Programming and Communications III: Databases + SQL + Tools
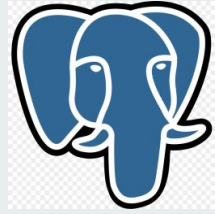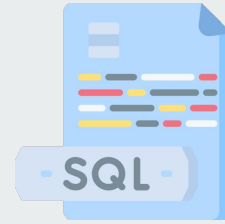
Jordi Ricard Onrubia Palacios

Departament d'Informàtica i Enginyeria Industrial Universitat de Lleida

Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

Programming and Communications III: Databases + SQL + Tools

Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

What Is a Database?

A database is any collection of data items. More technically we can describe it as a collection of integrated records.

A record is a representation of some physical or conceptual object. For example, A company storing the data of they employees. Each record will have multiple attributes, following the previous example, Name, Telephone, Directions, etc.

These records can be stored in different ways depending on the type of database we are using, if we use Relational Databases, specifically SQL Databases, we will use tables and create relations between them.

Database Table Example

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

Databases and the Relational Model

How relations work?

When we create a table we define some specific elements, one is an identifier or Primary Key, this column will be used for fast access and also for create relationships with other tables.

A primary key, is a type of column that is indexed, this means that is stored in way that enables the db to find them faster. We could think that is like a phonebook, where everyone inside is stored alphabetically.

The second one we can define is a Foreign Key, this column references a Primary Key from another table (not necessarily but recommended), this way we create a link between the two tables and establish a constrain the Primary Key from a table will be the same as the Foreign Key in another table.
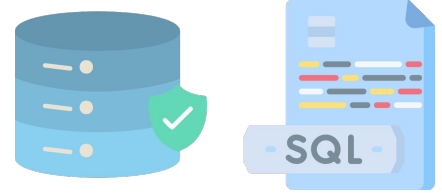
Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

Database Relations Example
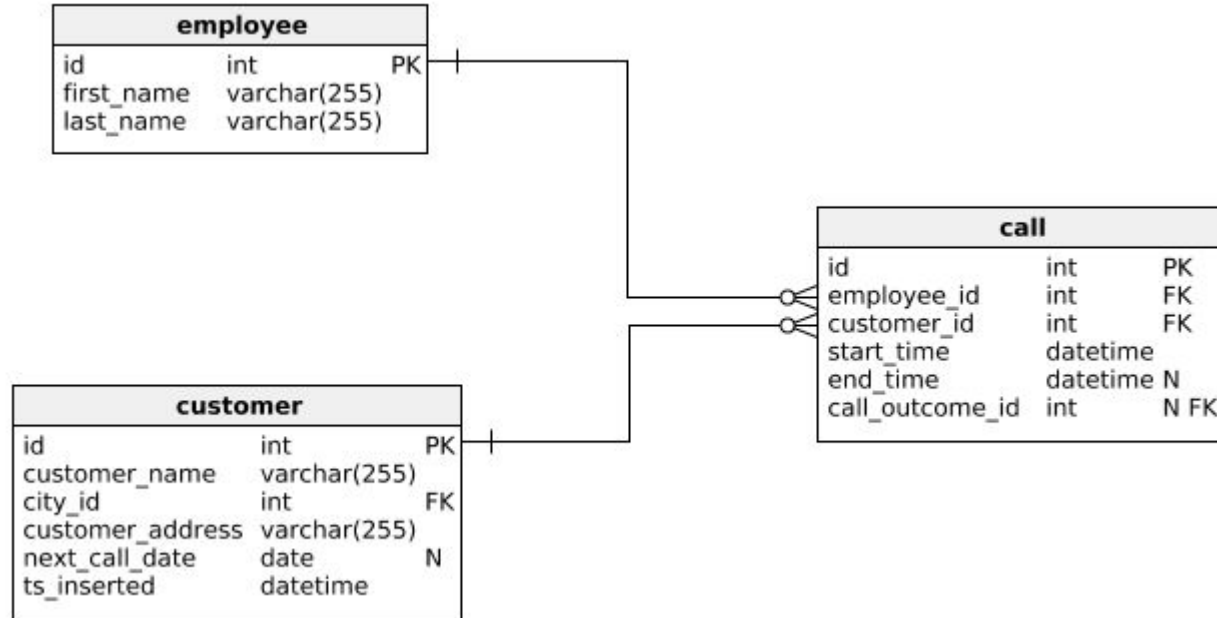
Databases and the Relational Model

So now, we know that in our relational db we are keeping our records inside tables, but, how do the database what records belong to which table?

Databases store metadata, data about data, in a an area called dictionary data, there the database will store all the data related to the tables, columns (previously named attributes), indexes, constraints an other items needed for a proper behaviour of our database.

What is SQL?
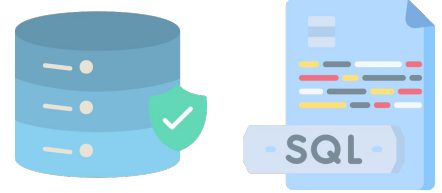
SQL stands for Structured Query Language is a domain-specific language used in programming and designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS).

 It is particularly useful in handling structured data for example in incorporating relations among entities and variables. SQL offers two main advantages.

 Firstly, it introduced the concept of accessing many records with one single command.

Secondly, it eliminates the need to specify how to reach a record, for example with or without an index.

SQL - Data Definition Language
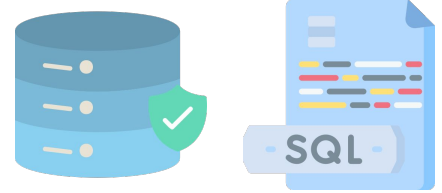
Data Definition Language (DDL) is in charge of describing what objects are going to be part of a database

- CREATE TABLE - creates a new table
- ALTER TABLE - modifies a table
- DROP TABLE - deletes a table

SQL - DDL


```
Table Commands SQL

CREATE TABLE <table_name> (
  <col_name> <col_value> <constrain>,
  <col_name> <col_value> <constrain>,
  <col_name> <col_value> <constrain>
  );

ALTER TABLE <table_name>
<ADD|DROP|ALTER> <column_name> <type|constrain>;

DROP TABLE <table_name>;
```
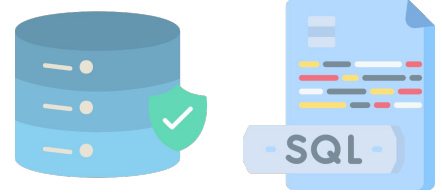
Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

SQL - Data Definition Language - Constraints

Constraints are rules we apply to a table to define what types we can insert in a column or a table, if can be missing values, if its a primary or a foreign key…

- Not null: The specified column must not contain empty values
- Primary key: The specified column/s act as an identifier for the table
- Foreign key: The specified column/s act as an identifier for another table
- Check: Define a range of values for a column
- Default: Specifies a default value in case none is applied

Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

# SQL - Data Definition Language - Types
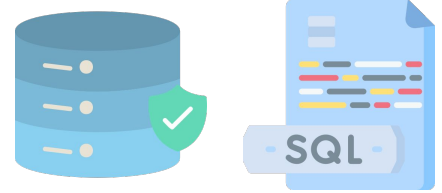
Databases can store a multiple types of values, the most common ones are the following:

- Numerics: Integer, Float, Double, Numeric, Decimal (The 2 latest allows us to specify the length of the number to store)
- Alphanumerics: Char (to specify strings of length n), Varchar (to specify strings of max length n), Text (to store strings without caring about the length)
- Dates: Date (day, month and year), Datetime (same as date but with hour, minutes and seconds too)
- Logics: Bit, and Boolean (1,0, True, False)

SQL - Data Manipulation Language
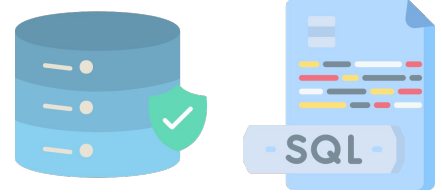
Data Manipulation Language (DMA) is in charge of the data manipulation, select it, insert it, modify it and remove it.

- SELECT - extracts data from a database
- UPDATE - updates data in a database
- DELETE - deletes data from a database
- INSERT INTO - inserts new data into a database

SQL Commands - Data Related Commands

```
● ● ●                    Data Commands SQL

SELECT <col_name1>, <col_name2>,...
FROM <table_name1>
-----Optional-----
WHERE <conditions>
JOIN <table_name2>
ON <table_name1.primary_key> = <table_name2.foreign_key>;

INSERT TABLE <table_name> (<column1>,<column2>,...)
VALUES (<value1>,<value2>,...)
```
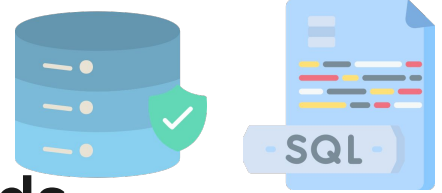
```
● ● ●                    Data Commands SQL

UPDATE <table_name>
SET <col1> = <value1>, <col2> = <value2>,...
WHERE <conditions>;


DELETE FROM <table_name> WHERE <conditions>;
```

SQL Commands - Data Related Commands

IMPORTANT: Take into account that if you do not put any "WHERE" with a proper condition while using an "UPDATE" or a "DELETE FROM" all of the data present in the column will be modified in case of the "UPDATE" and all the data will be deleted in case of the "DELETE FROM".

Be cautious with these sentences as they are powerful.

Programming and Communications III: Databases + SQL + Tools

- ❏ **Database & SQL**
  - ❏ What Is a Database?
  - ❏ Databases and the Relational Model
  - ❏ What is SQL?
  - ❏ SQL - DDL
  - ❏ SQL - DML
- ➢ Object Relational Mapping (ORM)
- ❏ Tools
  - ❏ SQLAlchemy
  - ❏ DBeaver

Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

Object Relational Mapping (ORM)

An ORM is a programming model that allows the structures of a relational database to be mapped onto a logical structure of entities in order to simplify and speed up the development of our applications.

The relational database structures are linked to the logical entities defined in the ORM, in such a way that the CRUD actions (Create, Read, Update, Delete) to be executed on the physical database are performed indirectly through of the ORM.

ORM - How does it work

ORMs create a model of the object-oriented program making a level of logic without the underlying details of the code. Mapping describes the relationship between an object and the data without knowing how the data is structured. The model can then be used to connect the application with the SQL code needed to manage data activities.
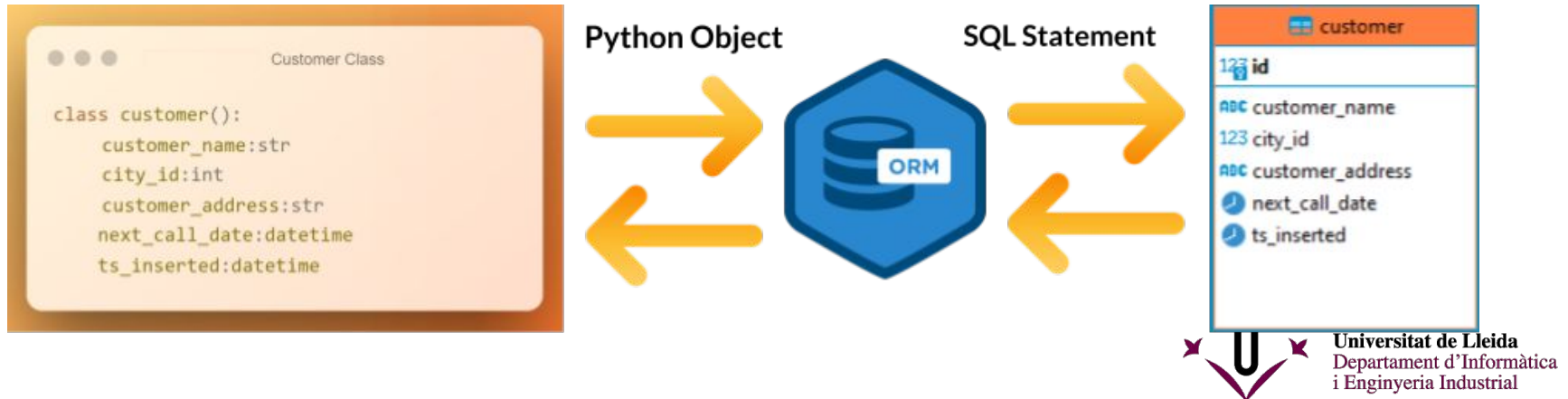
ORM - How does it work

ORMs create a model of the object-oriented program making a level of logic without the underlying details of the code. Mapping describes the relationship between an object and the data without knowing how the data is structured. The model can then be used to connect the application with the SQL code needed to manage data activities.
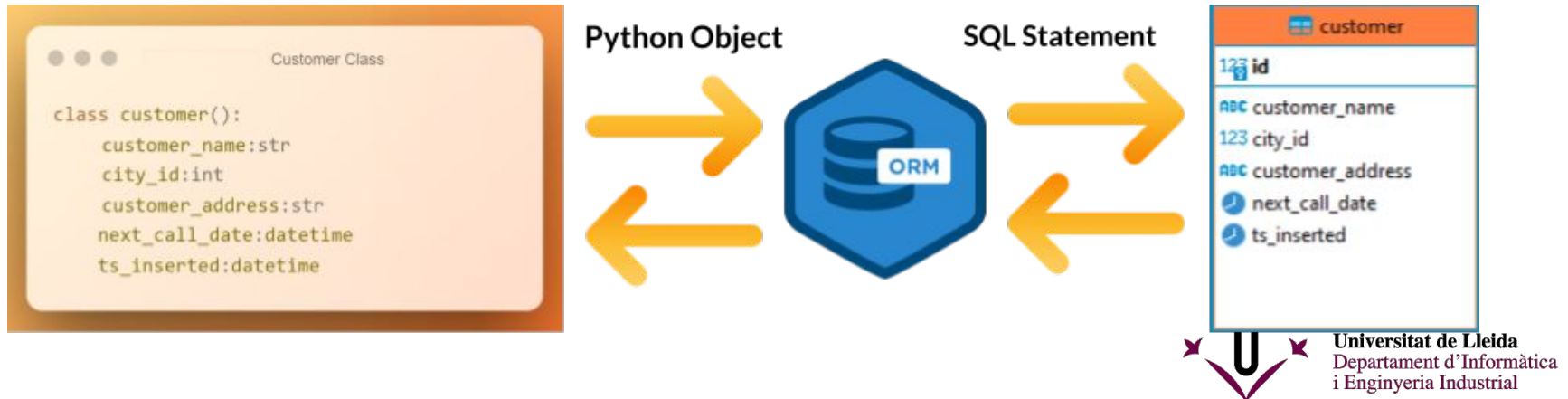
ORM - Advantages

- Productivity: Reduces time on writing database access codes.
- Application Design: We are not writing code for an specific DB, therefore our models are more easy to change if needed.
- Code Reuse: We do not create code specific for accessing the date, we are creating classes that can be used in our application too.
- Reduced Testing: The data access code is generated by the ORM therefore we do not need to test if we are accessing the database properly .

ORM - Disadvantages

- Performance: Sometimes if we are doing complex access, the generated queries might not be the more efficient.
- Complexity: As previously seen in some specific situations we might need to write raw sql queries, as the ORM is not able to generate proper queries, not only for performance but for requirements.

Programming and Communications III: Databases + SQL + Tools

❏ Database & SQL
  ❏ What Is a Database?
  ❏ Databases and the Relational Model
  ❏ What is SQL?
  ❏ SQL - DDL
  ❏ SQL - DML
❏ Object Relational Mapping (ORM)
➢ Tools
  ❏ SQLAlchemy
  ❏ DBeaver

Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

SQL Alchemy

Python library that facilitates access to a relational database, as well as the operations to perform on it.

It is independent of the database engine to be used and is compatible with most known relational databases: PostgreSQL, MySQL, Oracle, Microsoft SQL Server, Sqlite, ...

Although SQLAlchemy can be used using native SQL language queries, the main advantage of working with this library is achieved by using its ORM. The SQLAlchemy ORM maps tables to Python classes and automatically converts function calls within these classes to SQL statements

SQL Alchemy - Installation

To install this SQLAlchemy for python we need to run the following pip command

- pip install SQLAlchemy
- pip install SQLAlchemy-Utils
- pip install psycopg2 (might be installed with SQLAlchemy

Then import it in our Python program
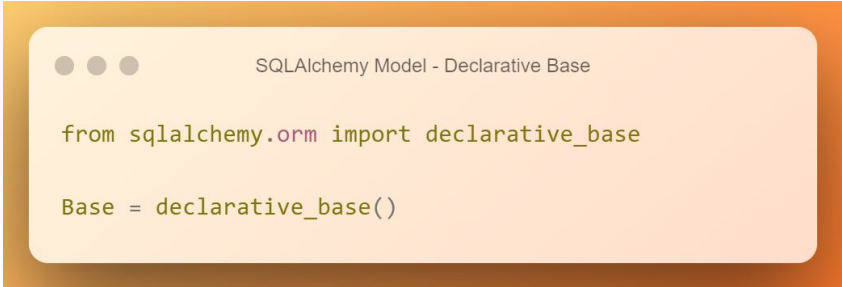
- import sqlalchemy
- import sqlalchemy_utils

Example: Model

To map the our objects with our tables we need to define the models so SQLAlchemy can create the relations.

To do this we need a Base, this base will be used to produce the appropriate table objects and create the proper maps.


```
SQLAlchemy Model - Declarative Base

from sqlalchemy.orm import declarative_base


Base = declarative_base()
```

Example: Model

Then, we can use it to create the models that will represent the tables in our database, note for defining the columns, types and some more, we might need to import more elements. For example, in the following example we will need the imports on the right.

SQLAlchemy Model - Extra Imports

```
from sqlalchemy import Column, Integer, String, ForeignKey, DateTime
```

Example: Model

```python
SQLAlchemy Model - Models

class Employee(Base):
    __tablename__ = "employee"
    id = Column(Integer, primary_key=True, autoincrement=True)
    first_name = Column(String, nullable=False)
    last_name = Column(String, nullable=False)


class City(Base):
    __tablename__ = "city"
    id = Column(Integer, primary_key=True, autoincrement=True)
    name = Column(String, nullable=False)
```

```python
SQLAlchemy Model - Models

class Customer(Base):
    __tablename__ = "customer"
    id = Column(Integer, primary_key=True, autoincrement=True)
    customer_name = Column(String, nullable=False)
    city_id = Column(Integer, ForeignKey("city.id"), nullable=False)
    customer_address = Column(String, nullable=False)
    next_call_date = Column(DateTime, nullable=False)
    ts_inserted = Column(DateTime, nullable=False)


class Call(Base):
    __tablename__ = "call"
    id = Column(Integer, primary_key=True, autoincrement=True)
    employee_id = Column(Integer, ForeignKey("employee.id"))
    customer_id = Column(Integer, ForeignKey("customer.id"))
    start_time = Column(DateTime)
    next_call_date = Column(DateTime)
    end_time = Column(DateTime)
    call_outcome_id = Column(Integer)
```

Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

Example: Insert

To insert data we need to connect with our database, for this we use an engine, this engine has a mandatory parameter which is the connection string to our database, as we are using Postgres the connection string should look like this.

postgresql://<user>:<password>@<direction>:<port>/<database>

The rest of the parameters are optional and are only there to show everything that is happening in our db and ensure that we are using the latest version.

The second step is optional, if we already know that our database exist we can skip, if we are not sure or want to create it we can check if not exists and create it or otherwise connect.

The last sentence will create all the models we have as tables in our db if they do not exists. This command is also optional

Example: Insert


```
SQLAlchemy Inserts - Connection

from sqlalchemy import create_engine
from sqlalchemy_utils import database_exists, create_database

engine = create_engine(
    "postgresql://postgres:postgrespw@localhost:49153/example",
    echo=True,
    future=True)

if not database_exists(engine.url):
    create_database(engine.url)
else:
    engine.connect()

Base.metadata.create_all(engine)
```

Example: Insert

To insert the data we need a Session, this session must belong to our engine.

Then we can simply create instances of our classes and with

- session.add() to insert a single instance
- session.add_all() to insert a list of instances

We can add data to our tables.

It is important to remark that we need to commit all the changes we do so this changes can be seen on our db, to do this we use session.commit()

Example: Insert


```
SQLAlchemy Inserts - Add

with Session(engine) as session:
    victor = Employee(first_name='Victor', last_name='Altés Gaspar')
    salma = Employee(first_name='Salma', last_name='Assiad Sebai')
    session.add_all([victor, salma])
    session.commit()

    lleida = City(name='Lleida')
    barcelona = City(name='Barcelona')
    murcia = City(name='Murcia')
    session.add_all([lleida, barcelona, murcia])
    session.commit()
```

Example: Insert


```
                        SQLAlchemy Inserts - Add

abraham = Customer(customer_name='Abraham Castro Criado', city_id=1,
customer_address='Calle de la Piruleta', next_call_date='2022-12-01',
ts_inserted='2022-12-01')

pelayo = Customer(customer_name='Pelayo Cobos Rodriguez', city_id=1, customer_address='Salchichon',
next_call_date='2022-05-01', ts_inserted='2022-05-01')

artur = Customer(customer_name='Artur Cullerés Cervera', city_id=2, customer_address='Sanjacobo
Street', next_call_date='2022-01-01', ts_inserted='2022-01-01')

didac = Customer(customer_name='Didac Colominas Abalde', city_id=3, customer_address='Albacete
Strassen', next_call_date='2022-08-01', ts_inserted='2022-08-01')

eduard = Customer(customer_name='Eduard de La Arada Janoher', city_id=1, customer_address='Wala wala
bing bong', next_call_date='2022-07-01',
ts_inserted='2022-07-01')

session.add_all([abraham, pelayo, artur, didac, eduard])
session.commit()
```

Example: Insert


```
SQLAlchemy Inserts - Add

call1 = Call(employee_id=1, customer_id=1, start_time='2022-12-01 12:05:06',
             end_time='2022-12-01 12:06:06', call_outcome_id=0)

call2 = Call(employee_id=1, customer_id=2, start_time='2022-05-01 17:11:06',
             end_time='2022-05-01 17:24:06', call_outcome_id=0)

call3 = Call(employee_id=2, customer_id=3, start_time='2022-01-01 22:01:06',
             end_time='2022-01-01 22:01:45', call_outcome_id=0)

call4 = Call(employee_id=2, customer_id=4, start_time='2022-08-01 11:43:06',
             end_time='2022-08-01 11:55:06', call_outcome_id=0)

call5 = Call(employee_id=1, customer_id=5, start_time='2022-07-01 04:22:06',
             end_time='2022-07-01 05:22:06', call_outcome_id=0)

session.add_all([call1, call2, call3, call4, call5])
session.commit()
```

Example: Insert

To select data we need a connection, we can reuse another connection from previous operations or we can build a new one, it is not recommended creating multiple connections, so, reuse always you can.

In the next slide, we are not checking if the database exists, as we are going to retrieve data, we are assuming everything is ready for that.

Example: Select


```
from sqlalchemy import create_engine, select
from sqlalchemy.orm import Session
from models import Employee, Customer, Call

engine = create_engine(
    "postgresql://<user>:<password>@<direction>:<port>/<database>",
    echo=True,
    future=True)


engine.connect()
```

Example: Insert

To select data from our database we need a Session, can be the same session we used before, or can be a new one.

To perform the most basic selects, the ones where we retrieve one or multiple values without any kind of condition we have:

1. session.execute(select(Class of object to retrieve)).scalars().all()
2. session.execute(select(Class of object to retrieve)).scalar()
3. session.execute(select(Class of object to retrieve)).scalars().one()
4. session.execute(select(Class of object to retrieve)).scalars().one_or_none()

Example: Insert

The first case will return a list of all the elements or an empty list if there is no element in our database.

The second and the fourth work exactly the same way, they will return the first element or None if we have no elements.

The third case will return the first element, but, if the table is empty it will rise an exception.

Note: When we are doing selects, we do not need to commit our session as we are not changing anything, this is only done for inserting and modifying data.

Example: Select


```
SQLAlchemy Select

with Session(engine) as session:
    print('------ SELECT ALL ------')
    employees = session.execute(select(Employee)).scalars().all()
    for employee in employees:
        print(employee.first_name)

    print('------ SELECT ONE ------')
    employee = session.execute(select(Employee)).scalar()
    print(employee.first_name)
```

Example: Insert

To apply conditions over our selects we need to modify the sentence :

- session.execute(select(Class1).filter_by(condition).scalars().all) #we can also use one or one_or_none or scalar, like in the previous example

filter_by will be translated as a 'WHERE' for our database.

And for joining different tables:

- session.execute(select(Class1).join(Class2, Class1.primary_key==Class2.foreign_key). filter(condition)).scalars().all()

With join we will join the different tables we specify by the primary_key and its respective foreign key, we can concatenate more joins or filter if needed.

Example: Select


```
                              SQLAlchemy Select

print('------ SELECT WITH FILTER ------')
customers = session.execute(select(Customer).filter_by(city_id=1)).scalars().all()
for customer in customers:
    print(customer.customer_name)


print('------ SELECT WITH JOIN ------')
calls = session.execute(select(Call).join(Customer,
Call.customer_id==Customer.id).filter(Customer.city_id==1)).scalars().all()
for call in calls:
    print(call.id)
```

Example: Modify

If we want to modify data, either update it or delete it, we need a connection and a session as we did in previous steps.

To update the data we have:

- session.execute(update(Class).where(condition).values(field=value).returning(Class.id)).scalar_one_or_none()

Update will update the table related to the class we pass as a parameter, the condition, as in SQL is important so we do not change all the fields of that table, with values() we can specify which column we must update and which value to apply, the returning is optional, but is a good practice to inform the id of the updated elements if any.

Example: Modify

To delete data we have:

- session.execute(delete(Class).where(condition).returning(Class.id)).scalar_one_or_none()

Delete will delete all the elements of the table related to the class passed as parameter under the condition specified, remember that if we do not specify a condition, everything will be deleted. Once again, the returning is not necessary but recommended.

Example: Modify


```
SQLAlchemy Modify

print('------ UPDATE ------')
updated_id = session.execute(
    update(Employee).where(Employee.id == 1).
    values(first_name="Paprika").
    returning(Employee.id)).scalar_one_or_none()

session.commit()

print('------ DELETE ------')
deleted_id = session.execute(
    delete(Call).where(Call.id == 1).
    returning(Call.id)).scalar_one_or_none()

session.commit()
```

DBeaver

DBeaver is a client SQL application and a database administration tool. It can be used with NoSQL databases thanks to its plugin system.

With DBeaver we can manage our databases by means of SQL sentences or scripts, or using the graphical interface (although it is not recommended).

Furthermore, the graphical interface allow us to visualize the data present in our database more easily as well as the tables, relations and constraints.

DBeaver - Installation

Download the installer from:

- https://dbeaver.io/download/

Follow the instructions (I bet you did not expect this)

☺

DBeaver - DB Configuration

Step 1: Click over new connection (the blue plug with the green +)

Step 2: Find the database we are using, in this case we are using PostgreSQL so we might have it between the popular ones, otherwise we can search by using the search bar on top or tabs on the left.
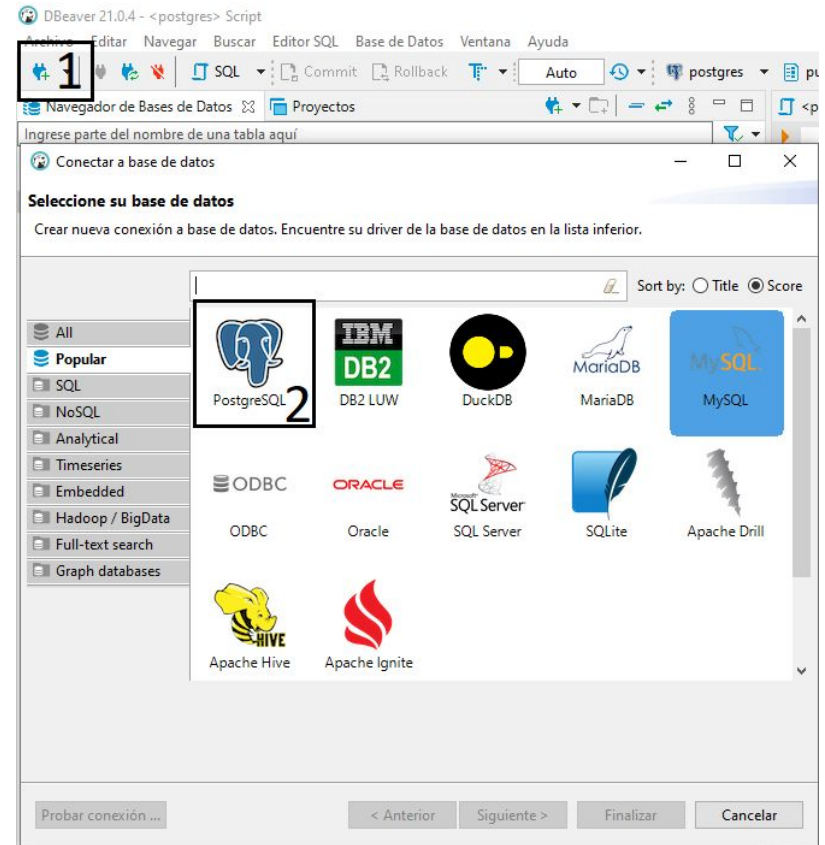
DBeaver - DB Configuration

Step 3: Here we have to configure the connection

Host: Here we specify where is our db (for a docker in our pc should be always localhost)

Post: Port ready for connection 5432 is the default but can change

Username: User with permissions in our db usually the admin

Password: Password of the user

Driver: By default DBeaver chooses the proper driver, but it can be changed by the different drivers provided o one downloaded.

Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

DBeaver - DB Configuration

Step 4 (Optional): Here it is recommended to check the "Show all databases" box, otherwise we will only see the created by default

Finally, we can click on "Test Connection" to see if everything works properly.

Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial
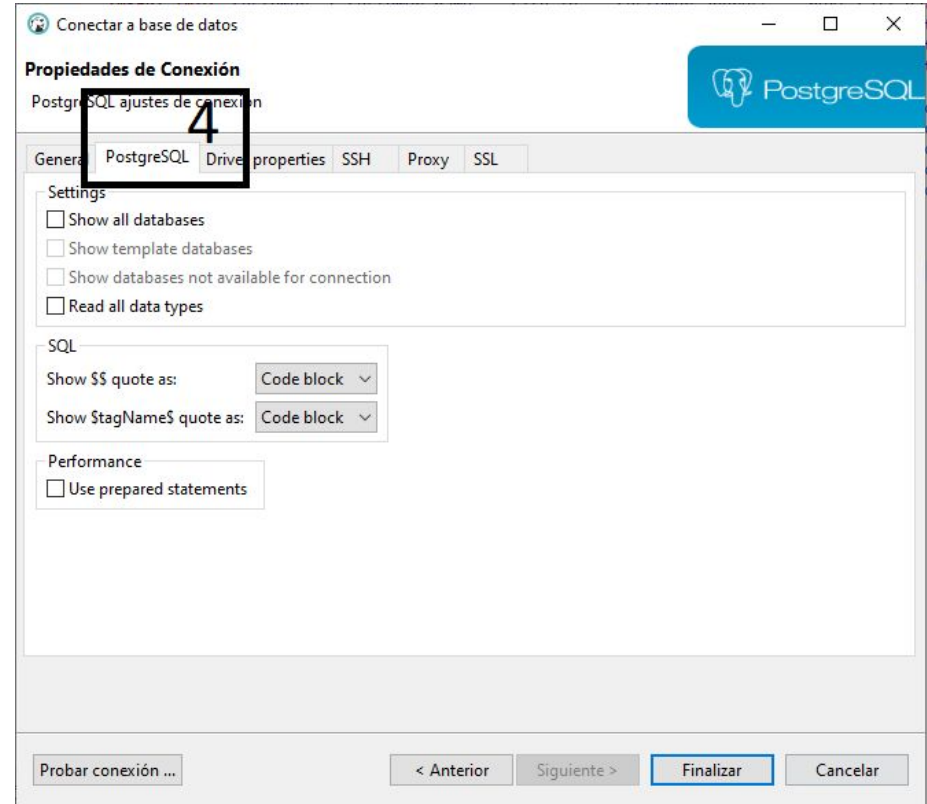
DBeaver - Environment
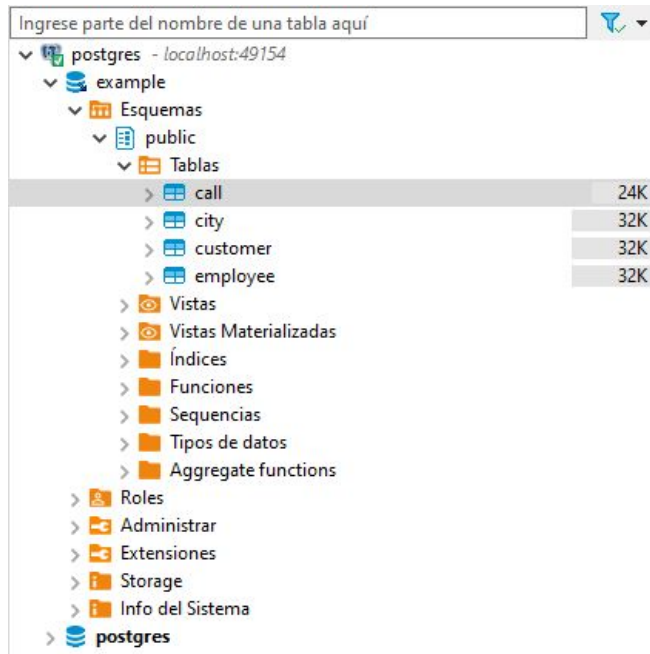
On the left we can see the database navigation window

In this window we can find all the databases configured, clicking on them will show all the data related from the database.

Inside Schema we will find all the tables created as well as different metadata such as constraints and more.

DBeaver - Environment

1   2   3   4   5

On the top bar we can find different options, as we have seen before, the first button to configure a connection, to refresh it, remove dbs.(1, 2)

The SQL button will allows us to open a screen where to write SQL commands to work with our database.(3)

A button where it shows the kind of database we are using at the moment and that we can use to switch to other.(4)

And a button that shows the database we are using, and also to switch between the different databases connected.(5)

DBeaver - Environment

If we click in any of our configured databases we will see the window on the right.

In this window we have 3 tabs

The first one will show the different properties from our table.

The second will show the data in that table, we can apply filters and order the data if required.

The third one will show the table as a diagram with all the related tables.

DBeaver - Environment

Finally, after pressing the SQL button you will see a blank tab, here you can write your SQL scripts to work with your database as shown below.

More info:

https://en.wikipedia.org/wiki/SQL

https://datubaze.files.wordpress.com/2016/03/a_taylor_sql_for_dummies_2003.pdf

https://www.w3schools.com/sql/default.asp

https://docs.sqlalchemy.org/en/14/orm/tutorial.html

http://www.leeladharan.com/sqlalchemy-query-with-or-and-like-common-filters

https://www.altexsoft.com/blog/object-relational-mapping/

https://drive.google.com/file/d/1SwcIcPijg0jSjLMAc86gDZV_iDmrKUih/view

https://dbeaver.io/

Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial