

# PostgreSQL

*Prepared by Ailton Moreira*

*[amv265@hotmail.com](mailto:amv265@hotmail.com)*

## Ailton Moreira

- Information System Engineer - PhD in Technology and Information System –  
Specialization in
  - Health Informatics
  - Omnichannel and Multichannel interaction in healthcare services
- Co-Leader of XiA – WP3
- Assistant Professor at University of Minho, 2022
- Software Engineer –
  - Frontend Engineer at ANDRE.AI 2024
  - Backend Engineer at ESC++, 2025 (freelancer)
- Training Academy, 2024
- Intelligent Data Systems Group, ALGORIMTI Center, 2018 (2024)

**Briefly introduce yourself:**

- Name
- Title or Position
- Department
- Experience with database systems and which type of database
- Expectations

# Agenda

- Module 1 – Introduction to PostgreSQL
- Module 2 – Data Modelling
- Module 3 – Efficient SQL Queries
- Module 4 – Organization and Structuring of Information
- Module 5 – Advanced Types and Extensions
- Module 6 – Security and Access Control
- Module 7 – Monitoring and Performance
- Module 8 – Strategic Data and Indicators
- Module 9 – Practical Project and Presentation ?

# Learning Outcomes

- Install, configure, and manage PostgreSQL 18 database, understanding its history and architecture.
- Design efficient data models using current data types and constraints.
- Write and optimize advanced SQL queries with tools like EXPLAIN.
- Organize data with schemas, partitioning, JSONB, and PL/pgSQL procedures.
- Utilize advanced types like arrays, geometric, and extensions such as PostGIS.
- Implement robust security measures, including roles, RLS, encryption, and backups.
- Monitor and tune performance using pg\_stat views and external tools.
- Create strategic reports, KPIs, and export data in formats like CSV and JSON.
- Apply concepts in real-world scenarios through Docker-based exercises.

# Module 1 - Introduction to PostgreSQL

- History and characteristics of PostgreSQL
- Basic installation and configuration (docker)
- Internal architecture (processes, buffers, write-ahead log)

# Introduction

- PostgreSQL is an open-source Object-Relational Database Management System (ORDBMS) based on PostgreSQL, known for its robustness, extensibility, and compliance with SQL standards.
- It originated from the POSTGRES project developed in the Computer Science Department at the University of California, Berkeley, in 1986, and evolved into PostgreSQL in 1996 with SQL support.

# Introduction - History

- With over two decades of development, PostgreSQL is now the most advanced open-source database available on the market.
- The POSTGRES project, led by Professor Michael Stonebraker, was sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc.
- In 1994, Andrew Yu and Jolly Chen added an SQL language interpreter to POSTGRES. Under a new name, Postgres95 was subsequently released on the internet to find its own path in the world as an open-source descendant of the original Berkeley POSTGRES code.
- In 1995, POSTGRES was renamed PostgreSQL.

# Introduction

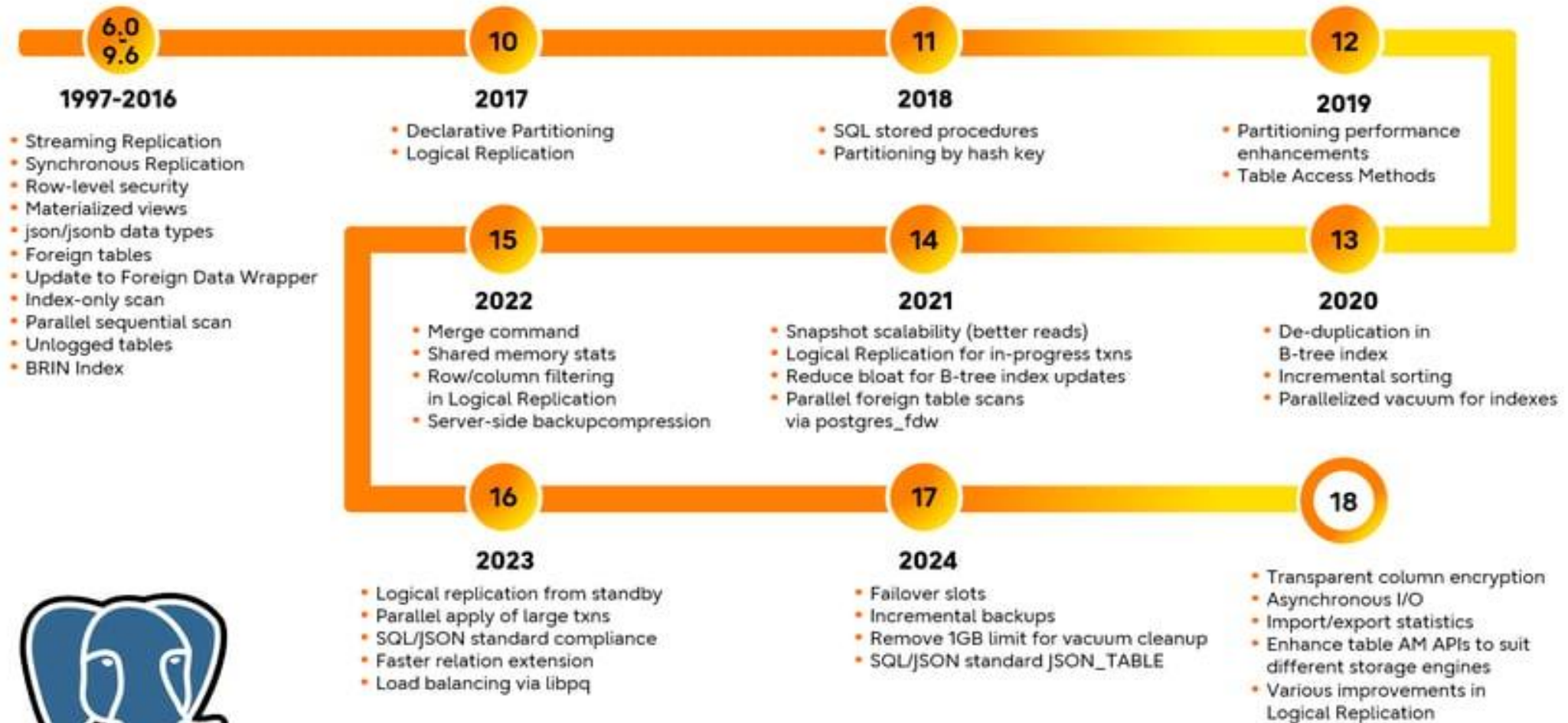
- PostgreSQL supports most of the SQL standard and many modern features :
  - Complex Queries
  - Foreign Keys
  - Triggers
  - Updatable Views
  - Transactional Integrity
  - Multi Version Concurrency Control (MVCC)
  - ACID compliance
  - Advanced data types
- PostgreSQL support some NOSQL features like:
  - Document Database - JSON
  - Key-Value pairs - HSTORE
  - JSONB – Binary JSON
  - Combining ANSI SQL Queries and JSON Queries

# Introduction

- PostgreSQL can be extended by the user in several ways, for example, by adding new:
  - Data Types
  - Functions
  - Operations
  - Aggregate Functions
  - Index Method
  - Procedural Language
  - Asynchronous replication
  - Nested transactions (savepoints)
- The recent versions of PostgreSQL support the following features:
  - Native Microsoft Windows Server version
  - Tablespace
  - Point-in-time recovery

# Introduction - Evolution

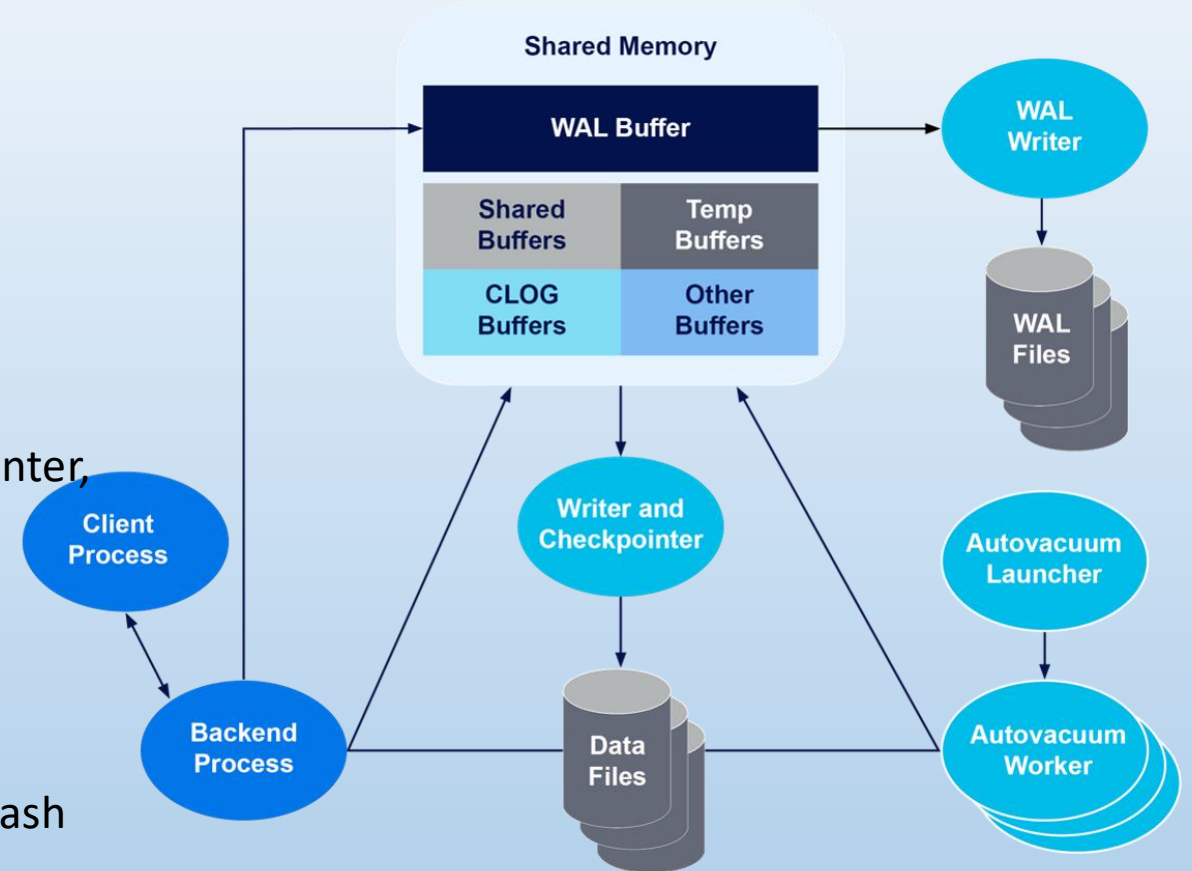
## Evolution of the OSS database PostgreSQL



Speculative, based on observations of discussions in the PostgreSQL Community

# Introduction – PostgreSQL Internal Architecture

- Client–Server architecture with a postmaster process
- One backend process per client connection
- Shared memory (Shared Buffers, WAL buffers) + local memory per backend
- Background processes: bgwriter, wal writer, checkpointer, autovacuum
- Physical storage uses heap files, 8KB pages, 1GB file segments
- Write-Ahead Logging (WAL) ensures durability and crash recovery
- Supports physical and logical replication for HA and scaling
- Connection pooling recommended for high concurrency workloads



Source: <https://www.instaclustr.com/blog/postgresql-architecture/>

# Installation

**Note:** PostgreSQL uses a client/server model. A PostgreSQL session consists of the following cooperating processes (programs):

- A server process, which manages the database files, accepts connections to the database from client applications, and performs database actions on behalf of the clients. The database server program is called postgres.
- The user's client (frontend) application that wants to perform database operations. Client applications can be very diverse in nature: a client could be a text-oriented tool, a graphical application, a web server that accesses the database to display web pages, or a specialized database maintenance tool. Some client applications are supplied with the PostgreSQL distribution; most are developed by users.

# Installation

- **Supported Operating Systems**

- Linux - several distributions
- Mac OSX
- MS Windows
- HP UX
- AIX
- Solaris/Sun OS
- Cygwin - Linux environment in MS Windows

- **Lots of Management Tools**

- PgAdmin
- phppgadmin
- DbVisualizer
- pgaccess
- pgShark

# Installation - Accessing Database

- To access to PostgreSQL database, we have several ways:

## 1. Using PostgreSQL Terminal:

PostgreSQL provides a terminal to access, is called psql.

```
$ psql dbname
```

## 2. Using GUI Tools:

GUI Tools that can access to PostgreSQL such as Navicat, PgAdmin, PHPPgAdmin, ...

## 3. Using Customize Applications:

Using one of the several available language bindings.

# Installation - Manipulated Database

- To create a database, we have two ways to create:

1. First way: we can use by command to create:

Ex: createdb dbname

2. Second way: we can use by sql standard to create:

Ex: create database dbname;

- To drop a database, we have two ways to drop:

1. First way: we can use by command to drop:

Ex: dropdb dbname

2. Second way: we can use by sql standard to drop:

Ex: drop database dbname;

# Installation - SQL Language

- Every Databases, they have to follow the SQL standard. The SQL provides as below:
  1. Data Definition Language (DDL)
  2. Data Manipulation Language (DML)
  3. Data Control Language (DCL)
  4. Transaction Control (TCL)

# Installation - DDL

- Data Definition Language (DDL) statements are used to define the database structure or schema.

Some examples:

1. CREATE - to create objects in the database
2. ALTER - alters the structure of the database
3. DROP - delete objects from the database
4. TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
5. COMMENT - add comments to the data dictionary
6. RENAME - rename an object

# Installation - DML

- Data Manipulation Language (DML) statements are used for managing data within schema objects.

Some examples:

1. SELECT - retrieve data from the database
2. INSERT - insert data into a table
3. UPDATE - updates existing data within a table
4. DELETE - deletes all records from a table, the space for the records remain
5. MERGE - UPSERT operation (insert or update)
6. CALL - call a PL/SQL or Java subprogram
7. EXPLAIN PLAN - explain access path to data
8. LOCK TABLE - control concurrency

# Installation - DCL

- Data Control Language (DCL) statements.

Some examples:

1. GRANT - gives user's access privileges to database
2. REVOKE - withdraw access privileges given with the GRANT command

# Installation - TCL

- Transaction Control (TCL) statements are used to manage the changes made by DML statements. It allows statements to be grouped together into logical transactions.

Some example:

1. COMMIT - save work done
2. SAVEPOINT - identify a point in a transaction to which you can later roll back
3. ROLLBACK - restore database to original since the last COMMIT
4. SET TRANSACTION - Change transaction options like isolation level and what rollback segment to use

# Installation

- **To install PostgreSQL, we have two options:**

1. Installing PostgreSQL from source code:

You can access the website to download it: <https://www.postgresql.org/ftp/source>

2. Install PostgreSQL from a file:

You can access the website to download the binary file: <https://www.postgresql.org/download/>

- **Docker Setup option**

# Module 1 - Introduction to PostgreSQL

- History and characteristics of PostgreSQL
- Basic installation and configuration (docker)
- Internal architecture (processes, buffers, write-ahead log)

# Module 2 – Data Modeling

- Entities, attributes, and relationships
- Normalization (1NF, 2NF, 3NF, BCNF)
- Denormalization: when to apply
- Data types and constraints (PK, FK, UNIQUE, CHECK)
- Logical and physical modeling

# Entities, attributes, and relationships

## Core entities (logical / schema-level)

- **Cluster** — collection of databases.
- **Database** — container for schemas; separate namespaces.
- **Schema** — grouping of objects (tables, views, functions).
- **Table (entity)** — holds rows (records).
  - *Typical attributes:* table\_name, owner, columns, constraints, tablespace, row\_type.
- **Column (attribute)** — name, data\_type, default, nullable, collation.
- **Index** — type (btree, gin), columns, unique, partial, method.
- **Constraint** — primary key, unique, foreign key, check, not null.
- **Sequence** — nextval/current value (often used for surrogate PKs).
- **View / Materialized View** — query result, dependent objects.
- **Function (stored procedure)** — name, arguments, return type, language.
- **Type** — built-in or user-defined composite/enum/domain.
- **Role (user/group)** — privileges, memberships.
- **Tablespace** — storage location mapping to filesystem.
- **Extension** — optional modules (e.g., PostGIS).

# Entities, attributes, and relationships

## Key relationships (how entities connect)

- **Schema** → **Tables**: one-to-many (schema contains many tables).
- **Table** → **Column**: one-to-many (table composed of columns).
- **Table** ↔ **Table (foreign key)**: one-to-many or many-to-many (via join table).
- **Table** ↔ **Index**: one-to-many (multiple indexes per table).
- **Table** ↔ **Sequence**: often one-to-one (serial/identity column uses sequence).
- **View** → **Table**: dependency (view references underlying tables).
- **Role** → **Object (permissions)**: many-to-many (roles granted privileges on many objects).
- **Tablespace** → **Table**: one-to-many (tables assigned to tablespaces).
- **Function** ↔ **Table / View**: dependency if function references objects.

# Entities, attributes, and relationships

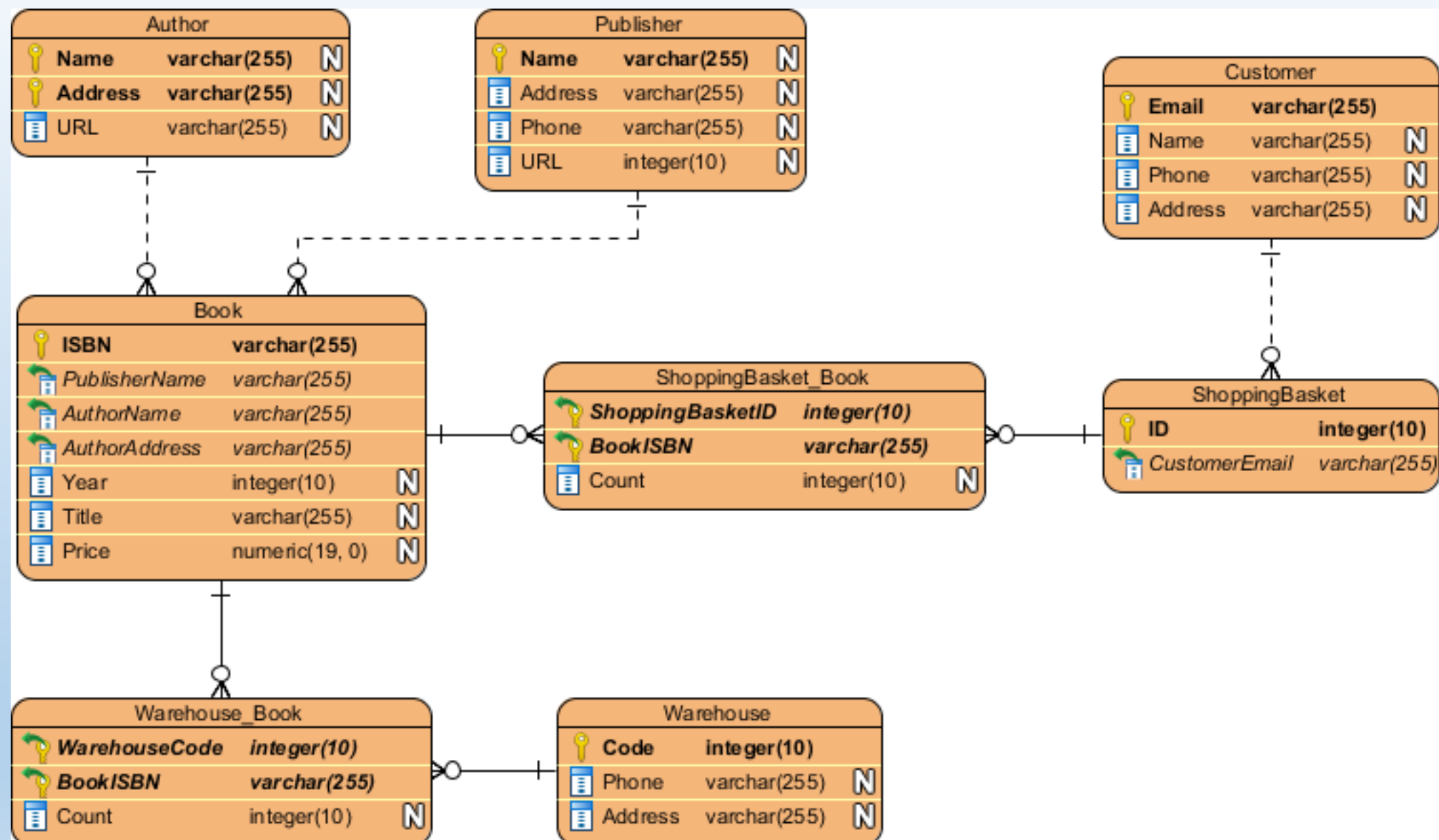
## Common relationship patterns (practical notes)

- **One-to-many:** implemented with FK on child table (e.g., orders.customer\_id → customers.id).
- **Many-to-many:** implemented by a junction table (e.g., product\_tags with product\_id, tag\_id).
- **Inheritance / table partitioning:** table INHERITS parent or uses declarative partitioning (parent→child tables).
- **Polymorphism / composite types:** composite columns or JSONB for flexible attributes.

# Entities, attributes, and relationships

```
CREATE TABLE customers (  
  id SERIAL PRIMARY KEY,  
  name VARCHAR(100),  
  email VARCHAR(50)  
);
```

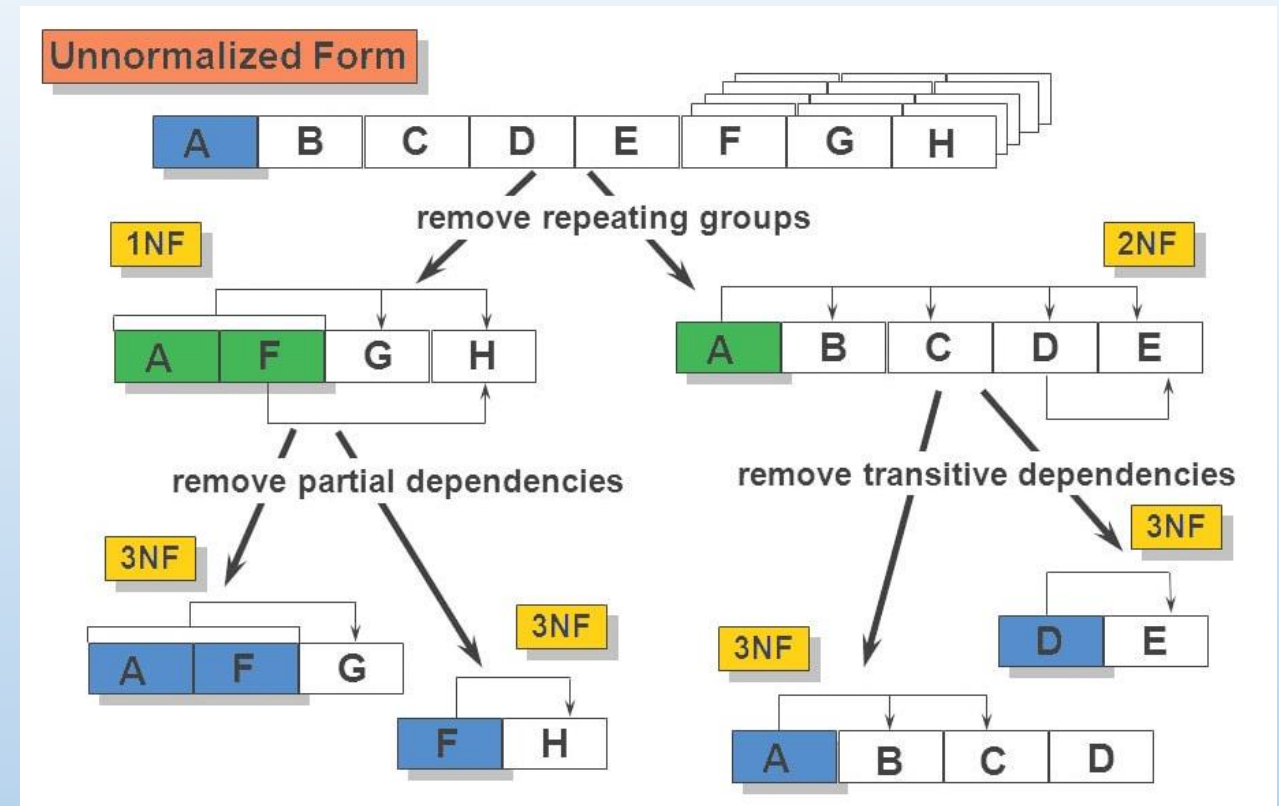
```
CREATE TABLE orders (  
  id SERIAL PRIMARY KEY,  
  customer_id INTEGER REFERENCES customers(id)  
);
```



# Normalization (1NF, 2NF, 3NF, BCNF)

**Normalization reduces redundancy:**

- 1NF (atomic values),
- 2NF (no partial dependencies),
- 3NF (no transitive dependencies),
- BCNF (every determinant is a candidate key).



| order_id | order_date | customer_id | customer_name | items       |
|----------|------------|-------------|---------------|-------------|
| 1001     | 2025-01-01 | 10          | Alice         | "P1:2,P2:1" |

**Example:**

Unnormalized table with repeating groups → Normalize to separate tables.

# Normalization (1NF, 2NF, 3NF, BCNF)

## 1NF — Atomic values (one value per cell)

**Rule:** each column holds atomic (indivisible) values; no repeating groups.

**Solution:** expand **items** into separate rows so each row is one order-item.

----- Order header

```
CREATE TABLE orders (  
  order_id serial PRIMARY KEY,  
  order_date date,  
  customer_id int  
);
```

----- Order items (one row per product in an order)

```
CREATE TABLE order_items (  
  order_id int REFERENCES orders(order_id),  
  product_id text,  
  quantity int,  
  PRIMARY KEY (order_id, product_id)  
);
```

## Example:

**orders:** (1001, '2025-01-01', 10)

**order\_items:** (1001, 'P1', 2), (1001, 'P2', 1)

Now values are atomic and easy to query  
(e.g., list all items for an order).

# Normalization (1NF, 2NF, 3NF, BCNF)

**2NF — No partial dependencies (applies when you have a composite primary key)**

**Rule:** if a table's primary key is composite, non-key columns must depend on the *whole* key, not just part of it.

Problem example (violation): order\_items PK = (order\_id, product\_id) but we store product\_name in the same table:

```
order_items
(order_id, product_id, product_name, quantity)
```

product\_name depends only on product\_id (part of the key) → partial dependency.

**Solution:** move product-specific info to a products table.

```
CREATE TABLE products (
  product_id text PRIMARY KEY,
  product_name text,
  price numeric
);
```

```
-- order_items keeps only keys & quantity
CREATE TABLE order_items (
  order_id int REFERENCES orders(order_id),
  product_id text REFERENCES products(product_id),
  quantity int,
  PRIMARY KEY (order_id, product_id)
);
```

**Now product\_name lives in products — no partial dependencies.**

# Normalization (1NF, 2NF, 3NF, BCNF)

## 3NF — No transitive dependencies

**Rule:** non-key columns must depend *directly* on the primary key, not via another non-key column.

Example of a transitive dependency: an employees table:

```
employees(emp_id PK, emp_name, dept_id, dept_name)
```

Here dept\_name depends on dept\_id, and dept\_id depends on emp\_id. So  $\text{emp\_id} \rightarrow \text{dept\_id} \rightarrow \text{dept\_name}$   
— dept\_name is transitively dependent on emp\_id. That's a 3NF violation.

**Solution:** pull department data into its own table.

```
CREATE TABLE departments (  
  dept_id serial PRIMARY KEY,  
  dept_name text UNIQUE  
);
```

```
CREATE TABLE employees (  
  emp_id serial PRIMARY KEY,  
  emp_name text,  
  dept_id int REFERENCES departments(dept_id)  
);
```

**Now employees only  
references dept\_id and  
department name is stored  
once in departments.**

# Normalization (1NF, 2NF, 3NF, BCNF)

**BCNF (Boyce–Codd Normal Form) — every determinant is a candidate key**

**Rule:** for every functional dependency  $A \rightarrow B$ ,  $A$  must be a candidate key (i.e., uniquely identify rows).

BCNF is stricter than 3NF. A common real-world scenario:

Table `course_assignments`:  $\rightarrow$  (`course_id`, `professor_id`, `room`)

- A course can have many professors (co-teaching), so no single column is a key.
- But every professor is assigned to exactly one room ( $\text{professor} \rightarrow \text{room}$ ).

Here  $\text{professor\_id} \rightarrow \text{room}$  but `professor_id` is not a candidate key of the table (it doesn't uniquely identify the full row)  $\rightarrow$  violates BCNF.

```
CREATE TABLE professors (  
  professor_id int PRIMARY KEY,  
  name text,  
  room text - - professor  $\rightarrow$  room  
);
```

```
CREATE TABLE course_professors (  
  course_id int,  
  professor_id int REFERENCES professors(professor_id),  
  PRIMARY KEY (course_id, professor_id)  
);
```

Now professor  room holds inside professors where `professor_id` is a key — BCNF satisfied.

# Denormalization: when to apply

Denormalization means **intentionally duplicating data** (or storing derived data) to make reads faster or simpler. Instead of always joining many tables to answer a query, you store the needed data together so some queries can run with fewer joins.

**Trade some extra storage and update complexity for faster reads**

## When to apply denormalization

Use it when:

- Workload is **read-heavy** (analytics, dashboards, API reads) and joins are the bottleneck.
- Need **low-latency** responses for specific queries (e.g., mobile app home screen).
- Can tolerate **slightly stale** data (or can update it synchronously if needed).
- The application is OK with added complexity to keep duplicates in sync.

Don't denormalize if:

- Have many concurrent writes and strict consistency requirements.
- Can't automate/update duplicates reliably.

# Normalization vs. Denormalization

## Comparing Advantages of Normalization and Denormalization

| Aspect               | Normalization                                                          | Denormalization                                                     |
|----------------------|------------------------------------------------------------------------|---------------------------------------------------------------------|
| Data Integrity       | Ensures consistency by reducing redundancy and enforcing relationships | May compromise integrity due to data duplication                    |
| Efficient Updates    | Easier to maintain and update individual data points                   | Requires updating in multiple places, increasing maintenance burden |
| Clear Relationships  | Clarifies data structure through foreign keys and normalization rules  | Can obscure logical data relationships due to flattened design      |
| Storage Optimization | Reduces storage by eliminating duplicate data                          | Consumes more space due to repeated data                            |
| Scalability          | Easier to evolve schema without risking inconsistency                  | Risk of inconsistency increases as system scales                    |

# Normalization vs. Denormalization

## Comparing Disadvantages of Normalization and Denormalization

| Aspect                       | Normalization                                                           | Denormalization                                                    |
|------------------------------|-------------------------------------------------------------------------|--------------------------------------------------------------------|
| Query Complexity             | Requires joins across multiple tables, increasing query complexity      | Simpler queries due to flat structure                              |
| Performance Overhead         | Slower reads in complex queries due to multiple joins                   | Faster read performance with reduced need for joins                |
| Development Time             | Requires thoughtful schema design and maintenance                       | Quicker setup for specific reporting or analytic needs             |
| Flexibility for BI/Analytics | Less suited for ad-hoc reporting; requires views or intermediate layers | Better aligned with analytics use cases due to consolidated data   |
| Risk of Anomalies            | Minimal if properly normalized                                          | Higher chance of anomalies from data duplication and inconsistency |

# Data types and constraints (PK, FK, UNIQUE, CHECK)

A **data type** defines **what kind of data** a column can store and **how PostgreSQL treats it** (storage, validation, operations).

## Numeric Types

| Data Type        | Description                      | Example                   |
|------------------|----------------------------------|---------------------------|
| SMALLINT         | Small integer (2 bytes)          | age SMALLINT              |
| INTEGER / INT    | Standard integer (4 bytes)       | quantity INT              |
| BIGINT           | Large integer (8 bytes)          | population BIGINT         |
| SERIAL           | Auto-increment integer           | id SERIAL PRIMARY KEY     |
| BIGSERIAL        | Large auto-increment integer     | id BIGSERIAL              |
| NUMERIC(p,s)     | Exact decimal (precision, scale) | price NUMERIC(10,2)       |
| REAL             | Floating-point (4 bytes)         | temperature REAL          |
| DOUBLE PRECISION | Floating-point (8 bytes)         | distance DOUBLE PRECISION |

# Data types and constraints (PK, FK, UNIQUE, CHECK)

A **data type** defines **what kind of data** a column can store and **how PostgreSQL treats it** (storage, validation, operations).

## Character (Text) Types

| Data Type  | Description          | Example              |
|------------|----------------------|----------------------|
| CHAR(n)    | Fixed-length text    | country_code CHAR(2) |
| VARCHAR(n) | Variable-length text | email VARCHAR(255)   |
| TEXT       | Unlimited text       | description TEXT     |

# Data types and constraints (PK, FK, UNIQUE, CHECK)

A **data type** defines **what kind of data** a column can store and **how PostgreSQL treats it** (storage, validation, operations).

## Date & Time Types

| Data Type                | Description               | Example                             |
|--------------------------|---------------------------|-------------------------------------|
| DATE                     | Calendar date             | <code>birth_date DATE</code>        |
| TIME                     | Time of day               | <code>start_time TIME</code>        |
| TIMESTAMP                | Date and time             | <code>created_at TIMESTAMP</code>   |
| TIMESTAMP WITH TIME ZONE | Time zone-aware timestamp | <code>created_at TIMESTAMPTZ</code> |
| INTERVAL                 | Time duration             | <code>duration INTERVAL</code>      |

# Data types and constraints (PK, FK, UNIQUE, CHECK)

A **data type** defines **what kind of data** a column can store and **how PostgreSQL treats it** (storage, validation, operations).

## Boolean Type

| Data Type | Description  | Example                        |
|-----------|--------------|--------------------------------|
| BOOLEAN   | True / False | <code>is_active BOOLEAN</code> |

## ENUM Type

```
CREATE TYPE order_status AS ENUM ('pending', 'paid', 'cancelled');
```

## UUID Type

| Data Type | Description                   | Example                                        |
|-----------|-------------------------------|------------------------------------------------|
| UUID      | Universally unique identifier | <code>id UUID DEFAULT gen_random_uuid()</code> |

# Data types and constraints (PK, FK, UNIQUE, CHECK)

A **data type** defines **what kind of data** a column can store and **how PostgreSQL treats it** (storage, validation, operations).

## JSON Types

| Data Type | Description                   | Example        |
|-----------|-------------------------------|----------------|
| JSON      | Stored as text                | payload JSON   |
| JSONB     | Binary JSON (indexed, faster) | metadata JSONB |

## Array Types

| Data Type | Description       | Example      |
|-----------|-------------------|--------------|
| ARRAY     | List of values    | tags TEXT[]  |
| INTEGER[] | Array of integers | scores INT[] |

# Data types and constraints (PK, FK, UNIQUE, CHECK)

## 1. PRIMARY KEY (PK)

### What it means:

- Uniquely identifies each row
- Cannot be NULL
- Only one PK per table

### Example:

```
CREATE TABLE customers (  
  customer_id SERIAL PRIMARY  
  KEY,  
  name TEXT NOT NULL  
);
```

### Practical meaning:

Every customer has a unique ID

PostgreSQL automatically creates an index for fast lookups

# Data types and constraints (PK, FK, UNIQUE, CHECK)

## 2. FOREIGN KEY (FK)

### What it means:

- Enforces relationships between tables
- Prevents orphan records

### Example:

```
CREATE TABLE orders (  
  order_id SERIAL PRIMARY KEY,  
  customer_id INT REFERENCES customers(customer_id),  
  order_date DATE  
);
```

What PostgreSQL enforces:

You **cannot insert** an order for a non-existing customer

You **cannot delete** a customer if orders still reference it (unless configured)

### Common options:

REFERENCES customers(customer\_id)

ON DELETE CASCADE -- delete child rows automatically

ON DELETE SET NULL -- keep order but remove customer link

# Data types and constraints (PK, FK, UNIQUE, CHECK)

## 3. UNIQUE

### What it means:

- Ensures values are unique across rows
- Allows only one occurrence of a value

### Example:

```
CREATE TABLE users (  
  user_id SERIAL PRIMARY KEY,  
  email TEXT UNIQUE  
);
```

### Practical use:

No two users can share the same email

PostgreSQL creates an index automatically

### Composite UNIQUE (multiple columns):

UNIQUE (order\_id, product\_id)

# Data types and constraints (PK, FK, UNIQUE, CHECK)

## 4. CHECK

### What it means:

- Enforces custom business rules
- Validates values at insert/update time

### Example:

```
CREATE TABLE products (  
  product_id SERIAL PRIMARY KEY,  
  price NUMERIC CHECK (price > 0),  
  stock INT CHECK (stock >= 0)  
);
```

### More realistic example:

- CHECK (status IN ('pending', 'paid', 'cancelled'))
- CHECK (end\_date >= start\_date)

# Logical and physical modeling

Logical: Abstract design (ERD). Physical: Implementation (tables, indexes).

## Exercises

- Design an ERD for a library system (books, users, loans).
- Implement a physical model with constraints and justify any denormalization.

# Module 2 – Data Modeling

- Entities, attributes, and relationships
- Normalization (1NF, 2NF, 3NF, BCNF)
- Denormalization: when to apply
- Data types and constraints (PK, FK, UNIQUE, CHECK)
- Logical and physical modeling

# Module 3 – Efficient SQL Queries

- Simple and advanced selections (SELECT, JOIN, GROUP BY, HAVING)
- Subqueries and Common Table Expressions (CTEs)
- Indexes and query optimization (EXPLAIN, ANALYZE)
- Views and materialized views

# Simple and advanced selections

SELECT picks columns/rows from tables. JOIN combines rows from two (or more) tables. GROUP BY aggregates rows into groups (e.g., totals per month). HAVING filters groups (like WHERE but for grouped results).

## Example:

```
CREATE TABLE customers (  
  customer_id SERIAL PRIMARY KEY,  
  name      TEXT  
);
```

```
CREATE TABLE orders (  
  order_id   SERIAL PRIMARY KEY,  
  customer_id INT REFERENCES customers(customer_id),  
  order_date DATE,  
  total_amount NUMERIC  
);
```

# Simple and advanced selections

## Simple select:

```
SELECT order_id, order_date, total_amount  
FROM orders  
WHERE total_amount > 100;
```

## JOIN (find customer name for each order):

```
SELECT o.order_id, c.name AS customer, o.total_amount  
FROM orders o  
JOIN customers c ON o.customer_id = c.customer_id;
```

# Simple and advanced selections

## **GROUP BY (total revenue per customer):**

```
SELECT c.name, SUM(o.total_amount) AS revenue  
FROM customers c  
JOIN orders o ON o.customer_id = c.customer_id  
GROUP BY c.name  
ORDER BY revenue DESC;
```

## **HAVING (only customers with revenue > 1000):**

```
SELECT c.name, SUM(o.total_amount) AS revenue  
FROM customers c  
JOIN orders o ON o.customer_id = c.customer_id  
GROUP BY c.name  
HAVING SUM(o.total_amount) > 1000;
```

# Subqueries and Common Table Expressions (CTEs)

A subquery is a SELECT inside another query. A CTE (WITH clause) is a named temporary result set you can reference in the main query — good for readability and breaking complex logic into steps.

## Example:

Subquery example (find customers with last order date)

```
SELECT name
FROM customers
WHERE customer_id IN (
  SELECT customer_id
  FROM orders
  WHERE order_date >= '2025-01-01'
);
```

# Subqueries and Common Table Expressions (CTEs)

A subquery is a SELECT inside another query. A CTE (WITH clause) is a named temporary result set you can reference in the main query — good for readability and breaking complex logic into steps.

## Example:

CTE example (more readable: recent revenue per customer)

```
WITH recent_orders AS (  
  SELECT customer_id, total_amount  
  FROM orders  
  WHERE order_date >= '2025-01-01'  
)  
SELECT c.name, SUM(r.total_amount) AS recent_revenue  
FROM customers c  
JOIN recent_orders r ON r.customer_id = c.customer_id  
GROUP BY c.name;
```

# Indexes and query optimization (EXPLAIN, ANALYZE)

Indexes let PostgreSQL find rows faster (like an index in a book). EXPLAIN shows the planner's chosen plan (Seq Scan vs Index Scan); ANALYZE executes and times.

EXPLAIN ANALYZE runs the query and reports actual times and row counts — essential to diagnose performance.

## Common index types (short)

- **B-tree** — default. Good for equality & range (=, <, >, ORDER BY).
- **GIN** — for jsonb, arrays, full-text.
- **GiST** — geospatial / PostGIS.
- **BRIN** — very large, naturally ordered tables (time-series).

## Practical indexing

- Index FK columns and columns used in WHERE and JOIN conditions.
- Composite indexes help when queries filter on several columns or do ORDER BY.
- Don't add too many indexes — each index slows writes (INSERT/UPDATE/DELETE).
- Use partial indexes for sparse values: `CREATE INDEX ON orders (order_date) WHERE status = 'SHIPPED';`

# Indexes and query optimization (EXPLAIN, ANALYZE)

```
-- create index on customer_id  
CREATE INDEX idx_orders_customer ON orders(customer_id);
```

```
-- examine plan before/after  
EXPLAIN ANALYZE  
SELECT * FROM orders WHERE customer_id = 42;
```

## Interpreting EXPLAIN ANALYZE

- Look if plan uses Index Scan vs Seq Scan.
- Check the "actual time" and number of rows.

## Common pitfalls

- Relying on indexes when queries return large portions of the table (planner may choose seq scan).
- Missing index on join columns.
- Not maintaining ANALYZE / autovacuum — stale stats lead to bad plans.

# Views and materialized views

Views: Virtual, e.g., CREATE VIEW.

A view is a saved query. When you query a view, PostgreSQL expands it into the underlying query — no storage. Good for encapsulation & simplifying SQL for users.

Benefits:

- Simplify complex queries (write once, use many times)
- Abstract underlying table structure
- Security (grant access to view, not base tables)
- Consistency (same logic everywhere)

## Example:

```
CREATE VIEW customer_revenue AS
SELECT c.customer_id, c.name, SUM(o.total_amount) AS revenue
FROM customers c
JOIN orders o ON o.customer_id = c.customer_id
GROUP BY c.customer_id, c.name;
```

**To retrieve the data use the customer\_revenue**

```
SELECT * FROM customer_revenue WHERE revenue > 1000;
```

# Views and materialized views

Materialized: Stored, REFRESH CONCURRENTLY

A materialized view stores the query results physically. Fast reads, but data can become stale — you must REFRESH it.

## Benefits:

- Fast reads (data is pre-computed)
- Can have indexes

## Drawbacks:

- Data can become stale
- Uses disk space
- Must be refreshed manually or on schedule

## Example:

```
CREATE MATERIALIZED VIEW mv_monthly_sales AS
SELECT date_trunc('month', order_date) AS month, SUM(total_amount) AS
revenue
FROM orders
GROUP BY month;
-- to refresh:
REFRESH MATERIALIZED VIEW mv_monthly_sales;
```

# Views and materialized views

| Feature           | View               | Materialized View        |
|-------------------|--------------------|--------------------------|
| Data Storage      | No (virtual)       | Yes (physical)           |
| Query Performance | Same as base query | Fast (pre-computed)      |
| Data Freshness    | Always current     | Snapshot (needs refresh) |
| Can have index    | No                 | Yes                      |
| Disk space        | None               | Use space                |
| Write overhead    | None               | Refresh time             |
| Updatable         | Sometimes          | No                       |
| Good for          | Simple abstraction | Complex aggregations     |
|                   | Security layer     | Reporting / Dashboards   |
|                   | Real-time data     | Search optimization      |

# Views and materialized views

## When to use which

- Use **views** for convenience when underlying data is small/updated often and queries are cheap.
- Use **materialized views** for expensive aggregations used frequently (dashboards), accepting some staleness.

## Performance tips

- Index materialized view if you query it by a column: `CREATE INDEX ON mv_monthly_sales(month);`
- Consider `CONCURRENTLY` refresh if you need reads during refresh: `REFRESH MATERIALIZED VIEW CONCURRENTLY mv_monthly_sales;` (requires unique index).

# Module 3 – Efficient SQL Queries

## Exercises

JOIN query on sample tables.

Optimize slow query with index, compare EXPLAIN.

Recursive CTE for graph traversal.

# Module 4 – Organization and Structuring of Information

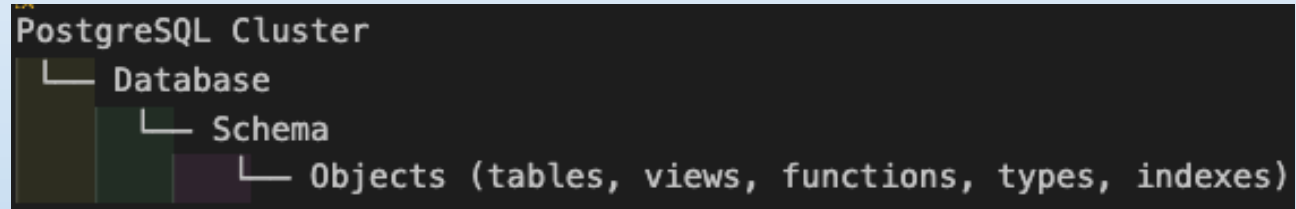
- Creation and maintenance of schemas and their objects
- Partitioned tables
- Document storage (JSON/JSONB)
- Stored procedures and functions (PL/pgSQL)

# Creation and maintenance of schemas and their objects

## What is a schema conceptually?

A **schema** is a **logical namespace** inside a PostgreSQL database.

## PostgreSQL hierarchy:



## Create schema:

```
CREATE SCHEMA academic;
```

## Full object name:

academic.students

## Create table inside schema:

```
CREATE TABLE academic.students (  
    student_id SERIAL PRIMARY KEY,  
    name TEXT NOT NULL  
);
```

# Creation and maintenance of schemas and their objects

## **search\_path**

PostgreSQL looks for objects using search\_path.

```
SHOW search_path;
```

## **Schema security**

```
GRANT USAGE ON SCHEMA academic TO professor;
```

```
GRANT SELECT, INSERT ON ALL TABLES IN SCHEMA academic TO professor;
```

## **Common mistakes**

- ✗ Putting everything in public
- ✗ Not qualifying object names in production
- ✗ Forgetting search\_path behavior

# Partitioned Tables

## What is partitioning really?

Partitioning is **horizontal splitting** of a table into **child tables** (partitions) based on a key.

To PostgreSQL:

- Parent table = logical view
- Partitions = physical storage

## Why partitioning improves performance

- Smaller indexes per partition
- Partition pruning (only relevant partitions scanned)
- Faster deletes (drop partitions)

Types of partitioning

| Type  | Example               |
|-------|-----------------------|
| RANGE | Dates, numeric, range |
| LIST  | Country, region       |
| HASH  | Even distribution     |

# Partitioned Tables

## Example:

### Parente table

```
CREATE TABLE logs (  
    log_id BIGSERIAL,  
    log_date DATE NOT NULL,  
    message TEXT  
) PARTITION BY RANGE (log_date);
```

### Child partitions

```
CREATE TABLE logs_2024 PARTITION OF logs  
FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');
```

```
CREATE TABLE logs_2025 PARTITION OF logs  
FOR VALUES FROM ('2025-01-01') TO ('2026-01-01');
```

### Insert behavior

```
INSERT INTO logs (log_date, message)  
VALUES ('2024-03-01', 'System started');
```

### PostgreSQL routes automatically.

### Partition pruning (important to explain)

```
SELECT * FROM logs WHERE log_date = '2024-03-01';
```

Only logs\_2024 is scanned.

### Maintenance advantage

```
DROP TABLE logs_2024;
```

Deletes millions of rows instantly.

# Document storage (JSON/JSONB)

## JSON vs JSONB

| JSON           | JSONB                |
|----------------|----------------------|
| Stored as text | Stored in binary     |
| Slower queries | Faster queries       |
| No indexing    | Supports GIN indexes |

### Example:

Storing flexible event metadata.

```
CREATE TABLE events (  
  event_id SERIAL PRIMARY KEY,  
  event_data JSONB NOT NULL  
);
```

# Document storage (JSON/JSONB)

## JSON vs JSONB

### Insert document

```
INSERT INTO events (event_data)
VALUES (
    '{
      "type": "purchase",
      "user": "alice",
      "items": [
        {"product": "A", "qty": 2},
        {"product": "B", "qty": 1}
      ],
      "total": 120.50
    }'
);
```

### Query JSONB

```
SELECT event_data->>'user' AS username
FROM events
WHERE event_data->>'type' = 'purchase';
```

### Query nested data

```
SELECT event_data->'items'
FROM events;
```

### Index JSONB

```
CREATE INDEX idx_events_data ON events USING GIN
(event_data);
```

# Stored procedures and functions (PL/pgSQL)

Procedure is used to perform database actions with PL/SQL.

Function is used to perform a calculation and return a result.

| Feature        | Function | Procedure |
|----------------|----------|-----------|
| Returns value  | ✓        | ✗         |
| Used in SELECT | ✓        | ✗         |
| Transactions   | ✗        | ✓         |

## Function

```
CREATE FUNCTION calculate_final_price(  
  base_price NUMERIC,  
  vat NUMERIC  
)  
RETURNS NUMERIC AS $$  
BEGIN  
  RETURN base_price + (base_price * vat);  
END;  
$$ LANGUAGE plpgsql;
```

## Usage

```
SELECT calculate_final_price(100, 0.23);
```

# Stored procedures and functions (PL/pgSQL)

## Procedure

```
CREATE PROCEDURE register_student(  
    student_name TEXT  
)  
LANGUAGE plpgsql  
AS $$  
BEGIN  
    INSERT INTO academic.students (name)  
    VALUES (student_name);  
END;  
$$;
```

## CALL:

```
CALL register_student('Maria');
```

## Error handling

```
CREATE OR REPLACE PROCEDURE academic.register_student(  
    student_name TEXT  
)  
LANGUAGE plpgsql  
AS $$  
BEGIN  
    BEGIN  
        INSERT INTO academic.students (name)  
        VALUES (student_name);  
    EXCEPTION  
        WHEN unique_violation THEN  
            RAISE NOTICE 'Student "%" already exists', student_name;  
    END;  
END;  
$$;
```

# Module 4 – Organization and Structuring of Information

## Exercises

- Create schema, tables.
- Partition log table, insert across partitions.
- PL/pgSQL proc to update JSONB.

# Module 4 – Organization and Structuring of Information

- Creation and maintenance of schemas and their objects
- Partitioned tables
- Document storage (JSON/JSONB)
- Stored procedures and functions (PL/pgSQL)

# Module 5 – Advanced Types and Extensions

- Arrays
- Geometric Types
- JSON Functions and Operators
- PostGIS

# Arrays

## Concept (what & why)

PostgreSQL supports **array types**, meaning a column can store **multiple values of the same type**.

Think of an array as: “a list inside a single cell”

This is useful when:

- The number of values is small and bounded
- Values belong tightly to the row
- You don't need complex relationships

# Arrays

```
CREATE TABLE students (  
  student_id SERIAL PRIMARY KEY,  
  name TEXT,  
  skills TEXT[]  
);
```

```
INSERT INTO students (name, skills)  
VALUES ('Ana', ARRAY['SQL', 'Python', 'Statistics']);
```

## Reading array values

```
SELECT skills FROM students;
```

## Access a specific position (1-based indexing):

```
SELECT skills[1] FROM students;
```

## Searching in arrays

```
SELECT * FROM students  
WHERE 'SQL' = ANY (skills);
```

## Check if array contains values:

```
SELECT * FROM students  
WHERE skills @> ARRAY['Python'];
```

## Updating arrays

Append element:

```
UPDATE students  
SET skills = array_append(skills, 'AI')  
WHERE name = 'Ana';
```

Remove element:

```
UPDATE students  
SET skills = array_remove(skills, 'Statistics')  
WHERE name = 'Ana';
```

## Indexing arrays

Use **GIN indexes**:

```
CREATE INDEX idx_students_skills ON students USING GIN (skills);
```

# Geometric Types

PostgreSQL includes **built-in geometric data types** to represent **2D spatial objects**, independent of PostGIS.

These are useful for:

- Simple geometry
- Games
- Layouts
- Basic spatial constraints

| Type    | Meaning            |
|---------|--------------------|
| POINT   | (x, y) coordinate  |
| LINE    | Infinite line      |
| LSEG    | Line segment       |
| BOX     | Rectangle          |
| PATH    | Sequence of points |
| POLYGON | Closed shape       |
| CIRCLE  | Center + radius    |

# Geometric Types

## **Example:**

### **storing points**

```
CREATE TABLE sensors (  
    sensor_id SERIAL PRIMARY KEY,  
    location POINT  
);
```

## **Insert:**

```
INSERT INTO sensors (location) VALUES (POINT(10, 20));
```

## **Geometric operations**

### **Distance between points:**

```
SELECT location <-> POINT(15, 25) as distance  
FROM sensors;
```

### **Check containment:**

```
SELECT * FROM sensors  
WHERE location <@ BOX '(0,0),(100,100)';
```

# JSON Functions and Operators

| Operator | Meaning                  |
|----------|--------------------------|
| ->       | Get JSON object          |
| ->>      | Get value as text        |
| #>       | Get nested object        |
| #>>      | Get nested value as text |
| @>       | Containment              |

## Example JSONB table

```
CREATE TABLE orders (  
  order_id SERIAL PRIMARY KEY,  
  data JSONB  
);
```

## Insert:

```
INSERT INTO orders (data)  
VALUES (  
  '{  
    "customer": "Alice",  
    "items": [  
      {"product": "Book", "qty": 2},  
      {"product": "Pen", "qty": 5}  
    ],  
    "total": 45.50  
  }'  
);
```

# JSON Functions and Operators

## Query JSON fields

```
SELECT data->>'customer' as customer_name  
FROM orders;
```

## Nested access:

```
SELECT data#>>'{items,0,product}' as product_name  
FROM orders;
```

## Filtering JSON

```
SELECT * FROM orders  
WHERE data @> '{"customer": "Alice"}';
```

## JSON functions

### Extract elements from arrays:

```
SELECT jsonb_array_elements(data->'items') as items  
FROM orders;
```

### OR specify to columns to table

```
SELECT  
    item->>'product' AS product,  
    (item->>'qty')::int AS quantity  
FROM orders o,  
     jsonb_array_elements(o.data->'items') AS item;
```

### Convert JSON to table:

```
SELECT  
    x.customer,  
    x.total  
FROM orders o,  
     jsonb_to_record(o.data) AS x(  
        customer TEXT,  
        total NUMERIC  
    );
```

# JSON Functions and Operators

JSON functions always work on JSON columns inside tables, and we must explicitly reference the table and column to extract arrays, convert JSON into columns, or index JSON efficiently.

| Task                    | Function                                                         |
|-------------------------|------------------------------------------------------------------|
| Extract array elements  | <code>jsonb_array_elements()</code>                              |
| Extract text value      | <code>-&gt;&gt;</code>                                           |
| Convert JSON to columns | <code>jsonb_to_record()</code>                                   |
| Index JSON              | GIN                                                              |
| Query JSON              | <code>@&gt;</code> , <code>-&gt;</code> , <code>-&gt;&gt;</code> |

## Indexing JSONB

```
CREATE INDEX idx_orders_data ON orders USING GIN (data);
```

## Index a specific JSON path

```
CREATE INDEX idx_orders_customer ON orders USING GIN ((data->'customer'));
```

## Index a specific JSON path for text search

```
CREATE INDEX idx_orders_customer_text ON academic.orders ((data->>'customer'));
```

# JSON and JSONB Operators

| Operator | Right Operand Type | Description                                | Example                                             | Example Result |
|----------|--------------------|--------------------------------------------|-----------------------------------------------------|----------------|
| ->       | int                | Get JSON array element (indexed from zero) | '[{"a":"foo"}, {"b":"bar"}, {"c":"baz"}] '::json->2 | { "c": "baz" } |
| ->       | text               | Get JSON object field by key               | '{"a": {"b":"foo"}} '::json->'a'                    | { "b": "foo" } |
| ->>      | int                | Get JSON array element as text             | '[1,2,3] '::json->>2                                | 3              |
| ->>      | text               | Get JSON object field as text              | '{"a":1, "b":2} '::json->>'b'                       | 2              |
| #>       | text[]             | Get JSON object at specified path          | '{"a": {"b":{"c": "foo"}}} '::json#>'a,b'           | { "c": "foo" } |
| #>>      | text[]             | Get JSON object at specified path as text  | '{"a": [1,2,3], "b": [4,5,6]} '::json#>>'a,2'       | 3              |

Table 9-41. Additional jsonb Operators

| Operator | Right Operand Type | Description                                                     | Example                                            |
|----------|--------------------|-----------------------------------------------------------------|----------------------------------------------------|
| =        | jsonb              | Are the two JSON values equal?                                  | '[1,2,3] '::jsonb = '[1,2,3] '::jsonb              |
| @>       | jsonb              | Does the left JSON value contain within it the right value?     | '{"a":1, "b":2} '::jsonb @> '{"b":2} '::jsonb      |
| <@       | jsonb              | Is the left JSON value contained within the right value?        | '{"b":2} '::jsonb <@ '{"a":1, "b":2} '::jsonb      |
| ?        | text               | Does the key/element <b>string</b> exist within the JSON value? | '{"a":1, "b":2} '::jsonb ? 'b'                     |
| ?        | text[]             | Do any of these key/element <b>strings</b> exist?               | '{"a":1, "b":2, "c":3} '::jsonb ?  array['b', 'c'] |
| ?&       | text[]             | Do all of these key/element <b>strings</b> exist?               | '["a", "b"] '::jsonb ?& array['a', 'b']            |

# JSON Creation Functions

| Function                                             | Description                                                                                                                                                                                                                                                                                                                                                                                                                                            | Example                                                                                                               | Example Result                                  |
|------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|-------------------------------------------------|
| <code>to_json(anyelement)</code>                     | Returns the value as JSON. Arrays and composites are converted (recursively) to arrays and objects; otherwise, if there is a cast from the type to <code>json</code> , the cast function will be used to perform the conversion; otherwise, a JSON scalar value is produced. For any scalar type other than a number, a boolean, or a null value, the text representation will be used, properly quoted and escaped so that it is a valid JSON string. | <code>to_json('Fred said "Hi."::text)</code>                                                                          | <code>"Fred said \"Hi.\""</code>                |
| <code>array_to_json(anyarray [, pretty_bool])</code> | Returns the array as a JSON array. A PostgreSQL multidimensional array becomes a JSON array of arrays. Line feeds will be added between dimension-1 elements if <code>pretty_bool</code> is true.                                                                                                                                                                                                                                                      | <code>array_to_json('{{1,5},{99,100}}'::int[])</code>                                                                 | <code>[[1,5],[99,100]]</code>                   |
| <code>row_to_json(record [, pretty_bool])</code>     | Returns the row as a JSON object. Line feeds will be added between level-1 elements if <code>pretty_bool</code> is true.                                                                                                                                                                                                                                                                                                                               | <code>row_to_json(row(1,'foo'))</code>                                                                                | <code>{"f1":1,"f2":"foo"}</code>                |
| <code>json_build_array(VARIADIC "any")</code>        | Builds a possibly-heterogeneously-typed JSON array out of a variadic argument list.                                                                                                                                                                                                                                                                                                                                                                    | <code>json_build_array(1,2,'3',4,5)</code>                                                                            | <code>[1, 2, "3", 4, 5]</code>                  |
| <code>json_build_object(VARIADIC "any")</code>       | Builds a JSON object out of a variadic argument list. By convention, the argument list consists of alternating names and values.                                                                                                                                                                                                                                                                                                                       | <code>json_build_object('foo',1,'bar',2)</code>                                                                       | <code>{"foo": 1, "bar": 2}</code>               |
| <code>json_object(text[])</code>                     | Builds a JSON object out of a text array. The array must have either exactly one dimension with an even number of members, in which case they are taken as alternating name/value pairs, or two dimensions such that each inner array has exactly two elements, which are taken as a name/value pair.                                                                                                                                                  | <code>json_object('{a, 1, b, "def", c, 3.5}')</code><br><br><code>json_object('{{a, 1},{b, "def"}},{c, 3.5}}')</code> | <code>{"a": "1", "b": "def", "c": "3.5"}</code> |
| <code>json_object(keys text[], values text[])</code> | This form of <code>json_object</code> takes keys and values pairwise from two separate arrays. In all other respects it is identical to the one-argument form.                                                                                                                                                                                                                                                                                         | <code>json_object('{a, b}', '{1,2}')</code>                                                                           | <code>{"a": "1", "b": "2"}</code>               |

# PostGIS (Geographic Information System)

## What is PostGIS?

PostGIS is a **PostgreSQL extension** that adds:

- Geographic data types
- Spatial indexes
- Hundreds of spatial functions

It turns PostgreSQL into a **full GIS database**.

## Enable PostGIS

`CREATE EXTENSION postgis;`

For docker use a postgis image!

## Core PostGIS types

| Type                       | Description       |
|----------------------------|-------------------|
| GEOMETRY                   | Planar (2D/3D)    |
| GEOGRAPHY                  | Earth coordinates |
| POINT, LINESTRING, POLYGON | Spatial objects   |

# PostGIS (Geographic Information System)

## Example:

### storing locations

```
CREATE TABLE cities (  
  city_id SERIAL PRIMARY KEY,  
  name TEXT,  
  location GEOGRAPHY(POINT, 4326)  
);
```

### Insert:

```
INSERT INTO cities (name, location)  
VALUES (  
  'Lisbon',  
  ST_SetSRID(ST_MakePoint(-9.1393, 38.7223), 4326)  
);
```

### Distance calculation

```
SELECT name,  
  ST_Distance(  
    location,  
    ST_SetSRID(ST_MakePoint(-8.6291, 41.1579), 4326)  
  ) / 1000 AS distance_km  
FROM cities;
```

# PostGIS (Geographic Information System)

## Spatial queries

**Find cities within 50km:**

```
SELECT name
FROM cities
WHERE ST_DWithin(
  location,
  ST_SetSRID(ST_MakePoint(-9.1393, 38.7223), 4326),
  50000
);
```

## Spatial indexes (critical)

```
CREATE INDEX idx_cities_location ON cities USING GIST
(location);
```

| Feature         | Use when              |
|-----------------|-----------------------|
| Arrays          | Small lists per row   |
| Geometric Types | Simple 2D geometry    |
| JSONB           | Semi-structured data  |
| PostGIS         | Real spatial/GIS data |

# Module 5 – Advanced Types and Extensions

**Exercises (Using Docker, PostGIS image for advanced)**

Array insert/query.

Geometric containment checks.

PostGIS spatial query on world cities.

# Module 6 – Security and Access Control

- Creation and management of users and roles
- Permissions: GRANT, REVOKE
- Policies and Row-Level Security (RLS)
- Encryption, backups and restoration
- Audit and compliance logs

# Creation and management of users and roles

## Users and Roles

### Core concept

In PostgreSQL, **users and roles are the same thing**.

A **role** can:

- Own database objects
- Have permissions
- Be assigned to other roles
- Act as a user if it has LOGIN

👉 Think of roles as **security identities**.

# Creation and management of users and roles

## Creating roles

### Create a login role (user):

```
CREATE ROLE alice LOGIN PASSWORD 'strong_password';
```

### Create a role without login (group role):

```
CREATE ROLE app_readonly;
```

### Assign role to user:

```
GRANT app_readonly TO alice;
```

## Role attributes

```
CREATE ROLE admin LOGIN
```

```
SUPERUSER
```

```
CREATEDB
```

```
CREATEROLE;
```

## Common attributes:

- LOGIN – can connect
- SUPERUSER – full access (avoid!)
- CREATEDB – create databases
- CREATEROLE – manage roles

# Permissions: GRANT, REVOKE

## Concept

Permissions control **who can do what** on **which objects**.

Objects include:

- Tables
- Views
- Sequences
- Functions
- Schemas

# Permissions: GRANT, REVOKE

## Table privileges

GRANT SELECT, INSERT ON orders TO app\_user;

Revoke:

REVOKE INSERT ON orders FROM app\_user;

## Schema permissions (often forgotten)

GRANT USAGE ON SCHEMA public TO app\_user; ← Without this, table access fails even with SELECT.

**Default privileges** ← Automatically grant permissions for future objects:

ALTER DEFAULT PRIVILEGES

GRANT SELECT ON TABLES TO app\_readonly;

## Example

CREATE ROLE reporting;

GRANT SELECT ON ALL TABLES IN SCHEMA public TO reporting;

# Policies and Row-Level Security (RLS)

## Concept

**Row-Level Security** restricts access to **rows**, not tables.

Two users can query the same table but see **different rows**.

## Use cases

- Multi-tenant systems
- User-specific data
- GDPR data isolation

# Policies and Row-Level Security (RLS)

## Example: employee data

```
CREATE TABLE employees (  
  id SERIAL PRIMARY KEY,  
  name TEXT,  
  department TEXT,  
  username TEXT  
);
```

## Enable RLS:

```
ALTER TABLE employees ENABLE ROW LEVEL SECURITY;
```

## Create policy

```
CREATE POLICY employee_policy  
ON employees  
USING (username = current_user);
```

## Result:

Each user sees only **their own row**

## Insert/update policies

```
CREATE POLICY employee_insert  
ON employees  
FOR INSERT  
WITH CHECK (username = current_user);
```

## Important rule

- ! RLS does **nothing** unless enabled
- ! Superusers bypass RLS

# Encryption, backups and restoration

## Encryption

### Encryption in transit (SSL)

ssl = on

Clients connect securely:

```
psql "sslmode=require"
```

### Encryption at rest

PostgreSQL does **not** encrypt data natively.

Use:

- Disk encryption (LUKS)
- Cloud provider encryption
- Encrypted file systems

# Encryption, backups and restoration

## Column-level encryption

```
CREATE EXTENSION pgcrypto;
```

```
INSERT INTO users (password)  
VALUES (crypt('mypassword', gen_salt('bf')));
```

## Verify:

```
SELECT * FROM users  
WHERE password = crypt('mypassword', password);
```

# Encryption, backups and restoration

## Backups

### Logical backups

```
pg_dump mydb > mydb.sql
```

### Restore:

```
psql mydb < mydb.sql
```

### Physical backups

```
pg_basebackup -D backup_dir
```

Used for:

- Replication
- Point-in-time recovery (PITR)

# Audit and compliance logs

## Logging configuration

In postgresql.conf:

```
logging_collector = on  
log_statement = 'ddl'  
log_connections = on  
log_disconnections = on
```

## pgAudit (advanced auditing)

### Enable:

```
CREATE EXTENSION pgaudit;
```

### Configure:

```
pgaudit.log = 'read, write, ddl'
```

## What pgAudit logs

- SELECT, INSERT, UPDATE, DELETE
- Schema changes
- Role changes

## Compliance use cases

- GDPR
- HIPAA
- ISO 27001
- Financial audits

# Module 6 – Security and Access Control

| Feature      | Purpose                   |
|--------------|---------------------------|
| Users/Roles  | Identity management       |
| GRANT/REVOKE | Permission control        |
| RLS          | Row-level data isolation  |
| Encryption   | Data protection           |
| Backups      | Disaster recovery         |
| Auditing     | Compliance & traceability |

# Module 6 – Security and Access Control

## Exercises

# Module 7 – Monitoring and Performance

- Monitoring with `pg_stat_activity`, `pg_stat_user_tables`, etc.
- Indexing: types of indexes and strategic use
- Performance tuning and parameters
- Integration with monitoring tools (PgAdmin, Grafana, Prometheus)

# Monitoring with pg\_stat\_activity, pg\_stat\_user\_tables

## Monitoring PostgreSQL Internals

PostgreSQL exposes its internal state through **system catalog views** (pg\_stat\_\*).

These views are your **first diagnostic tool**.

### pg\_stat\_activity – Session & Query Monitoring

#### What it shows

- Who is connected
- What queries are running
- Query duration
- Locks & waiting state

**“Task Manager for PostgreSQL”**

# Monitoring with pg\_stat\_activity, pg\_stat\_user\_tables

| Column          | Meaning                           |
|-----------------|-----------------------------------|
| pid             | Backend process ID                |
| username        | Database user                     |
| state           | active, idle, idle in transaction |
| query           | Current SQL                       |
| query_start     | When query began                  |
| wait_event_type | Lock, IO, CPU                     |

# Monitoring with pg\_stat\_activity, pg\_stat\_user\_tables

## Practical queries

### List active queries:

```
SELECT pid, username, state, query  
FROM pg_stat_activity  
WHERE state = 'active';
```

### Long-running queries:

```
SELECT pid, now() - query_start AS runtime, query  
FROM pg_stat_activity  
WHERE state = 'active'  
ORDER BY runtime DESC;
```

### Detect idle transactions (dangerous): ❌

```
SELECT pid, query  
FROM pg_stat_activity  
WHERE state = 'idle in transaction';
```

# Monitoring with pg\_stat\_activity, pg\_stat\_user\_tables

## pg\_stat\_user\_tables – Table-Level Statistics

### What it shows

- How tables are used:
- Reads (seq vs index)
- Inserts/updates/deletes
- Dead tuples
- Vacuum activity

| Column          | Meaning            |
|-----------------|--------------------|
| seq_scan        | Sequential scans   |
| idx_scan        | Index scans        |
| n_dead_tup      | Dead rows          |
| last_vacuum     | Last manual vacuum |
| last_autovacuum | Last auto vacuum   |

# Monitoring with pg\_stat\_activity, pg\_stat\_user\_tables

## Practical query

```
SELECT relname, seq_scan, idx_scan, n_dead_tup  
FROM pg_stat_user_tables  
ORDER BY n_dead_tup DESC;
```

## Interpretation

- High seq\_scan + low idx\_scan → missing indexes
- High n\_dead\_tup → vacuum pressure

# Indexing: types of indexes and strategic use

## Indexing – Types and Strategic Use

Indexes trade **write cost** for **read speed**.

- **B-tree (default)**

Use when

- Equality (=)
- Range queries (<, >, BETWEEN)
- ORDER BY

```
CREATE INDEX idx_orders_date ON orders(order_date);
```

- **Hash Index**

Use when

- Equality only
- Rarely better than B-tree

```
CREATE INDEX idx_users_email_hash ON users USING HASH (email);
```

 **Rarely recommended.**

# Indexing: types of indexes and strategic use

## Indexing – Types and Strategic Use

- **GIN (Generalized Inverted Index)**

**Use when**

- Arrays
- JSONB
- Full-text search

```
CREATE INDEX idx_users_profile ON users USING GIN (profile);
```

- **GiST (Generalized Search Tree)**

**Use when**

- Geospatial (PostGIS)
- Ranges
- Similarity search

```
CREATE INDEX idx_locations ON places USING GIST (location);
```

# Indexing: types of indexes and strategic use

## Indexing – Types and Strategic Use

- **BRIN (Block Range Index)**

**Use when**

- Very large tables
- Naturally ordered data (time-series)

```
CREATE INDEX idx_logs_time ON logs USING BRIN (created_at);
```

## Index selection strategy

| Scenario     | Index  |
|--------------|--------|
| High reads   | B-tree |
| JSON queries | GIN    |
| Geo data     | GiST   |
| Huge tables  | BRIN   |

# Performance tuning and parameters

PostgreSQL performance tuning balances:

- Memory
- CPU
- Disk I/O
- Query patterns

## Memory Parameters

### **shared\_buffers**

Memory used for caching data pages.

- `shared_buffers` = 25% of RAM

### **work\_mem**

Memory per operation (sort, hash join).

- `work_mem` = 16MB
  - Too high → OOM
  - Too low → disk spills

### **maintenance\_work\_mem**

Used by VACUUM, CREATE INDEX.

- `maintenance_work_mem` = 512MB

# Performance tuning and parameters

## Planner & I/O

### **effective\_cache\_size**

Estimate of OS cache.

- `effective_cache_size` = 75% of RAM

### **random\_page\_cost**

Cost of random disk access.

- `random_page_cost` = 1.1 # SSD

# Performance tuning and parameters

## **Autovacuum (critical)**

autovacuum = on

autovacuum\_vacuum\_scale\_factor = 0.2

Autovacuum:

- Cleans dead rows
- Prevents table bloat
- Updates statistics

# Performance tuning and parameters

## Query Analysis – EXPLAIN

### Why EXPLAIN matters

It shows **how PostgreSQL executes a query**.

### Example:

```
EXPLAIN ANALYZE
```

```
SELECT * FROM orders WHERE customer_id = 10;
```

Look for:

- Seq Scan vs Index Scan
- Execution time
- Rows processed

# Integration with monitoring tools (PgAdmin, Grafana, Prometheus)

## PgAdmin

### Features

- Live sessions view
- Query tool
- Dashboard metrics

### Best for:

- Beginners
- DBAs exploring issues

# Integration with monitoring tools (PgAdmin, Grafana, Prometheus)

## Prometheus + PostgreSQL Exporter

### Architecture

PostgreSQL → Exporter → Prometheus → Grafana

Exporter collects:

- Connections
- Transactions
- Cache hit ratio
- Replication lag

# Integration with monitoring tools (PgAdmin, Grafana, Prometheus)

## Grafana Dashboards

Visualize:

- QPS
- Latency
- Locks
- Autovacuum activity

Use for:

- Production monitoring
- Trend analysis
- Alerts

# Integration with monitoring tools (PgAdmin, Grafana, Prometheus)

## Performance Tuning Workflow

- Identify slow queries (pg\_stat\_activity)
- Analyze table usage (pg\_stat\_user\_tables)
- Use EXPLAIN ANALYZE
- Add indexes
- Tune memory parameters
- Monitor continuously

# Module 7 – Monitoring and Performance

## Exercises

# Module 8 – Strategic Data and Indicators

- Organizing data for reports
- Creating dashboards and reports with SQL
- KPIs and performance indicators
- Exporting data for external use (CSV, JSON, API)

# Organizing data for reports

## Operational vs Analytical Data

PostgreSQL is often used for:

- **OLTP (transactions)** → normalized tables
- **OLAP (analytics)** → aggregated, report-friendly structures

### Key principle

Reporting queries should not fight the transactional model.

## Reporting-Friendly Data Models

### a) Denormalized reporting tables

Pre-join and store data for fast reads.

```
CREATE TABLE sales_report (  
  order_id INT,  
  order_date DATE,  
  customer_name TEXT,  
  product_name TEXT,  
  quantity INT,  
  total_amount NUMERIC  
);
```

✓ Fast dashboards

✗ Data duplication (acceptable for reporting)

# Organizing data for reports

## b) Star schema (classic BI)

- **Fact table** → measurable data
- **Dimension tables** → descriptive data

### Example:

fact\_sales

- ↳ dim\_date
- ↳ dim\_customer
- ↳ dim\_product

```
CREATE TABLE fact_sales (  
  date_id INT,  
  customer_id INT,  
  product_id INT,  
  revenue NUMERIC  
);
```

# Organizing data for reports

## c) Materialized views (very important)

```
CREATE MATERIALIZED VIEW monthly_sales AS  
SELECT date_trunc('month', order_date) AS month,  
       SUM(total_amount) AS revenue  
FROM orders  
GROUP BY 1;
```

### **Refresh:**

```
REFRESH MATERIALIZED VIEW monthly_sales;
```

# Creating dashboards and reports with SQL

## Aggregation Fundamentals

```
SELECT
    date_trunc('month', order_date) AS month,
    COUNT(*) AS orders,
    SUM(total_amount) AS revenue,
    AVG(total_amount) AS avg_order_value
FROM orders
GROUP BY month
ORDER BY month;
```

## Window Functions (Dashboard Power)

Window functions allow **KPIs over time** without collapsing rows.

```
SELECT
    order_date,
    total_amount,
    SUM(total_amount) OVER (ORDER BY order_date) AS
running_total
FROM orders;
```

# Creating dashboards and reports with SQL

## Rankings & comparisons

```
SELECT
    product_id,
    SUM(total_amount) AS revenue,
    RANK() OVER (ORDER BY SUM(total_amount) DESC) AS rank
FROM sales
GROUP BY product_id;
```

## Time-based reports

```
SELECT
    date_trunc('week', order_date) AS week,
    SUM(total_amount) AS revenue
FROM orders
GROUP BY week;
```

# KPIs and performance indicators

## What is a KPI?

A **KPI (Key Performance Indicator)** is:

- Quantitative
- Measurable
- Actionable

Examples:

- Revenue growth
- Conversion rate
- Average order value (AOV)
- Customer retention

# KPIs and performance indicators

## KPI examples in SQL

### Revenue

```
SELECT SUM(total_amount) AS total_revenue FROM orders;
```

### Average Order Value (AOV)

```
SELECT AVG(total_amount) AS avg_order_value FROM orders;
```

### Month-over-Month Growth

```
SELECT  
  month,  
  revenue,  
  revenue - LAG(revenue) OVER (ORDER BY month) AS growth  
FROM monthly_sales;
```

### Conversion Rate

```
SELECT  
  COUNT(DISTINCT order_id)::float /  
  COUNT(DISTINCT session_id) AS conversion_rate  
FROM sessions;
```

# KPIs and performance indicators

## KPI tables

Store KPI results for dashboards.

```
CREATE TABLE kpi_daily (  
  kpi_date DATE,  
  revenue NUMERIC,  
  orders INT,  
  avg_order NUMERIC  
);
```

Populate via scheduled jobs.

# Exporting data for external use (CSV, JSON, API)

## **Export to CSV**

```
COPY (  
  SELECT * FROM monthly_sales  
) TO '/tmp/monthly_sales.csv' CSV HEADER;
```

Client-side (psql):

```
\copy monthly_sales TO 'monthly_sales.csv' CSV HEADER;
```

## **Export to JSON**

```
SELECT json_agg(monthly_sales)  
FROM monthly_sales;
```

## **Pretty format:**

```
SELECT jsonb_pretty(json_agg(row_to_json(ms)))  
FROM monthly_sales ms;
```

# Exporting data for external use (CSV, JSON, API)

## PostgreSQL as an API backend

Generate API-ready JSON:

```
SELECT json_build_object(  
  'month', month,  
  'revenue', revenue  
)  
FROM monthly_sales;
```

Used by:

- Node.js
- Python
- BI tools

## Integration with BI tools

PostgreSQL connects easily to:

- Power BI
- Tableau
- Metabase
- Superset

Dashboards usually query:

- Views
- Materialized views
- KPI tables

# Exporting data for external use (CSV, JSON, API)

## Strategic Design Best Practices

- ✓ Separate OLTP from reporting
- ✓ Use materialized views for heavy KPIs
- ✓ Precompute what changes slowly
- ✓ Keep raw data for traceability
- ✓ Version KPIs (definitions change!)

# Module 8 – Strategic Data and Indicators

## Exercises

# Postgresql

**Thank you!**

**Ailton Moreira**