Digital Career Institute

Python Course - Database - Basic Usage

Digital Career Institute

Goal of the Submodule

The goal of this submodule is to help the learners use databases in Python. By the end of this submodule, the learners should be able to understand how to:

- Connect to a PostgreSQL database using the terminal and a GUI like DBeaver.
- Create, modify, delete and populate tables with SQL.
- Work with basic column data types.
- Query database records.
- Create relationships between tables and perform simple queries on multiple tables.
- Define views.

Topics

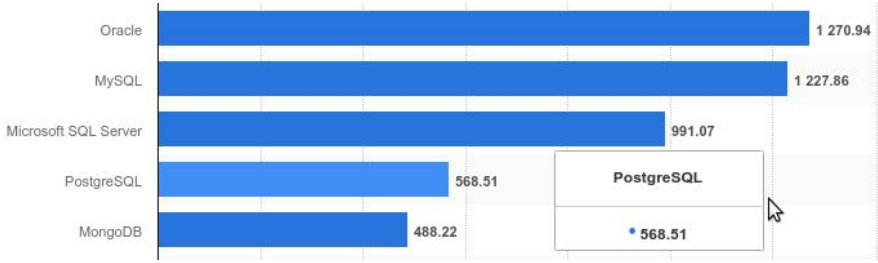
- Set up and connect to PostgreSQL
- Use DBeaver to work with PostgreSQL
- Explore the database structure
- Introduction to SQL
  - The Data Definition Language
  - The Data Manipulation Language
- Basic column data types
- Table relationships
  - Primary and foreign keys
  - Querying multiple tables
- Views

PostgreSQL

Introduction to PostgreSQL

PostgreSQL is a Relational Database Management System (RDBMS).

Rank			DBMS
Oct 2021	Sep 2021	Oct 2020	
1.	1.	1.	Oracle
2.	2.	2.	MySQL
3.	3.	3.	Microsoft SQL Server
4.	4.	4.	PostgreSQL
5.	5.	5.	MongoDB



It is the 4th most used database in the world
and the 2nd most used **open source** database.

Introduction to PostgreSQL

PostgreSQL is a RDBMS with pedigree

It is a descendant of **Postgres**,

... which evolved from Ingres (as in **Post** In**gres**),

... which was the first ever software implementation of the **Relational Model**,

... which was **introduced in 1970 by Edgar F. Codd** in his seminar paper “A Relational Model of Data for Large Shared Data Banks”,

... and has become **the most widely used data model**.

Introduction to PostgreSQL

PostgreSQL is a server

It runs on the background.

To interact with it, the user has to connect to the server and use a set of instructions.

It can hold many databases.

Graphical User Interface

Using a GUI

A **Graphical User Interface** (GUI) is another means of interacting with the database that does not require knowing the language and commands of the software.

A GUI is a software that often uses graphical means (like windows, menus and panels) in a visually attractive way and bases much of its user interaction on mouse click (or touch tap) actions.

A **GUI** is a computer program.

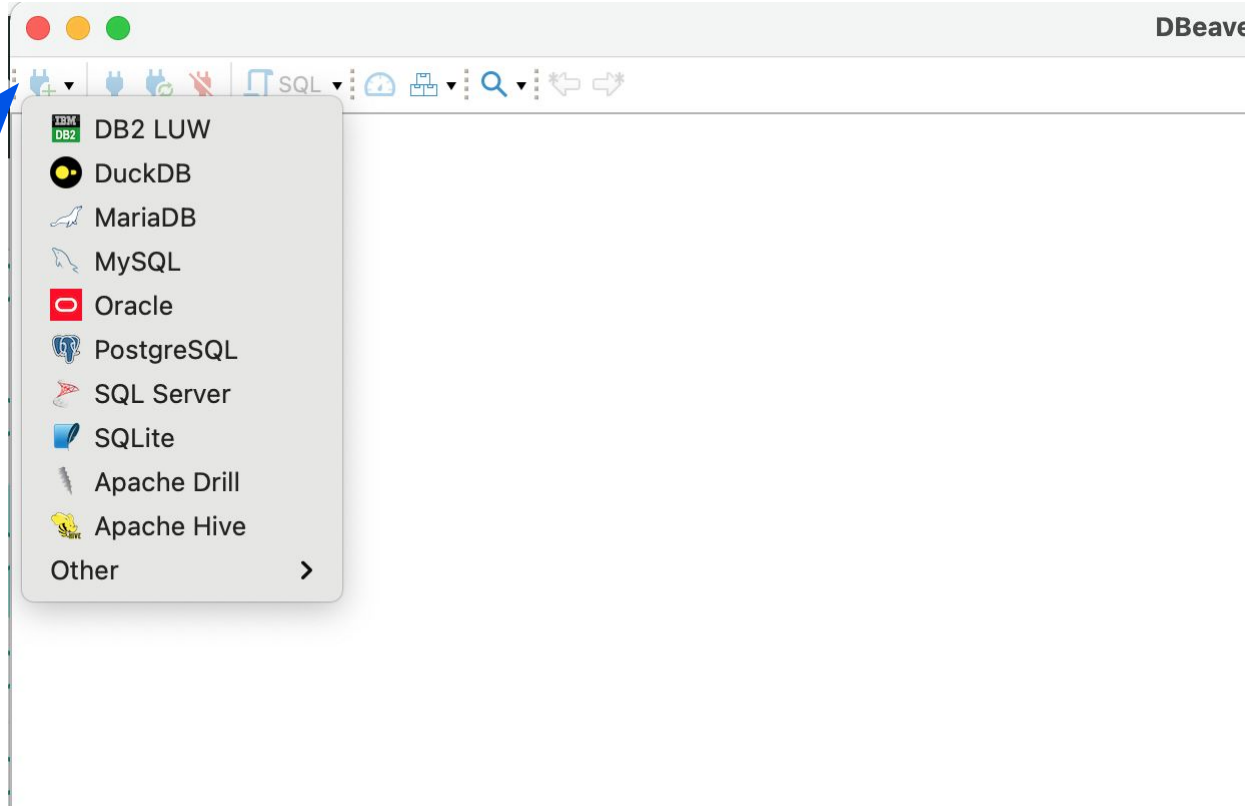
PostgreSQL GUI

There are different software applications that can be used to interact with a PostgreSQL server.

This section will show how to use **DBeaver**.

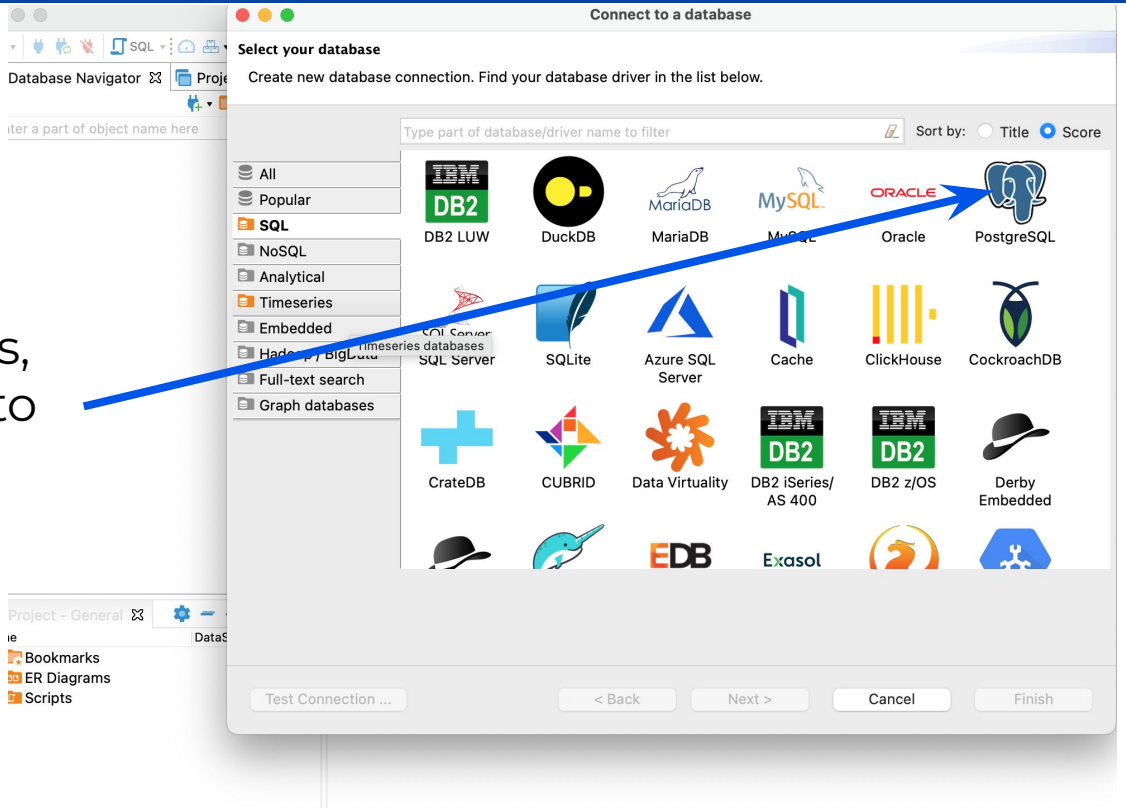
DBeaver: Creating a Connection

The first icon starts a new connection to a database server.



DBeaver: Connecting to PostgreSQL

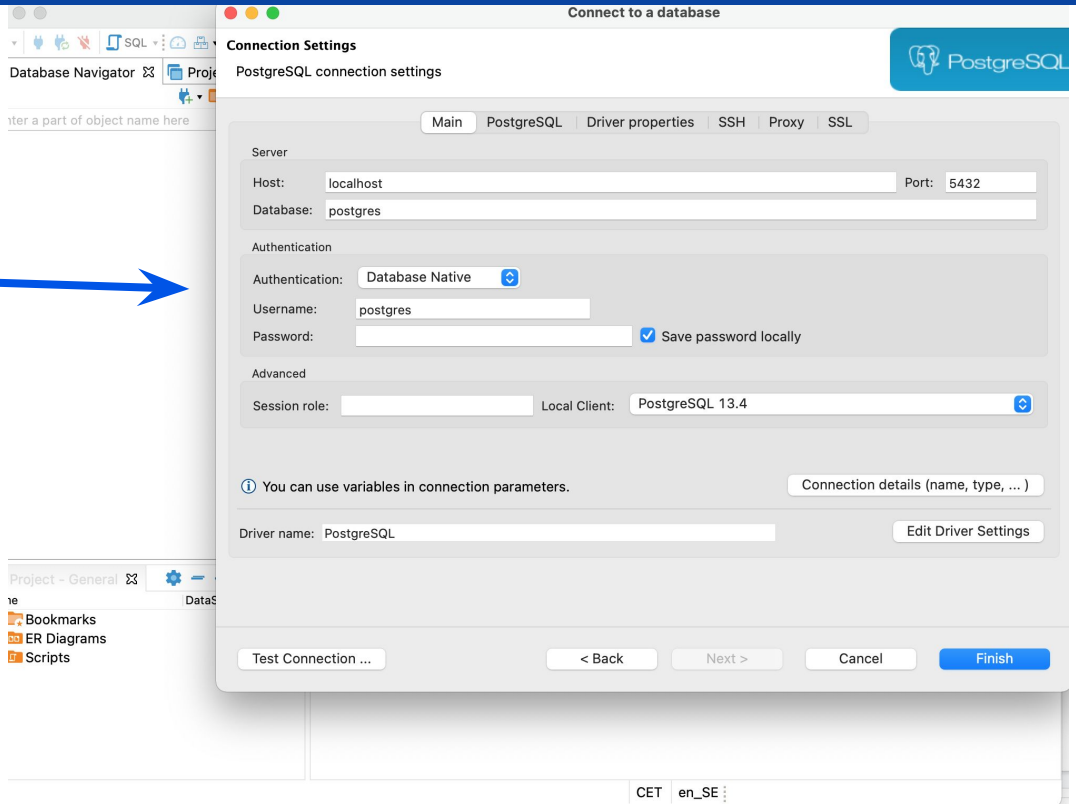
Amongst other options, DBeaver can connect to PostgreSQL.



DBeaver: Configuring PostgreSQL

The default values on the **`Main`** tab are usually enough.

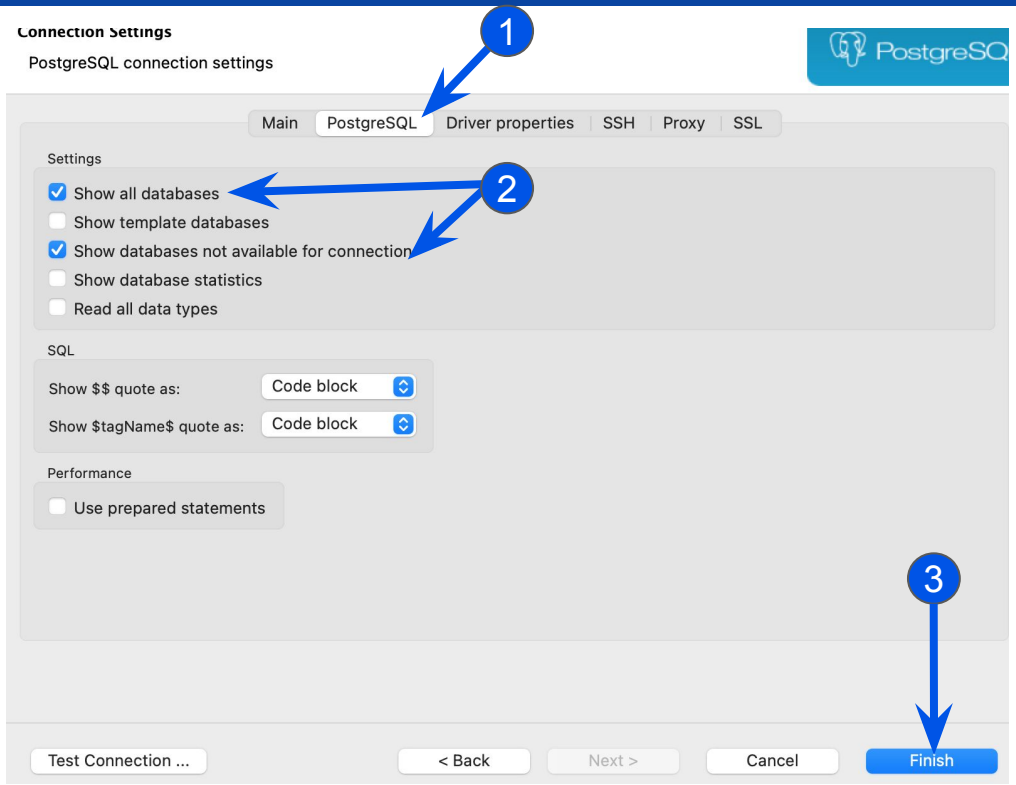
Different options can be used to connect to other servers in the network or use different users.



DBeaver - Configuring PostgreSQL

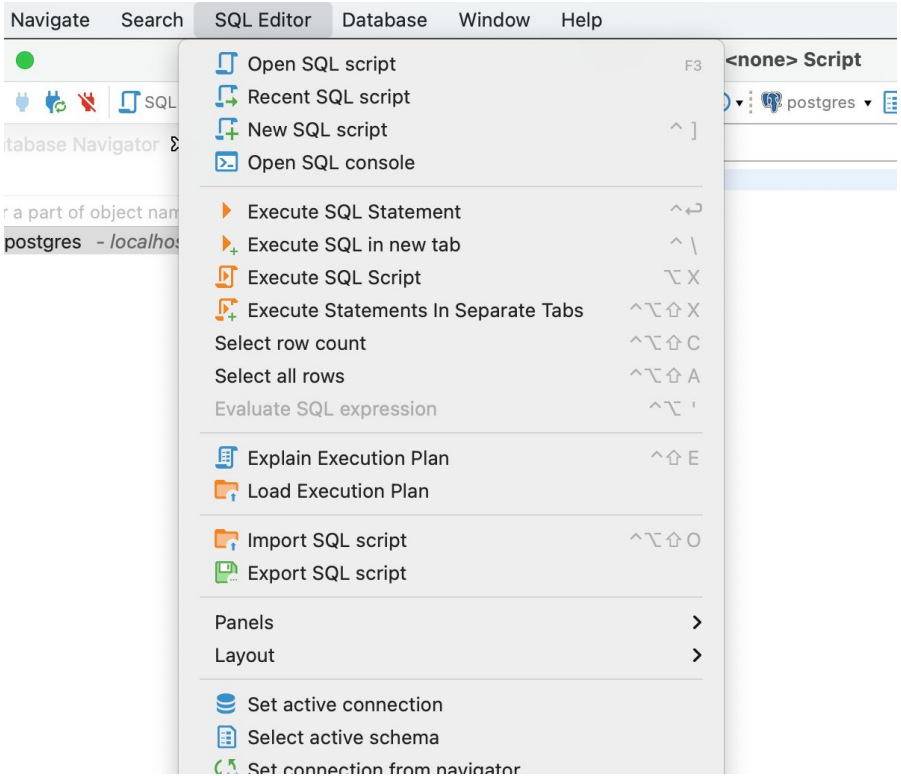
By default PostgreSQL defines a default database and DBeaver will display only the default database.

On the **`PostgreSQL`** tab, and to see all the databases, these two **`settings`** should be checked before selecting **`Finish`**.



DBeaver: Using the SQL Editor

In the menu bar, the **`SQL Editor`** menu includes options to open up a console, create SQL scripts or define the active connection.

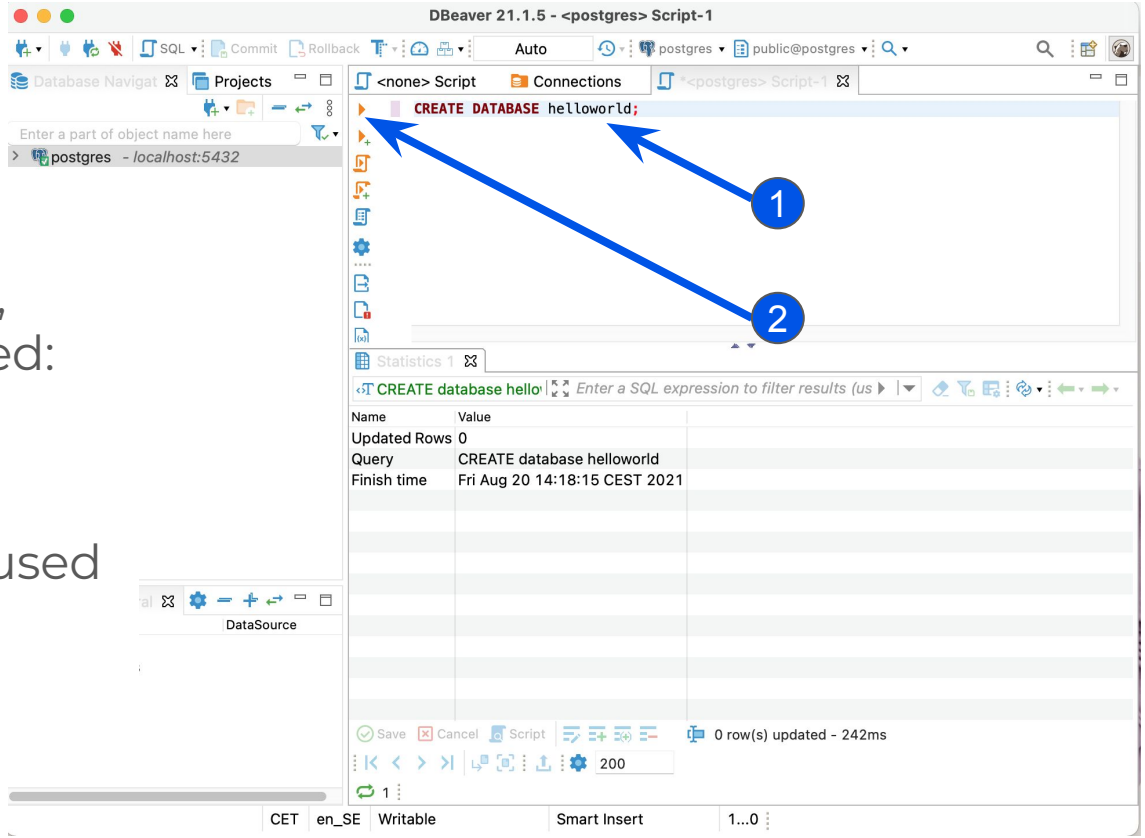


DBeaver: Executing Commands

1. In a new script window, commands can be typed:

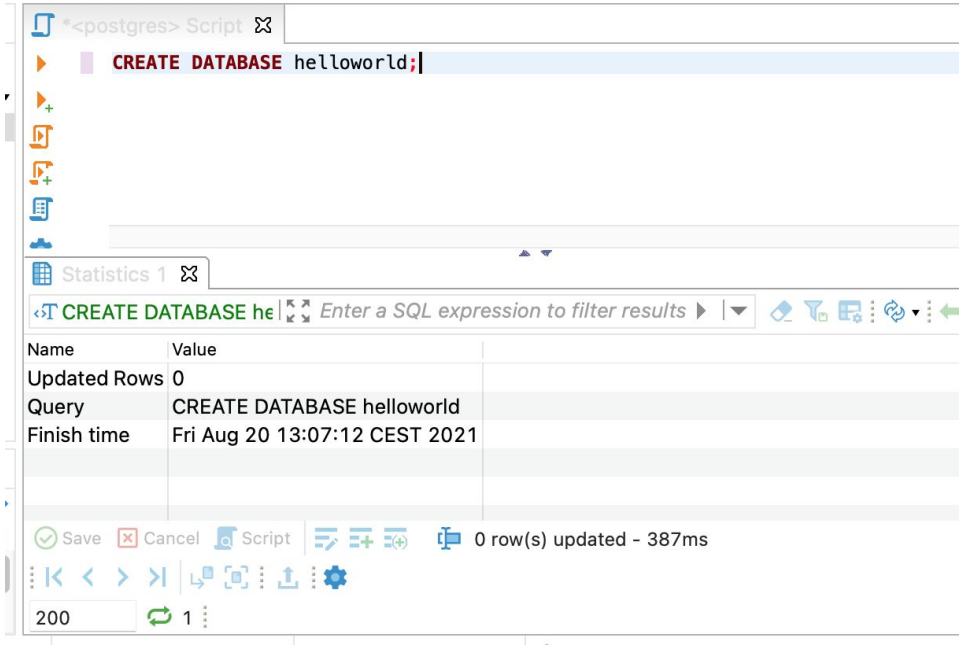
```
CREATE DATABASE helloworld;
```

2. The “play” icon can be used to execute the current script.

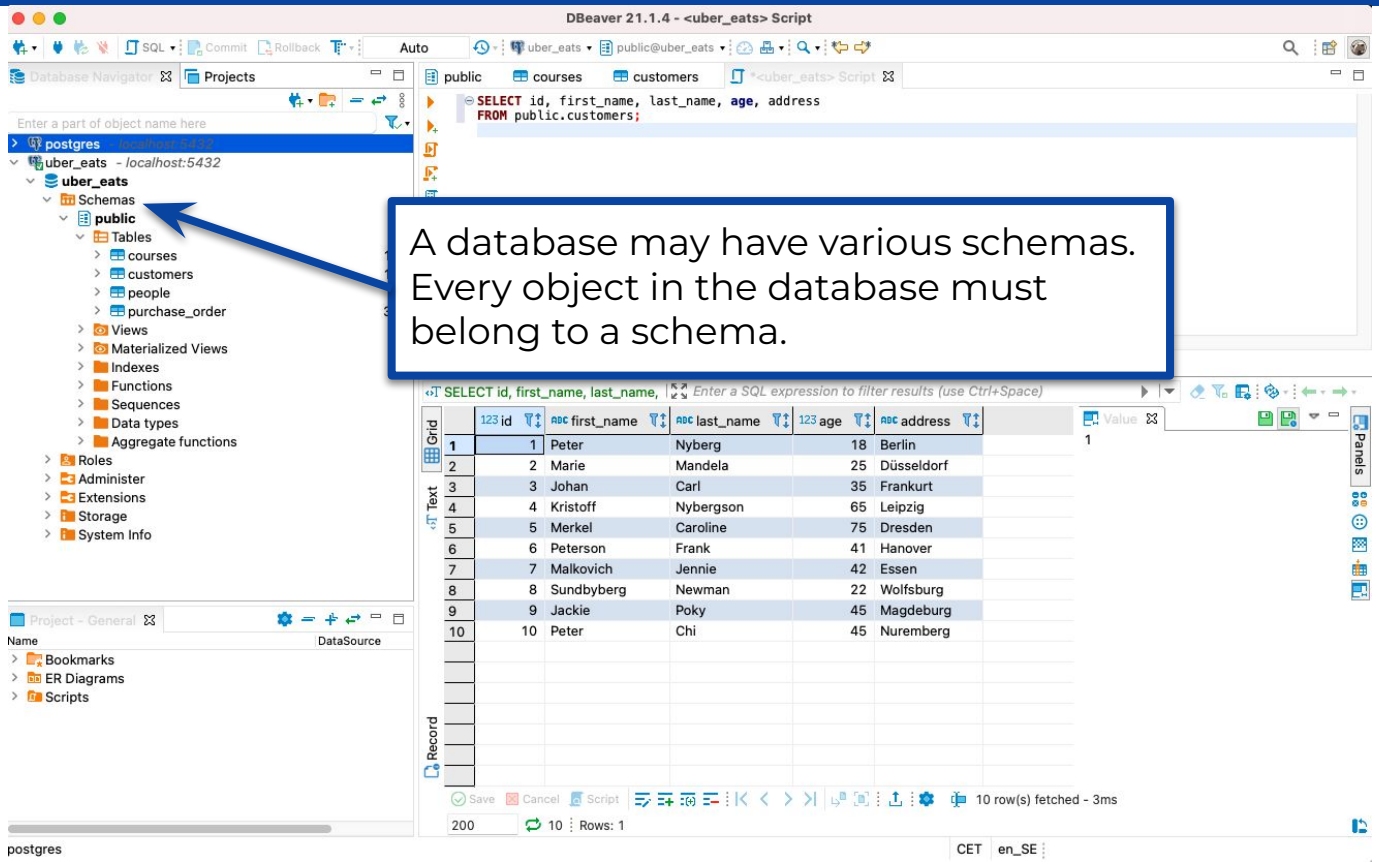


DBeaver: Results

The result of executing the instruction will appear below.



PostgreSQL: Schemas

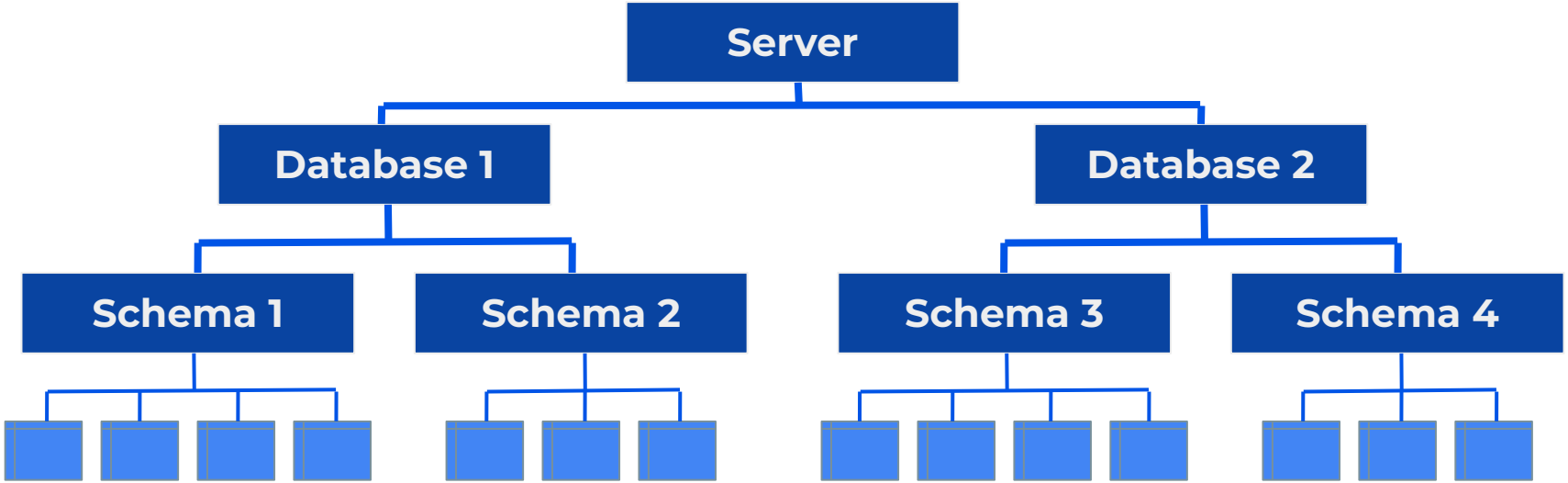


A database may have various schemas. Every object in the database must belong to a schema.

PostgreSQL Schemas

In many RDBMS, tables can be organized using one level of hierarchy: databases.

In PostgreSQL, two levels of hierarchy are used: databases and schemas.



Command-Line Interface

Introduction to CLI

Connecting to the PostgreSQL server can also be done using the command-line interface (CLI) of the terminal and typing:

```
$ psql
```

This will connect to the database server using the name of the current system user.

Alternatively, you can define a user with the `-U` parameter:

```
$ psql -U postgres
```

The PostgreSQL Console

Connecting to the PostgreSQL server logs the user into the **PostgreSQL console**.

```
$ psql -U postgres
psql (14.0)
Type "help" for help.

postgres=#
```

The PostgreSQL console is used to interact with the database using CLI commands and SQL.

List Databases

The `\l` command lists all databases.

```
postgres=# \l
```

```
                                   List of databases

           Name              |   Owner   | Encoding |   Collate   |    Ctype    |         Access privileges
-----------------------------+-----------+----------+-------------+-------------+-----------------------------------
 DCI                         | postgres  | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 uber_eats                   | postgres  | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 course_project              | postgres  | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 my_notes                    | postgres  | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
```

Connect to a Database

The `\c` command opens a connection to a specific database.

```
postgres=# \c uber_eats
```

```
psql (14.0)
You are now connected to database "uber_eats" as user "postgres".
uber_eats=#
```

Display all Schemas in a Database

The `\dn` command shows the list of schemas in the active database.

```
postgres=# \dn
```

```
  List of schemas
  Name   |  Owner
--------+----------
 public | postgres
(1 row)
```

Display all Objects in a Database

The `\d` command displays all objects in the current database.

```
uber_eats=# \d
```

```
                List of relations
 Schema |        Name         |   Type    |  Owner
--------+---------------------+-----------+----------
 public | customers           | table     | postgres
 public | customers_id_seq    | sequence  | postgres
 public | courses             | table     | postgres
```

Display Tables in a Database

The `\dt` command displays only the tables in the current database.

```
uber_eats=# \dt
```

```
                List of relations
 Schema |        Name         |   Type    |  Owner
--------+---------------------+-----------+----------
 public | customers           | table     | postgres
 public | courses             | table     | postgres
```

Summary of a Table

The `\d table_name` command describes a table.

```
uber_eats=# \d customers
```

```
                Table "public.customers"
 Column     |        Type              | Collation | Nullable | Default
------------+--------------------------+-----------+----------+---------
 id         | integer                  |           |          |
 first_name | character varying(255)   |           |          |
 last_name  | character varying(255)   |           |          |
 age        | integer                  |           |          |
 address    | character varying(255)   |           |          |
```

Full Description of a Table

The `\d+ table_name` will show a full description of the table.

```
uber_eats=# \d+ customers
```

```
 ...
Indexes:
    "friends_pkey" PRIMARY KEY, btree (id)
Referenced by:
    TABLE "message" CONSTRAINT "message_friend_id_fkey" FOREIGN KEY (friend_id)
REFERENCES friends(id) ON DELETE CASCADE
```

Executing Scripts

The `\i file_name.sql` command executes a file with instructions.

```
uber_eats=# \i file.sql
```

Can also be done outside the PostgreSQL console.

```
$ psql < file.sql
```

Console Commands

Using the **up arrow** and **down arrow** keys in the keyboard will loop through the instructions used during the current session.

Using the **tab** key will finish the command when we type part of that command. If multiple options are possible, it will show all options.

We learned ...

- How to use a graphical user interface like DBeaver to connect to PostgreSQL and execute database commands.

- How to use the terminal to open up a PostgreSQL console to list and connect to databases, display all tables and other objects and execute SQL scripts.

- Our first command: `CREATE DATABASE`.

SQL

sometimes pronounced ***sequel***

Why is SQL sexy?

PostgreSQL

MySQL

Microsoft
SQL Server

NoSQL

SQLite

NewSQL

Why is SQL sexy?

- The **Relational Model** is the set of concepts and principles behind relational databases.
- Edgar F. Codd defined 10 rules a RDBMS should follow.
- The rule number 5 states that "***A single language*** *must be able to define data, views, integrity constraints, authorization, transactions, and data manipulation.*".

This language is a standard and is called **Structured Query Language** (SQL).

One Single Language for Everything

One Language to Rule Them All

Definition

Manipulation

Query

Control

SQL “sub-languages”

SQL Categories

- **Data Definition Language (DDL)**
  Define and modify the database structure (tables, fields, relations, constraints,...).

- **Data Manipulation Language (DML)**
  Manage the data in the database (insert, update, delete).

- **Data Query Language (DQL)**
  Analyze and extract information from the data.

- Data Control Language (DCL)
  Define user access and privileges on the database objects.

SQL General Syntax

SQL is an **English-like** language

```
SELECT phone FROM friends WHERE name = 'Lisa';
```

An **SQL statement** starts with an **SQL command**. Commands look like verbs.

An SQL command is followed by an object's name in the database.

SQL General Syntax

SQL is an **English-like** language

```
SELECT phone FROM friends WHERE name = 'Lisa';
```

Each command may allow additional clauses that are often particular to that command.

The SQL statement must end with a semicolon.

SQL General Syntax

Comments in SQL are defined with --.

```
-- These first two lines are just comments,
-- they do not get executed. The next line does.
SELECT phone FROM friends WHERE name = 'Lisa';
```

SQL Categories & Commands

DDL

CREATE DATABASE,
DROP DATABASE,
CREATE TABLE,
ALTER TABLE,
DROP TABLE

DQL

SELECT

DML

INSERT,
UPDATE,
DELETE,
TRUNCATE

DCL

GRANT,
REVOKE

Data Definition Language

DDL Commands

The most common DDL commands are used to:

- `CREATE` databases and tables.
- `ALTER` the `TABLE` definition.
- `DROP` databases and tables.

Create a Database

```
CREATE DATABASE personal;
```

```
                                                List of databases

            Name              |   Owner   | Encoding |   Collate   |    Ctype    |         Access privileges
-----------------------------+-----------+----------+-------------+-------------+-----------------------------------
 DCI                         | postgres  | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 uber_eats                   | postgres  | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 course_project              | postgres  | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 my_notes                    | postgres  | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
 personal                    | postgres  | UTF8     | en_US.UTF-8 | en_US.UTF-8 |
```

Connect to a Database

```
postgres=# \c personal
```

The server may hold multiple databases.

Two tables with the same name can be defined in two different databases.

To know which of the two tables is being accessed, an active connection to its database must be established before.

Connecting to a database is one of the few operations that cannot be done with SQL in PostgreSQL.

Create a Schema

```
CREATE SCHEMA private;
```

```
personal=# \dn
  List of schemas
  Name    |  Owner
---------+----------
 private | postgres
 public  | postgres
(2 rows)
```

Create a Table

```
CREATE TABLE private.friends (
  -- The columns will
  -- be defined here.
);
```

The most basic definition of a table consists of:

- a table name. May be preceded by the schema name. If not, the default schema is used.
- a list of columns, wrapped in parentheses.

Create a Table: Columns

```
CREATE TABLE private.friends (
  first_name    varchar(20),
  last_name     varchar(50),
  phone         varchar(12),
  age           integer
);
```

Column definitions must be separated using commas.

`varchar` indicates a character string of varying length. The length is indicated in parentheses.

Each column is defined with a name and a type, separated by a whitespace. The column name must not include whitespaces or special keywords or characters.

Create a Table: Proper Styling

```
CREATE TABLE private.friends(first_name varchar(20),last_name varchar(50));
```

```
CREATE TABLE private.friends (
  first_name      varchar(20),
  last_name       varchar(50),
);
```

Change a Table: Add a Column

```
ALTER TABLE friends
ADD [COLUMN] address varchar(255);
```

```
personal=# \d friends
                        Table "public.friends"
   Column    |          Type          | Collation | Nullable | Default
-------------+------------------------+-----------+----------+---------
 first_name  | character varying(20)  |           |          |
 last_name   | character varying(50)  |           |          |
 phone       | character varying(12)  |           |          |
 age         | integer                |           |          |
 address     | character varying(255) |           |          |
```

Change a Table: Rename a Column

```
ALTER TABLE friends
RENAME [COLUMN] address TO location;
```

```
personal=# \d friends
                        Table "public.friends"
   Column    |          Type          | Collation | Nullable | Default
-------------+------------------------+-----------+----------+---------
 first_name  | character varying(20)  |           |          |
 last_name   | character varying(50)  |           |          |
 phone       | character varying(12)  |           |          |
 age         | integer                |           |          |
 location    | character varying(255) |           |          |
```

Change a Table: Change a Column's Type

```
ALTER TABLE friends
ALTER [COLUMN] location TYPE int;
```

```
personal=# ALTER TABLE friends ALTER location TYPE int;
ERROR:  column "location" cannot be cast automatically to type integer
HINT:  You might need to specify "USING location::integer".
```

Changing the type will require changing the type of the values that may be stored in that column.

Change a Table: Change a Column's Type

```
ALTER TABLE friends
ALTER [COLUMN] location TYPE int
USING location::integer;
```

```
personal=# \d friends
                        Table "public.friends"
   Column    |          Type          | Collation | Nullable | Default
------------+------------------------+-----------+----------+---------
 first_name | character varying(20)  |           |          |
 last_name  | character varying(50)  |           |          |
 phone      | character varying(12)  |           |          |
 age        | integer                |           |          |
 location   | integer                |           |          |
```

Change a Table: Remove a Column

```
ALTER TABLE friends
DROP [COLUMN] location;
```

```
personal=# \d friends
                        Table "public.friends"
   Column    |          Type          | Collation | Nullable | Default
------------+------------------------+-----------+----------+---------
 first_name | character varying(20)  |           |          |
 last_name  | character varying(50)  |           |          |
 phone      | character varying(12)  |           |          |
 age        | integer                |           |          |
```

Remove a Table

```
DROP TABLE friends;
```

Remove a Database

```
DROP DATABASE personal;
```

```
personal=# DROP DATABASE personal;
ERROR:  cannot drop the currently open database
personal=# \c postgres
postgres=# DROP DATABASE personal;
DROP DATABASE
```

Connecting to another database will release the lock on the database requiring deletion.

Remove Nonexistent Objects

```
ALTER TABLE friends DROP location;
DROP TABLE friends;
DROP DATABASE personal;
```

```
postgres=# ALTER TABLE friends DROP location;
ERROR:  column "location" of relation "friends" does not exist
postgres=# DROP TABLE friends;
ERROR:  table "friends" does not exist
postgres=# DROP DATABASE personal;
ERROR:  database "personal" does not exist
```

This is not a problem in this case, when using the statements once. But if this is part of a script, it will break the execution.

Remove Objects Only if they Exist

```
ALTER TABLE friends DROP IF EXISTS location;
DROP TABLE IF EXISTS friends;
DROP DATABASE IF EXISTS personal;
```

```
personal=# ALTER TABLE friends DROP IF EXISTS location;
NOTICE:  column "location" of relation "friends" does not exist, skipping
ALTER TABLE
personal=# DROP TABLE IF EXISTS friends;
NOTICE:  table "friends" does not exist, skipping
DROP TABLE
postgres=# DROP DATABASE IF EXISTS personal;
NOTICE:  database "personal" does not exist, skipping
DROP DATABASE
```

Data Manipulation Language

DML (& DQL) Commands

The most common DML (& DQL) commands are:

- **`INSERT`** to add data (DML).
- **`SELECT`** to retrieve data (DQL).
- **`UPDATE`** to change data (DML).
- **`DELETE`** to remove rows of data (DML).
- **`TRUNCATE`** to clear the table (DML).

Add Rows

Insert data in all fields.

```
INSERT INTO <table>
VALUES (<value1>, <value2>, <value3>, <value4>);
```

The values must be written in the same order as they were defined in the `CREATE TABLE` statement.

Add Rows

```
personal=# INSERT INTO friends
personal-# VALUES ('Lisa', 'Klepp', '916736453', 32);
INSERT 0 1
```

The values must be written in the same order as they were defined in the `CREATE TABLE` statement.

Add Rows

Insert data in some fields.

```
INSERT INTO <table>(<column2>, <column1>)
VALUES (<value2>, <value1>);
```

A different order may be specified in the first part of the statement.

If some fields allow NULL values, these can also be left out of the statement.

Add Rows

```
personal=# INSERT INTO friends(last_name, first_name)
personal-# VALUES ('Strum', 'Peter');
INSERT 0 1
```

The **phone** and **age** columns allow NULL values, so we can skip them.

Add Rows

Insert multiple rows.

```
INSERT INTO <table>(<column2>, <column1>)
VALUES (<value2.1>, <value1.1>),
       (<value2.2>, <value1.2>);
```

Multiple rows can be inserted in one statement, by adding more data in the **`VALUES`** clause and separating them with commas.

Add Rows

Insert multiple rows.

```
personal=# INSERT INTO friends(last_name, first_name)
personal-# VALUES ('Strum', 'Peter'), ('Sullivan', 'Regina');
INSERT 0 2
```

The output of the insert statement will indicate how many rows have been inserted.

Retrieve Rows

Retrieve all rows.

```
SELECT <columns> FROM <table>;
```

The **`<columns>`** is a comma-separated enumeration of field names.

Instead of an enumeration of fields names, all fields can be retrieved by writing **`*`** as **`<columns>`**.

Retrieve Rows

Retrieve all rows.

```
personal=# SELECT * FROM friends;
 first_name | last_name |   phone    | age
------------+-----------+-----------+-----
 Lisa       | Klepp     | 916736453 |  32
 Peter      | Strum     |           |
(2 rows)
```

```
personal=# SELECT first_name, phone FROM friends;
 first_name |   phone
------------+-----------
 Lisa       | 916736453
 Peter      |
(2 rows)
```

Retrieve Rows

Retrieve only some rows.

```
SELECT <columns> FROM <table>
WHERE <condition>;
```

The columns used in the **`<condition>`** can be also in the **`<columns>`** list, but it is not necessary.

A **`<condition>`** is a logical expression, a combination of operands and operators that produce a Boolean result.

Logical operators, such as **`AND`** and **`OR`** can be used.

Retrieve Rows

Retrieve some rows.

```
personal=# SELECT * FROM friends WHERE age = 32;
 first_name | last_name |   phone    | age
------------+-----------+------------+-----
 Lisa       | Klepp     | 916736453  |  32
(1 row)
```

```
personal=# SELECT first_name FROM friends WHERE last_name = 'Strum';
 first_name
------------
 Peter
(a row)
```

Update Data

Update all rows.

```
UPDATE <table>
SET <column1> = <value1>, <column2> = <value2>;
```

The **`UPDATE`** command uses the **`SET`** clause to identify what data has to be changed.

Multiple columns can be updated at the same time, separating them with commas.

Update Data

Update all rows.

```
personal=# UPDATE friends SET age = 33;
UPDATE 2
personal=# SELECT * FROM friends;
 first_name | last_name |   phone    | age
------------+-----------+-----------+-----
 Lisa       | Klepp     | 916736453 |  33
 Peter      | Strum     |           |  33
(2 rows)
```

Update Data

Update only some rows.

```
UPDATE <table> SET <column1> = <new_value>
WHERE <condition>;
```

Just as with the **`SELECT`** command, the **`UPDATE`** also allows for row selection using the **`WHERE`** clause and a **`<condition>`**.

Update Data

Update some rows.

```
personal=# UPDATE friends
personal=# SET phone = 923451762, first_name = 'Pete'
personal=# WHERE first_name = 'Peter';
UPDATE 1
personal=# SELECT * FROM friends;
 first_name | last_name |   phone    | age
------------+-----------+-----------+-----
 Lisa       | Klepp     | 916736453 |  33
 Pete       | Strum     | 923451762 |  33
(2 rows)
```

Delete Data

Delete all rows.

```
DELETE FROM <table>;
```

The `DELETE FROM` command removes rows from a table.

Delete Data

Clear table data.

```
TRUNCATE <tables>;
```

The `TRUNCATE` command is similar to the command in the previous slide.

It can only clear entire tables, but it can clear multiple tables at once, separated by commas.

When removing all rows from a table, this is the preferred method.

Delete Data

Delete some rows.

```
DELETE FROM <table>
WHERE <condition>;
```

The **`TRUNCATE`** command does not allow removing specific rows in a table.

The **`<condition>`** in the **`WHERE`** clause of the **`DELETE FROM`** command can be used to do so.

Delete Data

```
personal=# DELETE FROM friends
personal=# WHERE first_name = 'Pete';
DELETE 1
personal=# SELECT * FROM friends;
 first_name | last_name |   phone    | age
------------+-----------+-----------+-----
 Lisa       | Klepp     | 916736453 |  33
(1 row)
```

Data Query Language

Column Distinct Values

```
SELECT DISTINCT <columns>
FROM <table>;
```

The **DISTINCT** clause of the **SELECT** command returns only the values that are different.

Column Distinct Values

```
personal=# SELECT age
personal=# FROM friends;
 age
-----
  33
  20
  41
  33
  33
(5 rows)
```

```
personal=# SELECT DISTINCT age
personal=# FROM friends;
 age
-----
  20
  33
  41
(3 rows)
```

Column Distinct Values

```
personal=# SELECT age, phone
personal=# FROM friends;
 age |   phone
-----+-----------
  33 | 916736453
  20 |
  41 |
  33 |
  33 |
(5 rows)
```

```
personal=# SELECT
personal=# DISTINCT age, phone
personal=# FROM friends;
 age |   phone
-----+-----------
  20 |
  33 | 916736453
  33 |
  41 |
(4 rows)
```

If multiple columns are used, the result shows the records with different values in both columns.

Column Aliases

```
SELECT <column1> AS <alias1>
FROM <table>;
```

A column name can be retrieved with a different name, using an alias.

An alias is just a change on what the user sees, the table column name remains the same.

Column Aliases

```
personal=# SELECT first_name AS "Name", last_name AS "Family name"
personal-# FROM friends;
 Name        | Family name
------------+-----------
 Lisa        | Klepp
(1 row)
```

Limit the Results

```
SELECT <columns> FROM <table>
LIMIT <number>;
```

The **`LIMIT`** clause can be used to limit the amount of results returned, to the indicated **`<number>`**.

Limit the Results

```
personal=# SELECT first_name
personal=# FROM friends;
 first_name
------------
 Lisa
 Maria
 Lidia
 James
 Karen
(5 rows)
```

```
personal=# SELECT first_name
personal=# FROM friends
personal=# LIMIT 3;
 first_name
------------
 Lisa
 Maria
 Lidia
(3 rows)
```

Limit the Results

```
SELECT <columns> FROM <table>
OFFSET <number>;
```

The **`OFFSET`** clause will omit the first **`<number>`** of rows in the output.

Limit the Results

```
personal=# SELECT first_name
personal=# FROM friends;
 first_name
------------
 Lisa
 Maria
 Lidia
 James
 Karen
(5 rows)
```

```
personal=# SELECT first_name
personal=# FROM friends
personal=# OFFSET 3;
 first_name
------------
 James
 Karen
(2 rows)
```

Sort the Results

```
SELECT <columns> FROM <table>
ORDER BY <column1> [ASC|DESC];
```

The **`ORDER BY`** clause can be used to sort the results.

An additional clause can be used to define the direction of the sorting: **`ASC`**ending or **`DESC`**ending.

If this clause is not define, it will be sorted ascendingly.

Sort the Results

```
personal=# SELECT age
personal=# FROM friends;
 age
-----
  33
  20
  41
  33
  33
(5 rows)
```

```
personal=# SELECT age
personal=# FROM friends
personal=# ORDER BY age;
 age
-----
  20
  33
  33
  33
  41
(5 rows)
```

Sort the Results

```
SELECT <columns> FROM <table>
ORDER BY
    <column1> [ASC|DESC], <column2> [ASC|DESC];
```

The output can be sorted using multiple criteria.

It will be sorted first using the first criteria.

Those records with identical value in the first column will be sorted using the second criteria.

Sort the Results

```
personal=# SELECT age, phone
personal=# FROM friends
personal=# ORDER BY age;
 age |   phone
-----+-----------
  20 |
  33 | 916736453
  33 |
  33 |
  41 |
(5 rows)
```

```
personal=# SELECT age
personal=# FROM friends
personal=# ORDER BY age, phone;
 age |   phone
-----+-----------
  20 |
  33 |
  33 |
  33 | 916736453
  41 |
(5 rows)
```

Combining Clauses: Paginating

```
SELECT <columns> FROM <table>
OFFSET (<page> - 1) * <size>
LIMIT <size>;
```

The **`OFFSET`** and **`LIMIT`** clauses are often used together to provide a pagination feature.

For a page size of 10 rows:

<page>	**OFFSET**	**LIMIT**
1	0	10
2	10	10
...	...	...

Combining Clauses: Rankings

```
SELECT <columns> FROM <table>
ORDER BY <column>
LIMIT <size>;
```

The **`ORDER BY`** and **`LIMIT`** clauses are often used together to retrieve the top **`<size>`** records based on **`<column>`**.

Combining Clauses: Rankings

```
personal=# SELECT first_name, age
personal=# FROM friends
personal=# ORDER BY age DESC
personal=# LIMIT 3;
 first_name | age
------------+-----
 Karen      |  41
 Lisa       |  31
 Lidia      |  32
(5 rows)
```

The **three oldest** friends in the database.

```
personal=# SELECT first_name, age
personal=# FROM friends
personal=# ORDER BY age
personal=# LIMIT 1;
 first_name | age
------------+-----
 Maria      |  20
(5 rows)
```

The **youngest** friend in the database.

Combining Clauses: Rankings

```
SELECT <columns> FROM <table>
ORDER BY <column>
LIMIT 1
OFFSET <rank>;
```

Together with the **`OFFSET`** clause, the combination can be used to retrieve a rank (the Nth position in a ranking).

Combining Clauses: Rankings

```
personal=# SELECT first_name, age
personal=# FROM friends
personal=# ORDER BY age
personal=# LIMIT 1;
personal=# OFFSET 1;
 first_name | age
------------+-----
 Lisa       |  31
(5 rows)
```

The **second youngest** friend in the database.

```
personal=# SELECT first_name, age
personal=# FROM friends
personal=# ORDER BY age
personal=# LIMIT 2;
personal=# OFFSET 2;
 first_name | age
------------+-----
 Lidia      |  32
 James      |  33
(5 rows)
```

The **third and fourth youngest** friends in the database.

Combining Clauses: Rankings

```
personal=# SELECT DISTINCT age
personal=# FROM friends
personal=# ORDER BY age
personal=# LIMIT 2;
 age
-----
  20
  33
(2 rows)
```

The two youngest **ages** among the friends in the database.

```
personal=# SELECT DISTINCT age
personal=# FROM friends
personal=# ORDER BY age
personal=# LIMIT 3;
 age
-----
  20
  33
  41
(3 rows)
```

The three youngest **ages** among the friends in the database.

SQL Logical Expressions

Logical Expressions

```
SELECT <columns> FROM <table>
WHERE <logical expression>;
```

Logical expressions can be used with various commands (`SELECT`, `UPDATE`, `DELETE`), often in the `WHERE` clause.

They behave similarly to Python logical expressions.

Logical Operators

Like Python, it has the basic operators implemented.

AND

OR

NOT

Comparison Operators

To compare a value with another one we can use:

<	>	=
<=	>=	<>

The operator `IN` can be used to match the equality in a list:

```
<column_name> IN ('value1', 'value2')
```

Comparison Operators

The operator **IN** can be used to compare the value in the column with a list of valid matches:

```
SELECT <columns> FROM <table>
WHERE <column> IN (<value1>, <value2>,...);
```

Comparison Operators

The operator **BETWEEN** can be used
to compare the value with a range:

```
SELECT <columns> FROM <table>
WHERE <column> BETWEEN <value1> AND <value2>;
```

Equivalent to:

```
SELECT <columns> FROM <table>
WHERE <column> >= <value1> AND column <= <value2>;
```

Comparison Operators

Text fields have an additional operator named **`LIKE`**, that is used to match against patterns.

The **`LIKE`** operator uses the **`%`** symbol that matches against any number of characters.

```
SELECT * FROM friends
WHERE last_name LIKE 'O%';
```

This example returns a list of friends whose last name starts with the letter O.

We learned ...

- What is SQL and how it works.
- How to create a new database and define its tables.
- How to modify the database structure and remove it.
- How to populate the database tables with data.
- How to manipulate and update the data.
- How to remove data and clear entire tables.
- How to query the data and extract information from the database.
- How to manage with SQL features such as pagination and rankings.

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute