



# Advanced SQL with Postgres

Marc Linster  
Oct 2025

# Lesson Plan

- **14:00 - 17:00**
- **Start with review of prior day**
- **End with time to try things out and ask questions**
- **Hands on**

Monday	Quick intro to key concepts of relational databases Connecting to the Postgres database w. pgAdmin and psql Tables and query basics: Insert, update, delete, select <i>Time to exercise and ask questions</i>
Tuesday	<i>Review of prior session and Q&amp;A</i> Operators in the WHERE clause and computations in the SELECT clause Primary keys, constraints, and foreign key relationships Data Types: Numbers, Characters, JSON, and more <i>Time to exercise and ask questions</i>
Wednesday	<i>Review of prior session and Q&amp;A</i> Indexes Organizing data with schemas and entity-relationship diagrams Queries revisited (limit, distinct, not in, joins, views, and aggregates) <i>Time to exercise and ask questions</i>
Thursday	<i>Review of prior session and Q&amp;A</i> Data Normalization ACID compliance, Transactions, and Isolation Levels A simple project: Creating your own database with schema, tables, and foreign keys
Friday	<i>Review of prior session and Q&amp;A</i> Completing the project Stored procedures Common Table Expressions (CTE) Recap, Q&A, resources



# Intro and Advanced SQL Classes

## Practical Introduction to SQL with Postgres

Audience: Beginners

Tool: Graphical UI (pgAdmin)

Duration: 3 days @ 3 hours

Goal:

- Familiarize with SQL and relational databases
- Write simple queries
- Read and understand data models

## Advanced SQL with Postgres

Audience: Prior hands-on experience with database or programming tools

Tool: Command line (psql) & GUI (pgAdmin)

Duration: 5 days @ 3 hours

Goal:

- Read and write complex queries
- Understand transactions, indexes, and analytics concepts
- Develop normalized data models
- Develop stored procedures

# Day 1

- Connect to the VMs
  - Quick intro to key concepts of relational databases
  - Connecting to the Postgres database w. pgAdmin and psql
  - Tables and query basics: Insert, update, delete, select
-

---

# Connecting to the DLH Course VMs

---

# Quick Intro to Databases



# What is a database?

A **database** is an organized collection of structured data, typically stored electronically in a computer system.

## Use Case Examples

- All orders that were taken, shipped, invoiced and paid in a store or eCommerce applications
- List of all laws that are currently in place
- Register of all the inhabitants of a country with their age, marital status, and domicile
- Collections of documents, X-ray images, chat conversations, and emails
- The WWW
- Data can include numbers, text, geo-locations, images, videos, links, vectors, lists, ...

---

# Key Functions of a Database

- **Data Storage:** Securely store large volumes of data.
- **Data Retrieval:** Quickly fetch specific data using queries.
- **Data Manipulation:** Insert, update, or delete data as needed.
- **Data Integrity:** Ensure accuracy and consistency through constraints and rules.
- **Data Security:** Protect data from unauthorized access with authentication and encryption.
- **Data Analysis:** Support complex queries and reporting for decision-making.

---

# Components of a Database

- **Data:** The actual information stored, such as customer records, product details, or transaction logs.
- **Database Management System (DBMS):** Software that interacts with the database, enabling users to define, create, query, update, and manage data. Examples include MySQL, PostgreSQL, Oracle, and MongoDB.
- **Schema:** The structure or blueprint of the database, defining how data is organized (e.g., tables, fields, relationships).
- **Query Language:** A language used to interact with the database, such as SQL (Structured Query Language) for relational databases.

# History of Databases

## 1960s: The Birth of Databases

- File-Based Systems
- Hierarchical and Network Models

## Relational commercialization and initial open source

- Oracle, IBM DB2, and MS SQL Server
- Postgres
- MySQL
- Teradata

## Cloud databases and Open Source

- Postgres and MySQL on public cloud
- MongoDB Atlas
- Postgres integration of NoSQL and parallel query
- Databricks, Snowflake on public cloud

1970s

2000s

2010s

2020s

1960s

1980-90s

### Relational Databases

- E.F. Codd introduces relational model
- IBM System R
- Postgres precursor Ingres

### Rise of Big Data and NoSQL

- Explosion of data (Internet and IoT)
- NoSQL databases (MongoDB, Cassandra, Redis)
- Postgres maturing
- Postgres analytics fork: Greenplum

### Rise of AI

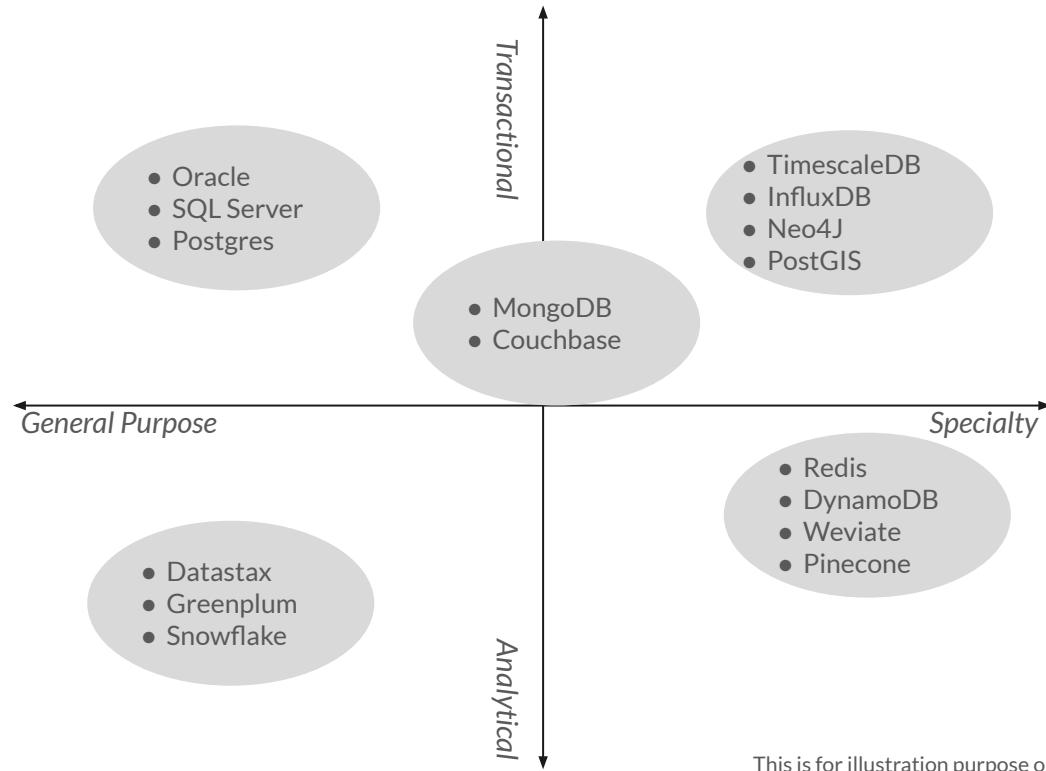
- Vector databases
- Open data formats
- Postgres w. AI integration



# Principal Types of Databases

- There are many, many types of databases
- Key database types (not exhaustive)
  - Relational ⇐ Our focus this week (Oracle, MySQL, Postgres, SQL Server, IBM DB2, Sybase, ...)
  - Document (MongoDB, Couchbase, ...)
  - Key Value Pair (Redis, RocksDB, Amazon Dynamo, ...)
  - Graph (Neo4J)
  - AI and vector (Pinecone, Weviate, ...)
- Features are merging!
  - Relational databases are adopting graph, document, KVP, and AI vector capabilities

# Database Landscape

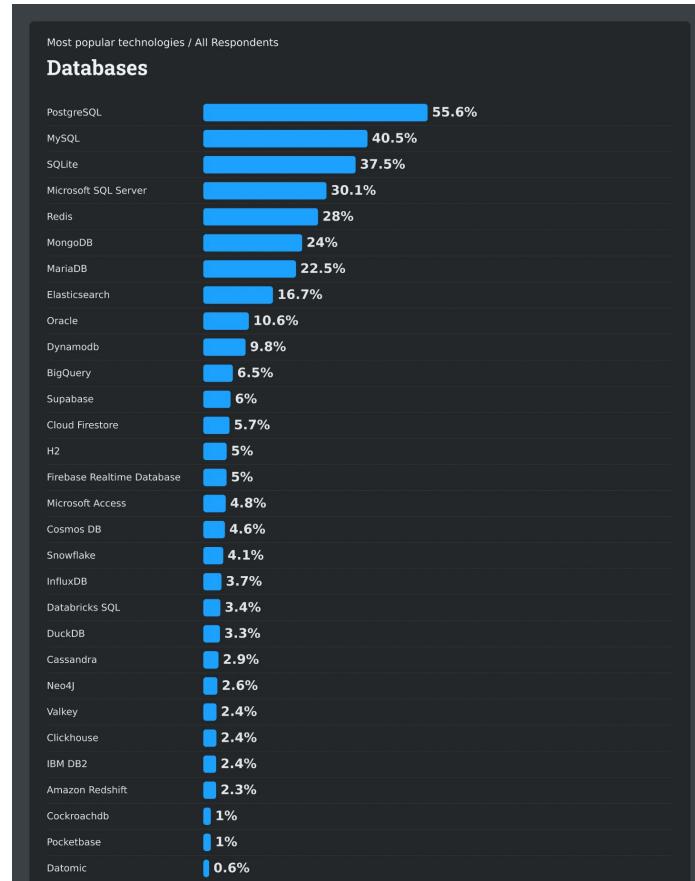


This is for illustration purpose only  
There are many, many databases and many ways to classify them

# Stackoverflow Developer Survey 2025



Year	Rank	Developers using Postgres	Key Competitors
2025	#1	55.6%	MySQL (40.5%), SQLite (37.5%)
2024	#1	51%	MySQL (45%), SQLite (34%)
2023	#1	47%	MySQL (47%), SQLite (32%)
2022	#2	46%	MySQL (47%), SQLite (32%)
2021	#2	40%	MySQL (51%), SQLite (32%)
2020	#2	38%	MySQL (55%), SQLite (32%)
2019	#2	34%	MySQL (58%), SQLite (32%)
2018	#3	32%	MySQL (58%), SQL Server (36%)



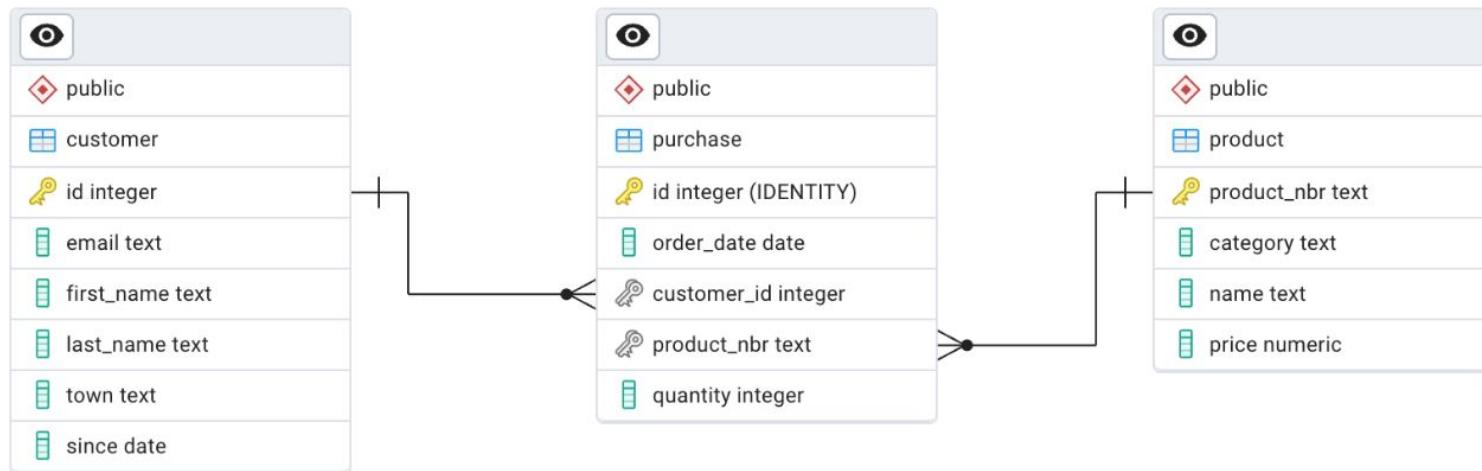


# Relational Databases

- Tables, e.g., products, customers
- Relations, e.g., which customer bought which product
- Schemas, e.g., structured collections of tables and relations
- Data types, e.g., integer, numeric, float, text, time, date, ...
- Data Definition Language (DDL), e.g. CREATE TABLE, CREATE INDEX, ....
- Data Manipulation Language (DML), e.g., SELECT name FROM product, INSERT (first\_name, last\_name) VALUES ('marc', 'linster') INTO customer

Global Product Sales Analysis - Q4 2025								
Category	Product Type	Sales Data (Units)		Customer Demographics		Geographic Breakdown		
		Total Sales	Avg Sales	Min Age	Max Age	North America	Europe	Oceania
Electronics	Laptops	1200	150	22	35	600	450	150
	Smartphones	2500	200	18	30	1000	800	700
	Tablets	800	100	15	25	400	300	100
	Headphones	1500	120	12	28	700	500	300
	Gaming Consoles	1000	180	18	32	500	400	100
	Smart Home Devices	900	110	16	26	450	350	100
	Cameras	600	80	14	24	300	200	100
	Peripherals	700	90	13	23	350	250	100
	Software Licenses	400	50	10	20	200	150	50
	Accessories	1100	130	11	27	550	400	150
Apparel	Clothing	1800	200	15	30	900	600	300
	Footwear	1500	180	14	28	750	550	250
	Hats & Caps	800	100	12	22	400	300	100
	Accessories	1000	120	11	24	500	350	150
	Leather Goods	600	140	16	30	300	200	100
	Outerwear	900	160	18	32	450	350	100
	Swimwear	500	80	10	20	250	150	50
	Sportswear	1200	180	15	30	600	400	200
	Denim	700	110	13	23	350	250	100
	Activewear	1000	130	12	22	500	350	150
Home & Living	Furniture	1000	150	18	32	500	400	100
	Decor	800	120	15	28	400	300	100
	Textiles	600	100	13	22	300	200	100
	Kitchenware	900	140	16	30	450	350	100
	Household Goods	700	110	14	26	350	250	100
	Storage Solutions	500	80	12	20	200	150	50
	Lighting	400	70	11	19	150	100	50
	Art & Prints	300	60	10	18	100	80	50
	Books & Stationery	200	50	9	17	80	60	50
	Leisure Goods	1000	160	17	31	500	400	100
Food & Beverage	Snacks	1200	180	14	26	600	450	150
	Drinks	1000	160	13	25	500	400	100
	Meat & Poultry	800	140	12	24	400	300	100
	Dairy Products	600	120	11	22	300	200	100
	Fresh Produce	900	150	15	27	450	350	100
	Baked Goods	700	130	13	23	350	250	100
	Condiments	500	90	10	19	200	150	50
	Spices & Herbs	400	80	9	18	150	100	50
	Breakfast Foods	300	70	8	17	100	80	50
	Snack Foods	1000	170	16	29	500	400	100
Automotive	Cars	1500	200	18	32	750	550	250
	Motorcycles	800	160	15	28	400	300	100
	Boats	600	140	13	26	300	200	100
	ATVs	500	120	12	24	250	150	100
	Trucks	900	180	16	30	450	350	100
	RVs	400	100	11	21	200	150	50
	Motorcycles	700	130	14	27	350	250	100
	ATVs	300	90	10	20	150	100	50
	Trucks	1000	190	17	31	500	400	100
	RVs	200	80	9	19	100	80	50
Sports & Outdoors	Footwear	1000	150	14	26	500	400	100
	Equipment	800	130	12	24	400	300	100
	Apparel	600	110	10	22	300	200	100
	Leisure Goods	900	160	15	29	450	350	100
	Accessories	700	140	13	27	350	250	100
	Footwear	500	100	11	21	200	150	50
	Equipment	400	90	10	20	150	100	50
	Apparel	300	80	9	19	100	80	50
	Leisure Goods	1000	170	16	30	500	400	100
	Accessories	200	70	8	18	100	80	50
Leisure	Leisure Goods	1200	180	15	28	600	450	150
	Accessories	1000	160	13	26	500	400	100
	Leisure Goods	800	140	12	24	400	300	100
	Accessories	600	120	10	22	300	200	100
	Leisure Goods	900	150	14	27	450	350	100
	Accessories	700	130	13	26	350	250	100
	Leisure Goods	500	100	11	21	200	150	50
	Accessories	400	90	10	20	150	100	50
	Leisure Goods	300	80	9	19	100	80	50
	Accessories	200	70	8	18	100	80	50
Business & Office	Office Equipment	1000	160	14	27	500	400	100
	Business Supplies	800	140	12	25	400	300	100
	Office Equipment	600	120	10	23	300	200	100
	Business Supplies	900	150	13	26	450	350	100
	Office Equipment	700	130	12	25	350	250	100
	Business Supplies	500	100	10	20	200	150	50
	Office Equipment	400	90	9	19	150	100	50
	Business Supplies	300	80	8	18	100	80	50
	Office Equipment	1000	170	15	28	500	400	100
	Business Supplies	200	70	7	17	100	80	50
Transportation	Automobiles	1500	200	16	30	750	550	250
	Commercial Vehicles	800	180	14	28	400	300	100
	Automobiles	600	160	12	26	300	200	100
	Commercial Vehicles	900	170	15	29	450	350	100
	Automobiles	700	150	13	27	350	250	100
	Commercial Vehicles	500	120	11	21	200	150	50
	Automobiles	400	110	10	20	150	100	50
	Commercial Vehicles	300	100	9	19	100	80	50
	Automobiles	1000	180	17	31	500	400	100
	Commercial Vehicles	200	80	7	16	100	80	50
Food & Beverage	Food	1200	180	15	28	600	450	150
	Drinks	1000	160	13	26	500	400	100
	Food	800	140	12	25	400	300	100
	Drinks	900	150	14	27	450	350	100
	Food	700	130	13	26	350	250	100
	Drinks	500	100	11	21	200	150	50
	Food	400	90	10	20	150	100	50
	Drinks	300	80	9	19	100	80	50
	Food	1000	170	16	29	500	400	100
	Drinks	200	70	7	16	100	80	50
Automotive	Cars	1800	220	18	32	800	600	200
	Commercial Vehicles	900	200	16	30	450	350	150
	Cars	700	180	14	28	350	250	100
	Commercial Vehicles	1000	190	17	31	500	400	100
	Cars	600	160	13	27	300	200	100
	Commercial Vehicles	500	130	11	21	200	150	50
	Cars	400	120	10	20	150	100	50
	Commercial Vehicles	300	110	9	19	100	80	50
	Cars	1000	210	17	30	500	400	100
	Commercial Vehicles	200	70	7	16	100	80	50
Sports & Outdoors	Footwear	1000	170	14	26	500	400	100
	Equipment	800	150	12	24	400	300	100
	Footwear	600	130	10	22	300	200	100
	Equipment	900	160	15	27	450	350	100
	Footwear	700	140	13	26	350	250	100
	Equipment	500	110	11	21	200	150	50
	Footwear	400	100	10	20	150	100	50
	Equipment	300	90	9	19	100	80	50
	Footwear	1000	180	16	28	500	400	100
	Equipment	200	70	7	16	100	80	50
Leisure	Leisure Goods	1300	190	15	27	650	500	150
	Accessories	1100	170	13	25	550	450	100
	Leisure Goods	900	150	12	24	450	350	100
	Accessories	700	130	10	22	350	250	100
	Leisure Goods	1000	160	14	26	500	400	100
	Accessories	500	110	11	21	200	150	50
	Leisure Goods	400	100	10	20	150	100	50
	Accessories	300	90	9	19	100	80	50
	Leisure Goods	1000	180	16	28	500	400	100
	Accessories	200	70	7	16	100	80	50
Business & Office	Office Equipment	1100	180	14	26	550	450	100
	Business Supplies	900	160	12	24	450	350	100
	Office Equipment	700	140	10	22	350	250	100
	Business Supplies	1000	170	13	25	500	400	100
	Office Equipment	500	110	11	21	200	150	50
	Business Supplies	400	100	10	20	150	100	50
	Office Equipment	300	90	9	19	100	80	50
	Business Supplies	200	80	8	18	100	80	50
	Office Equipment	1000	170	15	27	500	400	100
	Business Supplies	300	70	7	16	100	80	50
Transportation	Automobiles	1600	210	17	30	850	650	200
	Commercial Vehicles	900	190	15	28	480	380	150
	Automobiles	700	170	13	26	380	280	100
	Commercial Vehicles	1000	200	16	31	550	450	100
	Automobiles	600	150	12	25	320	220	100
	Commercial Vehicles	500	120	10	21	200	150	50
	Automobiles	400	110	10	20	150	100	50
	Commercial Vehicles	300	100	9	19	100	80	50
	Automobiles	1000	200	17	30	500	400	100
	Commercial Vehicles	200	70	7	16	100	80	50

# Entity-Relationship Diagrams





# Document Database

- Documents encapsulate and encode data, e.g., customer
- Collections gather similar documents, e.g., customers
- No standard schema, e.g., some customer documents can have fields that differ. For example
  - Customer1: firstname, lastname, address\_line\_1, city, postalcode
  - Customer2: firstname, middlename, lastname, address\_line\_1, address line 2, city, postalcode

---

# Document Database

```
{  
  "customer_id": "12345",  
  "name": "John Doe",  
  "email": "johndoe@example.com",  
  "phone": "+1234567890",  
  "address": {  
    "street": "123 Main St",  
    "city": "Anytown",  
    "state": "CA",  
    "zip": "90210"  
},
```

```
  "orders": [  
    {  
      "order_id": "67890",  
      "customer_id": "12345",  
      "date": "2023-10-01",  
      "total_amount": 150.75,  
      "items": [  
        {  
          "product_id": "p123",  
          "product_name": "Wireless  
Mouse",  
          "quantity": 1,  
          "price": 50.25  
        },  
        {  
          "product_id": "p456",  
          "product_name": "Bluetooth  
Keyboard",  
          "quantity": 1,  
          "price": 100.50  
        }  
      ]  
    },  
  ],
```



# Key Value Pair Database

- ‘Values’ associated with a ‘Key’
  - Customer: 123 name “Marc Linster” city “Junglinster”
  - Product:xyz price “192,17” color“blue”
- Extremely fast access to data
- Usually stored in memory
- Used as cache or working memory for applications



# Key Value Pair Database

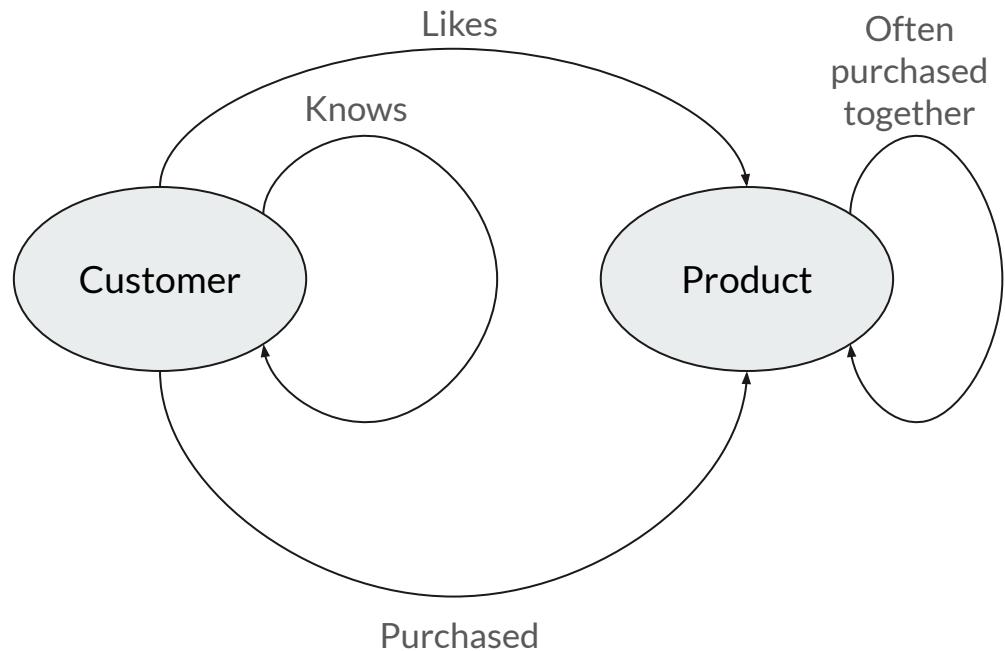
```
HSET customer:12345 name "John Doe" email "johndoe@example.com" phone "+1234567890"  
HSET customer:12345 address '{"street": "123 Main St", "city": "Anytown", "state": "CA", "zip": "90210"}'  
  
HSET customer:67890 name "Jane Smith" email "janessmith@example.com" phone "+0987654321"  
HSET customer:67890 address '{"street": "456 Elm St", "city": "Othertown", "state": "NY", "zip": "10001"}'
```

```
HSET order:67890 customer_id "12345" date "2023-10-01" total_amount "150.75"  
HSET order:67890 items '[{"product_id": "p123", "product_name": "Wireless Mouse", "quantity": 1, "price": 50.25}, {"product_id": "p456", "product_name": "Bluetooth Keyboard", "quantity": 1, "price": 100.50}]'  
  
HSET order:67891 customer_id "12345" date "2023-10-05" total_amount "75.00"  
HSET order:67891 items '[{"product_id": "p789", "product_name": "USB-C Hub", "quantity": 2, "price": 37.50}]'
```

---

# Graph Database

- Nodes, e.g., customer, product
- Relationships, e.g., purchased, likes, knows





# Graph Database

```
CREATE (c1:Customer {id: 1, name: "John Doe", email: "johndoe@example.com"});  
CREATE (c2:Customer {id: 2, name: "Jane Smith", email: "janeshmith@example.com"});  
CREATE (c3:Customer {id: 3, name: "Alice Johnson", email: "alicej@example.com"});  
  
CREATE (p1:Product {id: 101, name: "Wireless Mouse", price: 25.99});  
CREATE (p2:Product {id: 102, name: "Bluetooth Keyboard", price: 49.99});  
CREATE (p3:Product {id: 103, name: "USB-C Hub", price: 35.50});  
CREATE (p4:Product {id: 104, name: "Monitor", price: 199.99});  
  
MATCH (c1:Customer {id: 1}), (p1:Product {id: 101}) CREATE (c1)-[:LIKES]->(p1);  
MATCH (c1:Customer {id: 1}), (p2:Product {id: 102}) CREATE (c1)-[:LIKES]->(p2);  
MATCH (c2:Customer {id: 2}), (p3:Product {id: 103}) CREATE (c2)-[:LIKES]->(p3);  
MATCH (c3:Customer {id: 3}), (p4:Product {id: 104}) CREATE (c3)-[:LIKES]->(p4);
```

---

# Connecting to a Postgres Database

---

# Tools for this course

## Editor

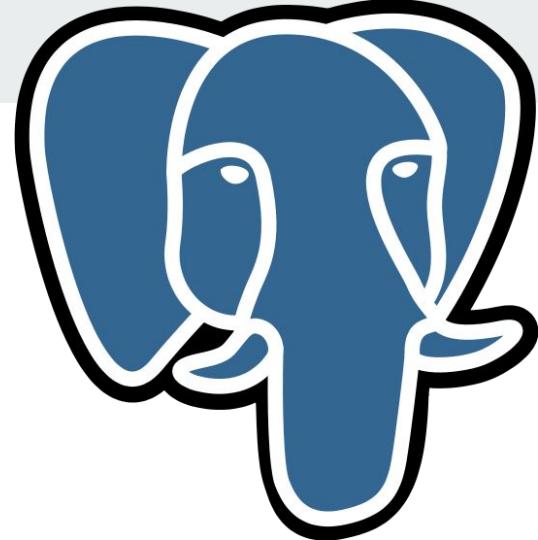
- pgAdmin 9.6
- License: Postgres License

## Postgres

- PostgreSQL V17.6
- License: Postgres License

## Exercises and Samples

- License: Creative Commons (no Rights Reserved)



It's all 100% Open Source

Take it home, use it,  
learn from it

# Basic elements of a Postgres database connection from pgAdmin

Connection name	The name in the GUI	<ul style="list-style-type: none"> <li>• <b>Postgres17</b> ←</li> <li>• My connection 1</li> </ul>
Where is the database server?	Network address (IP address and port)	<p>IP Address</p> <ul style="list-style-type: none"> <li>• <b>Localhost</b> ←</li> <li>• 188.115.7.175</li> <li>• mydb.aws.com</li> </ul> <p>Port</p> <ul style="list-style-type: none"> <li>• <b>5432</b> ←</li> <li>• 80</li> </ul>
Which database do I connect to?	Name of the database on the servers	<ul style="list-style-type: none"> <li>• <b>dlh1</b> ←</li> <li>• dlh2</li> <li>• dlh3</li> <li>• inventory</li> </ul>
Who am I?	Username and password	<p>Username</p> <ul style="list-style-type: none"> <li>• <b>postgres</b> ←</li> <li>• jDoe</li> </ul> <p>Password</p> <ul style="list-style-type: none"> <li>• <b>postgres</b> ←</li> <li>• xd67\$566</li> </ul>
Security parameters	Encryption on the network	<ul style="list-style-type: none"> <li>• Not applicable in training</li> </ul>

# Connection elements

- Network address
  - IP V4 example: 188.115.7.175
  - IP V6 example: 2001:7e8:c415:f601:c0fe:c98b:257a:a45b
  - url: [mydb.post.lu](http://mydb.post.lu)
  - Special address: **localhost** (always the local computer. In IPV4: "127.0.0.1", in IPV6: "::1")
- (Network) Port:
  - The port defines which service at a specific network address I want to use, and what 'language' (protocol) I can speak at that port
  - Examples:
    - Port 20: SMTP (email)
    - Port 80: HTTP (WWW)
    - Port 5432: Postgres

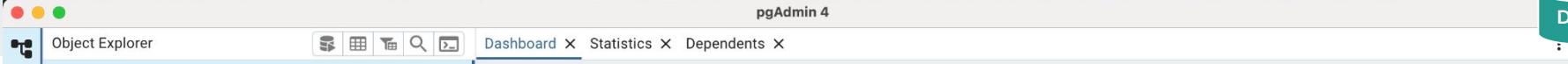
# pgAdmin

The screenshot shows the pgAdmin 4 application window on a Mac OS X desktop. The window title is "pgAdmin 4". The menu bar includes "pgAdmin 4", "Object", "Tools", "Edit", "View", "Window", and "Help". The top right corner shows the system status bar with icons for battery, signal, and date/time ("Mon Mar 24 09:26").

The main area is the "Dashboard" tab, which displays the following sections:

- Welcome:** Features the pgAdmin logo and the text "Management Tools for PostgreSQL". Below it are the sub-headings "Feature rich | Maximises PostgreSQL | Open Source" and a brief description of the tool's purpose.
- Quick Links:** Contains two buttons: "Add New Server" (with a gear icon) and "Configure pgAdmin" (with two interlocking gear icons).
- Getting Started:** Contains four links with icons:
  - "PostgreSQL Documentation" (blue elephant icon)
  - "pgAdmin Website" (blue globe icon)
  - "Planet PostgreSQL" (blue document icon)
  - "Community Support" (blue people icon)

At the bottom of the dashboard, there is a "Screenshot" button. The left sidebar is titled "Object Explorer" and shows a tree view with "Servers (3)" selected. The bottom left corner of the window has a gear icon.



**pgAdmin**  
Management Tools for PostgreSQL

Feature rich | Maximises PostgreSQL | Open Source

pgAdmin is an Open Source administration and management tool for the PostgreSQL database. It includes a graphical administration interface, an SQL query tool, a procedural code debugger and much more. The tool is designed to answer the needs of developers, DBAs and system administrators alike.

**Quick Links**

 Add New Server

 Configure pgAdmin

**Getting Started**

[PostgreSQL Documentation](#)

[pgAdmin Website](#)

[Planet PostgreSQL](#)

[Community Support](#)

pgAdmin 4

Servers (4)

- PostgreSQL 16
- dlhtraining
- Databases (7)
  - DLH1
  - DLH2
  - DLH3
  - DLH4
  - DLH5
  - DLH6
- postgres
  - Casts
  - Catalogs
  - Event Triggers
  - Extensions
  - Foreign Data Wrappers
  - Languages
  - Publications
  - Schemas (2)
    - public
    - test
  - Subscriptions
- Login/Group Roles
- Tablespaces

Dependents X Processes X DLH6 Script.sql X public.customer/D... X postgres/postgre... X DLH1/postgres@PostgreSQL 16\*

dlhtraining

General Connection Parameters SSH Tunnel Advanced Tags

Host name/address: localhost

Port: 5432

Maintenance database: DLH1

Username: postgres

Kerberos authentication?

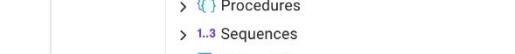
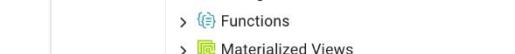
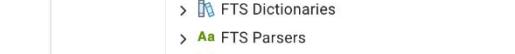
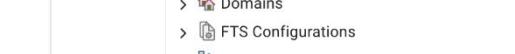
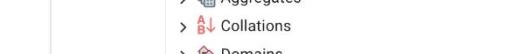
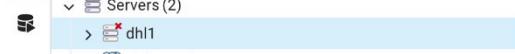
Role:

Service:

Data Output Messages Notifications

Showing rows: 1 to 1 Page No: 1 of 1

	current_database
1	DLH1



Please connect to the selected server to view the dashboard.

pgAdmin 4

Object Explorer

- > Aggregates
- > Collations
- > Domains
- > FTS Configurations
- > FTS Dictionaries
- > FTS Parsers
- > FTS Templates
- > Foreign Tables
- > Functions
- > Materialized Views
- > Operators
- > Procedures
- > 1..3 Sequences
- > Tables (6)
  - > address
  - > customer
    - > Columns (6)
      - id
      - customer\_number
      - first\_name
      - last\_name
      - address\_id
      - phone\_numbers
    - > Constraints (3)
      - customer\_address\_id\_fkey
      - customer\_customer\_number\_key
      - customer\_id\_key
    - > Indexes
    - > RLS Policies
    - > Rules
    - > Triggers
  - > manufacturer
  - > product
  - > sales\_transaction

Dashboard X Statistics X Dependents X

Activity State Configuration Logs System

Database sessions

Total Active Idle

Transactions per second

Transactions Commits Rollbacks

customer\_number

General Definition Constraints Variables Security SQL

Name customer\_number

Comment

Tuples in

Returned

Block I/O

Reads Hits

Screenshot

31

pgAdmin 4

Object Explorer

- > Aggregates
- > Collations
- > Domains
- > FTS Configurations
- > FTS Dictionaries
- > FTS Parsers
- > FTS Templates
- > Foreign Tables
- > Functions
- > Materialized Views
- > Operators
- > Procedures
- > 1..3 Sequences
- > Tables (6)
  - > address
  - > customer
    - > Columns (6)
      - id
      - customer\_number
      - first\_name
      - last\_name
      - address\_id
      - phone\_numbers
    - > Constraints (3)
      - customer\_address
      - customer\_customer
      - customer\_id\_key
    - > Indexes
    - > RLS Policies
    - > Rules
    - > Triggers
  - > manufacturer
  - > product

Dashboard X Statistics X Dependents X

Activity State Configuration Logs System

Database sessions

Total Active Idle

Transactions per second

Transactions Commits Rollbacks

Tuples in

Inserts Updates Deletes

Tuples out

Fetched Returned

Block I/O

Reads Hits

Count Rows

Create >

Delete

Delete (Cascade)

Refresh...

Restore...

Backup...

Import/Export Data...

Reset Statistics

ERD For Table

Maintenance...

Scripts >

Truncate >

View/Edit Data >

- All Rows
- First 100 Rows
- Last 100 Rows
- Filtered Rows...

Properties... es > customer

Servers > dhl1\_student > Databases > postgres

✓ Table rows counted: 500 ✘

32

pgAdmin 4

Object Explorer

- > Aggregates
- > Collations
- > Domains
- > FTS Configurations
- > FTS Dictionaries
- > FTS Parsers
- > FTS Templates
- > Foreign Tables
- > Functions
- > Materialized Views
- > Operators
- > Procedures
- > 1..3 Sequences
- > Tables (6)
  - > address
  - > customer
    - > Columns (6)
      - id
      - customer\_number
      - first\_name
      - last\_name
      - address\_id
      - phone\_numbers
    - > Constraints (3)
      - customer\_address\_id\_fkey
      - customer\_customer\_number\_key
      - customer\_id\_key
    - > Indexes
    - > RLS Policies
    - > Rules
    - > Triggers
  - > manufacturer
  - > product

Dashboard X Statistics X Dependents X public.customer/postgres/dhlstudent@dhl1\_student X

public.customer/postgres/dhlstudent@dhl1\_student

100 rows

Query History

```
1 SELECT * FROM public.customer
2 LIMIT 100
3
```

Data Output Messages Notifications

Showing rows: 1 to 100 Page No: 1 of 1

	id integer	customer_number character varying (25)	first_name character varying (25)	last_name character varying (25)	address_id integer	phone_numbers jsonb
1	1	cust00001	Ella	Walter	80900	{"home": "+352 32 07 30"}
2	2	cust00002	Enzo	Rutkowski	137013	{"home": "+352 621 375 964"}
3	3	cust00003	Clara	Arnold	26711	{"home": "+352 621 159 268", "work": "+352 79 23 31", "mobile": "+352 621 900 166"}
4	4	cust00004	Dylan	Ziółkowska	97658	{"home": "+352 62 09 85", "work": "+352 621 900 166", "mobile": "+352 621 707 758"}
5	5	cust00005	Felix	Kaczmarczyk	132220	{"home": "+352 77 24 31"}
6	6	cust00006	Emma	Mazurek	125998	{"home": "+352 57 64 83"}
7	7	cust00007	Enzo	Kaczmarczyk	15193	{"home": "+352 82 69 48", "work": "+352 621 707 758", "mobile": "+352 621 929 105"}
8	8	cust00008	Victoria	Urbański	175317	{"home": "+352 621 929 105"}
9	9	cust00009	Anna	Fischer	43145	{"home": "+352 621 665 543"}
10	10	cust00010	Lena	Lange	137847	{"home": "+352 20 15 75", "work": "+352 621 363 703", "mobile": "+352 621 900 166"}
11	11	cust00011	Nina	Wróblewski	102195	{"home": "+352 65 71 21"}
12	12	cust00012	Leonor	Hübner	147959	{"home": "+352 25 82 89", "work": "+352 621 900 166", "mobile": "+352 621 707 758"}

Table rows counted: 500 ✓

Servers > dhl1\_student > Databases > postgres > Schemas > public > Tables > customer | query complete 00:00:00.054

LF Ln 1, Col 1 33



## Object Explorer

- Servers (5)
  - PostgreSQL 16
    - Databases (8)
      - DLH1
      - DLH2
      - DLH3
      - DLH4
      - DLH5
      - DLH6
      - accesstest
      - postgres

> Login/Group Roles

> Tablespaces

> dhl1\_student

> dhltraining

> websitedata-analysis

> websitedataediting

Create >

Refresh...

Restore...

Backup...

CREATE Script

ERD For Database

Maintenance...

Grant Wizard...

Search Objects...

PSQL Tool

Query Tool

Properties...

Dashboard X Statistics X Dependents X Processes X DLH6 Script.sql X public.customer/D... X

⋮

Activity State Configuration Logs System

Database sessions

Total Active Idle

1  
0.75  
0.5  
0.25  
0

Transactions per second

Transactions Commits Rollbacks

3  
2  
1  
0

Tuples in

Inserts Updates Deletes

100  
75  
50  
25  
0

Tuples out

Fetched Returned

1K  
750  
500  
250  
0

Block I/O

Reads Hits

150  
100  
50  
0

Object Explorer

Servers (5)

PostgreSQL 16

Databases (7)

- > DLH1
- > DLH2
- > DLH3
- > DLH4
- > DLH5
- > DLH6
- > postgres

> Login/Group Roles (23)

> Tablespaces

> dh1\_student

> dh1training

> websitedata-analyst

> websitedataediting

Dashboard X Statistics X Dependents X Processes X DLH6 Script.sql X public.customer/D... X postgres/postgres@PostgreSQL 16 X

```
postgres=# psql (17.0, server 17.4 (Debian 17.4-1.pgdg120+2))
Type "help" for help.
```

Screenshot

# PSQL Cheat Sheet (essentials only)

## Connection

```
\c [connect] { [DBNAME| - USER| - HOST| - PORT} connect to database  
\conninfo display information about current connection  
\set displays info about current psql parameters  
\! clear clears the display
```

## Informational: (options: S = show system objects, + = additional detail)

\d [S+]	list tables, views, and sequences
\d [S+] NAME	describe table, view, sequence, or index
\dg [S+] [PATTERN]	list roles
\di [S+] [PATTERN]	list indexes
\dm [S+] [PATTERN]	list materialized views
\dn [S+] [PATTERN]	list schemas
\do [S+] [PATTERN]	list operators
\dp [PATTERN]	list table, view, and sequence access privileges

## Display etc

\x	Switch to expanded display (toggle)
\time	Measure execution time (toggle)
\watch	Repeated execution of query

\drds [PATRN1 [PATRN2]]	list per-database role settings
\ds [S+] [PATTERN]	list sequences
\dt [S+] [PATTERN]	list tables
\dT [S+] [PATTERN]	list data types
\du [S+] [PATTERN]	list roles
\dv [S+] [PATTERN]	list views
\l [+]	list databases
\sf [+]	show a function's definition
\sv [+]	show a view's definition

---

# Tables and queries: Basics

# Table Basics

- Rows = records (a.k.a. tuples in Postgres)
- Columns = data elements
- Example: mycustomer table
  - ID = a way to uniquely identify a customer ⇒ usually a large number
  - Email = the email address, a text field, sometimes limited in length
  - First name ⇒ a text field, sometimes limited in length
  - Last name ⇒ a text field, sometimes limited in length
  - Since => a date field

```
postgres=# SELECT * FROM mycustomer;
 id |          email           | first_name | last_name |      since
----+-----+-----+-----+-----+
  1 | marc@marclinster.com | Marc        | Linster    | 2025-01-01
  2 | jmu@gmail.lu       | Jeff        | Mueller    | 2025-01-02
  3 | bini@hotmail.lu    | Jean        | Bintner    | 2025-01-03
```

# Tables

- All rows/records have the same columns,
  - E.g., for every customer we have id, first\_name, last\_name, since
- Every column has a data type, which constrains what data can be in that position
  - E.g., ‘since’ is always a date, first\_name is always a text
- Most important data types
  - Boolean: Yes/No
  - Character types: char, varchar, text
  - Numeric: integer, floating point, ...
  - Temporal types: date, time, timestamp, interval
  - UUID: Unique identifies
  - Array: arrays of strings, numbers, etc
  - JSON: structured documents
  - Special types for network addresses, XML, geographic points, etc

<https://www.postgresql.org/docs/current/datatype.html>

More on data types later!



# Illustrating Select Postgres Data Types

Boolean: 'true' or 'false'

## Character Types

- Fixed length: CHARACTER(n), CHAR(n)
- Variable length w. length limit: CHARACTER VARYING (n), VARCHAR (n)
- Variable w.o. length limit: TEXT, VARCHAR

## Number Types

- INTEGER:
  - SMALLINT (+/- 32,768), INTEGER (+/- 2,147,483,648), BIGINT (+/- 9,223,372,036,854,775,808)
- NUMERIC/DECIMAL:
  - NUMERIC (5,2) - 5 digits before the period, two after the period, for example 172.99

# Let's try it out

- pgAdmin Query Tool
  - Make sure you are connected to DLH1!
    - Enter “`SELECT current_database();`” to check where you connected
  - Try this:

```
CREATE TABLE mycustomer (
    id INTEGER,
    email TEXT,
    first_name TEXT,
    last_name TEXT,
    since DATE
);
```

- Check out the list of tables in the pgAdmin tree browser (hit refresh)

The screenshot shows the pgAdmin 4 interface. On the left, the Object Explorer tree view is open, showing a hierarchy of databases, schemas, and tables. The 'Tables (1)' section under the 'mycustomer' schema is selected, revealing the 'mycustomer' table. On the right, the main window contains a query editor with the following SQL code:

```

-- create a simple table
-- add customer number as an explicitly entered unique ID
DROP TABLE IF EXISTS mycustomer;

CREATE TABLE mycustomer (
    id INTEGER,
    email TEXT,
    first_name TEXT,
    last_name TEXT,
    since DATE
);

INSERT INTO mycustomer (id, email, first_name, last_name, since)
VALUES
    (1,'marc@marcclintner.com','Marc','L'Inster','1.1.2025'),
    (2,'jeff@gmail.u','Jeff','Mueller','1.2.2025'),
    (3,'brian@briansmall.u','Jean','Bintner','1.3.2025');

```

The status bar at the bottom right indicates "Total rows: 3".

# Let's try something else

- pgAdmin Query Tool
  - Create the table mycustomer1 using the GUI
    - Add columns
      - id INTEGER,
      - email TEXT,
      - first\_name TEXT,
      - last\_name TEXT,
      - since DATE
  - Check out the list of tables in the pgAdmin tree browser (hit refresh)

The screenshot shows the pgAdmin 4 interface. On the left, the Object Explorer tree view displays a database structure under 'Servers' > 'PostgreSQL 16' > 'dhraining' > 'Databases' (containing 'DLH1'). Under 'Tables (1)', there is a node for 'mycustomer'. On the right, the main window has a 'Query' tab open with the following SQL script:

```
-- create a simple table
-- add customer number as an explicitly entered unique ID
DROP TABLE IF EXISTS mycustomer;

CREATE TABLE mycustomer (
    id INTEGER,
    email TEXT,
    first_name TEXT,
    last_name TEXT,
    since DATE
);

INSERT INTO mycustomer (id, email, first_name, last_name, since)
VALUES
    (1, 'marcmarclinter.com', 'Marc', 'L'Inster', '1.1.2025'),
    (2, 'jeffmuellr.ln', 'Jeff', 'Mueller', '1.2.2025'),
    (3, 'binniphomall.u', 'Jean', 'Blinter', '1.3.2025');
```

The status bar at the bottom right indicates 'Total rows: 3'.

# PSQL Tool

- Try:
  - \c (which database am I connected to?)
  - \c dlh1 (connect me to DLH1)
  - \l (list all the databases)

```
CREATE TABLE mycustomer2 (
    id INTEGER,
    email TEXT,
    first_name TEXT,
    last_name TEXT,
    since DATE
);
```

- \dt, \dt+ and \d mycustomer2
- DROP TABLE mycustomer1;
- DROP TABLE mycustomer2;
- \dt

The screenshot shows the pgAdmin 4 interface. On the left, the Object Explorer displays a tree view of database objects under 'DLH1'. In the center, a terminal window shows the following session:

```
pgAdmin 4
pgl (17.0, server 17.4 (Debian 17.4-1.pgdg120+2))
Type 'help' for help.

DLH1=# \dt mycustomer
              List of relations
 Schema |   Name    | Type | Owner
 public | mycustomer | table | postgres
(1 row)

DLH1=# \dt+ mycustomer
              List of relations
 Schema |   Name    | Type | Owner | Persistence | Access method | Size | Description
 public | mycustomer | table | postgres | permanent | heap          | 16 kB |
(1 row)

DLH1=# \d mycustomer
             Table "public.mycustomer"
 Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+-----+
 id    | integer |           |          |
 email | text   |           |          |
 first_name | text |           |          |
 last_name | text |           |          |
 since  | date  |           |          |
Indexes:
    "mycustomer_id_ung" UNIQUE CONSTRAINT, btree (id)
Referenced by:
    TABLE "myorder" CONSTRAINT "myorder_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES mycustomer(id)

DLH1=#
```

# Uppercase, lowercase, white spaces, etc

---

- Postgres ignores case - all names for tables, columns, functions, procedures etc. are lowercase
  - Postgres automatically lower cases all identifiers!!!
  - SQL identifiers and keywords must begin with a letter or an underscore (\_).
  - Subsequent characters can be letters, underscores, digits (0-9), or dollar signs (\$).
  - Note that dollar signs are not allowed in identifiers according to the letter of the SQL standard, so their use might render applications less portable.
- Customername = CustomerName = CUSTOMERNAME = customername <> "CustomerName" <> "customerName" <> "CUSTOMERNAME"
- Do not use camelCase, use snake\_case to improve readability of identifiers, e.g. customer\_name
- Spaces (' ') cannot be used in identifiers
- Double quotes "" can be used to create mixed case identifiers that include spaces

DON'T DO IT!!!

Mixed case and spaces cause all kinds of problems later on

---

# Query basics

---

# Query types

- SELECT: List data from tables that meet certain conditions
- UPDATE: Change data in records that meet certain conditions
- INSERT: Add records
- DELETE: Remove Records

## Let's try it out in the GUI Query Tool

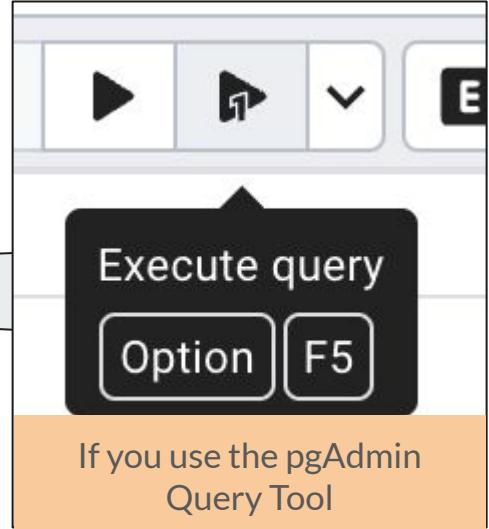
```
SELECT * FROM mycustomer;
```

- Remember the ';
- Click

```
INSERT INTO mycustomer (id, email, first_name, last_name, since)  
VALUES  
    (1,'marc@marclinster.com', 'Marc', 'Linster','1.1.2025');
```

```
INSERT INTO mycustomer (id, email, first_name, last_name, since)  
VALUES  
    (2,'jmu@gmail.lu','Jeff', 'Mueller','1.2.2025'),  
    (3,'bini@hotmail.lu', 'Jean','Bintner','1.3.2025');
```

```
SELECT first_name FROM mycustomer;  
SELECT first_name, last_name FROM mycustomer;  
SELECT * FROM customer WHERE last_name = 'Linster';  
SELECT * FROM customer WHERE first name LIKE 'Je%';
```



# Use the GUI to see the rows

- Go to dlh1/public/mycustomer
- Choose select all rows

The screenshot shows the pgAdmin 4 interface. In the Object Explorer, under the 'Tables (1)' section of the 'mycustomer' table, a context menu is open. The 'View/Edit Data' option is highlighted. Below it, the 'All Rows' button is also highlighted. The table data is displayed in a grid:

	<b>id</b>	<b>email</b>	<b>first_name</b>	<b>last_name</b>	<b>since</b>
1	1	marc@marclinster.c...	Marc	Linter	2025-01-...
2	2	jmuig@mail.lu	Jeff	Mueller	2025-01-...
3	3	bini@hotmail.lu	Jean	Bintner	2025-01-...

# Use PLSQL

- Add two more customers with id 4 and 5

# Building Blocks of the simple SELECT Query

- **SELECT**
  - Columns to be selected, maybe using AS to provide a meaningful name
  - For example `last_name AS ln`
- **FROM**
  - Table(s) from which to select. Can also use AS to provide a new name
  - For example `customer AS c`
- **WHERE**
  - Filter clauses to select a subset of records
  - Can use the new names from the AS clause
  - For example `c.ln LIKE 'Li%`
- **ORDER BY**
  - Organizes the output of the SELECT clause
  - Can use the new names from the AS clause
  - For example: `c.ln ASC, since DESC`

## Alias (AS)

- improves readability
- Only available in the context of this query
- Key word 'AS' is optional

More complex building blocks for SELECT queries (JOIN, GROUP BY, HAVING, GROUPING, WITH, ...) will be introduced later

## Let's try it out

```
UPDATE mycustomer SET first_name = 'Mark' WHERE id = 1;  
SELECT * FROM mycustomer;  
DELETE FROM mycustomer WHERE id = 1;  
SELECT * FROM mycustomer;  
SELECT * FROM mycustomer ORDER BY last_name ASC;
```

Try it in pgAdmin Query Tool and in PSQL

# Day 2

- Review of Day 1
  - Operators in the WHERE clause and computations in the SELECT clause
  - Primary keys, constraints, and foreign key relationships
  - Data Types: Numbers, Characters, JSON, and more
-

---

# Operators in the WHERE Clause

---

# Postgres Operators in the WHERE Clause

=	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
<>	Not equal to
!=	Not equal to
LIKE	Check if a value matches a pattern (case sensitive)
ILIKE	Check if a value matches a pattern (case insensitive)
AND	Logical AND
OR	Logical OR
IN	Check if a value is between a range of values
BETWEEN	Check if a value is between a range of values
IS NULL	Check if a value is NULL
NOT	Makes a negative result e.g. NOT LIKE, NOT IN, NOT BETWEEN

## Let's try it out

```
SELECT * FROM mycustomer WHERE first_name LIKE 'Je%';
```

```
SELECT * FROM mycustomer WHERE first_name LIKE 'je%';
```

```
SELECT * FROM mycustomer WHERE first_name ILIKE 'je%';
```

```
SELECT * FROM mycustomer WHERE first_name LIKE '_e%';
```

```
SELECT * FROM mycustomer WHERE first_name LIKE '_e%' AND last_name = 'Bintner';
```

```
SELECT * FROM mycustomer WHERE since  
    BETWEEN '2025-01-02' AND '2025-02-01';
```

---

# Computations in Select

# Postgres functions in the SELECT clause

Some of the key string functions

	String append	'a'  '_'  'b' → a_b
lpad/rpad	Create a padded string	rpad('hi', 5, 'xy') → hixyx
substring	Create a substring	substring('Thomas' from 2 for 3) → hom
::string	Cast a number (and other types) as string	1::TEXT  '_'  2::TEXT → 1_2

Many, many more in the Postgres documentation: <https://www.postgresql.org/docs/current/functions-string.html>



# Synonyms

- 'AS' defines a synonym
- Can be used for table names and column names
- Explicit form: `SELECT mc.email AS email_address FROM mycustomer AS mc;`
  - Using 'AS' makes the query more readable
  - Use snake\_case for the pretty names
- Short form: `SELECT mc.email email_address FROM mycustomer mc;`

---

## Lets try it out

```
SELECT
    FIRST_NAME || ' ' || LAST_NAME
    AS customer_name
FROM
    MYCUSTOMER
WHERE
    FIRST_NAME LIKE 'Je%';
```



# Other Calculations

- TO\_CHAR – convert a timestamp or a numeric value to a string.
  - `SELECT TO_CHAR (NOW(), 'DAY');`
  - `SELECT TO_CHAR (NOW(), 'MON');`
  - `SELECT TO_CHAR (CURRENT_DATE, 'Day Month, DD, YYYY');`
  - `SELECT TO_CHAR (12345.55, '99G999G999D99L');`
  - <https://www.postgresql.org/docs/current/functions-formatting.html>
- Format
  - `SELECT FORMAT ('This is a string with a %s and a %s', 'first placeholder', 'second placeholder');`
  - <https://www.postgresql.org/docs/current/functions-string.html#FUNCTIONS-STRING-FORMAT>
- Extract

# EXTRACT

- EXTRACT from timestamp or date
  - CENTURY
  - DAY (day of the month)
  - DOW (day of the week - Sunday(0) to Saturday (6)
  - ISODOW (ISO standard day of the week - Monday (1) to Sunday (7)
  - DOY (day of the year)
  - HOUR, MINUTE, ....
- Try it out:
  - ```
SELECT 'It is the ' || EXTRACT (DOY FROM current_date) || 'th day of the year';
```

  
?column?  
-----  

```
It is the 234th day of the year
```
- Name the column using AS
  - ```
SELECT 'It is the ' || EXTRACT (DOY FROM current_date) || 'th day of the year' as day_of_year;
```

## A better way to format strings: FORMAT

```
SELECT
    FORMAT ('%s %s', first_name, last_name)
        AS customer_name
FROM
    MYCUSTOMER
WHERE
    FIRST_NAME LIKE 'Je%';
```

# Calculations in the SELECT statement

```
SELECT
    FIRST_NAME || ' ' || LAST_NAME
        AS customer_name,
    FORMAT('Customer for %s days', CURRENT_DATE - since)
        AS how_long
FROM
    MYCUSTOMER;

customer_name |      how_long
-----+-----
Marc Linster | Customer for 111 days
Jeff Mueller  | Customer for 110 days
Jean Bintner  | Customer for 109 days
```

EXTRACT (field FROM source)

- Field: DAY, DOW, DOY, YEAR, ...
- SOURCE: Date, timestamp or interval

See:

<https://neon.tech/postgresql/postgresql-date-functions/postgresql-extract>



# Order By

```
SELECT * FROM mycustomer  
    ORDER BY last_name ASC;
```

```
SELECT * FROM mycustomer  
    WHERE first_name LIKE 'Je%'  
    ORDER BY last_name ASC, first_name ASC;
```

---

## Try it out

- Create a query that
  - Selects customers from the customer table
  - Puts the name in the format 'last name, first name', e.g., Linster, Marc
  - Adds a column entitled 'customer\_since'
  - Sorts descending by last name, ascending by first name

---

# Primary Keys, Constraints and Relationships

---

# Constraints and Relationships

- Constraints and relationships make sure your data in the tables is consistent
  - For example:
    - Orders always have quantities  $> 0$
    - Products with prices  $> 0$
    - Products each having different names
    - Orders referring to existing products
- Constraints and relationships complement the definition of tables, columns, and data types

---

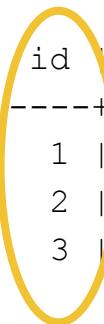
# Constraints and Relationships

- **Check Constraints:** The value in the column has to meet certain criteria, e.g.,  $\text{price} > 0$
- **Not-Null Constraints:** The column needs to have a value
- **Unique Constraints:** No two records have the same value
- **Primary Keys:** like the Unique Constraint, but excludes NULL and always creates an index
  - ⇒ A primary key can also be a combination of columns
- **Foreign Keys:** the data needs to be a valid reference to another table
  - You can assign a ‘trigger’ to handle deletions

---

# Primary Keys

- A unique way to find a row in the table
  - Cannot be NULL
  - Must be unique
- Can be a combination of rows



id	email	first_name	last_name	since
1	marc@marclinster.com	Marc	Linster	2025-01-01
2	jmu@gmail.lu	Jeff	Mueller	2025-01-02
3	bini@hotmail.lu	Jean	Bintner	2025-01-03

---

## This is what we are trying to do

- 3 tables
  - mycustomer
  - myproduct
  - myorder
- Orders refer to customers and products
  - Every order links to one customer and one product (simplified model)
  - Make sure only existing customer can place an order
  - Make sure that all products actually exist

# Table Basics

```
CREATE TABLE myproduct (
    product_nbr VARCHAR(10),
    name TEXT,
    price numeric
);
```

## Problems:

- Every product should have a unique id (or product number)
- No two products should have the same name
- Every product should have a price > 0

# Table Basics

```
DROP TABLE IF EXISTS myproduct;
```

```
CREATE TABLE myproduct (
    nbr VARCHAR(10) PRIMARY KEY,
    name TEXT UNIQUE NOT NULL,
    price NUMERIC NOT NULL CHECK (price > 0)
);
```

Try it out

- Insert 4 product records
- Use SELECT to make sure it worked
- Sort them by price ascending

- Add primary key to uniquely identify every record
  - PRIMARY KEY = UNIQUE + NOT NULL + indexed
  - Only one primary key per table
- Make sure that no products can have the same name
- All prices have to be greater than 0

SELECT * FROM myproduct;		
nbr	name	price
ham_001	Hammer 200gr	19.95
ham_002	Hammer 1kg	25.00
chisel_004	Chisel stone	39.95
pli_019	Pliers red, adjustable	13.50

---

## **CREATE TABLE myorder**

- Introducing new capabilities:
  - Automatically generate a primary key
  - Link the order to a customer
  - Link the order to a product

# Relations and foreign keys

```
DROP TABLE IF EXISTS myorder;
```

```
CREATE TABLE myorder (
    id INT PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
    date DATE,
    customer_id INTEGER NOT NULL,
    product_nbr VARCHAR(10) NOT NULL,
    qty INTEGER NOT NULL CHECK (qty > 0)
)
```

Nonsense order:

```
INSERT INTO myorder (date, customer_id, product_nbr, qty)
VALUES (current_date, 25, 'big_thing', 11);
```

- Create PRIMARY KEY
- Automatically generate a unique ID

Problems:

- No guarantee that the product\_nbr is actually in the myproduct table
- Not sure if the customer is actually defined!

# Relations and foreign keys

```
DROP TABLE IF EXISTS myorder;

CREATE TABLE myorder (
    id INT PRIMARY KEY GENERATED BY DEFAULT AS
IDENTITY,
    date DATE,
    customer_id INTEGER REFERENCES mycustomer (id),
    product_nbr VARCHAR(10) REFERENCES myproduct (nbr),
    qty INTEGER NOT NULL CHECK (qty > 0)
);
```

Definition fails as mycustomer (id) does not have a unique constraint

```
ALTER TABLE mycustomer ADD CONSTRAINT mycustomer_id_unq
UNIQUE (id);
```

try to create an order for a non-existing product

```
INSERT INTO myorder (date, customer_id,
product_nbr, qty)
VALUES (current_date, 25, 'big_thing', 11);
```

ERROR: insert or update on table  
"myorder" violates foreign key constraint  
"myorder\_customer\_id\_fkey"  
Key (customer\_id)=(25) is not present in  
table "mycustomer".

SQL state: 23503  
Detail: Key (customer\_id)=(25) is not  
present in table "mycustomer".

```

SELECT * FROM myproduct;
  nbr | name | price
-----+-----+-----+
ham_001 | Hammer 200gr | 19.95
ham_002 | Hammer 1kg | 25.00
chisel_004 | Chisel stone | 39.95
pli_019 | Pliers red, adjustable | 13.50

```

# Let's add orders

```

SELECT * FROM mycustomer;
  id | email | first_name | last_name | since
-----+-----+-----+-----+-----+
      1 | marc@marclinster.com | Marc | Linster | 2025-01-01
      2 | jmu@gmail.lu | Jeff | Mueller | 2025-01-02
      3 | bini@hotmail.lu | Jean | Bintner | 2025-01-03

```

\d myorder

Column	Type	Collation	Nullable	Default
id	integer		not null	generated by default ...
date	date			
customer_id	integer			
product_nbr	character varying(10)			
qty	integer		not null	

Indexes: "myorder\_pkey" PRIMARY KEY, btree (id)  
Check constraints: "myorder\_qty\_check" CHECK (qty > 0)  
Foreign-key constraints:  
 "myorder\_customer\_id\_fkey" FOREIGN KEY (customer\_id) REFERENCES mycustomer(id)  
 "myorder\_product\_nbr\_fkey" FOREIGN KEY (product\_nbr) REFERENCES myproduct(nbr)

```

INSERT INTO myorder (date, customer_id, product_nbr, qty)
VALUES
    (current_date, 1, 'ham_001', 1),
    (current_date, 1, 'ham_002', 1),
    (current_date-1, 2, 'chisel_004', 2),
    ('2025-03-01', 2, 'pli_019', 4);

```

---

# Queries that work on multiple tables

SELECT <column\_names> ← comma separated list

FROM <table\_names> ← comma separated list

WHERE

ORDER BY

}      Optional

```
SELECT * FROM myorder o,  
        myproduct p,  
        mycustomer c  
WHERE  
    o.product_nbr = p.nbr  
AND  
    o.customer_id = c.id;
```

```
- [ RECORD 1 ]-----  
id          | 1  
date        | 2025-02-02  
product_nbr | chisel_004  
qty         | 9  
customer_id | 1  
nbr         | chisel_004  
name        | Chisel stone  
price       | 39.95  
id          | 1  
email       | marc@marclinster.com  
first_name  | Marc  
last_name   | Linster  
since       | 2025-01-01
```

Uses the \x command in PSQL Tool to show the records in expanded mode

---

## Try it out

- 1) Add products to myproduct table (up to 5 total)
- 2) Add several customers (max 5)
- 3) Add at least 5 orders for different products and customers. Orders dating in February 2025
- 4) Create a query that displays the orders with
  - o Order id
  - o Order date (ascending)
  - o Customer name in format 'last\_name, first\_name'
  - o Product name
  - o Product price
  - o Order quantity
  - o Extended price (quantity \* price)

---

# Data Types in Postgres

# Key Data Types in SQL

Boolean	Logical true or false	true, false
Numbers	Integers: Decimals Floating-point	1,2, 45, 987 (smallint, <u>integers</u> , bigint) 7.25 3.14159265
Binary	Raw data	PDF, images
Character types	Fixed/limited length Unlimited length (max 1 GB)	
Date time	Date Time Timestamp Interval	2025-09-23 16:49:47.70598 2025-09-23 16:49:47.70598 1 day
JSON/JSONB	JSON/JSONB	{"home": "47 25 07", "mobile": "621 599-314"}
Array	INTEGER[]	{1,2,5,19,27}
Ranges	Date ranges Number ranges	[2025-03-01, 2025-03-31) [1, 9]
UUID	Universally Unique IDs	01995196-02ef-7db9-b11b-b906ee7b4926



# Numbers

Data Type	Range	Storage	Pro	Con
SMALINT	+/- 32,768	2 bytes	Fast computation Small storage	No fractions Automatic rounding
INTEGER	+/- 2,147 M	4 bytes		
BIGINT	+/- 9,223 B	8 bytes		
NUMERIC/ DECIMAL	Max. 131072 digits bef. decimal point; max 16383 digits aft. decimal point	Variable	No rounding errors	More storage Slower calculations
REAL	$-3.40282347 \times 1E38$ to $3.40282347 \times 1E38$	4 bytes	Imprecise w. rounding errors	Fast calculations Less storage than NUMERIC/DECIMAL
DOUBLE PRECISION	1E-307 to 1E+308	8 bytes		

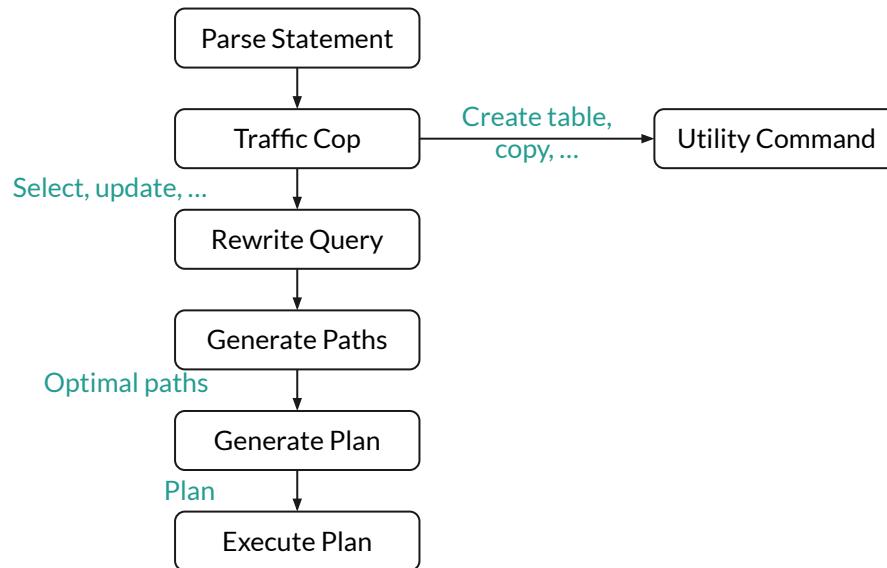
# Day 3

- **Review of Day 2**
  - **Indexes**
  - **Organizing data with schemas and entity-relationship diagrams**
  - **Queries revisited (joins, views, and aggregates)**
-

---

# Indexes

# How does Postgres process a query?



See <https://momjian.us/main/writings/pgsql/internalpics.pdf>



# How does a database find data?

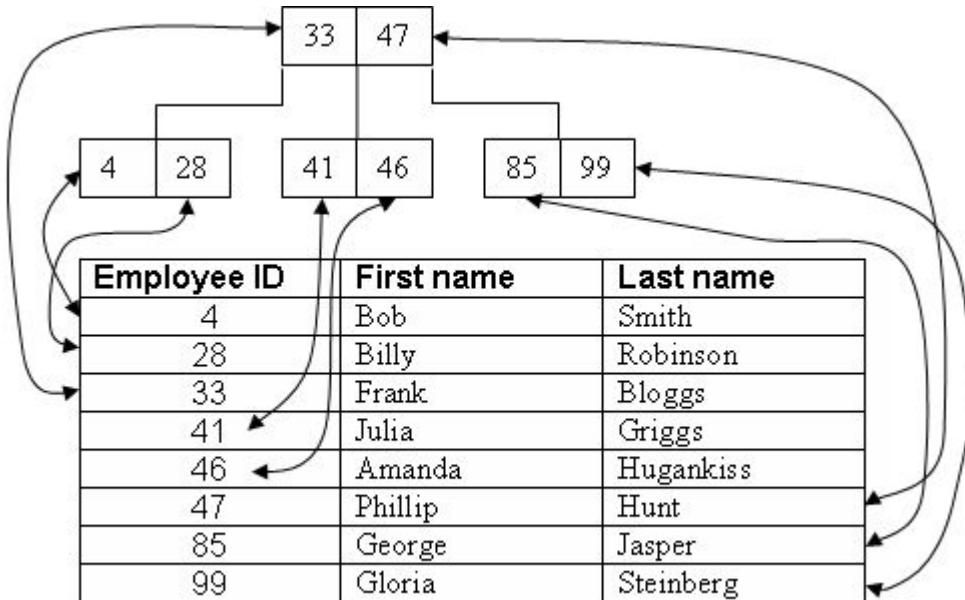
- Every table is stored in 8k data blocks on disk, in files that are 1GB (or less)
- Example: `SELECT * from customer WHERE last_name = 'Voigt';`
- Without Index
  - Sequential scan of all data blocks for that table
  - If block is not in memory (shared buffers), get it from disk and load it into memory
    - “**Sequential Scan**”
- With Index
  - Load the index into memory (usually much smaller than data table)
  - Ask the index which blocks have the data
  - Load only those blocks
    - “**Index Scan**”

# Many types of Indexes



- **Balanced Tree (B-Tree)**
  - Every database has it. The most common and used for 90%+ of all cases. The default in Postgres
  - Ideal for looking up unique values and maintaining unique indexes
  - High concurrency implementation - very fast
- Block-Range Index (BRIN)
  - Ideal for very large datasets that are organized according to the index
- Generalized Inverted Index (GIN)
  - Best for indexing values with many keys or values, e.g.,text documents, JSON, multi-dimensional data, arrays
  - Ideal for columns containing many duplicate
- Hash Index
  - Equality, non-equality lookups; no range lookups
- Generalized Search Tree (GiST)
  - Used for GIS, key-value store
- Space-Partitioned Generalized Search Tree (SP-GIST)
  - Specialized index

# Very Fast Access to Data

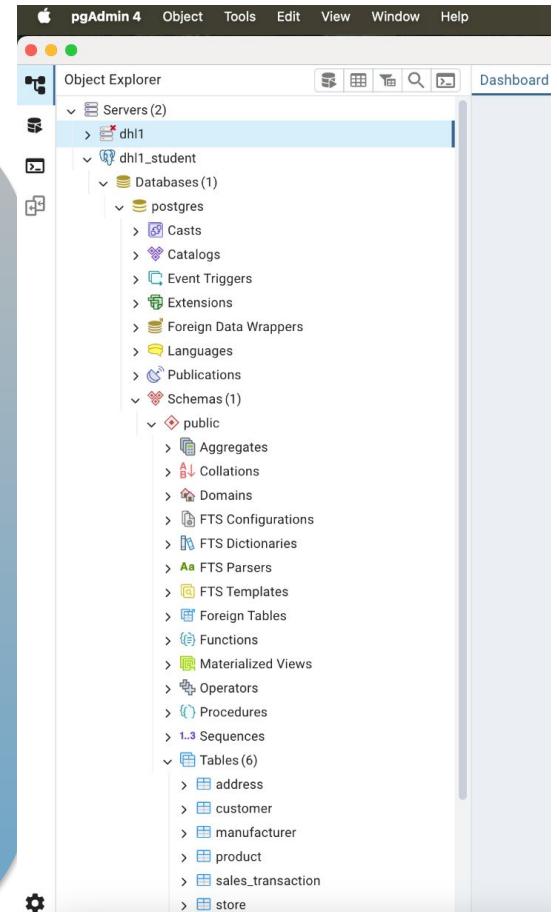
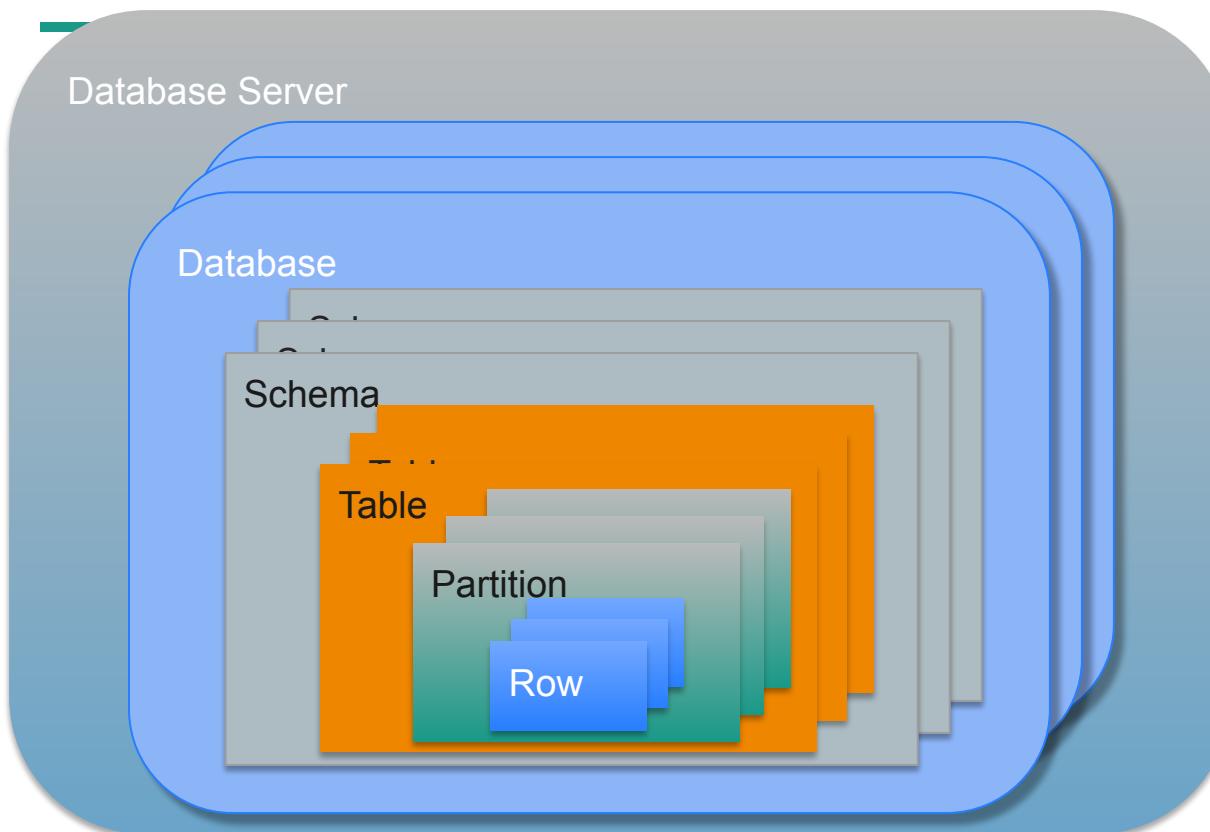


<https://en.wikipedia.org/wiki/Database>

---

# Organizing data - schemas, and entity relationship diagrams

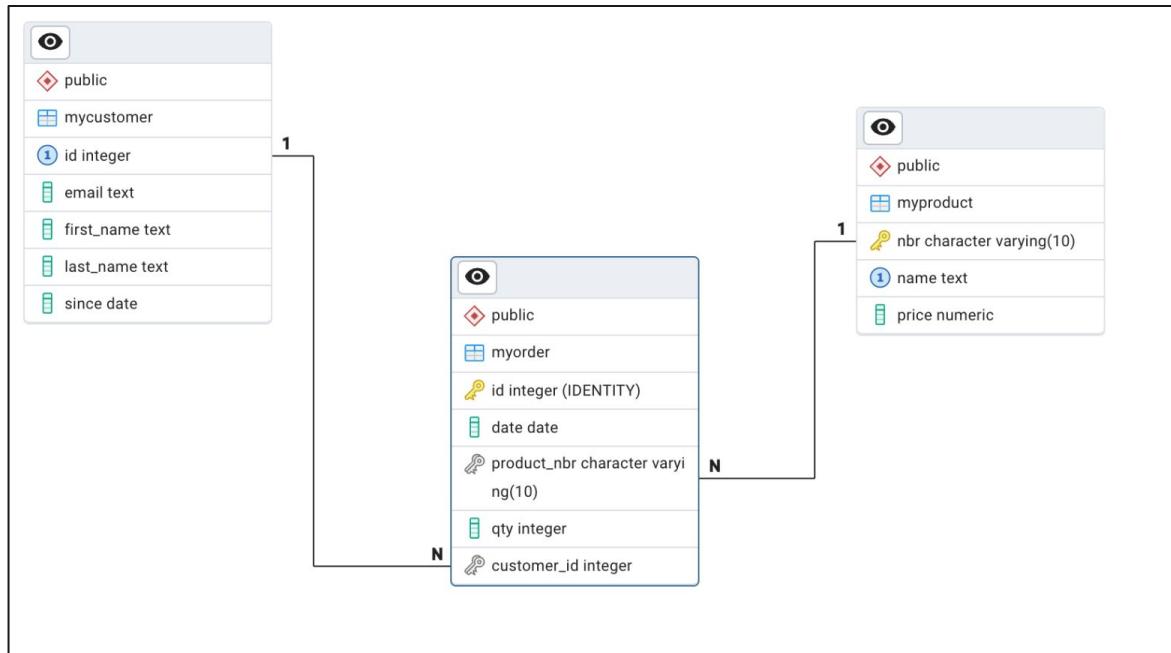
# Hierarchy of Postgres and SQL



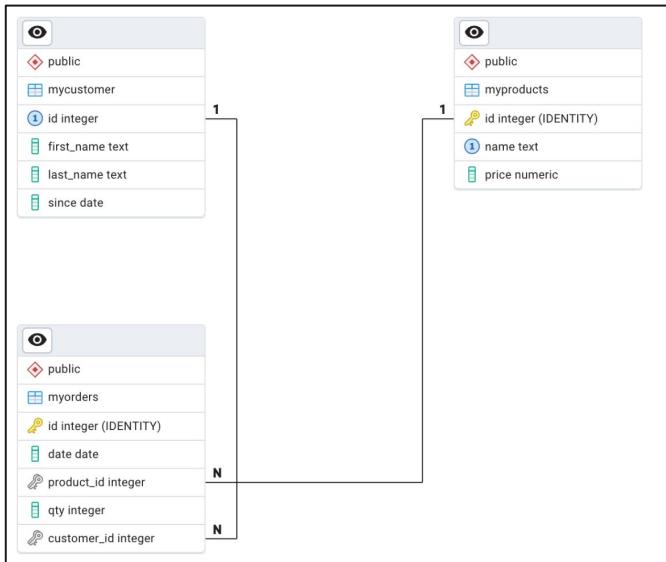
# Schemas

Graphical view of the schema

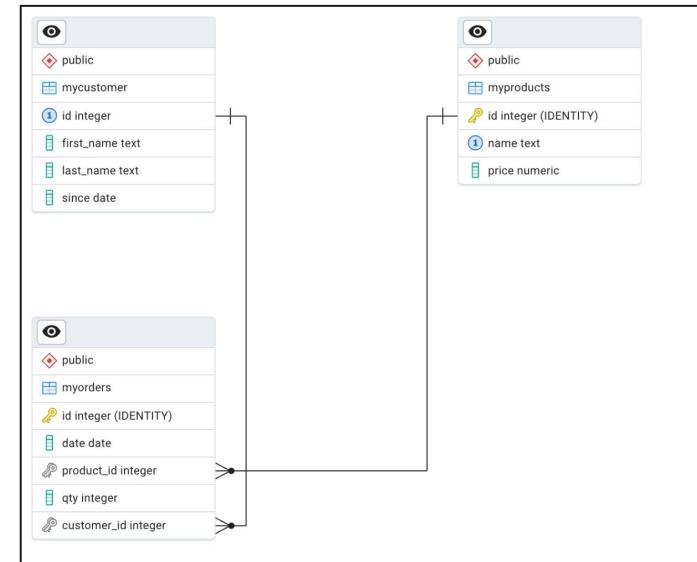
- Tables
- Columns
- Constraints
- Relationships
  - 1:1
  - 1:N
  - N:M



# Two Notations



Chen Notation



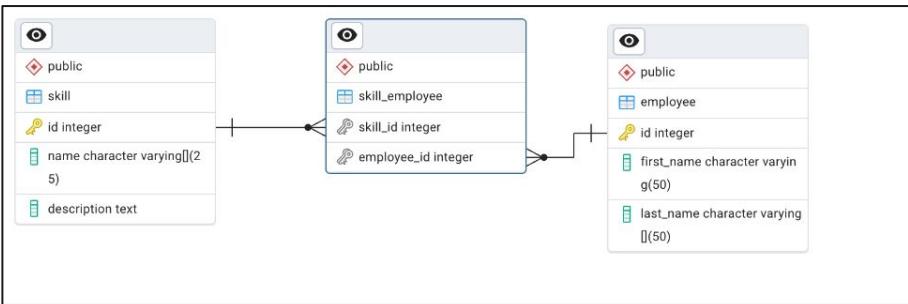
Crow Foot Notation

Different database

DLH2

# Use ERD Diagramming to create simple database

- Make sure you are connected to database DLH2
- Tables
  - Employees
  - Skills
  - Employees that have skill sets
- Use ERD tool to generate SQL
- Execute the generated SQL
- Inspect the results using the tree browser
- Connect to DLH2 with PSQL
- Inspect using the PSQL commands



Remove the BEGIN; and END;  
statement at the end of the  
generated SQL before executing

---

## **Queries revisited (joins, views, and aggregates)**

---

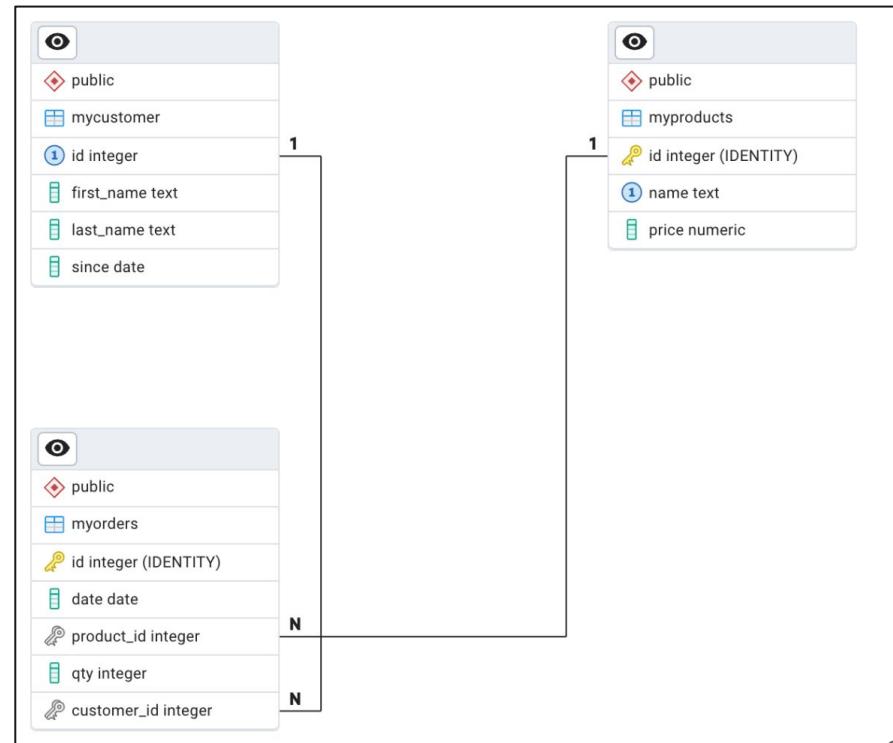
# Joins

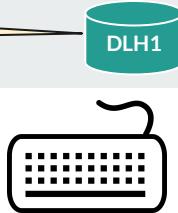
---

# 'JOIN ... ON ...' connects Two or more tables

Types of joins:

- (INNER) JOIN
- LEFT/RIGHT (OUTER) JOIN
- FULL (OUTER) JOIN
- CROSS JOIN





Different database

## Customers who ordered something

```
SELECT * FROM myorder
JOIN mycustomer
ON myorder.customer_id = mycustomer.id;
```

Uses the \x command in PSQL Tool to show the records in expanded mode

```
- [ RECORD 1 ] -----
id          | 1
date        | 2025-02-02
product_nbr | chisel_004
qty         | 9
customer_id | 1
nbr         | chisel_004
name        | Chisel stone
price       | 39.95
id          | 1
email       | marc@marclinster.com
first_name  | Marc
last_name   | Linster
since       | 2025-01-01
```

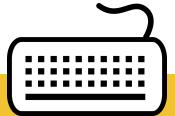


```
SELECT
    o.id AS order_id,
    o.date AS order_date,
    o.qty AS order_quantity,
    c.first_name || ' ' || c.last_name AS customer_name
FROM myorder o
    JOIN mycustomer c
        ON o.customer_id = c.id;
```

order_id	order_date	order_quantity	customer_name
1	2025-02-02	9	Marc Linster

- N.B.:
  - Explicit JOIN/ON improves readability
  - Using table alias (o,c) improves readability and avoids errors

How about the customers without orders?



# LEFT OUTER JOIN

```
SELECT * FROM mycustomer c
    LEFT OUTER JOIN myorder o
    ON c.id = o.customer_id;
```

id	first_name	last_name	since	id	date	product_id	qty	customer_id
1	Marc	Linster	2025-01-01	1	2025-02-02	3	3	1
2	Jeff	Mueller	2025-01-02					
3	Jean	Bintner	2025-01-03					

## LEFT (OUTER) JOIN:

- All records from the first (left) table
- Joined with second (right) table if available ; show NULL (empty) columns otherwise
- 'OUTER' is optional



# Explaining Left Join

mycustomer (left of join)

id	email	first_	last_	since
1	marc@marclinster.com	Marc	Linster	1/1/2025
2	jmu@gmail.lu	Jeff	Mueller	1/2/2025
3	bini@hotmail.lu	Jean	Bintner	1/3/2025

myorder (right of join)

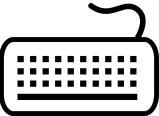
id	date	product_nbr	qty	customer_id
1	2/2/2025	chisel_004	9	1

## In simple join:

- align customers and orders using `mycustomer.id = myorder.customer_id`
- List only the customers that have a matching order, and the orders that have a matching customer

## In left join:

- align customers and orders using `mycustomer.id = myorder.customer_id`
- List all the customers (left) and if there is a matching order, list it too



## Joining more tables: Customer with orders and product details

```
SELECT * FROM mycustomer c
JOIN myorder o ON c.id = o.customer_id
JOIN myproduct p ON o.product_id = p.id;
```

id	first_name	last_name	since	id	date	product_id	qty	customer_id	id	name	price
1	Marc	Linster	2025-01-01	1	2025-02-02	3	3	3	1	pliers	2



# All customers, with orders and products (if they ordered anything)

```
SELECT * FROM mycustomer c
    LEFT JOIN myorder o ON c.id = o.customer_id
    LEFT JOIN myproduct p ON o.product_nbr = p.nbr;
```

id	first_name	last_name	since	id	date	product_id	qty	customer_id	id	name	price
1	Marc	Linster	2025-01-01	1	2025-02-02	3	3	1	3	pliers	2
2	Jeff	Mueller	2025-01-02								
3	Jean	Bintner	2025-01-03								

(3 rows)

- See the importance of meaningful alias
- 'id' appears three times.
- customer\_id, order\_id, product\_id would improve readability and reduce errors

---

## Try It Out

- Find the customers who didn't order anything.
  - Show only the name and since when they are customers
- Find the products that were never ordered

```
SELECT DISTINCT(p.*) FROM myproduct p
    LEFT JOIN myorder o ON o.product_nbr = p. nbr
    WHERE o.id IS NULL
```

Different  
database

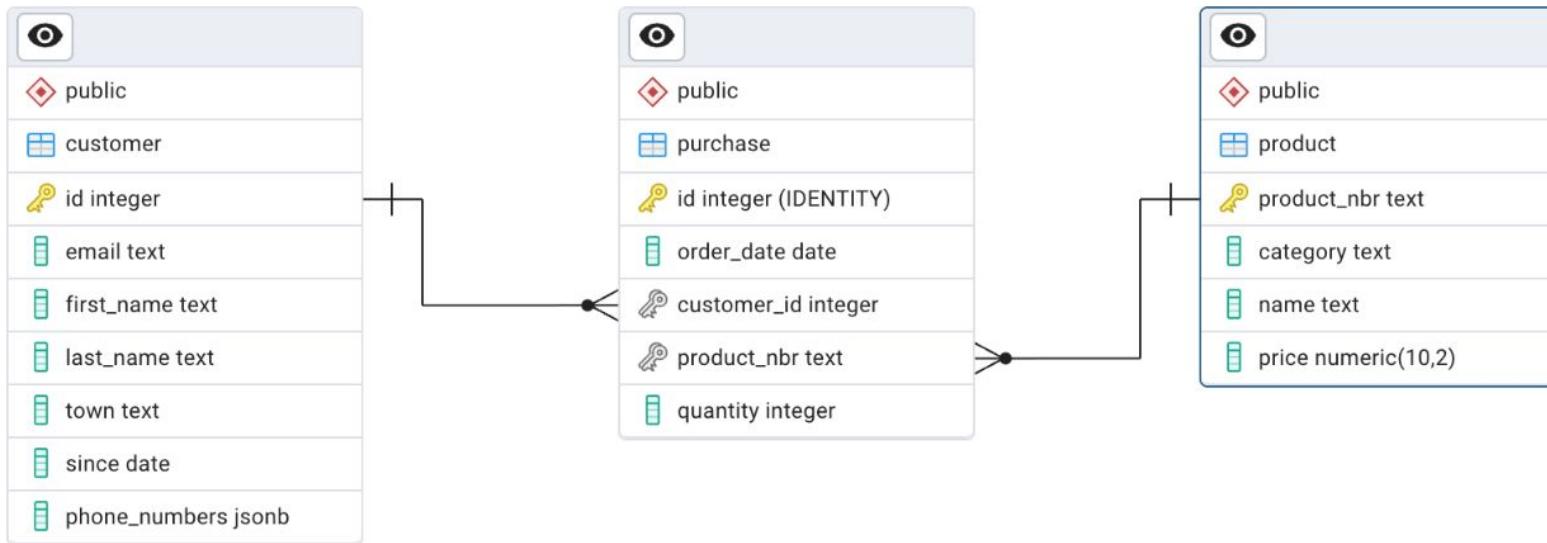
DLH3



## Explore the data set

- Connect to database DLH3
- Create the ERD Diagram
- Explore the tables
  - How many customers?
  - How many customers from Koerich?
  - List all the customers from Koerich that became a customer in June 2025.  
Order by date asc
  - How many orders were placed by customers from Differdange?
  - Which products did customers from Koerich order?
  - Which product categories did customers from Koerich order?

# ERD Diagram



---

# One more datatype: JSON for NoSQL

# JSON - a way to store unstructured data

---

- JSON(B) - JavaScript Object Notation (Binary)
- Allows to create complex data structures that are not subject to the usual rules of normalization and standardization

```
CREATE TABLE customer (
    id integer NOT NULL GENERATED BY DEFAULT AS IDENTITY,
    ...
    phone_numbers JSONB)

select * from customer where id = 6;
-[ RECORD 1 ]-----+
id          | 6
customer_number | cust00006
first_name    | Louis
last_name     | Lewandowski
address_id    | 111795
phone_numbers | {"home": "+352 621 453 975", "work": "+352 621 794 818", "mobile": "+352 80 37 20"}
```

# Complex JSON Example

```
"firstName": "John",           -- String Type
"lastName": "Smith",           -- String Type
"isAlive": true,               -- Boolean Type
"age": 25,                     -- Number Type
"height_cm": 167.6,            -- Number Type
"address": {                   -- Object Type
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
},
"phoneNumbers": [              // Object Array
    {}                         // Object
    {
        "type": "home",
        "number": "212 555-1234"
    },
    {
        "type": "office",
        "number": "646 555-4567"
    }
],
"children": [],
"spouse": null                 // Null
}
```

## 1. Number:

- Signed decimal number that may contain a fractional part and may use exponential notation.
- No distinction between integer and floating-point

## 2. String

- A sequence of zero or more Unicode characters.
- Strings are delimited with double-quotation mark
- Supports a backslash escaping syntax.

## 3. Boolean

- Either of the values true or false.

## 4. Array

- An ordered list of zero or more values,
- Each value may be of any type.
- Arrays use square bracket notation with elements being comma-separated.

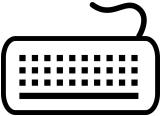
## 5. Object

- An unordered associative array (name/value pairs).
- Objects are delimited with curly brackets
- Commas to separate each pair
- Each pair the colon ':' character separates the key or name from its value.
- All keys must be strings and should be distinct from each other within that object.

## 6. null

- An empty value, using the word null

# Working with JSONB



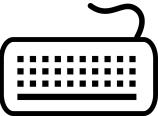
Try this in the PLSQL Tool or Query Tool

```
SELECT first_name, JSONB_PRETTY(phone_numbers) FROM customer LIMIT 5;
```

```
SELECT
    first_name,
    phone_numbers,
    phone_numbers ->> 'home' AS home_phone_number,
    phone_numbers ->> 'mobile' AS mobile_phone_number
FROM customer;
```

```
SELECT first_name, phone_numbers
FROM customer
WHERE phone_numbers ? 'mobile';
```

```
SELECT first_name, phone_numbers
FROM customer
WHERE NOT(phone_numbers ? 'home');
```



# Working with JSONB

- Adding to a JSON data element

```
UPDATE customer
    SET phone_numbers =
        jsonb_insert (phone_numbers, '{new}', '"123 456 789")'
WHERE id = 1;
```

- Removing a JSON data element

```
UPDATE customer
    SET phone_numbers =
        phone_numbers - 'new'
WHERE id = 1
```

<https://www.postgresql.org/docs/current/functions-json.html>



# What to know about JSONB?

- Use it for complex read-only data elements where data structure and type are less critical
  - Logs, telemetry, IOT data, data personalization
  - Tags for documents, multiple phone numbers, multiple addresses
  - Data collections with sparse columns
- Do not use it with foreign keys
- It can be indexed very efficiently using the GIN index
- It does not perform well for frequent updates - especially when the JSON data element is large
- Always prefer JSONB over JSON (older format)
- ROW\_TO\_JSON() and JSON\_TABLE make integration between JSON and standard tables very easy

# Querying JSON documents

```
SELECT id, nbrs.*  
FROM  
    customer,  
    JSON_TABLE (  
        phone_numbers,  
        '$' COLUMNS (  
            mobile TEXT PATH '$.mobile',  
            home TEXT PATH '$.home'  
        )  
    ) nbrs  
WHERE  
    nbrs.mobile IS NOT NULL  
LIMIT 5;
```

---

# Views

# CREATE VIEW

Views are named queries  
that are stored in the  
database server

```
CREATE VIEW customer_purchase_view AS
  SELECT
    c.id AS customer_id,
    c.first_name || ' ' || c.last_name AS
customer_name,
    c.town,
    p.product_nbr,
    pr.name AS product_name,
    pr.category as product_category,
    p.quantity,
    p.id AS purchase_id,
    p.order_date,
    (p.quantity * pr.price) AS total_price
  FROM customer c
  JOIN purchase p ON c.id = p.customer_id
  JOIN product pr ON p.product_nbr = pr.product_nbr;
```

pgAdmin 4

Servers (1)    Databases (3)    Schemas (1)    Views (1)

customer\_purchase\_view

Columns (10)

- customer\_id
- customer\_name
- town
- product\_nbr
- product\_name
- product\_category
- quantity
- purchase\_id
- order\_date
- total\_price

Total rows: 1

LF | Ln 1, Col 1

The screenshot shows the pgAdmin 4 interface. On the left, the Object Explorer tree displays a server named 'postgres17' with three databases: 'dlh1', 'dlh3', and 'postgres'. The 'dlh3' database is selected, showing its schemas ('public') and views ('customer\_purchase\_view'). The 'customer\_purchase\_view' view is expanded, revealing its 10 columns: customer\_id, customer\_name, town, product\_nbr, product\_name, product\_category, quantity, purchase\_id, order\_date, and total\_price. The main pane is a query editor titled 'Query' with a single digit '1' entered. Below the query editor are tabs for 'Data Output', 'Messages', and 'Notifications'. At the bottom, there are status bars for 'Total rows: 1' and 'Screenshot', along with a footer bar showing 'LF | Ln 1, Col 1'.

pgAdmin 4

Servers (1)    Databases (3)    Schemas (1)    Views (1)

customer\_purchase\_view

General    Definition    Code    Security    SQL

```
1  SELECT c.id AS customer_id,
2      (c.first_name || ' ' || c.last_name) AS customer_name,
3      c.town,
4      p.product_nbr,
5      pr.name AS product_name,
6      pr.category AS product_category,
7      p.quantity,
8      p.id AS purchase_id,
9      p.order_date,
10     p.quantity::numeric * pr.price AS total_price
11  FROM customer c
12  JOIN purchase p ON c.id = p.customer_id
13  JOIN product pr ON p.product_nbr = pr.product_nbr;
```

Columns (10)

- customer\_id
- customer\_name
- town
- product\_nbr
- product\_name
- product\_category
- quantity
- purchase\_id
- order\_date
- total\_price

Total rows: 0    Screenshot

Close    Reset    Save

The screenshot shows the pgAdmin 4 interface. On the left is the Object Explorer tree, which includes a 'Servers' section with one entry 'postres17' containing 'Databases', 'Schemas', and 'Views'. The 'Views' node is expanded, showing a single entry 'customer\_purchase\_view'. Below it is a 'Columns' node listing ten columns: customer\_id, customer\_name, town, product\_nbr, product\_name, product\_category, quantity, purchase\_id, order\_date, and total\_price. The main panel displays the SQL code for the view. At the bottom, there's a status bar with 'Total rows: 0' and a 'Screenshot' button. The bottom right corner of the status bar has 'LF' and 'Ln 1, Col 1'.

---

# Aggregates

---

# Questions that require aggregates

For example

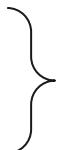
- How many customers do we have in every town?
- Which town has the most customers?
- How much did the top 10 customers buy?

# SQL Aggregates

- GROUP BY, e.g., group the customers by location
  - GROUP BY ... HAVING allows to limit the groups
- SUM, AVG, MIN, MAX, e.g, the total/ average/ minimum/ maximum within a group



After the WHERE clause



Part of the SELECT clause



# Aggregates

```
SELECT group_by_element_1, group_by_element_2, ..., aggregate_1, aggregate_2, ...
      FROM table1
      JOIN table2 ON ....
      WHERE condition
      GROUP BY group_by_element_1, group_by_element_2, ...
```

```
SELECT town, COUNT(id)
      FROM customer
      GROUP BY TOWN
      ORDER BY town ASC;
```

# Aggregates

```
SELECT town, COUNT(id) customer_count FROM customer  
GROUP BY TOWN  
ORDER BY town ASC;
```

```
SELECT town, COUNT(id) AS customer_count FROM customer  
WHERE since >= '2024-01-01' AND since < '2024-12-31'  
GROUP BY TOWN  
HAVING COUNT(id) > 1;
```

```
SELECT town, COUNT(id) AS customer_count FROM customer  
WHERE since >= '2024-01-01' AND since < '2024-12-31'  
GROUP BY TOWN  
HAVING COUNT(id) > 1  
ORDER BY customer_count DESC;
```

Single group - one aggregate (COUNT)

Single group - one aggregate; WHERE clause + limit of the groups

Single group - one aggregate; WHERE clause + limit of the groups + ORDER BY for groups

# Aggregates

```
SELECT
    town,
    EXTRACT (YEAR FROM since) AS since_year,
    COUNT(id) AS customer_count FROM customer
GROUP BY town, since_year
HAVING COUNT(id) > 1
ORDER BY town ASC, since_year ASC;
```

```
SELECT
    town,
    EXTRACT (YEAR FROM since) AS since_year,
    COUNT(id) AS customer_count FROM customer
WHERE TOWN <> 'ESCH'
GROUP BY town, since_year
HAVING COUNT(id) > 1
ORDER BY town ASC, since_year ASC;
```

Use the 'HAVING' clause to remove groups with small customer counts

Use the 'HAVING' clause to remove groups with small customer counts  
+

Use WHERE clause to filter on data set

---

# Exercises

- How many food items did we sell in 2024
  - Join purchases and products
  - Filter by category food
  - Count(\*)
- How many things did the top 10 customers buy?
  - Join the customer table with the purchase table
  - Group by customer and count the quantities
  - LIMIT 10
- How much did the top 10 customers spend every month?
  - Take a look at the view customer\_purchase\_view
  - Group by customer\_name and make an aggregate of the total price
  - Order by price
  - LIMIT 10

---

# GROUPING SETS, ROLLUPS and CUBE

ROLLUP (a,b,c) corresponds to  
GROUPING SETS

```
(  
    (a,b,c),  
    (a,b),  
    (c),  
    ()  
)
```

CUBE ( a, b, c ) corresponds to  
GROUPING SETS

```
(  
    ( a, b, c ),  
    ( a, b ),  
    ( a, c ),  
    ( a ),  
    ( b, c ),  
    ( b ),  
    ( c ),  
    ( )  
)
```



# GROUP BY ROLLUP

```
SELECT product_category, product_name, sum(quantity)
      FROM customer_purchase_view
 GROUP BY ROLLUP (product_category, product_name)
 ORDER BY product_category, product_name;
```

```
SELECT FORMAT ('%s living in %s', customer_name, town) AS customer_town ,
product_category, product_name, sum(quantity)
      FROM customer_purchase_view
     WHERE TOWN <> 'Esch'
 GROUP BY ROLLUP (customer_town, product_category, product_name)
 ORDER BY customer_town, product_category, product_name;
```



# GROUP BY ROLLUP

```
SELECT FORMAT ('%s living in %s', customer_name, town) AS customer_town ,  
product_category, product_name, sum(quantity)  
FROM customer_purchase_view  
WHERE TOWN <> 'Esch'  
GROUP BY CUBE (customer_town, product_category, product_name)  
ORDER BY customer_town, product_category, product_name;
```

---

# Guidelines for Queries



# Summary: Keys to Writing Good Queries

1. Limit the number of rows that are returned
2. Limit the number of columns that are returned
3. Leverage indexes where available
4. Define aliases in the SELECT clause using 'AS' to avoid duplicate column names
5. Use the aliases in the WHERE and GROUP clause to avoid typos and shorten the code
6. Define highly selective WHERE clauses, especially when using JOINS
7. Avoid CROSS JOINS (data set explodes!)
8. Use VIEWS to standardize complex, often repeated queries (e.g., sales\_transaction combined with sales\_transaction\_line)
9. Use snake\_case names for tables and columns
10. Study EXPLAIN plans to understand where complexity occurs

---

## FYI: SQL Order of Execution

1. **FROM/JOIN**: Specifies the tables from which to retrieve data.
2. **WHERE**: Filters the rows that meet the condition before grouping.
3. **GROUP BY**: Groups rows that share a property.
4. **HAVING**: Filters groups based on conditions, applied after grouping.
5. **SELECT**: Specifies the columns to retrieve or calculate.
6. **DISTINCT**: Removes duplicate rows from the result set.
7. **ORDER BY**: Sorts the result set by specified columns.
8. **LIMIT**: Specifies the maximum number of rows to return.
9. **OFFSET**: Specifies how many rows to skip before starting to return rows.

# Day 4

- Review of Day 3
- Data Normalization
- ACID compliance, Transactions, and Isolation Levels
- Creating your own database with schema, tables, and foreign keys



---

# Data Normalization



# Normalization and Avoiding Redundancy

- Data Normalization, a.k.a. 1NF/2NF,3NF is key to relational database design
- Data Normalization concepts do not exist in non-relational databases
  - Make sure every column only has one data element of the same type
  - Avoid redundancy (same data in multiple places)
  - Eliminate unintentional dependencies



# Database Normal Forms

- **1st Normal Form (1NF)**: Tables do not include other tables; every column has one data element of the same type
- **2nd Normal Form (2NF)**: All data elements depend on the whole primary key
- **3rd Normal Form (3NF)**: No internal dependencies between data elements

Other (higher) normal forms exist, but are rarely applied in practice

# Example

Article	Description	Brand	Country of origin	Sizes	Colors	Price
501 Jeans by Levis	The original button-fly jeans	Levis	USA	32/36 30/34 30/36	Blue	95.00
501 Jeans by Levis	The original button-fly jeans	Levis	USA	32/36 30/34 30/36	Black	110.99
Wrangler Cowboy Cut	Western jeans for a great day	Wrangler	USA	32/36 30/34 30/36	Blue	98.00
Wrangler Cowboy Cut	Western jeans for a great day	Wrangler	USA	32/36 30/34 30/36	Black	105.99
Wrangler Cowboy Cut	Western jeans for a great day	Wrangler	USA	32/36 30/34 30/36	Stone wash	101.99
Journey Trousers	Cotton gabardine Journey trousers	Brioni	Italy	36 38 40 42	Ivory White	125.95
Trucker Jacket	Rugged jeans jacket	Levis	USA	S M L	Blue	113.00

# Example

Sizes has multiple values in one column

Article	Description	Brand	Country of origin	Sizes		
501 Jeans by Levis	The original button-fly jeans	Levis	USA	32/36 30/34 30/36	B	95.00
501 Jeans by Levis	The original button-fly jeans	Levis	USA	32/36 30/34 30/36	Black	110.99
Wrangler Cowboy Cut	Western jeans for a great day	Wrangler	USA	32/36 30/34 30/36	Blue	98.00
Wrangler Cowboy Cut	Western jeans for a great day	Wrangler	USA	32/36 30/34 30/36	Black	105.99
Wrangler Cowboy Cut	Western jeans for a great day	Wrangler	USA	32/36 30/34 30/36	Stone wash	101.99
Journey Trousers	Cotton gabardine Journey trousers	Brioni	Italy	36 38 40 42	Ivory White	125.95
Trucker Jacket	Rugged jeans jacket	Levis	USA	S M L	Blue	113.00

Article	Description	Brand	Country of origin	Colors	Price
501 Jeans by Levis	The original button-fly jeans	Levis	USA	Blue	95.00
501 Jeans by Levis	The original button-fly jeans	Levis	USA	Black	110.99
Wrangler Cowboy Cut	Western jeans for a great day	Wrangler	USA	Blue	98.00
Wrangler Cowboy Cut	Western jeans for a great day	Wrangler	USA	Black	105.99
Wrangler Cowboy Cut	Western jeans for a great day	Wrangler	USA	Stone wash	101.99
Journey Trousers	Cotton gabardine Journey trousers	Brioni	Italy	Ivory White	125.95
Trucker Jacket	Rugged jeans jacket	Levis	USA	Blue	113.00

Article	Size
501 Jeans by Levis	32/36
501 Jeans by Levis	30/34
501 Jeans by Levis	30/36

To transform this into 1NF, we need to move the content of column Sizes to a separate table.



## **2NF: All data elements depend on the whole primary key**

To meet the requirements of 2NF, where all attributes must depend on the *whole* primary key, we must address the following:

1. Category and Brand depend on Article, that is, Article is the primary key for the content in columns Category and Brand
2. Article + Color are the combined primary key for Price

This leads us to create a separate table that links Article and Color to Price



## 2NF: Create a separate table that links Article and Color to Price

Article	Description	Brand	Country of origin
501 Jeans by Levis	The original button-fly jeans	Levis	USA
Wrangler Cowboy Cut	Western jeans for a great day	Wrangler	USA
Journey Trousers	Cotton gabardine Journey trousers	Brioni	Italy
Trucker Jacket	Rugged jeans jacket	Levis	USA

Article	Colors	Price
501 Jeans by Levis	Blue	95.00
501 Jeans by Levis	Black	110.99



## 3NF: No internal dependencies between data elements

Brand	Country of Origin
Levis	USA
Wrangler	USA
Brioni	Italy

Country of origin depends on brand (only). Move to separate table

# 3NF - No duplication; Easy access to data

Article	Description	Brand
501 Jeans by Levis	The original button-fly jeans	Levis
Wrangler Cowboy Cut	Pants	Wrangler
Journet Trousers	Pants	Brioni
Trucker Jacket	Outerwear	Levis

Brand	Country of Origin
Levis	USA
Wrangler	USA
Brioni	Italy

Article	Attributes	Price
501 Jeans by Levis	{color: Blue, size: 32/36}	95.00
501 Jeans by Levis	{color: Blue, size: 30/34}	95.00
501 Jeans by Levis	{color: Black, size: 30/34}	110.99
501 Jeans by Levis	{color: Black, size: 30/36}	110.99



## Normalization Matters!

- Avoids managing the same data in multiple places
- Avoids strange deletion situations, e.g., brand disappears when no article uses it
- Makes sure there is always a way to refer to data in a unique way
- Avoids designing the process into the data

---

# Transactions and ACID Compliance



# Transactions

- Sample Transaction

```
BEGIN
```

```
    Remove Euro 100 from Marc's account
```

```
    Add Euro 100 to 'Caves Wengler' account
```

```
COMMIT
```

- Key Considerations for a Database:

- **Atomic:** All or nothing
- **Consistent:** After the transaction both accounts are updated immediately and all constraints are validated (e.g., account > 0). Every user always see a consistent picture
- **Isolated:** Multiple transactions can happen at the same time. They either see the state before the transaction, or after the transaction - never in-between
- **Durable:** Once a transaction has completed, the result remains true.

# ACID - continued

- Atomic:
  - Multiple statements (INSERT, UPDATE, DELETE, ...) are executed as one atomic unit. If one of them fails, the others roll back. Even if power fails, an error occurs etc
  - Only completed transactions are written to disk and persisted
- Consistent:
  - All conditions (constraints, primary keys, foreign key constraints, data types) are true at the end of a transaction - otherwise the transaction roll back
  - Every transaction leaves the database in a consistent state
- Isolation:
  - Transactions 'in flight' and their results (e.g., Marc's account deducted by Euro 100, but funds are not yet in 'Caves Wengler' account) are not visible
  - Transactions work on a snapshot of what is true when the transaction starts
- Durable
  - Once a transaction completes (is committed), its results remain true
  - Databases log every transaction to disk (aka Write Ahead Log). 'COMMIT' only happens when the transaction has been written to non-volatile memory.



# Does every database provide ACID compliance

- ACID databases (e.g., SQL Server, Oracle, Postgres) prioritize consistency over availability
  - They reject transactions when ACID cannot be guaranteed
  - Important for business transactions
- Other databases prioritize availability over consistency
  - BASE: Basically Available, Soft state, and Eventually consistent
  - Used for high throughput systems where temporary inconsistency is OK, e.g., social media
- Isolation (in ACID) comes in multiple flavors
  - Read Uncommitted (not used in current relational databases)
  - Read Committed - most common model
  - Read Repeatable - needs locks on data and can impact performance
  - Read Serializable - locks the data used by the transaction and forces other transactions to wait (seldom used)

# Exploring ACID Transactions

```
CREATE TABLE account (id INT, amount NUMERIC);
```

```
INSERT INTO account VALUES (1, 100);
```

```
INSERT INTO account VALUES (2, 0);
```

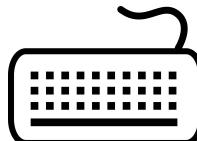
Create the tables and add the data

Verity with `SELECT * FROM account;`

Make sure you are in DLH5!!!

- `\c` (in PSQL)
- `SELECT current_database();` (in Query Tool)

id	amount
1	50
2	50



# Open up two pgAdmin Query Tools (or PSQL)

```
(1)  SELECT * FROM account  
      ORDER BY id ASC;
```

```
(3)  BEGIN;
```

```
(5)  UPDATE account  
      SET amount = amount - 50  
      WHERE id = 1;
```

```
(7)  SELECT * FROM account  
      ORDER BY id ASC;
```

```
(8)  UPDATE account  
      SET amount = amount + 50  
      WHERE id = 2;
```

```
(9)  COMMIT;
```

```
(2)  SELECT * FROM account  
      ORDER BY id ASC;
```

```
(4)  SELECT * FROM account  
      ORDER BY id ASC;
```

```
(6)  SELECT * FROM account  
      ORDER BY id ASC;
```

```
(10) SELECT * FROM account  
      ORDER BY id ASC;
```

# Nota Bene

- By default, “;” commits the transaction
  - ⇒ Every line of SQL is a transaction in pgAdmin (by Default)
  - From the Postgres docs:
    - PostgreSQL actually treats every SQL statement as being executed within a transaction. If you do not issue a `BEGIN` command, then each individual statement has an implicit `BEGIN` and (if successful) `COMMIT` wrapped around it. A group of statements surrounded by `BEGIN` and `COMMIT` is sometimes called a *transaction block*.
- More complex transactions with multiple statements must be wrapped into a transaction block

```
BEGIN;
```

```
...;
```

```
COMMIT;
```

- Try the prior example without the BEGIN/COMMIT



# Transaction Isolation Levels

<b>Read Uncommitted:</b>	Transactions can see uncommitted changes from other transactions, risking dirty reads. ( <b><i>Not available in Postgres</i></b> )
<b>Read Committed:</b>	Transactions can only see committed changes, preventing dirty reads but allowing non-repeatable reads. ( <b><i>Default in Postgres</i></b> )
<b>Read Repeatable:</b>	A transaction's reads are consistent, preventing dirty and non-repeatable reads but allowing phantom reads.
<b>Serializable:</b>	Transactions are fully isolated, executed in a serial order to prevent dirty, non-repeatable, and phantom reads

---

# Isolation Levels and Anomalies

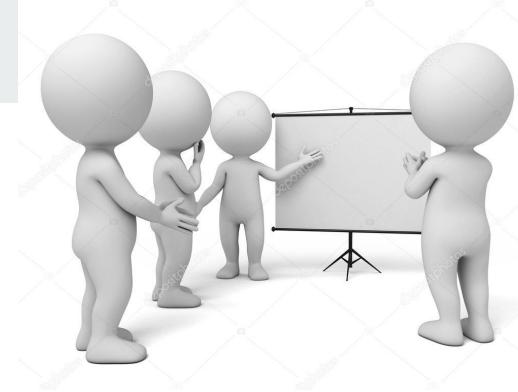
Isolation Level/ Anomaly	Dirty Read	Non-Repeatable Read	Phantom Read
Read Uncommitted	✗ Possible	✗ Possible	✗ Possible
Read Committed	✓ Not possible	✗ Possible	✗ Possible
Read Repeatable	✓ Not possible	✓ Not possible	✗ Possible
Serialized	✓ Not possible	✓ Not possible	✓ Not possible

---

# A sample project

---

# Project



- Build a database (dlhx) for a company with departments, employees and managers working in departments, projects, employee assignment to projects, and hours recorded for the employees working on the project
- Simple queries
  - Who worked on which project?
  - Who had the most hours total last month?
  - What were the total hours last month for each project?
  - Did we have any projects with 0 hours?
  - What is the reporting hierarchy in the company?

# Day 5

- Review of Day 4; Revisit and complete the project
- Completing the project
- Stored procedures
- Common Table Expressions (CTE)
- Recap, Q&A, resources



---

# Stored Procedures

# Stored Procedures Automate Repeated Tasks

```
CREATE PROCEDURE <procedure name> (
    IN/OUT p_parameter1 <DATATYPE>
        DEFAULT = <x>;
    ...
)
AS
$$
    DECLARE
        v_variable1 <DATATYPE>;
    ...
BEGIN
    <STATEMENT>;
    <STATEMENT>;
END
$$ LANGUAGE <language>;
```



Signature (name + parameters)



Variables that we need



Body of the procedure



Programming language (sql, plpgsql, plpython, ...)



# Simple Example

```
CREATE OR REPLACE PROCEDURE increase_price (IN category TEXT, IN increase NUMERIC)
LANGUAGE SQL
AS $$

    UPDATE product
    SET price = price + increase
    WHERE category = category;

$$;

SELECT * FROM product WHERE category = 'food';

CALL increase_price ('food', 1.00);
```

---

## Iterating through the records in a table

```
DECLARE
    <record_variable> RECORD;
    ...;
BEGIN
    FOR <record variable> IN SELECT * FROM product
    LOOP
        <statement 1>;
        ...
    END LOOP;
END;
```

---

# Controlling the Flow

```
IF <condition> THEN  
    <statement1>;  
    ...;  
ELSE  
    <statement1>;  
    ...;  
END IF;  
  
IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF;
```

```
CASE search-expression  
WHEN expression [, expression [ ... ]] THEN  
    statements  
[ WHEN expression [, expression [ ... ]] THEN  
    statements  
    ... ]  
[ ELSE  
    statements ]  
END CASE;
```

```
CREATE OR REPLACE PROCEDURE increase_price_differentiated
    (IN general_increase_percent NUMERIC, IN food_increase_percent NUMERIC)
LANGUAGE PLPGSQL
AS $$

DECLARE
    v_product RECORD;
BEGIN
    FOR v_product IN SELECT * FROM product
    LOOP
        IF v_product.category = 'food' THEN
            UPDATE product
            SET price = price + (price * food_increase_percent / 100)
            WHERE product_nbr = v_product.product_nbr;
            RAISE NOTICE 'Increased price of % by % percent', v_product.name, food_increase_percent;
        ELSE
            UPDATE product
            SET price = price + (price * general_increase_percent / 100)
            WHERE product_nbr = v_product.product_nbr;
            RAISE NOTICE 'Increased price of % by % percent', v_product.name, food_increase_percent;
        END IF;
    END LOOP;
END;
$$;

SELECT category, name, price FROM product
ORDER BY category, name;

CALL increase_price_differentiated (5, 10);
```



## Try it out

- add a parameter for the category 'snacks' with a different price increase
- Use IF / ELSEIF ....
- Use CASE

---

# Introduction to CTEs



# Common Table Expressions

What are Common Table Expressions?

1. Simplify the writing of complex queries by **decomposing** queries into separate pieces
  - o Avoid subqueries
  - o Comparable to writing local views that can only be used inside one query
2. Writing recursive queries
  - o Iterate through **hierarchical data structures**
  - o E.g. organizational charts, directory structures, or any data that has a parent-child relationship.

CTEs are available in most relational databases



# CTE Building Blocks

- 1) 'WITH' defines one or several named queries that creates subsets of data. These are temporary result sets
- 2) 'SELECT' statement can access those queries

```
WITH cte_name (column1, column2, ...) AS  
    (SELECT ...)  
-- Main query using the CTE  
SELECT ...FROM cte_name;
```

# CTE Building Blocks

- **WITH clause:** Introduces the CTE
- **CTE name:** Names the CTE. Name must be unique in scope of query.
- **Column List (optional):** If not provided, the column definitions are inherited from the SELECT clause
- **AS keyword:** The CTE definition.
- **CTE query:** This is a query that defines the CTE, which may include JOINS, WHERE, GROUP BY clauses, and other valid SQL constructs. Several queries, each with their own CTE name, are allowed.
- **Main query:** Can reference the CTE(s) by name. In the main query, you can use the CTE as if it were a regular table, simplifying the structure of complex queries.

```
WITH cte_name (column1, column2, ...) AS  
    (SELECT ...)  
-- Main query using the CTE  
SELECT ...FROM cte_name;
```



## Benefits of Using CTEs

- Readability and maintenance: CTEs are more readable and break complex queries down into simpler parts.
- Ease of debugging: Test parts of a complex query independently, making debugging easier.
- Performance: They can make query optimization by the planner more effective

---

## Putting CTEs to use

- What products are the Top 10 customers buying?
  - Create a CTE to find the Top 10 customers
  - Create a CTE that shows product sales by customer
  - Join both CTEs
  - Apply proper formatting

## Sample CTE: What did my overall top 10 customers by in June?

```
WITH top10_customers AS (
    SELECT customer_id, SUM(total_price) AS total_spent
    FROM customer_purchase_view
    GROUP BY customer_id
    ORDER BY total_spent DESC
    LIMIT 10
)
```

```
SELECT
    cpv.customer_name,
    cpv.product_name,
    SUM(cpv.quantity) AS total_qty,
    SUM(cpv.total_price) AS total_spent
FROM customer_purchase_view cpv
JOIN top10_customers tc ON cpv.customer_id = tc.customer_id
WHERE cpv.order_date >= '2025-06-01' AND cpv.order_date < '2025-07-01'
GROUP BY ROLLUP (cpv.customer_name, cpv.product_name)
ORDER BY cpv.customer_name, total_spent ASC;
```

} Common Table Expression - who are the overall top 10 customers?

} Given these customers:  
What did they buy in June 2025

---

## CTEs for Recursive Queries

- Used to process hierarchical type data structures
  - Bill of materials
  - Company hierarchy
  - ...
- The CTE results are appended to the base table

---

# Building Blocks of the Recursive CTE

```
WITH RECURSIVE cte_name (column1, column2, columnN)
AS (
    SELECT columns FROM table1 WHERE condition

    UNION [ALL]

    SELECT columns FROM cte_name
        WHERE recursive_condition
)
SELECT columns FROM cte_name;
```

- WITH RECURSIVE introduces the recursive CTE
- List of columns (optional)
- Initial SELECT clause to build the first record of the CTE (i.e. the non-recursive term)
- UNION to append the next clauses to the initial clause
- Recursive SELECT clause
- UNION appends the results of the recursive clause and the initial clause to build the result set
- WHERE ends the recursion
- Main query (last SELECT) works on the CTE result set as if it was a table



## Nota Bene

- The column counts and definitions between the non-recurring clause and the recurring clause must be identical
- UNION ALL brings both data sets together
- UNION (without ALL) removes duplicates
- UNION (without ALL) is much slower as it has to check for duplicates

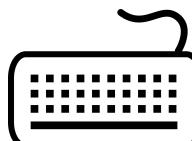
# Example Data Set

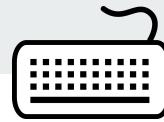
```
CREATE TABLE employee (
    id INT PRIMARY KEY,
    name TEXT,
    manager_id INT);
```

Dataset has been pre-loaded in DLH4.

Try `SELECT name ORDER BY salary ASC;`

id	name	manager_id	salary
15	Bill		75000
1	Marc	15	67500
2	Susi	15	68300
3	Jake	15	66450
4	Karin	2	62450
5	Bill	2	63000
6	John	4	49000
7	Frank	4	51000
8	Siggi	4	53000
9	Theo	4	52600
10	Barbara	9	46750

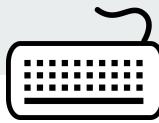




# CTE for Simple Hierarchy

```
WITH RECURSIVE employee_hierachy (id, name, manager_id)
AS (
    SELECT id, name, manager_id
        FROM employee
       WHERE manager_id IS NULL
  UNION ALL
    SELECT e.id, e.name, e.manager_id
        FROM employee e
     INNER JOIN employee_hierachy eh
           ON e.manager_id = eh.id
)
SELECT * FROM employee_hierachy;
```

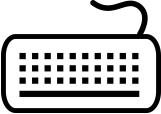
- WITH RECURSIVE introduces the recursive CTE
- List of columns (optional)
- Initial SELECT clause to build the first record of the CTE (i.e. the non-recursive term)
- UNION to append the next clauses to the initial clause
- Recursive SELECT clause
- UNION appends the results of the recursive clause and the initial clause to build the result set
- WHERE ends the recursion
- Main query (last SELECT) works on the CTE result set as if it was a table



# Hierarchy with Level Count

```
WITH RECURSIVE employee_hierarchy AS (
    SELECT id, name, manager_id, 1 AS level -- initiate level count
        FROM employee WHERE manager_id IS NULL -- define starting point
    UNION ALL
    SELECT e.id, e.name, e.manager_id,
        eh.level +1 -- add to the level count
        FROM employee e
        INNER JOIN employee_hierarchy eh ON e.manager_id = eh.id
)
SELECT * FROM employee_hierarchy;
```

# Hierarchy with Manager Name and Reporting Line

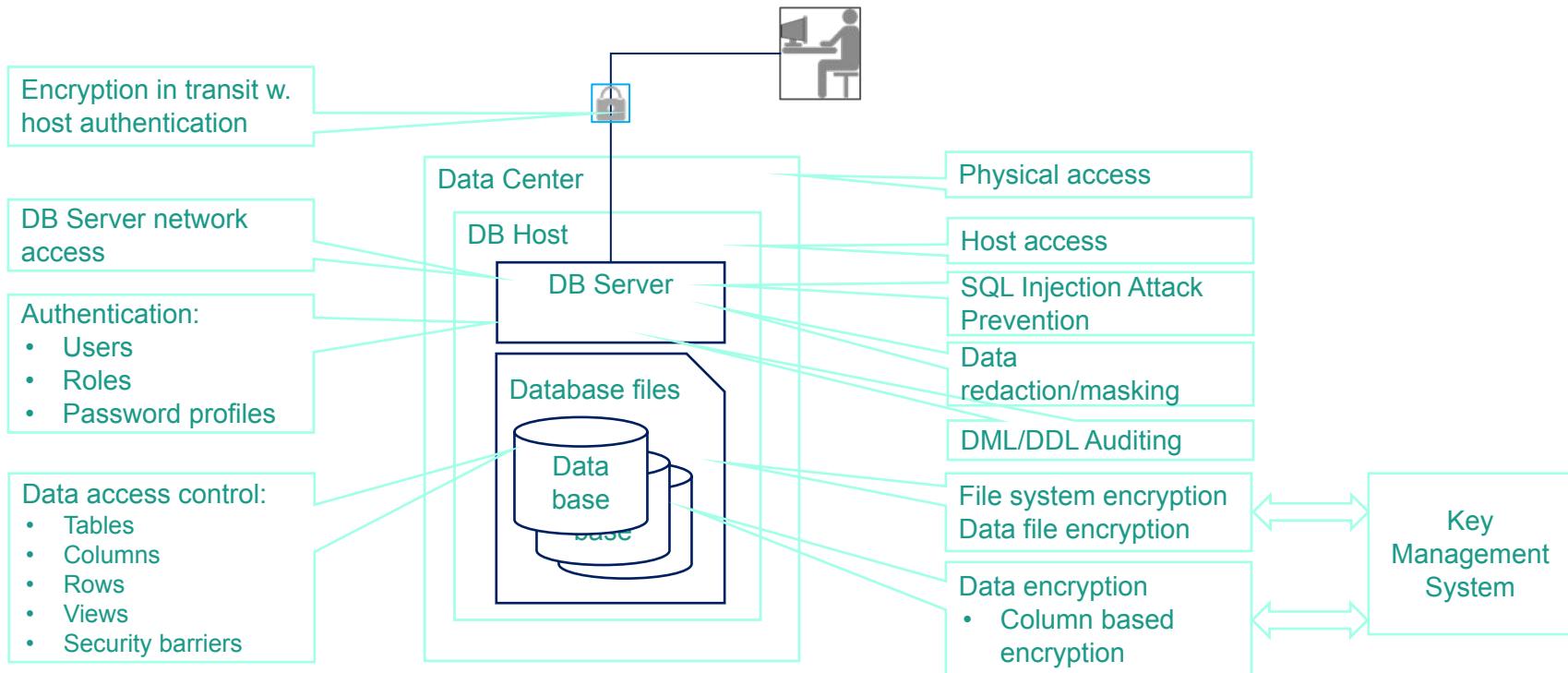


```
WITH RECURSIVE employee_hierachy AS (
    SELECT id, name, manager_id, NULL AS manager_name,
        1 AS level,
        'Top' as reporting_line
    FROM employee
    WHERE manager_id IS NULL
UNION ALL
    SELECT e.id, e.name, eh.id, eh.name AS manager_name, eh.level +1,
        FORMAT('%s/%s', eh.reporting_line, eh.name) -
    FROM employee e
    INNER JOIN employee_hierachy eh ON e.manager_id = eh.id
)
SELECT
    employee_hierachy.manager_id,
    employee_hierachy.manager_name,
    employee_hierachy.level as reporting_level,
    employee_hierachy.id as employee_id,
    employee_hierachy.name as employee_name,
    employee_hierachy.reporting_line
FROM employee_hierachy
ORDER BY reporting_level, manager_name ASC
```

---

# Users, permissions and access control

# MULTIPLE LAYERS OF SECURITY





# Today's Focus

- Roles, Users and Groups
  - **Users**: Roles with login rights
  - **Groups**: Roles without login rights; used the regroup rights definitions
- Access Controls
  - **Databases**
  - **Schemas**
  - **Tables**
  - Columns
  - Rows
  - Functions & procedures
  - Sequences
  - etc



# Managing Users

CREATE ROLE name [ [ WITH ] option [ ... ] ]

where option can be:

- SUPERUSER | NOSUPERUSER
- | CREATEDB | NOCREATEDB
- | CREATEROLE | NOCREATEROLE
- | INHERIT | NOINHERIT
- | LOGIN | NOLOGIN
- | REPLICATION | NOREPLICATION
- | BYPASSRLS | NOBYPASSRLS
- | CONNECTION LIMIT connlimit
- | [ ENCRYPTED ] **PASSWORD** 'password' | PASSWORD NULL
- | VALID UNTIL 'timestamp'
- | **IN ROLE** role\_name [, ...]
- | ROLE role\_name [, ...]
- | ADMIN role\_name [, ...]
- | SYSID uid

URL: <https://www.postgresql.org/docs/current/sql-createrole.html>;



## GRANT/REVOKE

# Access to database objects

#1 rule:

- Minimal access!
- Grant users the least amount of rights (privileges) that they need to do their work
- Start by removing all rights and privileges, and then grant only those that are necessary
- Use groups to manage the rights

SELECT

INSERT

UPDATE

DELETE

TRUNCATE

REFERENCES

TRIGGER

CREATE

CONNECT

TEMPORARY

EXECUTE

USAGE

SET

ALTER SYSTEM

MAINTAIN



# Example

- Goal:
  - Create a new database called `dlh6` with a single schema `websitedata` and a single table `customer`
  - Create two new user groups
    - `websiteanalyst` - they have read only access to the schema and the table
    - `website editor` - they have read/write access
- Approach:
  - Lets use `PSQL` - the Postgres command interface for most of this

# Create a new database, schema and table (use PSQL!)

---

Step	Description
Right click on PostgreSQL17/postgres and select PSQL	Connect to database <b>postgres</b> as user <b>postgres</b> (super user) using the PSQL Tool
\connectioninfo	Shows the database and user. Make sure we are in the right place (database <b>postgres</b> ) with the right login
CREATE DATABASE DLH6;	Create an empty database
\c DLH6;	Connect to the new database
\dn;	List the currently defined schemas
DROP SCHEMA public;	Drop unnecessary schema
CREATE SCHEMA websitedata;	Create an empty schema
ALTER DATABASE dlh6 SET SEARCH_PATH = websitedata;	Make sure that all commands will execute in this schema by default
CREATE TABLE customer (id INT, name TEXT); INSERT INTO customer VALUES (1, 'Marc Linster'); INSERT INTO customer VALUES (2, 'Jean Muller');	Create the table with some seed data

# Create Users and Groups

Step	Description
<pre>CREATE GROUP websiteeditor WITH     NOSUPERUSER     NOCREATEDB     NOCREATEROLE     INHERIT     NOLOGIN;</pre>	New group (i.e. role without login)
<pre>CREATE USER erika WITH     LOGIN     PASSWORD 'password'     IN ROLE websiteeditor;</pre>	New user, member of that role
<pre>CREATE GROUP websiteanalyst WITH     NOSUPERUSER     NOCREATEDB     NOCREATEROLE     INHERIT     NOLOGIN;</pre>	
<pre>CREATE USER jane WITH     LOGIN     PASSWORD 'password'     IN ROLE websiteanalyst;</pre>	

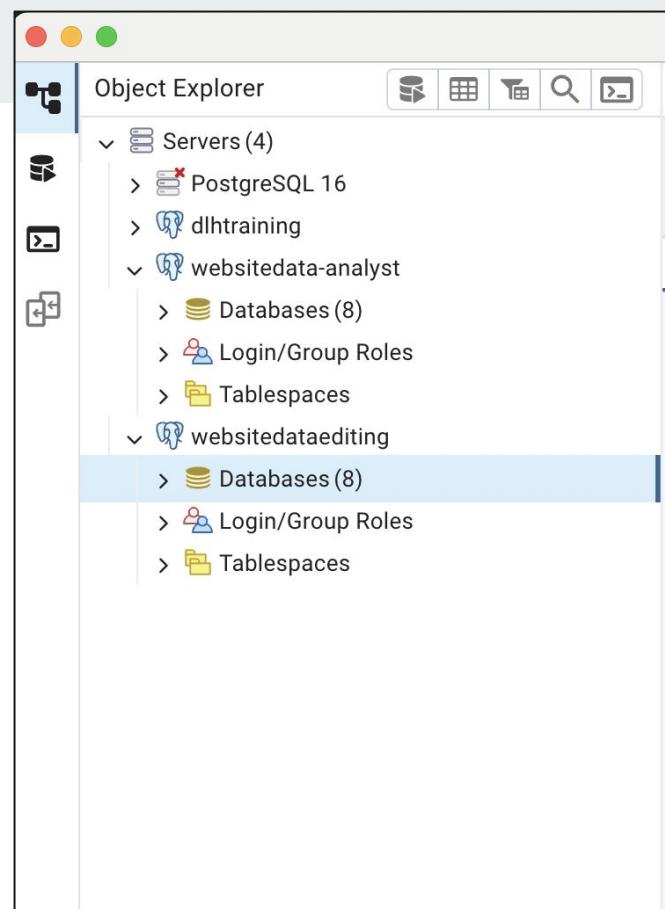
# Assign rights and access controls

---

Step	Description
GRANT CONNECT ON DATABASE "DLH6" TO websiteanalyst; GRANT CONNECT ON DATABASE "DLH6" TO websiteeditor;	Grant both groups the right to connect to the new database
GRANT USAGE ON SCHEMA websitedata TO websiteanalyst; GRANT USAGE ON SCHEMA websitedata TO websiteeditor;	Grant both groups the right to access the new schema
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA websitedata TO websiteeditor;	Grant the editor group (role) read/write access to the data
GRANT SELECT ON TABLE customer TO websiteanalyst;	Grant the analyst group (role) read-only access to the data

# Try it out

- Create a new server in pgAdmin (e.g., a new database connection) to DLH6 as user Jane
  - Call it 'websitedataanalysis'. Fill in the correct connection parameters
  - Query the customer table
  - Try to insert or update
- Create a new server (e.g., a new database connection) to DLH6 as user Erika
  - Call it 'websitedataediting'. Fill in the correct connection parameters
  - Query the customer table
  - Try to insert or update
  - Try to create a new table





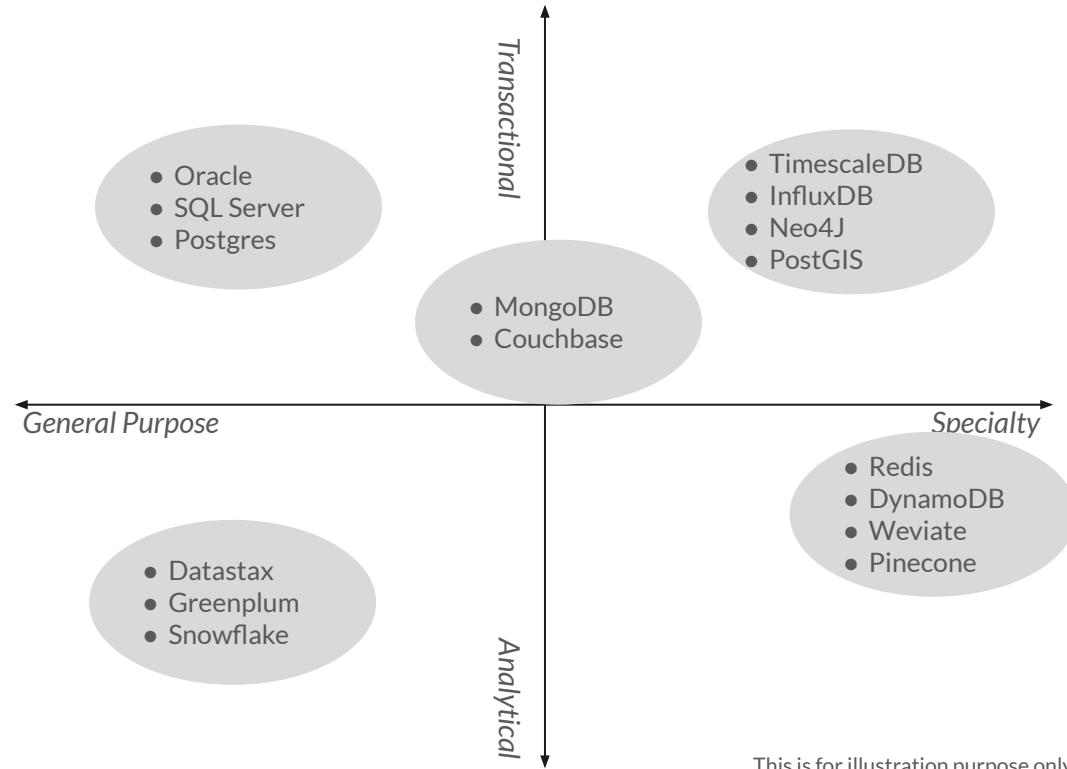
## Exercise

- Create a new database experiment with schema shopping
- Add a table
- Make this table editable for Erika and readable for Jane

---

# Recap

# Database Landscape



This is for illustration purpose only  
There are many, many databases and many ways to classify them

# Table Basics

- Rows = records (a.k.a. tuples in Postgres)
- Columns = data elements
- Example: mycustomer table
  - ID = a way to uniquely identify a customer ⇒ usually a large number
  - Email = the email address, a text field, sometimes limited in length
  - First name ⇒ a text field, sometimes limited in length
  - Last name ⇒ a text field, sometimes limited in length
  - Since => a date field

```
CREATE TABLE mycustomer (
    id INTEGER,
    email TEXT,
    first_name TEXT,
    last_name TEXT,
    since DATE
);
```

```
postgres=# SELECT * FROM mycustomer;
 id |          email           | first_name | last_name |      since
----+-----+-----+-----+-----+
  1 | marc@marclinster.com | Marc        | Linster   | 2025-01-01
  2 | jmu@gmail.lu       | Jeff        | Mueller   | 2025-01-02
  3 | bini@hotmail.lu    | Jean        | Bintner   | 2025-01-03
```

---

# Query types

- SELECT: List data from tables that meet certain conditions
- UPDATE: Change data in records that meet certain conditions
- INSERT: Add records
- DELETE: Remove Records

# Building Blocks of the simple SELECT Query

- **SELECT**
  - Columns to be selected, maybe using AS to provide a meaningful name
  - For example `last_name AS ln`
- **FROM**
  - Table(s) from which to select. Can also use AS to provide a new name
  - For example `customer AS c`
- **WHERE**
  - Filter clauses to select a subset of records
  - Can use the new names from the AS clause
  - For example `c.ln LIKE 'Li%`
- **ORDER BY**
  - Organizes the output of the SELECT clause
  - Can use the new names from the AS clause
  - For example: `c.ln ASC, since DESC`

## Alias (AS)

- improves readability
- Only available in the context of this query
- Key word 'AS' is optional

More complex building blocks for SELECT queries (JOIN, GROUP BY, HAVING, GROUPING, WITH, ...) will be introduced later

---

# Postgres Operators in the WHERE Clause

=	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
<>	Not equal to
!=	Not equal to
LIKE	Check if a value matches a pattern (case sensitive)
ILIKE	Check if a value matches a pattern (case insensitive)
AND	Logical AND
OR	Logical OR
IN	Check if a value is between a range of values
BETWEEN	Check if a value is between a range of values
IS NULL	Check if a value is NULL
NOT	Makes a negative result e.g. NOT LIKE, NOT IN, NOT BETWEEN

# Postgres functions in the SELECT clause

Some of the key string functions

	String append	'a'  '_'  'b' → a_b
lpad/rpad	Create a padded string	rpad('hi', 5, 'xy') → hixyx
substring	Create a substring	substring('Thomas' from 2 for 3) → hom
::string	Cast a number (and other types) as string	1::TEXT  '_'  2::TEXT → 1_2

Many, many more in the Postgres documentation: <https://www.postgresql.org/docs/current/functions-string.html>

---

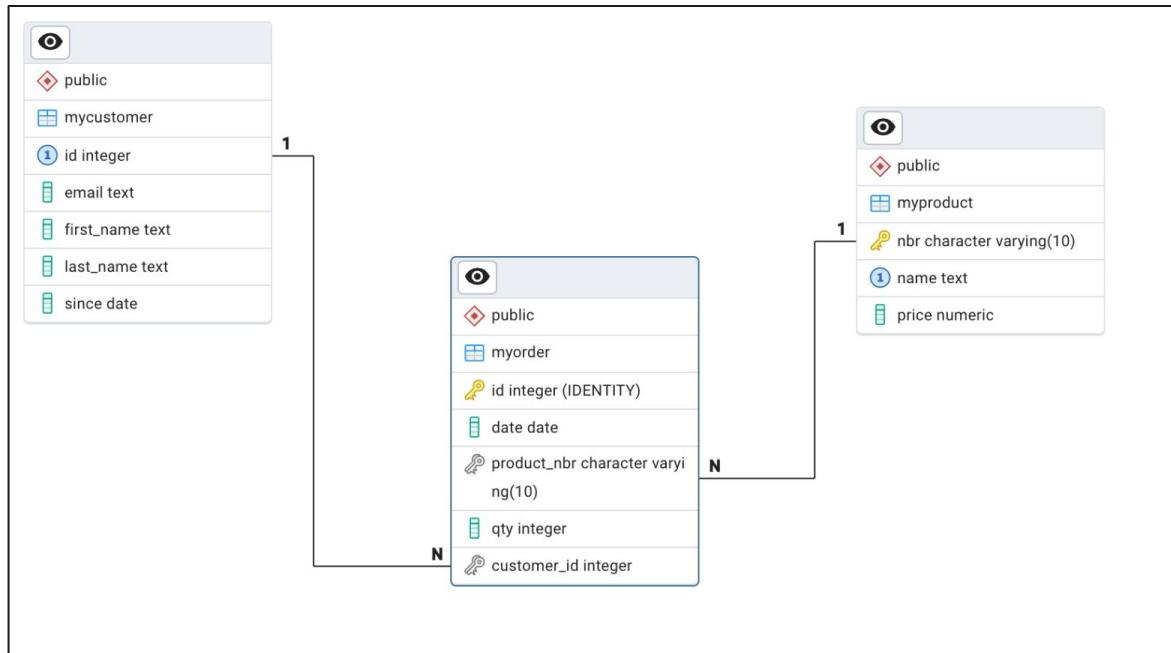
# Constraints and Relationships

- **Check Constraints:** The value in the column has to meet certain criteria, e.g.,  $\text{price} > 0$
- **Not-Null Constraints:** The column needs to have a value
- **Unique Constraints:** No two records have the same value
- **Primary Keys:** like the Unique Constraint, but excludes NULL and always creates an index
  - ⇒ A primary key can also be a combination of columns
- **Foreign Keys:** the data needs to be a valid reference to another table
  - You can assign a 'trigger' to handle deletions

# Schemas

Graphical view of the schema

- Tables
- Columns
- Constraints
- Relationships
  - 1:1
  - 1:N
  - N:M





## 3rd Normal Form

<u>Title</u>	<u>Author</u>	<u>Pages</u>	<u>Thickness</u>	<u>Publisher</u>	<u>Genre ID</u>
Beginning MySQL Database Design and Optimization	Chad Russell	520	Thick	Apress	1
The Relational Model for Database Management: Version 2	E.F.Codd	538	Thick	Addison-Wesley	2

<u>Title</u>	<u>Format</u>	<u>Price</u>
Beginning MySQL Database Design and Optimization	Hardcover	49.99
Beginning MySQL Database Design and Optimization	E-book	22.34
The Relational Model for Database Management: Version 2	E-book	13.88
The Relational Model for Database Management: Version 2	Paperback	39.99

<u>Publisher</u>	<u>Country</u>
Apress	USA
Addison-Wesley	USA

<u>Genre ID</u>	<u>Name</u>
1	Tutorial
2	Popular science

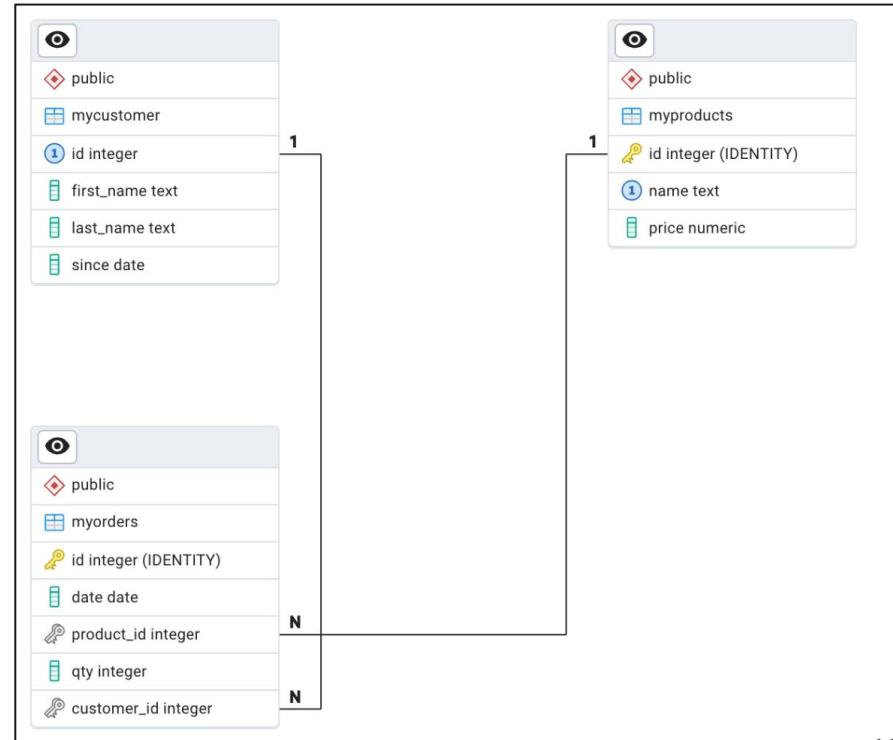
<u>Author</u>	<u>Author Nationality</u>
Chad Russell	American
E.F.Codd	British

---

# 'JOIN ... ON ...' connects Two or more tables

Types of joins:

- (INNER) JOIN
- LEFT/RIGHT (OUTER) JOIN
- FULL (OUTER) JOIN
- CROSS JOIN



# CREATE VIEW

Views are named queries  
that are stored in the  
database server

```
CREATE VIEW customer_purchase_view AS
  SELECT
    c.id AS customer_id,
    c.first_name || ' ' || c.last_name AS
customer_name,
    c.town,
    p.product_nbr,
    pr.name AS product_name,
    pr.category as product_category,
    p.quantity,
    p.id AS purchase_id,
    p.order_date,
    (p.quantity * pr.price) AS total_price
  FROM customer c
  JOIN purchase p ON c.id = p.customer_id
  JOIN product pr ON p.product_nbr = pr.product_nbr;
```

# SQL Aggregates

- GROUP BY, e.g., group the customers by location
  - GROUP BY ... HAVING allows to limit the groups
  - GROUP BY GROUPING SETS (...) creates sub totals
- SUM, AVG, MIN, MAX, e.g, the total/ average/ minimum/ maximum within a group

} After the WHERE clause

} Part of the SELECT clause

---

# Many types of Indexes



- Balanced Tree (B-Tree)
  - Every database has it. The most common and used for 90%+ of all cases. The default in Postgres
  - Ideal for looking up unique values and maintaining unique indexes
  - High concurrency implementation - very fast
- Block-Range Index (BRIN)
  - Ideal for very large datasets that are organized according to the index
- Generalized Inverted Index (GIN)
  - Best for indexing values with many keys or values, e.g.,text documents, JSON, multi-dimensional data, arrays
  - Ideal for columns containing many duplicate
- Hash Index
  - Equality, non-equality lookups; no range lookups
- Generalized Search Tree (GiST)
  - Used for GIS, key-value store
- Space-Partitioned Generalized Search Tree (SP-GIST)
  - Specialized index



# Transactions

- Sample Transaction

```
BEGIN
```

```
    Remove Euro 100 from Marc's account
```

```
    Add Euro 100 to 'Caves Wengler' account
```

```
COMMIT
```

- Key Considerations for a Database:

- **Atomic:** All or nothing
- **Consistent:** After the transaction both accounts are updated immediately and all constraints are validated (e.g., account > 0). Every user always see a consistent picture
- **Isolated:** Multiple transactions can happen at the same time. They either see the state before the transaction, or after the transaction - never in-between
- **Durable:** Once a transaction has completed, the result remains true.



# Summary: Keys to Writing Good Queries

1. Limit the number of rows that are returned
2. Limit the number of columns that are returned
3. Leverage indexes where available
4. Define aliases in the SELECT clause using 'AS' to avoid duplicate column names
5. Use the aliases in the WHERE and GROUP clause to avoid typos and shorten the code
6. Define highly selective WHERE clauses, especially when using JOINS
7. Avoid CROSS JOINS (data set explodes!)
8. Use VIEWS to standardize complex, often repeated queries (e.g., sales\_transaction combined with sales\_transaction\_line)
9. Use snake\_case names for tables and columns
10. Study EXPLAIN plans to understand where complexity occurs

---

## FYI: SQL Order of Execution

1. **FROM/JOIN**: Specifies the tables from which to retrieve data.
2. **WHERE**: Filters the rows that meet the condition before grouping.
3. **GROUP BY**: Groups rows that share a property.
4. **HAVING**: Filters groups based on conditions, applied after grouping.
5. **SELECT**: Specifies the columns to retrieve or calculate.
6. **DISTINCT**: Removes duplicate rows from the result set.
7. **ORDER BY**: Sorts the result set by specified columns.
8. **LIMIT**: Specifies the maximum number of rows to return.
9. **OFFSET**: Specifies how many rows to skip before starting to return rows.

---

# Resources

---

# Resources

## Books:

- PostgreSQL 16 Administration Cookbook; Gianni Ciolfi, Boris Mejias, Jimmy Angelakos, Vibhor Kumar, Simon Riggs, 2023
- PostgreSQL Up and Running; Regina Obe and Leo Hsu, 2020
- Introduction to PostgreSQL for the data professional; Ryan Booz and Grant Fritchey 2025 (coming from SQL Server)
- Mastering PostgreSQL 17: Elevate your database skills with advanced deployment, optimization, and security strategies 6th ed. Hans-Jürgen Schönig, 2023

## Blogs & online:

- Postgres documentation: <https://www.postgresql.org/docs/current/index.html>
- Neon's Postgres Tutorial: <https://neon.tech/postgresql/tutorial>
- Bruce Momjian's blog about SQL and Postgres: <https://momjian.us/main/presentations/sql.html>
- Bruce Momjian's blog about Postgres performance: <https://momjian.us/main/presentations/performance.html>
- Laetitia Avrot's excellent blog about DBA topics: <https://mydbanotebook.org/>
- Vibhor Kumar's blog: <https://vibhorkumar.wordpress.com/>
- Stormatics' blog: <https://stormatics.tech/our-blogs>
- pgAdmin Website (with videos): <https://www.pgadmin.org/>

