CONSTRAINT boat_pk PRIMARY KEY (bid)  
);
```

-- The boat table uses a named primary key constraint.

-- Ex. 3 Create a table reserve with a composite key and foreign keys.

```
CREATE TABLE reserve (  
    sid INTEGER,  
    bid INTEGER,  
    day INTEGER,  
    CONSTRAINT reserve_pk PRIMARY KEY (sid, bid, day),  
    CONSTRAINT reserve_boat_FK FOREIGN KEY (bid) REFERENCES  
boat(bid),
```

```
    CONSTRAINT reserve_sailor_FK FOREIGN KEY (sid) REFERENCES
sailor(sid)
```

```
);
```

-- The composite key (sid, bid, day) ensures no duplicate reservations. Foreign keys enforce referential integrity.

4.3.2. Altering and Dropping Table

-- Ex. 4 Add a new column email to the sailor table.

```
ALTER TABLE sailor ADD email VARCHAR(255);
```

-- New column is added. Existing rows will have NULL in this column until updated.

-- Ex. 5 Drop the column email from sailor.

```
ALTER TABLE sailor DROP COLUMN email;
```

4.3.3. CRUD Operations

-- Ex. 6 Insert sample sailors.

```
INSERT INTO sailor (sname, age, rating)
```

```
VALUES
```

```
('Dustin', 45, 7),
```

```
('Brutus', 33, 1),
```

```
('Lubber', 55.5, 8);
```

-- Note: sid is auto-generated.

-- Ex. 7 Insert boats

```
INSERT INTO boat (bid, bname, color)
```

```
VALUES
```

```
(101, 'Interlake', 'blue'),
```

```
(102, 'Clipper', 'red');
```

-- Ex. 8 Increase the rating of sailor with sid = 1.

```
UPDATE sailor
```

```
SET rating = rating + 1
```

```

WHERE sid = 1;

-- Ex.9 Increase age of all sailors by 1 year.
    UPDATE sailor
    SET age = age + 1;

-- Ex.10 Delete sailor with name Brutus.
        DELETE FROM sailor WHERE sname = 'Brutus';

-- Ex.11 Delete all rows from reserve.
        DELETE FROM reserve;

-- Ex.12 Enable foreign key constraints in SQLite.
        PRAGMA foreign_keys = ON;

-- Ex.13 Attempt to insert an invalid reservation.
        INSERT INTO reserve (sid, bid, day) VALUES (999,
999, 5);

-- This fails because sailor 999 and boat 999 do not exist.
Foreign key constraint prevents invalid data.

```

4.4. Troubleshooting

- Primary Key and AUTOINCREMENT
 - In SQLite, INTEGER PRIMARY KEY is an alias for the hidden ROWID.
 - Adding AUTOINCREMENT ensures IDs are never reused, but it adds overhead — use only if necessary.
- Primary Key must be NOT NULL
 - According to the SQL standard, primary keys are always NOT NULL.
 - Some older SQLite versions allowed NULL primary keys — avoid relying on this behavior.
- Foreign Keys Disabled by Default
 - In SQLite, foreign key constraints are off by default.
 - Enable them explicitly at the start of each session:


```
PRAGMA foreign_keys = ON;
```

- If you forget this, invalid data (orphaned references) may be inserted without error.
- Deleting Rows with Dependencies
 - If you try to delete a sailor who has reservations, you may get: SQL Error [19]: FOREIGN KEY constraint failed
 - Solution: delete dependent rows in reserve first, or define cascading behavior (ON DELETE CASCADE).
- Reserved Keywords as Column Names
 - Avoid using words like day, user, order as column names.
 - If unavoidable, quote them: "day". Better: choose safe alternatives (e.g., reservation_day).
- ALTER TABLE Limitations in SQLite
 - You can add, rename, or drop columns, but advanced changes (like removing constraints) may require recreating the table.
- UPDATE without WHERE
 - Running an UPDATE without a WHERE clause updates all rows.
 - Example:


```
UPDATE sailor SET rating = 0;
```
 - → resets the rating for every sailor. Always double-check your WHERE.
- DELETE without WHERE
 - Similarly, DELETE FROM sailor; removes all rows while keeping the table.
 - Use with caution.

4.5. Proposed Exercises

The following exercises are designed to practice schema definition, constraints, and CRUD operations in SQLite. Use DBeaver or another SQLite client. Submit both the SQL queries and, where possible, their results.

4.5.1. Creating and Modifying Schemas (DDL)

- Create a student table with attributes (sid, name, age, gpa) where sid is the primary key.
- Create a course table with attributes (cid, title, credits) where cid is the primary key.
- Create an enrolled table with attributes (sid, cid, grade) and:
 - composite primary key (sid, cid),
 - foreign keys referencing student(sid) and course(cid)
- Add a column email to the student table.
- Rename the column title in course to course_name.
- Drop the gpa column from the student table.
- Drop the enrolled table.

4.5.2. Inserting Data (CREATE in CRUD)

- Insert 5 students into the student table with realistic names and ages.

- Insert 3 courses into the course table.
 - Insert enrollments linking students to courses, including different grades.
 - Try to insert an enrollment for a student who does not exist and note the result (foreign key violation).
 - Insert multiple rows in one INSERT statement into the student table.
- 4.5.3. Updating Data (UPDATE in CRUD)
- Update the age of all students by +1 year.
 - Increase the credits of all courses by 1.
 - Update the grade of a specific student in a course from B to A.
 - Decrease the grade of all students in Math101 by one level (e.g., A $\rightarrow$ B, B $\rightarrow$ C).
- 4.5.4. Deleting Data (DELETE in CRUD)
- Delete one student by sid.
 - Delete all students older than 30.
 - Try to delete a course that has enrolled students and observe the foreign key error.
 - Delete all rows from the enrolled table without dropping the table.
- 4.5.5. Working with Foreign Keys and Constraints
- Create a new table library with a foreign key referencing student(sid). Insert rows and test what happens if you delete a student.
 - Redefine the enrolled table with ON DELETE CASCADE and test deleting a student who has enrollments.
 - Insert a reservation in the reserve table with a NULL sid. Does SQLite accept it? Why?
 - Enable and disable foreign keys using PRAGMA foreign_keys. Show the difference in behavior
- 4.5.6. Mixed
- Design and implement a small schema for a library system with tables Book, Author, Borrow.
 - Insert at least 10 rows in each table.
 - Write queries to:
 - find books borrowed by a specific student,
 - find authors who have at least 2 books in the library,
 - delete a book and observe cascading effects on Borrow.
 - Recreate the classic Sailor–Boat–Reserve schema from previous labs, but enforce stricter constraints (e.g., no NULLs in sid or bid).

Laboratory no. 5

Working with PostgreSQL: Installation, Schemas, and Sample Databases

5.1. Objectives

The aim of this laboratory is to introduce students to PostgreSQL as a relational database management system and to gain practical skills in database installation, configuration, and usage with sample datasets. By the end of this session, students should be able to:

- Understand the role of PostgreSQL as an object-relational DBMS and its advantages compared to other systems.
- Install PostgreSQL on a local machine and configure the environment (PATH, superuser account, default port).
- Use psql (command-line client) and DBeaver (GUI tool) to connect to PostgreSQL databases.
- Differentiate between a database and a schema in PostgreSQL, and understand the concept of the public schema.
- Create and manage databases with SQL (CREATE DATABASE, DROP DATABASE) and GUI tools.
- Load and explore a sample database (e.g., Northwind, Pagila) into PostgreSQL.
- Perform CRUD operations (INSERT, SELECT, UPDATE, DELETE) on sample datasets.
- Write more complex SQL queries involving joins, grouping, filtering, and aggregation.
- Create reports such as customer orders by year, total revenues, top products, and customer payments.
- Understand best practices in working with PostgreSQL users, roles, and privileges.

5.2. Theoretical Foundation

5.2.1. Introduction to PostgreSQL

PostgreSQL is an open-source, object-relational database management system (ORDBMS). It extends the relational model with advanced features such as:

- user-defined data types,
- table inheritance,
- functions and stored procedures,
- strong support for ACID transactions.

Compared to lightweight systems like SQLite, PostgreSQL is designed for multi-user environments, supports concurrency, and provides extensive functionality for both OLTP (transactional workloads) and OLAP (analytical workloads).

5.2.2. Installing PostgreSQL

- On Windows, installation is done via the official installer (EnterpriseDB).
- During installation, the user specifies:
 - superuser (postgres) password,
 - default port (5432),
 - data directory.
- After installation, PostgreSQL is available as a Windows service.

Environment variables (e.g., PATH) may need to be updated so that the psql client can be used from the command line.

5.2.3. PostgreSQL Tools: psql and DBeaver

- psql: a command-line client shipped with PostgreSQL.
 - Connect to a database:

```
psql -U postgres -d mydb
```
 - Useful meta-commands:
 - \l → list databases,
 - \dt → list tables,
 - \q → quit session.
- DBeaver: a GUI database management tool.
 - Allows easy creation of connections, schema browsing, SQL execution, and visualization of results.
 - Particularly useful for working with larger databases like Northwind or Pagila.

5.2.4. Databases and Schemas in PostgreSQL

- A database is a separate collection of objects (tables, views, functions, etc.).
- Within each database, objects are organized in schemas.
- By default, every new PostgreSQL database has a schema named public.
- Example:
 - mydb = database,
 - public = default schema,
 - public.customers = table customers inside schema public.

Schemas help organize data and allow multiple users or applications to work in the same database without conflicts.

5.2.5. Sample Databases

For educational and testing purposes, PostgreSQL supports importing sample datasets such as:

- Northwind – a small sales database with customers, orders, products, employees.

- Pagila – a movie rental database, inspired by Sakila from MySQL.

These datasets allow practice with:

- CRUD operations (INSERT, UPDATE, DELETE, SELECT),
- joins across multiple tables (e.g., customers → orders → products),
- reporting queries (e.g., total sales per customer, best-selling product).

5.2.6. User Management

PostgreSQL uses a role-based access model:

- A role can act as a user or a group.
- Common commands:

```
CREATE ROLE analyst LOGIN PASSWORD 'pass123';
```

```
GRANT CONNECT ON DATABASE mydb TO analyst;
```

```
GRANT USAGE ON SCHEMA public TO analyst;
```

- Superuser (postgres) can create new users and assign privileges.

5.3. Lab Exercises

5.3.1. Installation and Connection

```
-- Ex. 1 Connect to PostgreSQL using psql
```

```
psql -U postgres -d postgres
```

```
-- Connects as superuser postgres to the default postgres database.
```

```
-- Ex. 2 Create a new database for practice.
```

```
CREATE DATABASE lab5db;
```

```
-- Creates an empty database named lab5db.
```

5.3.2. Working with Schemas

```
Ex. 3 Create a custom schema.
```

```
CREATE SCHEMA training;
```

```
-- A new schema training is added to the current database.
```

```
-- Ex. 4 Create a table inside the schema.
```

```
CREATE TABLE training.students (  
    sid SERIAL PRIMARY KEY,  
    name TEXT,
```

```

    age INTEGER,
    gpa NUMERIC(3,2)
);
-- Table students is created in schema training.
5.3.3. Loading a Sample Database (Northwind)
-- Ex. 5 Import Northwind into PostgreSQL.
-- Step 1: download the northwind.sql dump
-- Restore with:
psql -U postgres -d lab5db -f northwind.sql
-- The Northwind schema and data are now available.

```

```

-- Ex. 6 Connect to PostgreSQL using psql
psql -U postgres -d postgres
-- Connects as superuser postgres to the default postgres
database.

```

5.3.4. CRUD Operations in Northwind

```

-- Ex. 7 Insert a new customer.
INSERT INTO customers (customer_id, company_name,
contact_name, country)
VALUES ('CUSTX', 'DataLab Inc.', 'Alice Johnson', 'USA');
-- A new row is inserted into the customers table.

```

```

-- Ex. 8 Update a customer's country.
UPDATE customers
SET country = 'Canada'
WHERE customer_id = 'CUSTX';
-- Updates the country column for the inserted customer.

```

```

-- Ex. 9 Delete the inserted customer.
DELETE FROM customers WHERE customer_id = 'CUSTX';
-- Removes the test row from the customers table.

```

5.3.5. Querying and Reporting

-- Ex. 10 List all orders placed in 1997.

```
SELECT order_id, customer_id, order_date
FROM orders
WHERE EXTRACT(YEAR FROM order_date) = 1997
ORDER BY order_date;
```

-- Ex. 11 Find the total number of orders per customer.

```
SELECT customer_id, COUNT(*) AS total_orders
FROM orders
GROUP BY customer_id
ORDER BY total_orders DESC;
```

-- Ex. 12 List the top 5 products by quantity ordered.

```
SELECT p.product_name, SUM(od.quantity) AS total_qty
FROM order_details od
JOIN products p ON od.product_id = p.product_id
GROUP BY p.product_name
ORDER BY total_qty DESC
LIMIT 5;
```

5.3.6. User Management

-- Ex. 13 Create a new role.

```
CREATE ROLE analyst LOGIN PASSWORD 'labpass';
```

-- Ex. 14 Grant privileges on the Northwind database.

```
GRANT CONNECT ON DATABASE lab5db TO analyst;
GRANT USAGE ON SCHEMA public TO analyst;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO analyst;
```

5.4. Troubleshooting

- psql not recognized

- Add PostgreSQL's bin folder to your PATH (e.g., C:\Program Files\PostgreSQL\16\bin), then reopen the terminal.
- Can't connect / port in use
 - Default port is 5432. Check service status and port conflicts:
 - Windows Services: ensure PostgreSQL service is Running.
 - `netstat -ano | find "5432"` (Windows) to see who uses the port.
 - Change port in `postgresql.conf` if needed.
- Authentication failed
- Wrong user/password or DB name. Try:


```
psql -U postgres -d postgres -h localhost -p 5432
```
- If local auth issues persist, review `pg_hba.conf` (method md5/scram vs trust) and reload:


```
SELECT pg_reload_conf();
```
- Server not starting
 - Check the data directory permissions and free disk space.
 - Review logs in `.../data/pg_log/` or `.../data/log/`.
- DBeaver can't connect
 - Verify driver selection (PostgreSQL), host, port, DB, user.
 - If SSL is required/forbidden by your instance, match the SSL setting in the connection.
- Semicolons & psql meta-commands
 - SQL statements must end with `;`.
 - `\l` (list DBs), `\dn` (schemas), `\dt` (tables), `\du` (roles), `\q` (quit).
- "Permission denied" / role errors
 - Grant minimal required privileges:


```
GRANT CONNECT ON DATABASE lab5db TO analyst;
GRANT USAGE ON SCHEMA public TO analyst;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO analyst;
```
- New tables? Also set default privileges (optional):


```
ALTER DEFAULT PRIVILEGES IN SCHEMA public
GRANT SELECT ON TABLES TO analyst;
```
- Database vs. schema confusion
 - Fully qualify when unsure: `SELECT * FROM public.customers;`
 - Check current schema search path:


```
SHOW search_path;
```
- Restore/import fails

- Use psql for .sql dumps:


```
psql -U postgres -d lab5db -f northwind.sql
```
- For .dump/custom formats, use pg_restore:


```
pg_restore -U postgres -d lab5db northwind.dump
```
- Foreign key / constraint errors
 - Insert parent rows before child rows.
 - Delete child rows (or use ON DELETE CASCADE) before deleting parents.
- Case sensitivity / quoted identifiers
 - Unquoted identifiers are folded to lower case. If you created "Customers" with quotes, you must always quote it:


```
SELECT * FROM "Customers";
```
- Locale/encoding oddities
 - Check:


```
SHOW lc_collate; SHOW lc_ctype; SHOW client_encoding;
```
- Transaction surprises
 - In psql, a failed statement may abort the transaction until ROLLBACK.
 - Use COMMIT; / ROLLBACK; explicitly when batching changes.

5.5. Proposed Exercises

The following exercises are designed to strengthen your understanding of PostgreSQL installation, database management, schemas, CRUD operations, and reporting using a sample dataset (e.g., Northwind). Submit both the SQL queries and, where possible, the outputs.

5.5.1. Installation and Connection

- Install PostgreSQL on your machine and connect with psql as user postgres.
- Create a new database called lab5db and connect to it.
- Create a new schema training and verify it appears with \dn.

5.5.2. Schemas and Tables

- Inside schema training, create a table employees with columns (eid SERIAL, name TEXT, position TEXT, salary NUMERIC(8,2)).
- Create a table departments with columns (did SERIAL, dept_name TEXT) and set did as primary key.
- Add a column email to the employees table.
- Drop the column email from the employees table.

5.5.3. Loading Sample Database (Northwind)

- Import the Northwind dataset into lab5db.
- Verify that tables customers, orders, and products are available.
- Inspect the first 10 rows of the customers table.

5.5.4. CRUD Operations

- Insert three new customers into the customers table.
 - Update the contact name of one customer.
 - Delete one of the inserted customers.
 - Insert five new products into the products table.
 - Update the price of all products in the category “Beverages” by +10%.
 - Delete all products with stock quantity = 0.
- 5.5.5. Querying and Reporting
- List all orders placed in 1998.
 - Find the total number of orders per customer.
 - Find the top 5 customers by number of orders.
 - Show the total revenue per year, combining orders and order_details.
 - List the employees and the number of orders handled by each.
 - Display the top 3 products by total sales amount.
 - Find all customers who never placed an order.
 - Show the average unit price per category.
- 5.5.6. User Management
- Create a role analyst with password labpass.
 - Grant analyst read-only access to the customers and orders tables.
 - Create another role dataentry with privileges to insert rows into products.
 - Revoke INSERT privileges from dataentry and test the result.
- 5.5.7. Mixed
- Create a report showing customers and their total spending, sorted from highest to lowest.
 - Write a query to find the product(s) that generated the highest revenue overall.
 - Show for each employee the total revenue from the orders they managed.
 - Write a query to display the number of orders per month in 1997.
 - Create a view top_customers that lists the top 10 customers by sales.
 - Export the result of a query (e.g., total sales per category) from DBeaver as a CSV file.

Laboratory no. 6

Integrity Constraints, Referential Rules, and Views in PostgreSQL

6.1. Objectives

The objectives of this laboratory are to extend the students' understanding of data integrity, database constraints, and views, ensuring that databases store only valid and consistent information.

By the end of this session, students should be able to:

- Understand the concept of integrity constraints and their role in maintaining data correctness.
- Define and apply NOT NULL, UNIQUE, and CHECK constraints to enforce data quality rules.
- Implement referential integrity using FOREIGN KEYS with options such as ON DELETE CASCADE, SET NULL, and SET DEFAULT.
- Modify existing tables to add or alter constraints using ALTER TABLE statements.
- Create and manage unique indexes to enforce data uniqueness in existing databases.
- Work with binary data (BLOBs) and images in database tables using DBever.
- Use sequences and serial types to automatically generate primary key values.
- Apply self-joins to represent hierarchical relationships within a single table (e.g., employees and managers).
- Create and query views as virtual tables to improve data security and abstraction.
- Design and improve a database schema by adding appropriate constraints and views, and test these constraints for enforcement.

6.2. Theoretical Foundation

6.2.1. Integrity Constraints in Database

A well-designed database must guarantee that the data it stores are accurate, consistent, and meaningful.

Integrity constraints are rules applied to the schema that restrict the values allowed in tables, ensuring that invalid or contradictory data cannot be inserted or maintained.

In SQL, these constraints are checked automatically by the Database Management System (DBMS) whenever data are inserted, updated, or deleted. If a modification violates a constraint, the DBMS either rejects the change or performs a compensating action (such as cascading updates or deletions).

Integrity constraints protect the logical correctness of the database and prevent accidental damage or data corruption.

6.2.2. Types of Integrity Constraints

6.2.2.1. Domain Constraints

- A domain constraint restricts the set of valid values for an attribute.
- It is enforced by specifying a data type and, optionally, additional restrictions such as NOT NULL or CHECK.
- name VARCHAR(40) NOT NULL
- This ensures that every row must have a non-null name value.

6.2.2.2. NOT NULL Constraint

- The NOT NULL constraint prohibits the insertion of NULL values for a column.
- Attributes that are essential (e.g., employee name, product price) should be declared NOT NULL to prevent incomplete data.

```
ALTER TABLE public.shippers
```

```
ALTER COLUMN phone SET NOT NULL;
```

- If any row already has NULL in that column, the change will fail until data are corrected.

6.2.2.3. UNIQUE Constraint

- A UNIQUE constraint ensures that the combination of specified columns contains only distinct values.
- It can be applied to a single column or a group of columns.

```
CREATE UNIQUE INDEX shippers_company_name_idx
```

```
ON public.shippers (company_name);
```

- This guarantees that two shippers cannot have the same company name.

6.2.2.4. CHECK Constraint

- A CHECK constraint enforces that values in a column satisfy a logical condition.
- It can validate ranges, enumerations, or logical expressions.

```
ALTER TABLE public.products
```

```
ADD CONSTRAINT products_check CHECK (unit_price > 0);
```

- This prevents negative or zero prices from being entered into the products table.
- If a statement such as the below one is executed, it will fail due to the CHECK constraint.

```
UPDATE products SET unit_price = -21 WHERE product_id = 2;
```

6.2.2.5. Referential Integrity (FOREIGN KEY)

- Referential integrity ensures that relationships between tables remain consistent.
- A foreign key in one table references a primary key in another table, guaranteeing that referenced data actually exists.

```
CREATE TABLE course (  
    course_id VARCHAR(8),  
    title VARCHAR(50),  
    dept_name VARCHAR(20),  
    credits NUMERIC(2,0) CHECK (credits > 0),  
    PRIMARY KEY (course_id),  
    FOREIGN KEY (dept_name) REFERENCES department  
        ON DELETE CASCADE  
        ON UPDATE CASCADE  
);
```

- ON DELETE CASCADE: if a department is deleted, all related courses are deleted automatically.
- ON UPDATE CASCADE: if the department name changes, the related courses are updated accordingly.
- Other options include SET NULL or SET DEFAULT.

6.2.2.6. Constraint Naming and Management

- Constraints can be explicitly named to facilitate maintenance or debugging.

```
ALTER TABLE products  
ADD CONSTRAINT products_check_price CHECK (unit_price > 0);
```

- If needed, a constraint can later be dropped using its name:

```
ALTER TABLE products DROP CONSTRAINT products_check_price;
```

6.2.3. Working with Binary and Large Objects (BLOB/CLOB)

Modern databases can store binary data such as images or documents using BLOB (Binary Large Object) columns.

In DBeaver, users can:

- Load or save BLOB data directly from files.
- View images (JPEG, PNG, GIF, BMP) directly within the interface.
- Edit and manage binary data using the Value View Panel (shortcut F7).

This feature supports multimedia applications and document management systems.

6.2.4. Sequences and Serial Columns

A sequence is an independent object that generates unique numeric values, often used for primary keys.

```
CREATE SEQUENCE region_region_id_seq;  
ALTER TABLE region  
ALTER COLUMN region_id SET DEFAULT  
nextval('region_region_id_seq');  
SELECT setval('region_region_id_seq', 4);
```

Alternatively, PostgreSQL provides the shorthand data type SERIAL for automatic numbering.

6.2.5. SELF Join

A self-join allows a table to be joined with itself.

This is especially useful for hierarchical data (e.g., employees reporting to managers).

```
SELECT  
    e.employee_id, e.first_name, e.last_name,  
    m.first_name AS manager_first_name, m.last_name AS  
    manager_last_name  
FROM employees e  
LEFT JOIN employees m ON e.reports_to = m.employee_id;
```

This query lists each employee alongside their manager.

6.2.6. Views

A view is a virtual table defined by a query.

It presents data from one or more tables without storing it physically, allowing for abstraction, customization, and security.

```
CREATE VIEW employees_with_managers AS  
SELECT e.employee_id, e.first_name, e.last_name,  
       m.first_name || ' ' || m.last_name AS manager  
FROM employees e  
LEFT JOIN employees m ON e.reports_to = m.employee_id;
```

Advantages:

- Simplifies complex queries.
- Restricts access to sensitive data (e.g., hiding salaries or personal details).
- Reflects real-time data since it is computed dynamically when queried.

6.3. Lab Exercises

In this laboratory, you will work with PostgreSQL (via DBeaver) to explore and enforce various database constraints, create and test views, and extend the Northwind or a similar sample database.

If needed, reconnect to the lab5db database created in the previous session.

6.3.1. Working with NOT NULL and UNIQUE Constraints

```
-- Ex. 1 Adding a NOT NULL constraint
-- Open the public.shippers table in DBeaver.
-- Identify the column phone, which currently allows NULL
values.
-- Verify that no rows contain NULL in this column:
SELECT * FROM shippers WHERE phone IS NULL;
-- The result should be empty before applying the constraint.
-- Add the constraint
ALTER TABLE public.shippers
ALTER COLUMN phone SET NOT NULL;
-- Test the constraint:
INSERT INTO shippers (shipper_id, company_name) VALUES (9,
'ExFed');
-- The command should fail because phone cannot be null
-- Insert a valid row
INSERT INTO shippers (shipper_id, company_name, phone)
VALUES (9, 'ExFed', '1-800-555-2222');
-- Ex. 2 Enforcing Uniqueness
-- Add a unique index on company_name:
CREATE UNIQUE INDEX shippers_company_name_idx
ON public.shippers (company_name);
-- Attempt to insert a duplicate name
INSERT INTO shippers (shipper_id, company_name, phone)
```

```
VALUES (10, 'FedEx', '1-800-555-3333');  
-- This should fail due to violation of the unique constraint.
```

6.3.2. Working with CHECK Constraints

```
-- Ex. 3 Prevent invalid product prices  
-- Open the products table and observe the unit_price column.  
-- Add a check constraint to ensure all prices are positive:  
ALTER TABLE public.products  
ADD CONSTRAINT products_check_price CHECK (unit_price > 0);  
-- Try to insert or update a product with a negative price:  
UPDATE products SET unit_price = -10 WHERE product_id = 2;  
-- The command should fail with a CHECK constraint violation.
```

6.3.3. Enforcing Referential Integrity

```
-- Ex. 4 Creating and Testing a Foreign Key
```

```
-- Create a new table departments:
```

```
CREATE TABLE departments (  
    dept_id SERIAL PRIMARY KEY,  
    dept_name VARCHAR(40) NOT NULL UNIQUE  
);
```

```
-- Create a related table employees:
```

```
CREATE TABLE employees (  
    emp_id SERIAL PRIMARY KEY,  
    emp_name VARCHAR(40) NOT NULL,  
    dept_id INT,  
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)  
        ON DELETE CASCADE  
        ON UPDATE CASCADE  
);
```

```
-- Insert data:
```

```
INSERT INTO departments (dept_name) VALUES ('Sales'), ('IT');
```

```
INSERT INTO employees (emp_name, dept_id) VALUES ('Alice', 1),
('Bob', 2);
```

```
-- Test referential actions:
```

```
DELETE FROM departments WHERE dept_id = 1;
```

```
-- The employee "Alice" will also be deleted due to ON DELETE
CASCADE.
```

6.3.4. Working with Sequences and Serial Columns

```
-- Ex. 5 Converting a column into a sequence
```

```
-- Check the region table contents:
```

```
SELECT * FROM region;
```

```
-- Create a sequence:
```

```
CREATE SEQUENCE region_region_id_seq;
```

```
-- Link it to the primary key column:
```

```
ALTER TABLE region
```

```
ALTER      COLUMN      region_id      SET      DEFAULT
nextval('region_region_id_seq');
```

```
ALTER SEQUENCE region_region_id_seq OWNED BY region.region_id;
```

```
-- Initialize the sequence to the current maximum ID:
```

```
SELECT setval('region_region_id_seq', (SELECT MAX(region_id)
FROM region));
```

```
-- Test by inserting a new region:
```

```
INSERT INTO region (region_description) VALUES ('My New
Region');
```

6.3.5. Self-Join Practice

```
-- Ex. 6 Display employees with their managers
```

```
-- The employees table includes a reports_to column
referencing another employee.
```

```
-- Use a self-join to display hierarchical relationships:
```

```
SELECT
```

```
    e.employee_id,
```

```
    e.first_name, e.last_name, e.title,
```

```

    m.first_name AS manager_first_name, m.last_name AS
manager_last_name
FROM employees e
LEFT JOIN employees m ON e.reports_to = m.employee_id
ORDER BY e.employee_id;

```

6.3.6. Working with Views

```

-- Ex. 7 Creating a view
-- Define a view to show employees along with their managers:
CREATE VIEW employees_with_managers AS
SELECT
    e.employee_id, e.first_name, e.last_name, e.title,
    m.first_name || ' ' || m.last_name AS manager
FROM employees e
LEFT JOIN employees m ON e.reports_to = m.employee_id;
-- Query the view:
SELECT * FROM employees_with_managers;
-- Modify or add employees and observe that the view updates
automatically.

```

6.3.7. Working with BLOB Data (Images)

```

-- Ex. 8 Loading an Image into a Table
-- Open the categories table (which includes a picture column).
-- Right-click on a row's picture cell → Edit → Load from file.
-- Select a local image (e.g., category1.png) and save changes.
-- DBeaver will automatically render the image within the cell
viewer.

```

6.3.8. Self-Join Practice Continued

Even with correct SQL syntax, working with constraints and schema modifications can produce unexpected errors.

The following tips will help you diagnose and resolve common issues when working with constraints, sequences, BLOBs, and views.

6.4. Troubleshooting

6.4.1. Constraint Violations

ERROR: null value in column "phone" of relation "shippers" violates not-null constraint

Cause: Trying to insert or update a row with a NULL value in a NOT NULL column.

Solution: Ensure all NOT NULL columns have valid data before insertion. Use the following in order to locate problematic rows.

```
SELECT * FROM shippers WHERE phone IS NULL;
```

6.4.2. Unique Constraint Errors

- ERROR: duplicate key value violates unique constraint "shippers_company_name_idx"
- Cause: Attempting to insert or update a record with a value that already exists in a column declared as UNIQUE.
- Solution: Check for duplicates before insertion or update the existing row instead of inserting a new one.

```
SELECT company_name, COUNT(*) FROM shippers GROUP BY  
company_name HAVING COUNT(*) > 1;
```

6.4.3. CHECK Constraint Failures

- ERROR: new row violates check constraint "products_check_price"
- Cause: The new value violates the logical condition in the CHECK clause.
- Solution: Review the condition using:

```
\d products
```

- and ensure all values satisfy `unit_price > 0` before attempting to update.

6.4.4. Foreign Key Constraint Failures

- ERROR: insert or update on table "employees" violates foreign key constraint
- Cause: The foreign key value does not match any primary key in the referenced table.
- Solutions:
 - Insert the parent row first.
 - Enable cascading actions (ON DELETE CASCADE / ON UPDATE CASCADE).
 - If deletions fail, delete dependent rows first or use:

```
ALTER TABLE employees
```

```
DROP CONSTRAINT employees_dept_id_fkey,
```

```
ADD CONSTRAINT employees_dept_id_fkey
```

```
FOREIGN KEY (dept_id)
```

```
REFERENCES departments(dept_id)
```

```
ON DELETE CASCADE;
```

6.4.5. Sequence / Serial Issues

- ERROR: duplicate key value violates unique constraint "region_pkey"
- Cause: The sequence is not synchronized with existing data.
- Solution: Reset the sequence to the correct value:

```
SELECT setval('region_region_id_seq', (SELECT MAX(region_id)
FROM region));
```

6.4.6. BLOB and Image Handling Problems

- Issue: Images do not display correctly in DBeaver.
- Possible causes:
 - Unsupported file format (only png, jpg, gif, bmp supported).
 - The column type is not properly set as BYTEA (PostgreSQL).
- Solution:
 - Use the “Value View Panel” (F7) in DBeaver to view or edit images.
 - Ensure the column type is:

picture BYTEA;

6.4.7. View-Related Errors

- ERROR: column does not exist in view definition
- Cause: A column used in the view was renamed or deleted in the base table.
- Solution: Drop and recreate the view:

```
DROP VIEW employees_with_managers;
```

```
CREATE VIEW employees_with_managers AS
```

```
SELECT e.employee_id, e.first_name, e.last_name,
       m.first_name || ' ' || m.last_name AS manager
```

```
FROM employees e
```

```
LEFT JOIN employees m ON e.reports_to = m.employee_id;
```

6.4.8. Permission or Role Issues

- ERROR: permission denied for relation employees
- Cause: The user lacks privileges to query or modify the table.
- Solution:
 - Grant proper privileges:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON employees TO analyst;
```

6.4.9. Data Type Mismatch

- ERROR: column "budget" is of type numeric but expression is of type text
- Cause: Attempt to insert data with an incompatible type.
- Solution: Use explicit casting or correct the input data:

```
INSERT INTO department (dept_name, building, budget)
VALUES ('IT', 'North', 50000::NUMERIC);
```

6.4.10. General Best Practices

- Always check for NULL values before adding NOT NULL constraints.
- Use meaningful names for constraints (e.g., products_check_price, fk_emp_dept).
- Back up data before applying schema changes.
- Avoid disabling constraints temporarily unless absolutely necessary.
- Re-run constraint validation after imports or bulk data updates.

6.5. Proposed Exercises

The following exercises are designed to help you apply and test your understanding of database constraints, sequences, self-joins, and views.

Work with the PostgreSQL database (for example, lab5db or a new database lab6db). Submit both the SQL scripts and, where possible, screenshots or outputs of query results.

6.5.1. Exploring and Adding Constraints

- Create a new table suppliers with the following columns:
 - supplier_id SERIAL PRIMARY KEY
 - company_name VARCHAR(50) NOT NULL
 - contact_name VARCHAR(50)
 - phone VARCHAR(20) NOT NULL
- Add a unique constraint on company_name.
- Insert 3–4 valid rows, then try to insert a duplicate company_name and note the error message.
- Add a CHECK constraint to ensure that phone starts with '+' (e.g., CHECK (phone LIKE '+%')).
- Try inserting a phone number without + and observe the constraint violation.

6.5.2. Referential Integrity

- Create a new table orders_summary that references the existing orders table:

```
CREATE TABLE orders_summary (
    summary_id SERIAL PRIMARY KEY,
    order_id INT NOT NULL,
    total_amount NUMERIC(10,2) CHECK (total_amount >= 0),
    FOREIGN KEY (order_id) REFERENCES orders(order_id)
    ON DELETE CASCADE
);
```

- Test cascading behavior by deleting an order from orders and verifying that the related record in orders_summary is automatically removed.
- Modify the foreign key behavior to SET NULL and observe the difference.

6.5.3. Sequences and Serial Columns

- Create a sequence named movies_seq.
- Create a table movies using this sequence:

```
CREATE TABLE movies (
    movie_id INT DEFAULT nextval('movies_seq') PRIMARY KEY,
    title VARCHAR(100) NOT NULL,
    year INT CHECK (year > 1900)
);
```

- Insert several rows and display the generated movie_id values.
- Use SELECT currval('movies_seq'); to check the current sequence value.

6.5.4. Self-Join Practice

- Extend the employees table by ensuring the reports_to column references employee_id.
- Write a query to list:
 - Each employee's full name, title, and their manager's full name.
 - Employees who have no manager should still appear in the result.

```
SELECT e.employee_id, e.first_name, e.last_name,
       m.first_name AS manager_first_name, m.last_name AS
manager_last_name
FROM employees e
LEFT JOIN employees m ON e.reports_to = m.employee_id;
```

6.5.5. Views and Data Security

- Create a view named employee_public_info that shows only non-sensitive employee data:

```
CREATE VIEW employee_public_info AS
SELECT employee_id, first_name, last_name, title, city,
country
FROM employees;
```

- Query the view to confirm it returns the expected columns.
- Grant SELECT privileges on the view to a new role public_viewer.
- Attempt to query the base table employees using that role and verify access is restricted.

6.5.6. Working with BLOBs

- In the categories table, add or replace an image for one row using DBeaver:
 - Right-click → Edit → Load from file.
 - Choose a .jpg or .png file and save.
- Test displaying the image in DBeaver's Value View Panel (F7).
- Try saving the image back to a local file and document the process.

6.5.7. Advanced - Custom Database Design

Design and implement a small movie database inspired by IMDb, including the following:

- Tables:
 - movies (movie_id, title, release_year, genre_id, rating)
 - genres (genre_id, name)
 - actors (actor_id, name, birth_year)
 - movie_cast (movie_id, actor_id, role)
- Constraints:
 - All IDs are SERIAL PRIMARY KEY.
 - Foreign keys with ON DELETE CASCADE.
 - CHECK (rating BETWEEN 0 AND 10).
- Views:
 - topRated_movies showing only movies with rating > 8.
 - actors_in_multiple_movies listing actors appearing in more than 3 movies.
- Bonus:
 - Add a sequence manually instead of using SERIAL.
 - Use a self-join to show actors who have appeared together in the same movie.
 - Export one of your views as a CSV file from DBeaver.

Laboratory no. 7

Database Design and Backup in PostgreSQL

7.1. Objectives

The objectives of this laboratory are to deepen students' understanding of database design principles, conceptual modeling, and data preservation techniques using PostgreSQL.

By the end of this session, students should be able to:

- Understand the database design process, including conceptual, logical, and physical design stages.
- Describe and model real-world systems using the Entity–Relationship (E-R) model.
- Identify entities, attributes, relationships, and mapping cardinalities in a system.
- Differentiate between strong and weak entities, and represent them properly in an E-R diagram.
- Explain and apply normalization concepts to minimize redundancy and ensure data integrity.
- Select appropriate PostgreSQL data types (numeric, character, date/time) when designing schemas.
- Use serial and sequence data types to generate surrogate keys automatically.
- Design and implement a university database in PostgreSQL using E-R concepts and suitable data types.
- Perform database backup and restoration using PostgreSQL utilities (pg_dump, pg_restore, psql).
- Understand the difference between plain-text, custom, and tar backup formats and their use cases.
- Integrate backup strategies as part of good database administration practices.

7.2. Theoretical Foundation

7.2.1. Database Design Overview

Database design is the process of defining the structure, storage, and relationships of data to support a specific set of applications.

It consists of three major stages:

- Conceptual Design — identifying entities, attributes, and relationships using high-level models such as the Entity–Relationship (E–R) model.
- Logical Design — translating the conceptual model into relational tables, primary and foreign keys, and constraints.

- Physical Design — defining storage details, indexes, partitions, and optimization techniques within a specific DBMS (e.g., PostgreSQL).

A well-designed database minimizes redundancy, prevents inconsistencies, and improves query performance.

7.2.2. The Entity–Relationship (E–R) Model

The E–R model is a conceptual data model used to represent the structure of data in a system. It identifies:

- Entity sets – real-world objects (e.g., Student, Course, Instructor).
- Attributes – properties of entities (e.g., name, ID, birth date).
- Relationship sets – associations between entities (e.g., “enrolls in”, “teaches”).

E–R diagrams are graphical representations of this model and are widely used as the first step in designing relational databases.

Entities and Attributes

Entities:

- An entity is a distinguishable object or concept, such as a student or a department.
- Entities are grouped into entity sets that share the same properties.
- Each entity has a set of attributes that describe it.

Attributes can be:

- Simple: indivisible (e.g., age, salary).
- Composite: composed of sub-parts (e.g., name → first_name, last_name).
- Derived: calculated from other attributes (e.g., age derived from birth_date).
- Null: representing missing or unknown data.

Each attribute has a domain, which defines the permissible values (e.g., rating between 1–10).

7.2.3. Relationships and Relationship Sets

A relationship represents an association among entities.

For instance, the relationship advisor connects an Instructor and a Student entity.

A relationship set is a collection of relationships of the same type (e.g., all advisor–student pairs).

Relationships can have their own attributes (called descriptive attributes) — e.g., grade in the takes relationship between Student and Course.

7.2.4. Mapping Cardinalities

Cardinalities define how many entities of one set can be associated with entities of another set:

Type	Description
One-to-One (1:1)	Each entity in A is linked to at most one in B, and vice versa.
One-to-Many (1:N)	One entity in A can relate to multiple entities in B.
Many-to-One (N:1)	Many entities in A relate to one entity in B.
Many-to-Many (M:N)	Entities in both A and B can have multiple associations.

Participation can be:

- Total (Mandatory) – every entity participates in at least one relationship.
- Partial (Optional) – some entities may not participate.

Example:

- Every student must have an advisor → total participation.
- Some instructors may have no students → partial participation.

Weak and Strong Entities

- A strong entity has its own primary key.
- A weak entity depends on another entity (called the owner) for identification.
 - It cannot exist without the related strong entity.
 - It uses a discriminator (partial key) combined with the owner's key for uniqueness.

Example:

- Section is a weak entity identified by Course_ID (from Course) and a discriminator such as Section_No.

In E–R diagrams:

- Strong entities are represented by rectangles.
- Weak entities are represented by double rectangles.
- Identifying relationships are shown with double diamonds.

7.2.5. Normalization and Redundancy Avoidance

Two common design pitfalls:

- Redundancy — storing the same data in multiple places (e.g., repeating course title in each section).
- Incompleteness — missing important relationships or attributes.

Normalization techniques (1NF, 2NF, 3NF, BCNF) help remove redundancy and ensure that:

- Each attribute depends on the key, the whole key, and nothing but the key.
- Data updates and deletions do not cause anomalies.

7.2.6. PostgreSQL Data Types

When implementing the logical schema, selecting appropriate data types is crucial for performance and integrity.

Numeric Types:

- smallint, integer, bigint - whole numbers of various sizes.
- numeric or decimal - exact precision for financial data.
- real, double precision - approximate values for scientific calculations.
- serial / bigserial - auto-incrementing integer keys backed by sequences.

Note: Numeric types are slower but precise; use them when exact values matter.

Character Types:

- char(n) - fixed-length string (padded with spaces).
- varchar(n) - variable-length string up to n characters.
- text - unlimited length (recommended for general text fields).

In PostgreSQL, performance is identical across these types; text is often preferred for flexibility.

Date and Time Types

- date - stores a calendar date.
- timestamp [with|without time zone] — stores date and time values.
- time [with|without time zone] — time only.
- interval - duration (e.g., 3 days, 2 hours).

PostgreSQL stores timestamps internally in UTC.

Use timestamp with time zone when working with multiple geographic regions.

7.2.7. Backup and Recovery in PostgreSQL

Data protection is a critical part of database management.

Backup and recovery ensure that your work can be restored in case of hardware failure, corruption, or accidental deletion.

Backup

The primary tool for logical backup is `pg_dump`, which creates a portable dump of a single database.

Basic command:

```
pg_dump -U postgres dbname > backup_file.sql
```

Useful options:

Option	Description
<code>-F c</code>	Custom compressed format
<code>-F t</code>	Tar format
<code>--inserts</code>	Dumps data as INSERT statements (portable but slower)
<code>-s</code>	Dumps schema only (no data)
<code>-v</code>	Verbose mode

Example:

```
pg_dump -U postgres --inserts northwind > northwind.sql
```

7.2.8. Restoration

To restore a plain-text backup

```
createdb -U postgres newdb
```

```
psql -U postgres -f backup_file.sql newdb
```

To restore a custom or tar backup

```
pg_restore -U postgres -d newdb -F c backup_file.backup
```

Common options:

- ``- a`` – data only
- ``- s`` – schema only
- ``- C`` – create database before restoring
- ``- c`` – drop existing objects before creation

7.2.9. Best Practices

- Always back up before major schema changes or upgrades.
- Store backups on external or cloud storage.
- Test restoration regularly.
- Automate scheduled backups for production environments.

- Remember: minor PostgreSQL versions are backward-compatible, but major versions are not.

7.3. Lab Exercises

In this laboratory, you will design and implement a PostgreSQL database using the Entity–Relationship model, apply appropriate data types, and perform backup and restoration operations.

You may use DBeaver or the psql console to execute the exercises.

7.3.1. Conceptual Modeling – E-R Diagram

Objective: Model a real-world university database.

- Identify and draw the E–R diagram for a university system containing the following entities:
 - Department (dept_id, dept_name, building, budget)
 - Instructor (instr_id, name, salary, dept_id)
 - Student (stud_id, name, age, major_dept_id)
 - Course (course_id, title, credits, dept_id)
 - Section (section_id, semester, year, course_id)
 - Takes (relationship between Student and Section, with attribute grade)
 - Advisor (relationship between Instructor and Student)
- Represent the following relationships:
 - Each student is advised by one instructor (1:N).
 - Each instructor belongs to exactly one department (N:1).
 - Each course belongs to exactly one department (N:1).
 - Each section is part of one course but may differ by semester/year (weak entity).
 - Students enroll in sections (M:N relationship via takes).
- Specify the cardinalities (1:1, 1:N, M:N) and participation (total/partial) on the diagram.
- Tip: Use any ER modeling tool (e.g., draw.io, Lucidchart, DBeaver Diagram Editor).

7.3.2. Logical Design – Implementing the Schema in PostgreSQL

Translate the E–R model into relational tables.

Example SQL script:

```
CREATE TABLE department (
    dept_id SERIAL PRIMARY KEY,
    dept_name VARCHAR(40) NOT NULL UNIQUE,
    building VARCHAR(30),
    budget NUMERIC(12,2) CHECK (budget >= 0)
```

);

```
CREATE TABLE instructor (  
    instr_id SERIAL PRIMARY KEY,  
    name VARCHAR(50) NOT NULL,  
    salary NUMERIC(10,2) CHECK (salary >= 0),  
    dept_id INT REFERENCES department(dept_id)  
        ON DELETE SET NULL ON UPDATE CASCADE  
);
```

```
CREATE TABLE student (  
    stud_id SERIAL PRIMARY KEY,  
    name VARCHAR(50) NOT NULL,  
    age INT CHECK (age > 15),  
    major_dept_id INT REFERENCES department(dept_id)  
);
```

```
CREATE TABLE course (  
    course_id SERIAL PRIMARY KEY,  
    title VARCHAR(60) NOT NULL,  
    credits INT CHECK (credits > 0),  
    dept_id INT REFERENCES department(dept_id)  
);
```

```
CREATE TABLE section (  
    section_id SERIAL PRIMARY KEY,  
    semester VARCHAR(10),  
    year INT CHECK (year >= 2000),  
    course_id INT REFERENCES course(course_id)  
);
```

```
CREATE TABLE takes (  
    stud_id INT REFERENCES student(stud_id),  
    section_id INT REFERENCES section(section_id),  
    grade CHAR(2),  
    PRIMARY KEY (stud_id, section_id)  
);
```

```
CREATE TABLE advisor (  
    stud_id INT REFERENCES student(stud_id),  
    instr_id INT REFERENCES instructor(instr_id),  
    PRIMARY KEY (stud_id)  
);
```

7.3.3. Inserting Sample Data

Insert a small dataset to set your schema.

```
INSERT INTO department (dept_name, building, budget)  
VALUES ('Computer Science', 'IT Hall', 500000),  
       ('Mathematics', 'Science Block', 300000);
```

```
INSERT INTO instructor (name, salary, dept_id)  
VALUES ('Dr. Miller', 80000, 1),  
       ('Dr. James', 75000, 2);
```

```
INSERT INTO student (name, age, major_dept_id)  
VALUES ('Alice', 20, 1),  
       ('Bob', 21, 2);
```

```
INSERT INTO course (title, credits, dept_id)  
VALUES ('Databases', 5, 1),  
       ('Calculus', 4, 2);
```

```
INSERT INTO section (semester, year, course_id)
VALUES ('Fall', 2025, 1),
      ('Spring', 2026, 2);
```

```
INSERT INTO takes (stud_id, section_id, grade)
VALUES (1, 1, 'A'), (2, 2, 'B');
```

```
INSERT INTO advisor (stud_id, instr_id)
VALUES (1, 1), (2, 2);
```

7.3.4. Query Practice

-- Ex. 1 List all students with their advisors:

```
SELECT s.name AS student, i.name AS advisor
FROM student s
JOIN advisor a ON s.stud_id = a.stud_id
JOIN instructor i ON i.instr_id = a.instr_id;
```

-- Ex. 2 Show each department and the number of instructors in it:

```
SELECT d.dept_name, COUNT(i.instr_id) AS instructor_count
FROM department d
LEFT JOIN instructor i ON d.dept_id = i.dept_id
GROUP BY d.dept_name;
```

-- Ex. 3 Display students and the courses they are taking:

```
SELECT s.name AS student, c.title AS course
FROM takes t
JOIN student s ON t.stud_id = s.stud_id
JOIN section sec ON t.section_id = sec.section_id
JOIN course c ON sec.course_id = c.course_id;
```

-- Ex. 4 Find the instructor(s) advising the highest number of students:

```
SELECT i.name, COUNT(a.stud_id) AS advisee_count
FROM advisor a
JOIN instructor i ON a.instr_id = i.instr_id
GROUP BY i.name
ORDER BY advisee_count DESC;
```

7.3.5. Backup and Restore Operations

Backup the database

-- Use the `pg_dump` command to create a backup of your university database:

```
pg_dump -U postgres university > university_backup.sql
```

-- For a compressed binary format:

```
pg_dump -U postgres -F c -f university.backup university
```

Restore the Database

-- Create a new database to test the restore:

```
createdb -U postgres university_restored
```

-- Restore from a plain-text backup:

```
psql -U postgres -f university_backup.sql university_restored
```

-- Restore from a custom format backup:

```
pg_restore -U postgres -d university_restored -F c
university.backup
```

7.4. Troubleshooting

Designing and managing databases involves several steps that can lead to errors during schema creation, data insertion, or backup and restoration.

This section summarizes the most common issues encountered during database design, PostgreSQL implementation, and backup management, along with suggested solutions.

7.4.5. Foreign Key Constraint Violations

- Error:
 - ERROR: insert or update on table "student" violates foreign key constraint
- Cause:

- The referenced key does not exist in the parent table (e.g., inserting a student with a non-existent department ID).
- Solution:
 - Always insert data in referential order — parent (department) first, child (student) second.
- Check parent table values:

```
SELECT * FROM department;
```

If needed, temporarily remove and re-add the constraint:

```
ALTER TABLE student DROP CONSTRAINT
student_major_dept_id_fkey;
```

```
ALTER TABLE student ADD CONSTRAINT student_major_dept_id_fkey
FOREIGN KEY (major_dept_id) REFERENCES department(dept_id);
```

7.4.6. Primary Key or Unique Constraint Violations

- Error:
 - ERROR: duplicate key value violates unique constraint "course_pkey"
- Cause:
 - Attempting to insert a row with a duplicate primary key or unique value.
- Solution:
 - Verify uniqueness before inserting:

```
SELECT course_id FROM course WHERE course_id = <value>;
```

- Use SERIAL or GENERATED ALWAYS AS IDENTITY to ensure automatic key assignment.

7.4.7. CHECK Constraint Failures

- Error:
 - ERROR: new row violates check constraint "instructor_salary_check"
- Cause:
 - The inserted value does not satisfy the logical condition (salary >= 0).
- Solution:
 - Review the constraint definition:

```
\d instructor
```

- Update data to meet the rule:

```
UPDATE instructor SET salary = 1000 WHERE salary < 0;
```

7.4.8. Type Mismatch or Casting Errors

- Error:
 - ERROR: column "credits" is of type integer but expression is of type text
- Cause:
 - Inserted data type does not match the column type.
- Solution:
 - Ensure proper literal formats:

```
INSERT INTO course (title, credits) VALUES ('Databases',  
4);
```

- Use casting if needed:

```
INSERT INTO course (credits) VALUES ('4'::INT);
```

7.4.9. Sequence Synchronization Problems

- Error:
 - ERROR: duplicate key value violates unique constraint "student_pkey"
- Cause:
 - The sequence value is behind existing data (common after data import).
- Solution:
 - Reset the sequence to the current maximum value:

```
SELECT      setval('student_stud_id_seq',      (SELECT  
MAX(stud_id) FROM student));
```

7.4.10. Table or Column Does Not Exist

- Error:
 - ERROR: relation "advisor" does not exist
- Cause:
 - Trying to reference a table or column that was not created or has a typo.
- Solution:
 - Check existing objects:
`\dt`
- Review spelling and case sensitivity (PostgreSQL treats unquoted names as lowercase).
 - "Course" ≠ course

7.4.11. Permission Denied

- Error:
 - ERROR: permission denied for relation department
- Cause:

- The connected user lacks privileges on the table.
- Solution:
 - Grant privileges explicitly:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON department TO lab_user;
```

7.4.12. Backup/Restore Errors

- Error:
 - pg_restore: [archiver] input file appears to be a text format dump. Please use psql.
- Cause:
 - Attempting to restore a plain-text backup using pg_restore.
- Solution:
 - Use psql for text-based .sql dumps:

```
psql -U postgres -f university_backup.sql university
```

- Use pg_restore only for binary (.backup) or tar files:

```
pg_restore -U postgres -d university -F c university.backup
```

7.4.13. Encoding or Locale Issues During Restore

- Error:
 - ERROR: invalid byte sequence for encoding "UTF8"
- Cause:
 - Backup was created with a different encoding (e.g., Latin1).
- Solution:
 - Specify encoding during restore:

```
psql -U postgres --set=client_encoding=LATIN1 -f backup.sql newdb
```

- or recreate database with correct locale:

```
createdb -U postgres newdb --encoding=LATIN1 --lc-collate='en_US.UTF-8'
```

7.4.14. DBeaver-Specific Issues

- Case 1:
 - Problem: Cannot execute .backup file in DBeaver SQL Editor.
 - Cause: DBeaver only handles plain-text SQL scripts.
 - Solution: Convert binary backup to SQL using:

```
pg_restore -f dump.sql -F c university.backup
```

- Then open and run dump.sql in DBeaver.
- Case 2:
 - Problem: Schema missing after import.

- Cause: The target database did not include schema creation statements.
- Solution: Ensure you use:

```
pg_dump -U postgres -C university > university_full.sql
```

- or create the database manually before restore.

7.4.15. Common Logical Design Mistakes

Issue	Symptom	Fix
Missing relationships	Data inconsistency across tables	Add foreign keys
Wrong data types	Truncated or invalid values	Use proper PostgreSQL types (text, numeric, date)
Redundant attributes	Duplication of info	Normalize (2NF, 3NF)
Overuse of NULLs	Missing critical info	Define NOT NULL and CHECK constraints

7.4.16. Best Practices

- Always start with an E-R diagram before implementing tables.
- Use constraints to enforce integrity, not just application logic.
- Regularly back up your database, especially before altering schema.
- Use versioned SQL scripts to reproduce schema changes.
- Test your restore procedure — a backup is only useful if it can be restored successfully!

7.5. Proposed Exercises

The following exercises are designed to reinforce your understanding of conceptual modeling, normalization, PostgreSQL implementation, and backup strategies.

They should be completed after reading the theoretical background and performing the in-lab exercises.

For each exercise, submit both the E–R diagram and the corresponding SQL implementation (DDL + sample data).

7.5.5. Conceptual Design

Design an E–R diagram for a Library Management System that includes:

- Entities: Book, Author, Member, Loan, Publisher
- Relationships:
 - A book can have one or more authors.
 - A member can borrow multiple books.
 - Each loan records the date borrowed and date returned.
 - Each book belongs to a publisher.

- Identify primary keys, foreign keys, and cardinalities.
- Include at least one weak entity (e.g., Loan dependent on Member).

Deliverables:

- Draw the E–R diagram using a modeling tool (draw.io, DBeaver, Lucidchart).
- Describe the type of each relationship (1:1, 1:N, M:N).
- Specify participation (total or partial).

7.5.6. Logical Design and Normalization

Convert the E–R model from Exercise 1 into a relational schema using SQL CREATE TABLE statements.

Ensure:

- All relationships are correctly implemented with foreign keys.
- Composite keys are used where necessary (e.g., for Loan).
- The schema satisfies Third Normal Form (3NF).

Challenge:

- Explain any normalization steps performed ($1NF \rightarrow 2NF \rightarrow 3NF$) and justify attribute separation decisions.

7.5.7. Implementation in PostgreSQL

Implement your schema in PostgreSQL.

Insert at least 5–10 records per table.

Test the following queries:

- List all books borrowed by a specific member.
- Show the total number of books per author.
- Find members who have not returned any books yet.
- Display all publishers with their number of published books.
- Update the Loan table by adding a new column `late_fee` NUMERIC(5,2) with a default value of 0.00.

7.5.8. Data Types and Constraints

For the implemented tables, apply suitable data types and constraints:

- NOT NULL, CHECK, UNIQUE, DEFAULT
- Use SERIAL or GENERATED ALWAYS AS IDENTITY for primary keys.
- Use NUMERIC(10,2) for prices, and DATE or TIMESTAMP for time-based data.

Validate constraints by attempting invalid inserts (e.g., a negative fee or a missing date).

7.5.9. Advanced Query Practice

- Using the Library database, create SQL queries to:
- Display members who borrowed books from more than two authors.
- List all books that have never been borrowed.
- Retrieve the top 3 authors based on the number of distinct books written.
- Compute total late fees collected per month (using GROUP BY date_part('month', return_date)).

7.5.10. Backup and Recovery

Perform and document a complete backup and restore cycle for your Library database.

Backup:

```
pg_dump -U postgres -F c -f library.backup librarydb
```

Drop and recreate the database:

```
dropdb -U postgres librarydb
```

```
createdb -U postgres librarydb
```

Restore:

```
pg_restore -U postgres -d librarydb -F c library.backup
```

Verify restoration with:

```
SELECT COUNT(*) FROM book;
```

Deliverable:

- Take screenshots showing each step and result.

7.5.11. Extended Challenge (Optional)

Design and implement a Hospital Management Database including:

- Entities: Patient, Doctor, Department, Appointment, Treatment
- Requirements:
 - A patient can have multiple appointments.
 - Each appointment belongs to one doctor and one department.
 - Treatment depends on Appointment (weak entity).
- Implement the schema in PostgreSQL.
- Add one view that summarizes appointments per doctor per week.
- Create and test both plain SQL and custom-format backups.

Laboratory no. 8

Database Normalization and Data Consistency

8.1. Objectives

The main objective of this laboratory is to understand and apply the process of normalization in relational database design, ensuring data consistency, elimination of redundancy, and integrity maintenance.

Students will work with a dataset describing people and their automobiles, normalize the schema into Third Normal Form (3NF), and test the resulting design in PostgreSQL.

By the end of this lab, students should be able to:

- Understand the purpose and importance of database normalization.
- Identify anomalies that occur in unnormalized or partially normalized databases.
- Distinguish between 1NF, 2NF, and 3NF, and understand when each form applies.
- Apply normalization techniques to transform a denormalized dataset into 3rd Normal Form.
- Create new tables and define foreign key relationships for normalized entities.
- Use SQL commands to migrate data from unnormalized tables to normalized structures.
- Implement and enforce referential integrity between related tables.
- Use Mockaroo or a similar data generator to populate tables with thousands of test records.
- Create views that combine normalized data into readable denormalized formats for analysis.
- Execute SQL queries to explore data relationships in the normalized schema.
- Understand the trade-offs between normalization (data integrity) and performance (query speed).

8.2. Theoretical Foundation

8.2.1. Introduction to Database Normalization

Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity.

Its primary goal is to ensure that each piece of information is stored only once, making updates, inserts, and deletions more efficient and consistent.

A database that is not normalized can lead to:

Redundancy – the same information stored multiple times.

- Update anomalies – modifying data in one place but not everywhere.
- Insert anomalies – inability to insert data because of missing related values.
- Delete anomalies – accidental loss of important data when deleting related records.

Normalization transforms a large, unstructured table into a set of smaller, well-structured tables connected through foreign keys.

8.2.2. Anomalies in Unnormalized Databases

Example: A single table `person_cars` stores both personal and vehicle information:

person_id	name	car_model	color	plate_number
1	John	Audi A4	Red	AB-100
2	Maria	Toyota Yaris	Blue	CD-200
3	John	BMW 320i	Red	EF-300

Problems:

- Redundancy: John’s name and color “Red” appear multiple times.
- Update anomaly: If John’s name changes, we must update every occurrence.
- Insertion anomaly: We cannot insert a new person without a car.
- Deletion anomaly: Deleting all of John’s cars would remove his personal data too.

8.2.3. The Normalization Process

Normalization occurs in stages, called normal forms (NF).

Each stage builds upon the previous one, addressing specific structural issues.

8.2.3.1. First Normal Form (1NF)

A relation is in 1NF if:

- Each column contains atomic (indivisible) values.
- Each record is unique (no duplicate rows).
- There are no repeating groups or arrays.

Transforming `person_cars` to 1NF:

- Ensure one car per row.
- Split multi-value fields (like colors or phones) into separate rows or tables.

Example:

```
CREATE TABLE person (
  person_id SERIAL PRIMARY KEY,
  name VARCHAR(50)
```

);

```
CREATE TABLE car (  
    car_id SERIAL PRIMARY KEY,  
    model VARCHAR(50),  
    color VARCHAR(20),  
    plate_number VARCHAR(10)  
);
```

8.2.3.2. Second Normal Form (2NF)

A relation is in 2NF if:

- It is already in 1NF.
- Every non-key attribute depends on the whole primary key, not just a part of it.

This issue appears in tables with composite primary keys.

Example:

- If a table `person_car(person_id, car_id, color)` exists and `color` depends only on `car_id`, not on both columns — that's a violation of 2NF.

Fix: Move the `color` attribute to the `car` table.

8.2.3.3. Third Normal Form (3NF)

A relation is in 3NF if:

- It is already in 2NF.
- There are no transitive dependencies — non-key attributes do not depend on other non-key attributes.

Example:

- If in the `car` table, the attribute `model` determines `color`, then `color` is transitively dependent on `model`, not the key (`car_id`).

Fix: Separate the `model` and `color` data:

```
CREATE TABLE model (  
    model_id SERIAL PRIMARY KEY,  
    model_name VARCHAR(50)  
);
```

```
CREATE TABLE color (
    color_id SERIAL PRIMARY KEY,
    color_name VARCHAR(20)
);
```

```
ALTER TABLE car
ADD COLUMN model_id INT REFERENCES model(model_id),
ADD COLUMN color_id INT REFERENCES color(color_id);
```

Example: From Unnormalized Data to 3NF

Unnormalized table:

person_name	car_model	color	plate_number
John	Audi A4	Red	AB-100
Maria	Toyota Yaris	Blue	CD-200
John	BMW 320i	Red	EF-300

Step 1 – 1NF:

Separate repeated groups into rows; ensure atomicity.

Step 2 – 2NF:

Isolate data depending only on part of a composite key — create person, car, and ownership tables.

Step 3 – 3NF:

Eliminate transitive dependencies — move color and model into their own tables, referenced by foreign keys.

Resulting schema:

- person(person_id, name)
- color(color_id, color_name)
- model(model_id, model_name)
- car(car_id, model_id, color_id, plate_number)
- ownership(person_id, car_id)

This ensures:

- No redundancy in names or colors.
- Updates are localized to one table.

- Referential integrity between entities.

8.2.4. Denormalization (Reverse Process)

Sometimes databases are denormalized to improve read performance, especially in reporting or data warehousing.

This involves combining tables or storing derived attributes.

Example:

```
CREATE VIEW person_car_view AS
SELECT p.name, m.model_name, c.color_name, car.plate_number
FROM person p
JOIN ownership o ON p.person_id = o.person_id
JOIN car ON car.car_id = o.car_id
JOIN model m ON car.model_id = m.model_id
JOIN color c ON car.color_id = c.color_id;
```

Views provide a denormalized perspective while maintaining normalized storage.

8.2.5. Advantages of Normalization

Benefit	Description
Eliminates redundancy	Each fact stored only once.
Improves consistency	Fewer update anomalies.
Simplifies maintenance	Changes in one place propagate safely.
Improves query accuracy	Relationships are well defined.
Supports referential integrity	Clear parent–child relations.

8.2.6. When to stop Normalizing

Over-normalization can reduce performance by requiring multiple joins.

Practical systems often stop at 3NF or BCNF (Boyce-Codd Normal Form).

Further normalization (4NF, 5NF) is used in specialized cases like multi-valued dependencies.

8.2.7. Tools and Resources

- Mockaroo – for generating realistic random datasets (<https://mockaroo.com>)
- DBeaver ER Diagram Generator – for visualizing relationships between normalized tables.
- pgAdmin or psql – for verifying constraints and dependencies using:

\d table_name

8.3. Lab Exercises

In this laboratory, you will normalize a denormalized dataset step-by-step, transforming it into Third Normal Form (3NF) and then testing the final design in PostgreSQL using DBeaver.

You will also learn how to populate your tables with generated data using Mockaroo and verify that constraints enforce data integrity.

8.3.1. Starting from an Unnormalized Table

--Objective: Begin with a flat structure that mixes multiple entities (people and cars) in a single table.

-- Create a database named lab8db:

```
CREATE DATABASE lab8db;
```

-- Create an unnormalized table person_car:

```
CREATE TABLE person_car (  
    person_id SERIAL PRIMARY KEY,  
    person_name VARCHAR(50),  
    car_model VARCHAR(50),  
    car_color VARCHAR(20),  
    plate_number VARCHAR(10)  
);
```

-- Insert sample data:

```
INSERT INTO person_car (person_name, car_model, car_color,  
plate_number)
```

```
VALUES
```

```
('John', 'Audi A4', 'Red', 'AB-100'),  
('Maria', 'Toyota Yaris', 'Blue', 'CD-200'),  
('John', 'BMW 320i', 'Red', 'EF-300');
```

-- Display the current dataset:

```
SELECT * FROM person_car;
```

-- Observe that both John and Red appear multiple times – indicating redundancy.

8.3.2. Step 1 – Convert to First Normal Form

-- Goal: Ensure atomic values and eliminate repeating groups.
-- Verify there are no multi-value cells (e.g., lists of cars or colors).

-- Each car entry now represents one record – so 1NF is already achieved.

-- Add a unique constraint to avoid duplicate license plates:

```
ALTER TABLE person_car
```

```
ADD CONSTRAINT unique_plate UNIQUE (plate_number);
```

-- The table now satisfies 1NF – all attributes are atomic and rows are unique.

8.3.3. Step 2 – Convert to Second Normal Form

-- Goal: Remove partial dependencies – each non-key attribute must depend on the whole key.

-- Since the primary key person_id does not combine multiple columns, we must instead separate person and car information.

-- Create separate tables:

```
CREATE TABLE person (  
    person_id SERIAL PRIMARY KEY,  
    name VARCHAR(50)  
);
```

```
CREATE TABLE car (  
    car_id SERIAL PRIMARY KEY,  
    model VARCHAR(50),  
    color VARCHAR(20),  
    plate_number VARCHAR(10) UNIQUE  
);
```

-- Create a relationship table to represent ownership:

```
CREATE TABLE ownership (  
    person_id INT REFERENCES person(person_id),
```

```

        car_id INT REFERENCES car(car_id),
        PRIMARY KEY (person_id, car_id)
    );
-- Migrate data manually:
-- Insert distinct people:
INSERT INTO person (name)
SELECT DISTINCT person_name FROM person_car;
-- Insert distinct cars:
INSERT INTO car (model, color, plate_number)
SELECT DISTINCT car_model, car_color, plate_number FROM
person_car;
-- Recreate ownership relationships:
INSERT INTO ownership (person_id, car_id)
SELECT p.person_id, c.car_id
FROM person_car pc
JOIN person p ON p.name = pc.person_name
JOIN car c ON c.plate_number = pc.plate_number;
-- Verify the structure:
SELECT * FROM person;
SELECT * FROM car;
SELECT * FROM ownership;
-- The design is now in 2NF – all non-key attributes depend
only on full primary keys.
8.3.4. Step 3 – Convert to Third Normal Form
-- Goal: Remove transitive dependencies – no attribute should
depend on another non-key attribute.
-- Create tables for repeating descriptive data (model,
color):
CREATE TABLE color (
    color_id SERIAL PRIMARY KEY,
    color_name VARCHAR(20)

```

```
);
```

```
CREATE TABLE model (  
    model_id SERIAL PRIMARY KEY,  
    model_name VARCHAR(50)  
);
```

```
-- Modify the car table to reference those:
```

```
ALTER TABLE car  
ADD COLUMN model_id INT REFERENCES model(model_id),  
ADD COLUMN color_id INT REFERENCES color(color_id);
```

```
-- Migrate data:
```

```
INSERT INTO color (color_name)  
SELECT DISTINCT color FROM car;
```

```
INSERT INTO model (model_name)  
SELECT DISTINCT model FROM car;
```

```
UPDATE car
```

```
SET color_id = (SELECT color_id FROM color WHERE  
color.color_name = car.color),  
model_id = (SELECT model_id FROM model WHERE  
model.model_name = car.model);
```

```
-- Remove redundant columns:
```

```
ALTER TABLE car  
DROP COLUMN model,  
DROP COLUMN color;
```

```
-- The database now satisfies 3NF – all non-key columns depend  
directly on the key (car_id), not indirectly.
```

8.3.5. Step 4 – Verify the Normalized Schema

```
-- Check final relationships:
\d person
\d car
\d color
\d model
\d ownership
-- The structure should show:
-- Foreign key from car → model
-- Foreign key from car → color
-- Foreign keys in ownership linking person and car
```

8.3.6. Step 5 – Populating Data with Mockaroo

```
-- Go to https://mockaroo.com
-- Generate data for:
-- person (fields: name)
-- model and color (string values)
-- car (model_id, color_id, plate_number)
-- ownership (person_id, car_id)
-- Download each dataset as a .csv file.
-- Import into DBeaver using:
-- Right-click on table → Import Data → CSV
-- Match column names correctly.
-- Tip: PostgreSQL accepts CSV imports via the COPY command:
COPY person(name) FROM '/path/person.csv' DELIMITER ',' CSV
HEADER;
```

8.3.7. Step 6 – Testing the Normalized Design

```
-- Run the following queries to validate data consistency:
-- List all people and their cars:
SELECT p.name, m.model_name, c.color_name, car.plate_number
FROM ownership o
```

```

JOIN person p ON o.person_id = p.person_id
JOIN car ON o.car_id = car.car_id
JOIN model m ON car.model_id = m.model_id
JOIN color c ON car.color_id = c.color_id;
-- Count how many cars each person owns:
SELECT p.name, COUNT(o.car_id) AS total_cars
FROM ownership o
JOIN person p ON o.person_id = p.person_id
GROUP BY p.name
ORDER BY total_cars DESC;
-- Find which color appears most often:
SELECT c.color_name, COUNT(*) AS count
FROM car
JOIN color c ON c.color_id = car.color_id
GROUP BY c.color_name
ORDER BY count DESC;

```

8.3.8. Step 7 – Creating a View for Reporting

```

-- To visualize denormalized information:
CREATE VIEW person_car_view AS
SELECT p.name AS person,
       m.model_name AS model,
       c.color_name AS color,
       car.plate_number
FROM person p
JOIN ownership o ON o.person_id = p.person_id
JOIN car ON o.car_id = car.car_id
JOIN model m ON car.model_id = m.model_id
JOIN color c ON car.color_id = c.color_id;
-- Query the view:

```

```
SELECT * FROM person_car_view LIMIT 10;
```

-- This view provides an easy-to-read combined dataset while maintaining the integrity of the normalized schema.

8.4. Troubleshooting

During the normalization process and implementation in PostgreSQL, you may encounter several issues related to constraints, data migration, and CSV imports.

This section summarizes the most common errors, their causes, and practical solutions.

8.4.1. Foreign Key Constraint Violations

- Error:
 - ERROR: insert or update on table "ownership" violates foreign key constraint "ownership_person_id_fkey"
- Cause:
 - The record being inserted references a person_id or car_id that does not exist in the parent table.
 - Data migration order is incorrect (child inserted before parent).
- Solution:
 - Always insert data into parent tables first (person, car) and child tables later (ownership).
 - Use the following to check referential consistency:

```
SELECT o.*
```

```
FROM ownership o
```

```
LEFT JOIN person p ON p.person_id = o.person_id
```

```
WHERE p.person_id IS NULL;
```

- Delete invalid rows or correct the IDs before reapplying constraints.

8.4.2. Duplicate Key Errors

- Error:
 - ERROR: duplicate key value violates unique constraint "car_plate_number_key"
- Cause:
 - Two or more rows have the same license plate number, violating the unique constraint.
- Solution:
 - Identify duplicates:

```
SELECT plate_number, COUNT(*)
```

```
FROM car
```

```
GROUP BY plate_number
```

HAVING COUNT(*) > 1;

- Delete or update redundant rows before re-inserting data.

8.4.3. Data Type Mismatch

- Error:
 - ERROR: column "model_id" is of type integer but expression is of type text
- Cause:
 - You are inserting or importing a string into a numeric foreign key column.
- Solution:
 - Convert or map string names to corresponding IDs before insertion:

UPDATE car

```
SET model_id = (SELECT model_id FROM model WHERE
model.model_name = car.model);
```

- Always verify column types with:

\d car

8.4.4. NULL Value Constraint Violations

- Error:
 - ERROR: null value in column "person_id" violates not-null constraint
- Cause:
 - Missing reference values during import or migration.
 - Data loaded out of order, leaving foreign keys uninitialized.
- Solution:
 - Insert data into parent tables first.
 - Use ALTER TABLE to temporarily allow NULL, then correct values:

```
ALTER TABLE ownership ALTER COLUMN person_id DROP NOT NULL;
```

8.4.5. Incorrect Data Import from CSV

- Problem:
 - Data imported from a .csv file appears truncated, misaligned, or missing.
- Common causes:
 - Wrong delimiter (use , not ;).
 - Header row not skipped or mismatched.
 - Quotation marks missing around text with commas.
- Solution:
 - Ensure your CSV uses correct settings:

- Delimiter: ,
 - Encoding: UTF-8
 - Quote: "
 - Header line: included
 - Use the correct import command:
 - COPY person(name)
 - FROM '/path/person.csv'
 - DELIMITER ',' CSV HEADER;
 - Or in DBeaver:
 - Right-click table → Import Data → CSV → Map columns properly.
- #### 8.4.6. Relationship Missing After Import
- Problem:
 - Data seems imported correctly, but queries return empty join results.
 - Cause:
 - Foreign key columns (person_id, car_id) are empty or mismatched.
 - Solution:
 - Rebuild the ownership table using matching logic:

```
INSERT INTO ownership (person_id, car_id)
SELECT p.person_id, c.car_id
FROM person_car pc
JOIN person p ON p.name = pc.person_name
JOIN car c ON c.plate_number = pc.plate_number;
```

- Always validate with:

```
SELECT COUNT(*) FROM ownership;
```

8.4.7. View Creation Errors

- Error:
 - ERROR: column "model_name" does not exist
- Cause:
 - A column was renamed or removed in the base table after the view was created.
- Solution:
 - Drop and recreate the view:

```
DROP VIEW person_car_view;
CREATE VIEW person_car_view AS
SELECT p.name, m.model_name, c.color_name, car.plate_number
FROM person p
JOIN ownership o ON o.person_id = p.person_id
```

```
JOIN car ON o.car_id = car.car_id
JOIN model m ON car.model_id = m.model_id
JOIN color c ON car.color_id = c.color_id;
```

8.4.8. Case Sensitivity Issues

- Problem:
 - PostgreSQL returns: ERROR: column "Person_Name" does not exist
- Cause:
 - Identifiers created with mixed case and double quotes become case-sensitive.
- Solution:
 - Avoid quotes in table/column creation:

```
CREATE TABLE person (name VARCHAR(50));
```

- If case sensitivity is necessary, always use double quotes consistently:

```
SELECT "Person_Name" FROM "Person";
```

8.4.9. Constraint Naming Conflicts

- Error:
 - ERROR: constraint "unique_plate" for relation "car" already exists
- Cause:
 - You attempted to recreate an existing constraint with the same name.
- Solution:
 - Drop it first:

```
ALTER TABLE car DROP CONSTRAINT IF EXISTS unique_plate;
```

- Then recreate:

```
ALTER TABLE car ADD CONSTRAINT unique_plate UNIQUE
(plate_number);
```

8.4.10. DBeaver Import or Encoding Errors

- Problem:
 - Data imported from CSV shows "???" or corrupted characters (especially for names with accents).
- Cause:
 - CSV file not saved in UTF-8 encoding.
- Solution:

- Open file in a text editor (e.g., VS Code or Notepad++).
- Convert encoding to UTF-8 and re-import.
- In DBeaver import wizard → choose File Encoding: UTF-8.

8.4.11. Logical Design Mistakes

Issue	Symptom	Fix
Too many attributes in one table	Repetition of data	Split into multiple related tables
Missing relationships	Broken joins or NULL foreign keys	Define explicit FOREIGN KEY constraints
Overuse of NULLs	Inconsistent data	Apply NOT NULL or CHECK constraints
No indexing	Slow queries	Add indexes on FK and frequently filtered columns

8.4.12. Best Practices

- Always analyze data dependencies before splitting tables.
- Enforce referential integrity early using foreign keys.
- Name constraints explicitly (e.g., fk_ownership_person).
- Keep a backup before running mass updates or drops.
- Test imports on a copy of the database first.
- Use views to simplify joins and reports.
- Check constraint status with:

```
SELECT conname, convalidated FROM pg_constraint WHERE
convalidated = false;
```

8.5. Proposed Exercises

The following exercises are designed to help you consolidate your understanding of database normalization, data dependencies, and schema refinement.

Work in PostgreSQL using DBeaver or pgAdmin, and document your work with screenshots and SQL scripts.

8.5.1. Conceptual Design and Anomaly Detection

You are given the following unnormalized dataset representing employee project assignments:

EmpName	Department	Project	Manager	HoursWorked
Alice	IT	CRM System	John	40

Bob	IT	CRM System	John	35
Alice	IT	Inventory	Mike	20
Carol	HR	Recruitment	Laura	30

Tasks:

- Identify the data anomalies present (update, insertion, deletion).
- Determine which attributes are dependent on others.
- Draw an initial E–R diagram based on this table.
- Write down functional dependencies (FDs) such as:
 - EmpName $\rightarrow$ Department
 - Project $\rightarrow$ Manager
 - (EmpName, Project) $\rightarrow$ HoursWorked

8.5.2. Normalization to 3NF

Normalize the dataset from Exercise 1 step-by-step:

- Decompose it into 1NF – ensure atomic values and unique rows.
- Derive 2NF – remove partial dependencies.
- Derive 3NF – remove transitive dependencies.

For each step:

- Show the resulting tables and their primary keys.
- Write the SQL DDL statements for each table (CREATE TABLE).
- Document each foreign key that enforces referential integrity.

8.5.3. Implementation in PostgreSQL

Implement the normalized schema in PostgreSQL.

- Create tables:
 - employee(emp_id, emp_name, dept_id)
 - department(dept_id, dept_name)
 - project(project_id, project_name, manager_id)
 - assignment(emp_id, project_id, hours_worked)
- Insert test data (minimum 8–10 rows).
- Test join queries such as:

```
SELECT e.emp_name, d.dept_name, p.project_name, p.manager_id,
a.hours_worked
FROM assignment a
JOIN employee e ON e.emp_id = a.emp_id
JOIN department d ON d.dept_id = e.dept_id
JOIN project p ON p.project_id = a.project_id;
```

- Verify that no redundancy exists by checking duplicate counts:

```
SELECT emp_name, COUNT(*) FROM employee GROUP BY emp_name
HAVING COUNT(*) > 1;
```

8.5.4. Using Mockaroo to Populate Data

Use Mockaroo to generate large datasets for your normalized schema.

- Create realistic data for:
 - employee (names, emails, department IDs)
 - department (names)
 - project (names, managers)
 - assignment (emp_id, project_id, hours_worked)
- Export to .csv files and import into PostgreSQL using DBeaver or:

```
COPY employee(emp_name, dept_id)
FROM '/path/employee.csv'
DELIMITER ',' CSV HEADER;
```

- Run aggregation queries such as:

```
SELECT d.dept_name, SUM(a.hours_worked)
FROM assignment a
JOIN employee e ON e.emp_id = a.emp_id
JOIN department d ON d.dept_id = e.dept_id
GROUP BY d.dept_name;
```

8.5.5. Denormalization for Reporting

- Create a view that presents a denormalized summary of employees and projects:

```
CREATE VIEW employee_project_summary AS
SELECT e.emp_name, d.dept_name, p.project_name, p.manager_id,
a.hours_worked
FROM assignment a
JOIN employee e ON e.emp_id = a.emp_id
JOIN department d ON d.dept_id = e.dept_id
JOIN project p ON p.project_id = a.project_id;
```

- Query the view:

```
SELECT * FROM employee_project_summary LIMIT 10;
```

- Then export the view's output as a .csv report.

8.5.6. Normalization Analysis

Given the table below describing student exam records:

Student	Group	Course	Teacher	Grade
Ana	101	DB	Popescu	10
Mihai	101	DB	Popescu	9
Ana	101	AI	Ionescu	8

- Identify functional dependencies.
- Normalize up to Third Normal Form (3NF).
- Draw the E–R diagram of the resulting design.
- Create the corresponding PostgreSQL schema.
- Write a query that displays each student with the average grade per teacher.

8.5.7. Challenges

Design and normalize a Sales Management System.

Include:

- Entities: Customer, Product, Invoice, Invoice_Detail, Salesperson.
- Normalize to 3NF.
- Implement in PostgreSQL.
- Insert sample data and generate a sales summary view:

```
CREATE VIEW total_sales AS
```

```
SELECT c.name AS customer, SUM(id.quantity * id.price) AS  
total
```

```
FROM invoice_detail id
```

```
JOIN invoice i ON id.invoice_id = i.invoice_id
```

```
JOIN customer c ON i.customer_id = c.customer_id
```

```
GROUP BY c.name;
```

Laboratory no. 9

Specialized Database Modeling: Specialization and Generalization

9.1. Objectives

The objective of this laboratory is to introduce students to the advanced modeling concepts of specialization and generalization in the Entity–Relationship (E-R) model, and their implementation in relational databases.

These concepts extend the basic E-R model by allowing designers to represent hierarchical relationships between entity types and to model shared or distinct attributes more effectively.

By the end of this lab, students should be able to:

- Understand the conceptual difference between specialization and generalization.
- Identify and design ISA hierarchies (superclass–subclass relationships).
- Model entities that share common attributes through attribute inheritance.
- Distinguish between disjoint and overlapping specializations.
- Create E-R diagrams that represent hierarchical entity structures.
- Translate specialization/generalization concepts into relational schemas.
- Implement foreign key constraints that maintain superclass–subclass relationships.
- Use views to represent combined data from superclass and subclass entities.
- Understand how inheritance of attributes and relationships works in E-R design.
- Design and implement practical university-based specialization databases (e.g., person, student, employee, instructor).
- Develop custom specialization models for other real-world systems (e.g., hospitals, organizations).

9.2. Theoretical Foundation

In database design, specialization and generalization extend the traditional Entity–Relationship (E–R) model to represent hierarchical relationships between entities that share common characteristics.

These techniques help organize data models more efficiently when some entities are subtypes (specializations) of a more general supertype (generalization).

9.2.1. Generalization

Generalization is a bottom-up approach in which two or more entity types with similar attributes are combined into a single superclass (supertype).

Example:

- Suppose we have two entities, Student and Instructor, each having the attributes name, email, and address.
- These shared attributes can be factored into a generalized entity Person.

Student(name, email, address, year)

Instructor(name, email, address, salary)

↓

Person(name, email, address)

- Student and Instructor become subclasses of Person.
- Person is the superclass that stores the common attributes.

Purpose:

- Reduce redundancy by grouping shared attributes.
- Simplify relationships that apply to all subclasses.

9.2.2. Specialization

Specialization is the reverse - a top-down process in which a general entity is divided into specialized sub-entities based on distinguishing characteristics.

Example:

- Starting from a general entity Person, we can define specialized sub-entities:
- Student (with attributes like year, major)
- Instructor (with attributes like salary, department)

Person

├── Student

└── Instructor

Purpose:

- Capture specific attributes or relationships of subtypes.
- Provide flexibility in modeling complex real-world hierarchies.

9.2.3. Attribute Inheritance

In a specialization hierarchy:

- Subclasses inherit all attributes and relationships of their superclass.
- Additional attributes are defined only at the subclass level.

For example:

- Person: person_id, name, email
- Student: inherits above + adds year, major
- Instructor: inherits above + adds salary, dept_id

Thus, all subclasses share the same unique identifier (person_id).

9.2.4. Disjoint vs. Overlapping Specialization

Type	Description	Example
Disjoint	An entity instance belongs to only one subclass .	A Person can be either a Student or an Instructor, not both.
Overlapping	An entity instance can belong to multiple subclasses simultaneously .	A Person can be both a Student and an Employee.

- Disjoint constraints are represented with a “d” in E-R diagrams.
- Overlapping constraints are represented with an “o”.

9.2.5. Total vs. Partial Specialization

Type	Description	Example
Total specialization	Every instance of the superclass must belong to at least one subclass.	Every Person must be either a Student or an Instructor.
Partial specialization	Some instances of the superclass may not belong to any subclass.	Some Persons may be just visitors (neither Student nor Instructor).

Notation in E-R diagrams:

- Double line from superclass to subclasses → Total specialization
- Single line → Partial specialization

9.2.6. E–R Representation Example

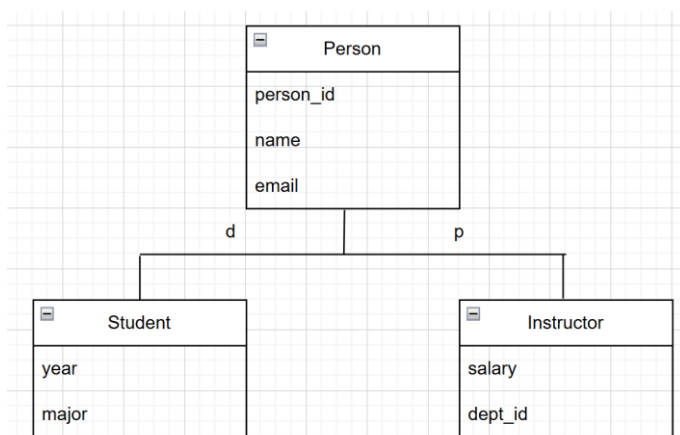


Fig. 9.1. E-R Diagram

- The “d” indicates a disjoint specialization.
- The single line means it’s partial (not every person must be one of the subclasses).

9.2.7. Implementation in Relational Databases

There are three main ways to implement specialization/generalization in SQL:

9.2.7.1. Single Table Inheritance (All-in-One Table)

All superclass and subclass attributes are stored in one table, with some columns left NULL.

```
CREATE TABLE person (
    person_id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    email VARCHAR(100),
    role VARCHAR(20),
    year INT,
    major VARCHAR(50),
    salary NUMERIC(10,2),
    dept_id INT
);
```

Simple but wastes space and can lead to NULL-heavy tables.

9.2.7.2. Class Table Inheritance (Separate Tables)

The superclass and each subclass have their own table.

```
CREATE TABLE person (
    person_id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    email VARCHAR(100)
);
```

```
CREATE TABLE student (
    person_id INT PRIMARY KEY REFERENCES person(person_id),
    year INT,
    major VARCHAR(50)
```

);

```
CREATE TABLE instructor (  
    person_id INT PRIMARY KEY REFERENCES person(person_id),  
    salary NUMERIC(10,2),  
    dept_id INT  
);
```

Most commonly used approach — enforces inheritance and integrity.

Requires joins to access full entity data.

9.2.7.3. Shared Primary Key (1:1 Mapping)

Subclasses reuse the primary key of the superclass to maintain a one-to-one relationship.

Example:

```
SELECT p.name, s.major  
FROM person p  
JOIN student s ON p.person_id = s.person_id;
```

9.2.8. Example: University Database

Entities:

- Person(person_id, name, email)
- Student(person_id, year, major)
- Instructor(person_id, salary, dept_id)

Relationships:

- Instructor teaches Course.
- Student enrolls in Course.

This can be extended with overlapping specializations, such as Employee (for non-academic staff).

9.2.9. Advantages of Specialization and Generalization

Aspect	Benefit
Clarity	Easier to model real-world hierarchies
Reuse	Common attributes appear once in superclass
Integrity	Reduces redundancy and inconsistency

Flexibility	Supports partial and overlapping membership
Query poer	Enables combined or filtered subclass queries

9.2.10. Example Queries

-- Get all persons and their type:

```
SELECT p.name,
       CASE
         WHEN s.person_id IS NOT NULL THEN 'Student'
         WHEN i.person_id IS NOT NULL THEN 'Instructor'
         ELSE 'Other'
       END AS role
```

FROM person p

LEFT JOIN student s ON p.person_id = s.person_id

LEFT JOIN instructor i ON p.person_id = i.person_id;

-- Retrieve all instructors with department name:

```
SELECT p.name, i.salary, d.dept_name
```

FROM instructor i

JOIN person p ON p.person_id = i.person_id

JOIN department d ON d.dept_id = i.dept_id;

9.3. Lab Exercises

In this lab you will design and implement a specialization/generalization (ISA) hierarchy in PostgreSQL, practice disjoint vs. overlapping subtypes, and use views to present unified data. Use DBeaver or psql.

9.3.1. Conceptual Modeling (E-R)

Goal: Draw an E-R diagram for a university personnel system with a superclass and two subclasses.

Superclass: Person(person_id, name, street, city, email)

Subclasses:

- Student(tot_cred, residential_college)
- Employee(salary) → further specialized into Instructor(rank) and Secretary(hours_per_week)

Indicate:

- Disjoint specialization between Instructor and Secretary
- Overlapping specialization between Student and Employee (a person can be both)
- Participation: Employee is partial of Person; Student is partial of Person

Deliverable: a clean E-R diagram with ISA notations and participation (total/partial) marks.

9.3.2. Logical Design (Class-Table Inheritance)

Goal: Translate the E-R into relational tables using shared primary keys for 1:1 superclass–subclass links.

-- Superclass

```
CREATE TABLE person (
    person_id SERIAL PRIMARY KEY,
    name       VARCHAR(80) NOT NULL,
    street     VARCHAR(80),
    city       VARCHAR(80),
    email      VARCHAR(120)
);
```

-- First-tier subclasses

```
CREATE TABLE student (
    person_id INT PRIMARY KEY REFERENCES person(person_id) ON
DELETE CASCADE,
    tot_cred  INT CHECK (tot_cred >= 0),
    residential_college VARCHAR(80)
);
```

```
CREATE TABLE employee (
    person_id INT PRIMARY KEY REFERENCES person(person_id) ON
DELETE CASCADE,
    salary     NUMERIC(10,2) CHECK (salary >= 0)
);
```

-- Second-tier subclasses of Employee (disjoint intent-enforced later)

```
CREATE TABLE instructor (
```

```

    person_id INT PRIMARY KEY REFERENCES employee(person_id) ON
DELETE CASCADE,
    rank      VARCHAR(40)
);
CREATE TABLE secretary (
    person_id INT PRIMARY KEY REFERENCES employee(person_id)
ON DELETE CASCADE,
    hours_per_week NUMERIC(4,1) CHECK (hours_per_week BETWEEN 0
AND 60)
);

```

9.3.3. Sample Data

```

INSERT INTO person (name, street, city, email) VALUES
('Ana Pop', 'Str. Unirii 10', 'Cluj', 'ana@example.com'),
('Mihai Ionescu', 'Bd. Eroilor 5', 'Cluj',
'mihai@example.com'),
('Laura Stan', 'Str. Universitatii 1', 'Bucuresti',
'laura@example.com');

```

Ana = Student

```

INSERT INTO student (person_id, tot_cred, residential_college)
SELECT person_id, 120, 'Collegium' FROM person WHERE name =
'Ana Pop';

```

--Mihai = Employee -> Instructor

```

INSERT INTO employee (person_id, salary)
SELECT person_id, 7800 FROM person WHERE name = 'Mihai
Ionescu';
INSERT INTO instructor (person_id, rank)
SELECT person_id, 'Associate' FROM person WHERE name = 'Mihai
Ionescu';

```

```
--Laura = Employee -> Secretary
INSERT INTO employee (person_id, salary)
SELECT person_id, 5200 FROM person WHERE name = 'Laura Stan';
INSERT INTO secretary (person_id, hours_per_week)
SELECT person_id, 30 FROM person WHERE name = 'Laura Stan';
```

9.3.4. Enforcing Disjoint vs. Overlapping

9.3.4.1. Disjoint rule for Instructor vs. Secretary

(An employee can be at most one of these.)

-- Option 1: Exclusion constraint via triggers (portable approach)

```
CREATE OR REPLACE FUNCTION enforce_disjoint_employee_roles()
RETURNS TRIGGER AS $$
BEGIN
    IF TG_TABLE_NAME = 'instructor' THEN
        IF EXISTS (SELECT 1 FROM secretary WHERE person_id =
NEW.person_id) THEN
            RAISE EXCEPTION 'Employee % already a secretary
(disjoint rule)', NEW.person_id;
        END IF;
    ELSIF TG_TABLE_NAME = 'secretary' THEN
        IF EXISTS (SELECT 1 FROM instructor WHERE person_id =
NEW.person_id) THEN
            RAISE EXCEPTION 'Employee % already an instructor
(disjoint rule)', NEW.person_id;
        END IF;
    END IF;
    RETURN NEW;
END; $$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER trg_disjoint_instructor
BEFORE INSERT OR UPDATE ON instructor
```

```
FOR          EACH          ROW          EXECUTE          FUNCTION
enforce_disjoint_employee_roles();
```

```
CREATE TRIGGER trg_disjoint_secretary
```

```
BEFORE INSERT OR UPDATE ON secretary
```

```
FOR          EACH          ROW          EXECUTE          FUNCTION
enforce_disjoint_employee_roles();
```

9.3.4.2. Overlapping rule for Student vs. Employee

No extra constraint: the same person_id may appear in both student and employee.

9.3.5. Queries

List all persons with their roles (Student/Instructor/Secretary/Other):

```
SELECT p.person_id, p.name,
       CASE
         WHEN s.person_id IS NOT NULL AND i.person_id IS NOT
NULL THEN 'Student & Instructor'
         WHEN s.person_id IS NOT NULL AND sec.person_id IS NOT
NULL THEN 'Student & Secretary'
         WHEN s.person_id IS NOT NULL THEN 'Student'
         WHEN i.person_id IS NOT NULL THEN 'Instructor'
         WHEN sec.person_id IS NOT NULL THEN 'Secretary'
         ELSE 'Other'
       END AS role
FROM person p
LEFT JOIN student  s  ON s.person_id = p.person_id
LEFT JOIN employee e  ON e.person_id = p.person_id
LEFT JOIN instructor i ON i.person_id = p.person_id
LEFT JOIN secretary sec ON sec.person_id = p.person_id
ORDER BY p.person_id;
```

All instructors with salary and (example) department:

-- (Add your department(dept_id, dept_name) and FK in employee
if needed.)

```
-- Example if employee has dept_id FK:
-- SELECT p.name, e.salary, d.dept_name
-- FROM instructor i
-- JOIN employee e ON e.person_id = i.person_id
-- JOIN person    p ON p.person_id = e.person_id
-- JOIN department d ON d.dept_id = e.dept_id;
```

```
--Secretaries working ≥ 20 hours/week:
SELECT p.name, sec.hours_per_week
FROM secretary sec
JOIN person p ON p.person_id = sec.person_id
WHERE sec.hours_per_week >= 20
ORDER BY sec.hours_per_week DESC;
```

9.3.6. Unified Views (Virtual “flattened” tables)

```
CREATE OR REPLACE VIEW view_instructor AS
SELECT p.person_id, p.name, p.city, e.salary, i.rank
FROM instructor i
JOIN employee  e ON e.person_id = i.person_id
JOIN person    p ON p.person_id = i.person_id;
```

```
CREATE OR REPLACE VIEW view_student AS
SELECT    p.person_id,    p.name,    p.city,    s.tot_cred,
s.residential_college
FROM student s
JOIN person  p ON p.person_id = s.person_id;
```

Views give security/abstraction and recompute results dynamically without duplicating data.

9.3.7. Integrity Checks & Maintenance

Missing subclass rows: find Person entries that are neither Student nor Employee:

```

SELECT p.*
FROM person p
LEFT JOIN student s  ON s.person_id = p.person_id
LEFT JOIN employee e ON e.person_id = p.person_id
WHERE s.person_id IS NULL AND e.person_id IS NULL;

```

Invalid second-tier membership: find employees who are neither instructors nor secretaries:

```

SELECT e.*
FROM employee e
LEFT JOIN instructor i ON i.person_id = e.person_id
LEFT JOIN secretary s  ON s.person_id = e.person_id
WHERE i.person_id IS NULL AND s.person_id IS NULL;

```

9.3.8. Optional Extensions

- Add courses and relationships:
 - Course(course_id, title, credits)
 - Teaches(instructor_id → instructor.person_id, course_id, semester, year)
 - Takes(student_id → student.person_id, course_id, section, semester, year)
- Model total vs. partial specializations with CHECK+trigger logic (e.g., require every Person to be at least Student or Employee for total).
- Seed bulk test data via CSV imports and validate constraints.

9.4. Troubleshooting

Designing ISA hierarchies often triggers integrity and modeling issues. Use the checks below to diagnose and fix typical problems related to superclass–subclass links, disjoint vs. overlapping specializations, attribute/relationship inheritance, and views. Concepts referenced: specialization, generalization, ISA, inheritance, disjoint/overlapping, and using views to expose combined data

9.4.1. Orphan subclass rows (missing superclass)

- Symptom:
 - INSERT into student/instructor/secretary fails or later joins miss data.
- Cause:

- Subclass PK must also exist in the superclass (shared key).
- Fix:
 - Insert into person (and employee for second-tier roles) before inserting subclass rows.
- Quick audit:

-- Student without Person

```
SELECT s.person_id FROM student s
LEFT JOIN person p ON p.person_id = s.person_id
WHERE p.person_id IS NULL;
```

-- Instructor/Secretary without Employee

```
SELECT i.person_id FROM instructor i
LEFT JOIN employee e ON e.person_id = i.person_id
WHERE e.person_id IS NULL;
```

9.4.2. Superclass exists but no subclass where required (intended “total” specialization)

- Symptom:
 - Persons that should be Student or Employee are “plain” Person rows.
- Cause:
 - You modeled it as total specialization conceptually, but didn’t enforce it.
- Fix (trigger example):

```
CREATE OR REPLACE FUNCTION enforce_total_specialization()
RETURNS TRIGGER AS $$
BEGIN
    PERFORM 1 FROM student WHERE person_id = NEW.person_id;
    IF FOUND THEN RETURN NEW; END IF;
    PERFORM 1 FROM employee WHERE person_id = NEW.person_id;
    IF FOUND THEN RETURN NEW; END IF;
    RAISE EXCEPTION 'Person % must be Student or Employee',
    NEW.person_id;
```

```
END; $$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER trg_total_spec  
AFTER INSERT OR UPDATE ON person  
FOR EACH ROW EXECUTE FUNCTION enforce_total_specialization();
```

9.4.3. Violating disjointness between sibling subclasses

- Symptom:
 - Same employee appears in both instructor and secretary.
- Cause:
 - Disjoint specialization wasn't enforced.
- Fix (mutual-exclusion trigger pair):

```
CREATE OR REPLACE FUNCTION disjoint_employee_roles()  
RETURNS TRIGGER AS $$  
BEGIN  
    IF TG_TABLE_NAME = 'instructor' AND EXISTS (SELECT 1 FROM  
secretary WHERE person_id = NEW.person_id)  
        THEN RAISE EXCEPTION 'Employee % already a secretary',  
NEW.person_id;  
    ELSIF TG_TABLE_NAME = 'secretary' AND EXISTS (SELECT 1 FROM  
instructor WHERE person_id = NEW.person_id)  
        THEN RAISE EXCEPTION 'Employee % already an instructor',  
NEW.person_id;  
    END IF;  
    RETURN NEW;  
END; $$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER disjoint_instructor BEFORE INSERT OR UPDATE ON  
instructor
```

```
FOR EACH ROW EXECUTE FUNCTION disjoint_employee_roles();
```

```
CREATE TRIGGER disjoint_secretary BEFORE INSERT OR UPDATE ON  
secretary
```

```
FOR EACH ROW EXECUTE FUNCTION disjoint_employee_roles();
```

-- (Overlapping specializations, like Person being both Student and Employee, should not use this rule.)

9.4.4. Cascading deletes remove subclass data unexpectedly

- Symptom:
 - Deleting a person removes student/employee (and instructor/secretary) rows.
- Cause:
 - You used ON DELETE CASCADE on FK from subclass to superclass, which is correct for shared PK, but be deliberate about it.
- Fix:
 - If you want to block deletion when specialized rows exist, use ON DELETE RESTRICT (or drop the row from subclasses first).

9.4.5. Attribute inheritance not reflected in schema or queries

- Symptom:
 - You cannot see superclass attributes when querying subclasses.
- Cause:
 - Inheritance is conceptual; in the relational model, you must join subclass to superclass to “inherit” attributes in results.
- Fix:
 - Always join subclass to person (and to employee for second-tier roles), or use views:

```
CREATE OR REPLACE VIEW view_instructor AS
SELECT p.person_id, p.name, p.city, e.salary, i.rank
FROM instructor i
JOIN employee e ON e.person_id = i.person_id
JOIN person p ON p.person_id = i.person_id;
-- (Using views for unified outputs is recommended in the doc.)
```

Wrong participation (partial vs. total) or wrong specialization type (disjoint vs. overlapping)

Symptom: Data model contradicts business rules (e.g., some Persons shouldn't be allowed to be “Other”; or Employees show up as both Instructor and Secretary).

Cause: The chosen constraints don't match the intended participation or overlap/disjoint semantics.

- Fix:
 - Total participation → add validation trigger (Tip #2).
 - Disjoint siblings → add exclusion triggers (Tip #3).
 - Overlapping siblings → do not add mutual-exclusion logic.
- 9.4.6. “Column does not exist” in views after schema changes
- Symptom:
 - Views fail after you rename or drop columns in base tables.
 - Cause:
 - Views depend on base columns.
 - Fix:
 - Drop and recreate the view:

```
DROP VIEW IF EXISTS view_student;
```

```
CREATE VIEW view_student AS
```

```
SELECT    p.person_id,    p.name,    p.city,    s.tot_cred,
s.residential_college
```

```
FROM student s
```

```
JOIN person p ON p.person_id = s.person_id;
```

9.4.7. Insert order errors & FK violations

- Symptom:
 - FK errors when inserting into subclass before superclass.
- Fix (insert order):
 - person → employee (if needed) → instructor/secretary or student.
- Audit quickly:

```
-- Employees lacking a second-tier role
```

```
SELECT e.person_id FROM employee e
```

```
LEFT JOIN instructor i ON i.person_id = e.person_id
```

```
LEFT JOIN secretary s ON s.person_id = e.person_id
```

```
WHERE i.person_id IS NULL AND s.person_id IS NULL;
```

9.4.8. Case-sensitive identifiers cause “relation/column does not exist”

- Symptom:
 - Errors referencing "Student" vs student.
- Cause:
 - Quoted identifiers become case-sensitive in PostgreSQL.
- Fix:
 - Prefer unquoted, lowercase identifiers in DDL and queries.

9.4.9. Performance: too many joins on reads

- Symptom:
 - Complex queries feel slow.
- Cause:
 - Class-table inheritance requires joins to assemble entities.
- Mitigations:
 - Add indexes on FK columns (person_id, dept_id).
 - Use views for convenience; optionally materialized views for heavy reports (refresh as needed).
 - Denormalize selectively for read-mostly workloads (keep writes consistent).
- Quick Health Checks
 - Unclassified persons (neither Student nor Employee):

```
SELECT p.* FROM person p
LEFT JOIN student s ON s.person_id = p.person_id
LEFT JOIN employee e ON e.person_id = p.person_id
WHERE s.person_id IS NULL AND e.person_id IS NULL;
```

- Role conflicts (if disjoint expected for Instructor vs Secretary):

```
SELECT e.person_id
FROM employee e
JOIN instructor i ON i.person_id = e.person_id
JOIN secretary s ON s.person_id = e.person_id;
```

9.5. Proposed Exercises

These exercises will help you practice designing and implementing specialization and generalization hierarchies (ISA relationships) in relational databases, focusing on disjoint vs. overlapping specializations, total vs. partial participation, and class-table inheritance in SQL.

9.5.1. Conceptual Modeling: University People

- Design an E–R diagram for a university system with the following entities:
 - Person(person_id, name, email, address)
 - Student(person_id, year, major)
 - Employee(person_id, salary)
 - Instructor(person_id, rank, department)
 - Secretary(person_id, hours_per_week)
- Tasks:
 - Identify which entity should be the superclass and which should be subclasses.
 - Define ISA hierarchies and specify whether they are disjoint or overlapping.

- Define participation constraints (total or partial).
- Draw the final E–R diagram clearly showing inheritance arrows and constraint symbols.

9.5.2. Logical Design: From ER to Relational Schema

- Translate your E–R diagram into a relational schema using class-table inheritance (a separate table for each class).
 - Ensure that each subclass uses the same primary key as the superclass.
 - Add foreign key constraints to maintain referential integrity.
 - For example:

```
CREATE TABLE person (
  person_id SERIAL PRIMARY KEY,
  name VARCHAR(80),
  email VARCHAR(100),
  address VARCHAR(120)
);
```

```
CREATE TABLE student (
  person_id INT PRIMARY KEY REFERENCES person(person_id),
  year INT,
  major VARCHAR(60)
);
```

- Document the relationships and constraints (using ON DELETE CASCADE where appropriate).

9.5.3. Enforcing Disjoint and Overlapping Rules

- Scenario A (Disjoint):
 - An employee can be either an Instructor or a Secretary, but not both.
- Scenario B (Overlapping):
 - A person can be both a Student and an Employee.
- Tasks:
 - Implement the disjoint rule using triggers or constraints.
 - Allow overlapping participation by omitting mutual exclusions.
 - Insert test data that validates both cases:
 - One person as both student and instructor.
 - One person incorrectly as both instructor and secretary → should raise an exception.

9.5.4. Data Population and Queries

Insert realistic data:

- At least 5 Persons, covering all subclass combinations.
- Demonstrate overlapping and disjoint membership.
- Write SQL queries to:
 - Retrieve all persons with their roles.
 - List all instructors earning above a threshold.
 - Find students who are also employees.
 - Display secretaries who work more than 25 hours/week.

9.5.5. Using Views

Create views to simplify querying of combined data:

```
CREATE VIEW all_persons AS
SELECT p.person_id, p.name,
       CASE
         WHEN s.person_id IS NOT NULL AND i.person_id IS NOT
NULL THEN 'Student & Instructor'
         WHEN s.person_id IS NOT NULL THEN 'Student'
         WHEN i.person_id IS NOT NULL THEN 'Instructor'
         WHEN sec.person_id IS NOT NULL THEN 'Secretary'
         ELSE 'Other'
       END AS role
FROM person p
LEFT JOIN student s  ON s.person_id = p.person_id
LEFT JOIN employee e  ON e.person_id = p.person_id
LEFT JOIN instructor i ON i.person_id = e.person_id
LEFT JOIN secretary sec ON sec.person_id = e.person_id;
```

- Test filtering on the view, e.g. `SELECT * FROM all_persons WHERE role LIKE '%Instructor%';`

9.5.6. Extended Case: Hospital Database

Design and implement a specialization hierarchy for a hospital system:

- Superclass: Person(person_id, name, address, phone)
- Subclasses:
 - Doctor(specialization, salary)
 - Nurse(shift, hours)

- Patient(blood_type, admission_date)

Requirements:

- The specialization is overlapping (a person can be both doctor and patient).
- Model total participation (every person must be at least one of the subclasses).
- Implement the database in PostgreSQL and populate it with sample data.
- Create queries to:
 - Find all patients who are also employees (doctor or nurse).
 - Show the total number of people per subclass.

9.5.7. Corporate Staff Hierarchy

Build a more complex hierarchy for a company database:

- Person → Employee → (Manager, Engineer, Clerk)
- Managers can supervise other employees (self-referencing FK).
- Engineers may have multiple specialties (link table).
- Normalize all tables and define the appropriate ISA constraints.

Test queries:

- List all managers and the employees they supervise.
- Show engineers who are also managers (overlapping scenario).
- Verify that no employee is both a manager and a clerk (disjoint rule).

9.5.8. Advanced Integrity Enforcement

Write SQL code or triggers that enforce:

- Total participation: every Person must be at least in one subclass.
- Disjoint constraint: Manager and Clerk cannot overlap.
- Overlapping rule: Engineer and Manager can overlap.

Document your enforcement logic (via triggers, constraints, or checks) and test them by inserting violating records.

Laboratory no. 10

Databases Through the Eyes of a Python Developer: Connecting, Querying, and Basic Data Processing

10.1.Objectives

The objective of this laboratory is to introduce database interaction from Python, focusing on how a developer connects to a relational database, executes queries safely, and processes results programmatically.

By the end of this session, students should be able to:

Understand the difference between SQL execution in the database and data processing in Python.

- Set up a Python environment and install required libraries.
- Connect to a SQLite database using Python.
- Execute SQL queries from Python using the DB-API 2.0 interface.
- Retrieve query results and process them using Python structures.
- Use parameterized queries to prevent SQL injection.
- Load query results into a pandas DataFrame.
- Perform basic data processing (filtering, grouping, sorting).
- Export results to CSV files.

This lab reuses the Sailing Database (sailors, boats, reserves) from previous laboratories.

10.2.Theoretical Foundation

Why SQLite in this Lab?

In this laboratory, SQLite is used as a lightweight relational database system that does not require a separate server installation or configuration.

This allows students to focus on the fundamentals of Python–database interaction, such as opening connections, executing queries, retrieving results, and processing data programmatically.

In the next laboratory, these concepts will be extended to PostgreSQL, where students will work with a client-server database system.

10.2.1. Why Python + Databases?

In real-world applications, SQL and Python play complementary roles:

SQL is used to:

- filter and join data efficiently,
- aggregate results close to the data source,
- enforce constraints and data integrity.

Python is used to:

- orchestrate database operations,
- apply business logic,
- transform and format results,
- integrate with files (CSV, Excel), APIs, and user interfaces.

Practical rule:

- Use SQL to get the right data, and Python to process what you get.

10.2.2. DB-API 2.0 (Python Database Interface)

Most Python database libraries follow a standard interface:

- Create a connection
- Create a cursor
- Execute SQL queries
- Fetch results
- Close resources

General pattern:

```
import sqlite3
```

```
conn = sqlite3.connect("sailing.db")
```

```
cur = conn.cursor()
```

```
cur.execute("SELECT * FROM sailors")
```

```
rows = cur.fetchall()
```

```
conn.close()
```

Typical workflow:

- Open connection
- Create Cursor
- Execute query
- Fetch results
- Process data
- Close connection

10.2.3. Safe Query Parameterization

Building SQL queries using string concatenation is unsafe:

WRONG (SQL injection risk)

```
query = "SELECT * FROM sailors WHERE sname = '" + user_input  
+ "'"
```

Instead, use parameterized queries:

```
cur.execute(  
    "SELECT * FROM sailors WHERE sname = ?",  
    (user_input,) )
```

- SQLite uses ?
- PostgreSQL drivers (e.g., psycopg2) use %s

10.2.4. Fetching Results

Common methods:

- fetchone() → one row
- fetchall() → all rows
- fetchmany(n) → limited rows

Example:

```
rows = cur.fetchall()  
for row in rows:  
    print(row)
```

10.2.5. pandas for Data Processing

The pandas library allows working with query results as tables:

```
import pandas as pd
```

```
df = pd.read_sql_query("SELECT * FROM sailors", conn)
```

Advantages:

- easy filtering
- grouping and aggregation
- exporting to CSV
- rapid data exploration

10.3.Lab Exercises

10.3.1. Environment Setup

Step 1: Create a virtual environment

```
python -m venv venv
```

Step 2: Activate it

Windows:

```
venv\Scripts\activate
```

Linux/macOS:

```
source venv/bin/activate
```

Step 3: Install packages

```
pip install pandas jupyter
```

Step 4: Start Jupyter Notebook

```
jupyter notebook
```

10.3.2. Connecting to SQLite

Create a connection to the existing database:

```
import sqlite3
```

```
conn = sqlite3.connect("sailing.db")
```

```
cur = conn.cursor()
```

Test query:

```
cur.execute("SELECT sname, age FROM sailors")
```

```
print(cur.fetchall())
```

10.3.3. Basic Query Execution

Exercise 1: Retrieve all sailors

```
cur.execute("SELECT * FROM sailors")
rows = cur.fetchall()
```

```
for row in rows:
    print(row)
```

Exercise 2: Filter sailors by age

```
cur.execute("SELECT sname FROM sailors WHERE age > ?", (30,))
print(cur.fetchall())
```

10.3.4. Working with Results in Python

Convert results to dictionaries:

```
columns = [desc[0] for desc in cur.description]

results = []
for row in cur.fetchall():
    results.append(dict(zip(columns, row)))

print(results)
```

10.3.5. Loading Data into pandas

```
import pandas as pd
```

```
df = pd.read_sql_query("SELECT * FROM sailors", conn)
print(df.head())
```

10.3.6. Data Processing in pandas

Exercise 3: Filter data

```
df[df["age"] > 30]
```

Exercise 4: Sort data

```
df.sort_values(by="age", ascending=False)
```

Exercise 5: Grouping and aggregation

```
df.groupby("rating")["age"].mean()
```

Exercise 6: Multiple aggregations

```
df.groupby("rating")["age"].agg(["mean", "min", "max"])
```

Exercise 7: Retrieve the oldest sailors

```
df.nlargest(3, "age")
```

10.3.7. Working with Multiple Tables

Load joined data:

```
query = """  
SELECT s.sname, b.bname, r.day  
FROM sailors s  
JOIN reserves r ON s.sid = r.sid
```

```
JOIN boats b ON b.bid = r.bid
"""
```

```
df = pd.read_sql_query(query, conn)
print(df)
```

10.3.8. Modifying Data from Python

Python can also be used to insert, update, or delete records in the database.

Example: insert a new sailor

```
cur.execute(
    "INSERT INTO sailors (sid, sname, rating, age) VALUES (?,
    ?, ?, ?)",
    (99, "Alice", 5, 29)
)

conn.commit()
```

The commit() call is necessary to save changes permanently in the database.

Without commit(), the modification may not be stored.

10.3.9. Error Handling

Database operations may fail because of missing files, invalid SQL syntax, or constraint violations.

For this reason, Python code should handle errors safely.

Example:

```
try:
    conn = sqlite3.connect("sailing.db")
    cur = conn.cursor()
```

```
cur.execute("SELECT * FROM sailors")
rows = cur.fetchall()
print(rows)

except Exception as e:
    print("Error:", e)

finally:
    conn.close()
```

The finally block ensures that the database connection is closed even if an error occurs.

10.3.10. Handling Missing Values

```
df.fillna("Unknown")
```

or

```
df.dropna()
```

10.3.11. Exporting Results

```
df.to_csv("output.csv", index=False)
```

10.3.12 Closing the Connection

```
conn.close()
```

10.4. Troubleshooting

- Database not found → Check file path to sailing.db
- Module not found → Run pip install pandas
- Empty results → Verify database contains data

- SQL errors → Test query directly in DBeaver first
- Connection not closed → Always call `conn.close()`

10.5. Proposed Exercises

10.5.1. Basic Queries

- Retrieve all boats and their colors.
- Display sailors older than 35.
- Count total reservations.

10.5.2. pandas Processing

- Load sailors into a DataFrame.
- Compute average age per rating.
- Find the oldest sailor.

10.5.3. Join Analysis

- Show all reservations with sailor names and boat names.
- Count reservations per boat.

10.5.4. Export

- Export joined data to CSV.
- Create a report with:
 - sailor name
 - number of reservations

10.5.5. Challenge

Build a Python script that:

- connects to the database
- runs at least 3 queries
- processes results using pandas
- exports a final report

Laboratory no. 11

Connecting Python to PostgreSQL:

Client–Server Databases and Advanced Querying

11.1.Objectives

The objective of this laboratory is to extend Python–database interaction by working with a client-server database system using PostgreSQL.

By the end of this session, students should be able to:

Understand the difference between embedded databases (SQLite) and client-server systems (PostgreSQL).

- Connect to a PostgreSQL database using Python.
- Use the psycopg2 library to execute SQL queries.
- Work with parameterized queries in PostgreSQL.
- Retrieve and process query results in Python.
- Load PostgreSQL data into pandas DataFrames.
- Perform data analysis using pandas.
- Understand basic transaction handling (commit and rollback).
- Compare SQLite and PostgreSQL usage in Python applications.

11.2.Theoretical Foundation

In the previous laboratory, SQLite was used as an embedded database stored in a local file.

PostgreSQL is a client-server database system:

- The database runs as a server process.
- Clients (such as Python applications) connect over a network.
- Authentication (username/password) is required.
- Multiple users can access the database concurrently.

This architecture is widely used in real-world applications.

11.2.1. Python and PostgreSQL (psycopg2)

To connect Python to PostgreSQL, we use the psycopg2 library.

Install it using:

```
pip install psycopg2-binary
```

Typical Connection Pattern

```
import psycopg2
```

```

conn = psycopg2.connect(
    host="localhost",
    database="lab5db",
    user="postgres",
    password="your_password"
)

cur = conn.cursor()

cur.execute("SELECT * FROM customers LIMIT 5")
rows = cur.fetchall()

conn.close()

```

11.2.2. Parameterized Queries in PostgreSQL

Unlike SQLite (which uses ?), PostgreSQL uses %s:

```

cur.execute(
    "SELECT * FROM customers WHERE country = %s",
    ("USA",)
)

```

11.2.3. Transactions

PostgreSQL uses transactions by default.

Changes must be committed:

```
conn.commit()
```

If something goes wrong:

```
conn.rollback()
```

11.3. Lab Exercises

11.3.1. Environment Setup

Install required packages:

```
pip install psycopg2-binary pandas jupyter
```

Start Jupyter:

```
jupyter notebook
```

11.3.2. Connecting to PostgreSQL

```
import psycopg2
```

```
conn = psycopg2.connect(  
    host="localhost",  
    database="lab5db",  
    user="postgres",  
    password="your_password"  
)
```

```
cur = conn.cursor()
```

Test query:

```
cur.execute("SELECT * FROM customers LIMIT 5")  
print(cur.fetchall())
```

11.3.3. Basic Query Execution

Exercise 1: Retrieve all customers

```
cur.execute("SELECT * FROM customers")  
rows = cur.fetchall()
```

```
for row in rows:  
    print(row)
```

Exercise 2: Filter customers by country

```
cur.execute(  
    "SELECT company_name FROM customers WHERE country = %s",
```

```
        ("USA",)  
    )
```

```
print(cur.fetchall())
```

11.3.4. Working with Results in Python

Convert results to dictionaries:

```
columns = [desc[0] for desc in cur.description]
```

```
results = []
```

```
for row in cur.fetchall():  
    results.append(dict(zip(columns, row)))
```

```
print(results)
```

11.3.5. Loading Data into pandas

```
import pandas as pd
```

```
df = pd.read_sql_query("SELECT * FROM customers", conn)
```

```
print(df.head())
```

11.3.6. Data Processing in pandas

Exercise 3: Filter data

```
df[df["country"] == "USA"]
```

Exercise 4: Sort data

```
df.sort_values(by="company_name")
```

Exercise 5: Grouping and aggregation

```
df.groupby("country").size()
```

Exercise 6: Multiple aggregations

```
df.groupby("country").agg({"customer_id": "count"})
```

11.3.7. Working with Multiple Tables

```
query = """
```

```
SELECT c.company_name, o.order_id, o.order_date
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
"""
```

```
df = pd.read_sql_query(query, conn)
```

```
print(df)
```

11.3.8. Modifying Data

Insert example:

```
cur.execute(
    "INSERT INTO customers (customer_id, company_name,
country) VALUES (%s, %s, %s)",
    ("CUSTX", "DataLab Inc.", "USA")
)
```

```
conn.commit()
```

Update example:

```
cur.execute(
    "UPDATE customers SET country = %s WHERE customer_id =
%s",
    ("Canada", "CUSTX")
)
```

```
conn.commit()
```

11.3.9. Transactions and Rollback

try:

```

cur.execute(
    "INSERT INTO customers (customer_id, company_name)
VALUES (%s, %s)",
    ("CUSTX", "Duplicate Test")
)

```

```

conn.commit()

```

```

except Exception as e:
    print("Error:", e)
    conn.rollback()

```

11.3.10. Error Handling

```

try:
    cur.execute("SELECT * FROM invalid_table")

```

```

except Exception as e:
    print("Error:", e)

```

11.3.11. Exporting Results

```

df.to_csv("customers_report.csv", index=False)

```

11.3.12. Closing the Connection

```

conn.close()

```

11.4. Troubleshooting

- Connection refused → Check if PostgreSQL service is running
- Authentication failed → Verify username and password
- Module not found → Install psycopg2
- SQL errors → Test queries in DBeaver

11.5. Proposed Exercises

11.5.1. Basic Queries

- Retrieve all orders.
- Display customers from a specific country.
- Count total orders.

11.5.2. pandas Processing

- Load orders into a DataFrame.
- Compute number of orders per customer.

- Find top 5 customers by number of orders.
- 11.5.3. Join Analysis
- Combine customers and orders.
 - Compute total orders per country.
- 11.5.4. Transactions
- Insert test data.
 - Rollback changes.
 - Verify that data was not saved.
- 11.5.5. Challenge

Build a Python script that:

- connects to PostgreSQL
- runs multiple queries
- processes data using pandas
- generates a final report (CSV)

Laboratory no. 12

Building a Data Processing Pipeline: From Database to Report

12.1.Objectives

The objective of this laboratory is to integrate SQL and Python skills into a complete data processing workflow, where data is extracted from a database, processed programmatically, and transformed into a final report.

By the end of this session, students should be able to:

- Understand the concept of a data processing pipeline.
- Extract data from a PostgreSQL database using Python.
- Combine multiple queries into a single workflow.
- Process and transform data using pandas.
- Create derived metrics and summaries.
- Export processed data into CSV files.
- Automate a simple reporting workflow using Python scripts.
- Structure code in a clean and reusable way.

12.2.Theoretical Foundation

12.2.1. What is a Data Pipeline?

A data pipeline is a sequence of steps that moves and transforms data from a source to a final output.

Typical stages:

- Data extraction (from database)
- Data transformation (cleaning, filtering, aggregating)
- Data loading/export (CSV, Excel, reports)

12.2.2. Pipeline Architecture

Database → SQL Queries → Python → pandas → Output (CSV / Report)

Separation of Responsibilities

SQL:

- filtering
- joins
- aggregations close to data

Python:

- combining results
- custom logic
- formatting outputs

12.2.3. Reusable Code Structure

Instead of writing everything in one block, scripts should be structured:

- connection function
- query functions
- processing functions
- export function

12.3.Lab Exercises

12.3.1. Setup

Ensure PostgreSQL is running and the database (e.g., Northwind or lab5db) is available.

Install required libraries:

```
pip install psycopg2-binary pandas
```

12.3.2. Creating a Connection Function

```
import psycopg2
```

```
def get_connection():  
    return psycopg2.connect(  
        host="localhost",  
        database="lab5db",  
        user="postgres",  
        password="your_password"  
    )
```

12.3.3. Extracting Data

```
import pandas as pd
```

```
conn = get_connection()
```

```
customers_df = pd.read_sql_query("SELECT * FROM customers",  
conn)
```

```
orders_df = pd.read_sql_query("SELECT * FROM orders", conn)
```

12.3.4. Data Transformation

Merge datasets:

```
df = customers_df.merge(orders_df, on="customer_id")
```

Compute number of orders per customer:

```
orders_per_customer =  
df.groupby("customer_id").size().reset_index(name="total_orders")
```

Find top customers:

```
top_customers =  
orders_per_customer.sort_values(by="total_orders",  
ascending=False).head(5)
```

12.3.5. Creating Derived Metrics

```
avg_orders = orders_per_customer["total_orders"].mean()  
print("Average orders per customer:", avg_orders)
```

12.3.6. Exporting Results

```
orders_per_customer.to_csv("orders_per_customer.csv",  
index=False)  
top_customers.to_csv("top_customers.csv", index=False)
```

12.3.7. Structuring the Pipeline

```
def extract_data(conn):  
    customers = pd.read_sql_query("SELECT * FROM customers",  
conn)  
    orders = pd.read_sql_query("SELECT * FROM orders", conn)  
    return customers, orders
```

```
def transform_data(customers, orders):  
    df = customers.merge(orders, on="customer_id")  
    result =  
df.groupby("customer_id").size().reset_index(name="total_orders")  
    return result
```

```
def load_data(df):  
    df.to_csv("report.csv", index=False)
```

```
def main():
    conn = get_connection()
    customers, orders = extract_data(conn)
    result = transform_data(customers, orders)
    load_data(result)
    conn.close()
```

```
if __name__ == "__main__":
    main()
```

12.3.8. Error Handling

```
try:
    main()
except Exception as e:
    print("Pipeline error:", e)
```

12.4. Troubleshooting

- Connection issues → Check PostgreSQL service
- Empty DataFrame → Verify query results
- Merge errors → Check column names
- File not generated → Verify export path

12.5. Proposed Exercises

12.5.1. Data Extraction

- Load customers, orders, and products tables.
- Verify data sizes.

12.5.2. Data Transformation

- Compute total orders per customer.
- Compute total quantity sold per product.

12.5.3. Analysis

- Find top 5 products.
- Find top 5 customers.
- Compute average orders per country.

12.5.4. Reporting

- Export all results to CSV files.
- Create a summary report with:
 - total customers
 - total orders

- top customer

12.5.5. Challenge

Build a complete data pipeline that:

- extracts data from PostgreSQL
- processes multiple tables
- computes at least 3 metrics
- exports a final report

Bibliography

Elmasri, R., & Navathe, S. B. (2016). Fundamentals of database systems (7th ed.). Pearson Education.

Coronel, C., & Morris, S. (2019). Database systems: Design, implementation, and management (13th ed.). Cengage Learning.

Ramakrishnan, R., & Gehrke, J. (2003). Database management systems (3rd ed.). McGraw-Hill.

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). Database system concepts (7th ed.). McGraw-Hill Education.

Rob, P., & Coronel, C. (2007). Database systems: Design, implementation, and management (8th ed.). Thomson Course Technology.

Connolly, T. M., & Begg, C. E. (2015). Database systems: A practical approach to design, implementation, and management (6th ed.). Pearson Education.

Date, C. J. (2003). An introduction to database systems (8th ed.). Addison-Wesley.

PostgreSQL Global Development Group. (2025). PostgreSQL 16.4 documentation. Retrieved October 2025, from <https://www.postgresql.org/docs/>

W3Schools. (2025). SQL tutorial. Retrieved October 2025, from <https://www.w3schools.com/sql/>