Challenge@PoliTO_**By Students**
Mapping PoliTO spaces – Kickoff meeting
6-7 March 2025

Mapping PoliTO spaces

Prof: Elisabetta Colucci, Emere Arco, Andrea Ajmar, Tommaso Calò, Simone Preziosi

Mentor: Alessandra Spreafico, Tommaso Calò, Simone Preziosi, Emere Arco

Click: Orazio Maria Pennisi, Laura Ronchetto

SPATIAL DATABASES

The relational model

The **Relational Model** is a way of structuring and managing data in a database using tables (also called **relations**).

It was introduced by **E.F. Codd** in 1970 and is the foundation for modern **Relational Database Management Systems (RDBMS)** such as MySQL, PostgreSQL, Oracle, and SQL Server.

The relational model

Let's start with an example: there is a need to develop a computer system that stores **data about Italian roads**.

Each **road** will be associated with certain values (**attributes**), such as a possible name, the number of lanes, etc. It is also important to know the list of **Italian regions** that each road passes through (and vice versa). Each road belongs to a **functional classification** (e.g., highway, main extra-urban road, neighborhood urban road, etc.).

The relational model

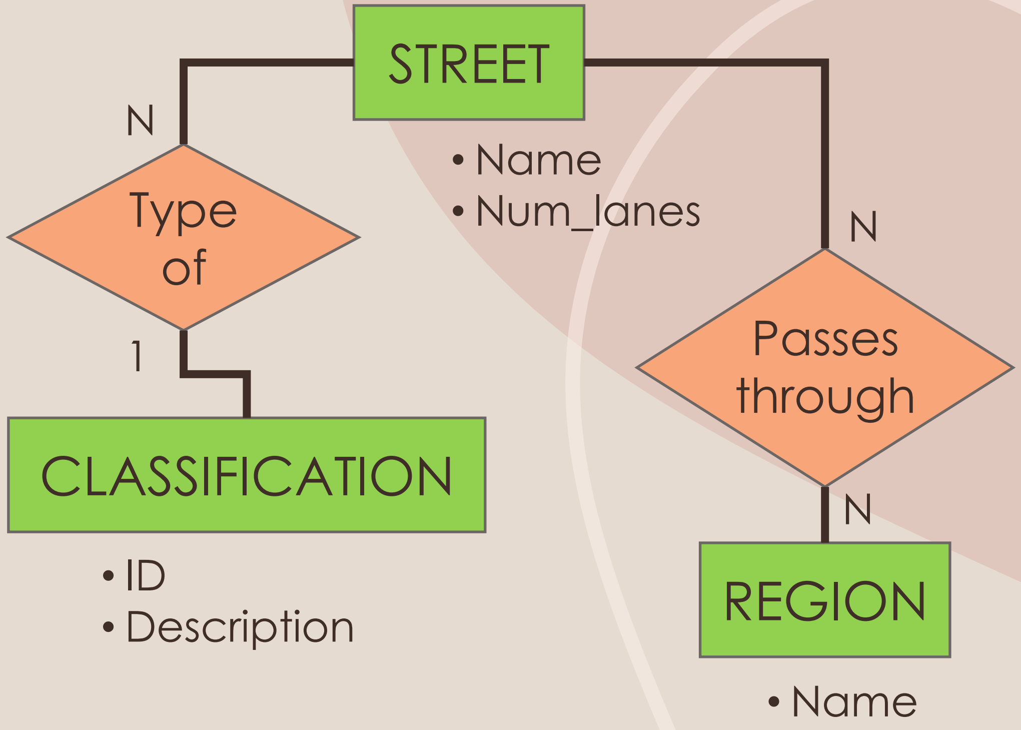
Entities: correspond to **distinguishable classes of objects** in the database. Entities are characterized by associated attributes.

Relationships: connect entities to each other. For example, roads and regions are linked by an inclusion relationship (a road passes through a region). Relationships can be of various **types**:

- One-to-one: Each object in one entity corresponds to one and only one object in the other.
- One-to-many: Each object in one entity corresponds to one or more objects in the other. For example, each functional classification (highway) corresponds to a series of roads of that type (A1, A13, ...).
- Many-to-many: Each object in one entity corresponds to one or more objects in the other, and vice versa. For example, many roads pass through a given region, and conversely, a road can pass through multiple regions.

Entities – relationships diagram

1. Each entity (e.g., a road) corresponds to a **table**.
2. Each attribute (e.g., name) of an entity corresponds to a **column**.
3. Each individual object (e.g., a specific road) corresponds to **a row** in the related table.
4. Relationships are implemented:
   a. implicitly **through attributes and constraints** (if the cardinality is 1–1 or 1–n);
   b. through **an additional table** (if the cardinality is n–n).



Implementation of entities – relationships diagram

Street

Name	Num_lanes	Class
A1	4	01
A22	4	01
Aurelia	4	02
Emilia	2	02

Classification

Code	Description
01	Highway
02	Primary road
03	Secondary road

Street_Region

Street	Region
A1	Lazio
A1	Campania
Aurelia	Toscana
Aurelia	Lazio

Region

Name
Lazio
Toscana
Campania

Keys

In a relational database, the order of rows in a table is not predefined, so a method is needed to identify a specific row. This is achieved through the **primary key**, which is a column (or a set of columns) containing unique values for each row. The primary key uniquely identifies each record in the table.

In some cases, a unique **numeric identifier** (a progressive number) is added to serve as the primary key.

In addition to primary keys, **foreign keys** exist. These are special columns that establish relationships between tables.

Data types

Attributes of entities are associated with a specific **data type.** The main data types are:

- **Text:** Contains words and phrases, with a fixed or variable length (e.g., street names).
- **Numbers:** Stores numerical values, with subtypes like integers and floating-point numbers (single or double precision).
- **Dates, Times, and Intervals:** Represents temporal data (e.g., creation dates, time measurements).

Data Types

Attributes of entities are associated with a specific **data type.** The main data types are:

- **Boolean:** Represents truth values, storing either true or false (road accessibility status).
- **Special Types**: Includes specialized formats like **GEOMETRY**, which allows storing geographic data.

Each table column must be assigned a data type, ensuring that all values within that column belong to the same type. For example, a numeric column cannot contain text, maintaining **data consistency**.

Indexes

Indexes are a fundamental component of a database, though they often remain hidden. They are structures associated with one or more columns of a table and are **used to speed up data retrieval**.

To ensure optimal database performance, an index should be created for any data expected to be frequently searched. Once created, an index is automatically updated when the data changes.

Additionally, there are **spatial indexes**, which allow for the efficient handling of geographic data.

Transactions

Databases are often used by multiple users simultaneously. While individual operations on data are guaranteed, it is sometimes necessary to ensure that **a sequence of updates** is executed consistently.

For example, imagine booking two adjacent seats on a plane. If the seats were assigned one at a time, another user might book one in between, disrupting the reservation.

The **transaction mechanism** allows multiple data modifications to be **grouped into a single unit**. These changes are executed **atomically** - either all succeed together, or none are applied, ensuring **data consistency**.

Schemas

Real-world databases contain a large number of tables, making management complex. To address this, database systems use **schemas, which function like folders for organizing tables**. Schemas allow tables to be grouped into separate containers. Tables in different schemas can have the same name, but within the same schema, table names must be unique.

Normalization

The process of organizing data to eliminate redundancy and improve data integrity. The **First, Second, and Third Normal Forms (1NF, 2NF, 3NF)** define a set of rules for designing a well-structured database.

1. Every table must have a **primary key**.
2. Attributes must store **atomic values** (i.e., no lists or multiple values in a single field).
3. Data should not be **redundant or repeated**.

For example, if a table contains both **birth dates and zodiac signs** for individuals, it is not normalized because the zodiac sign can be derived from the birth date, leading to redundancy**.**

Documents vs databases

Many of the applications we commonly use (word processors, image editors, etc.) follow a **"working document" principle.** In these applications, users can create a new document, save it to disk, and reload it later. Changes can be made, and users decide whether to save them or not, meaning there is a distinction between the document in memory and the saved document. Typically, a document corresponds to a file on disk, which can be moved, copied, or shared. Additionally, in most cases, documents can only be edited by one person at a time, preventing simultaneous collaboration.

Documents vs databases

In contrast, **databases work differently**: there is no distinction between data in memory and data on disk, as all changes are assumed to be instantly stored. There is no concept of saving or loading a document; instead, users connect to the data. Most databases operate on **remote servers**, where users do not directly access files. Additionally, databases are designed for **multi-user concurrent access**, allowing multiple people to work on the same data simultaneously.

Advantages of the Relational Model

- **Data Integrity & Consistency**: Ensures accuracy through constraints like primary and foreign keys.
- **Flexibility**: Allows complex queries using SQL.
- **Scalability**: Suitable for large datasets with optimized indexing and query processing.
- **Data Independence**: Logical and physical data independence allow changes without affecting applications.

Interacting with a database

Queries

Databases store large amounts of information, and the most common operation performed on them is querying. A **query** extracts only the relevant data based on specific criteria. Typical queries include:

- Selecting specific **columns**
- Filtering rows based on certain **conditions**
- Combining data from multiple tables using **joins**
- Performing **spatial queries** (in spatial databases)

Interacting with a database

Views

Frequently used queries can be saved as **views.** A view stores the **query structure**, not the actual results. Views behave like tables but are **dynamic**, meaning their content updates automatically when the underlying data changes.

Spatial Queries

In spatial databases, queries can involve **spatial relationships**, such as:

- Finding objects within a certain distance of a point.
- Identifying adjacent objects.

SQL (Structured Query Language)

SQL (Structured Query Language) is a **standard textual language** used for working with databases.

Being a standard language, it is mostly independent of the specific database software (such as Oracle, Microsoft SQL Server, PostgreSQL, MySQL, etc.).

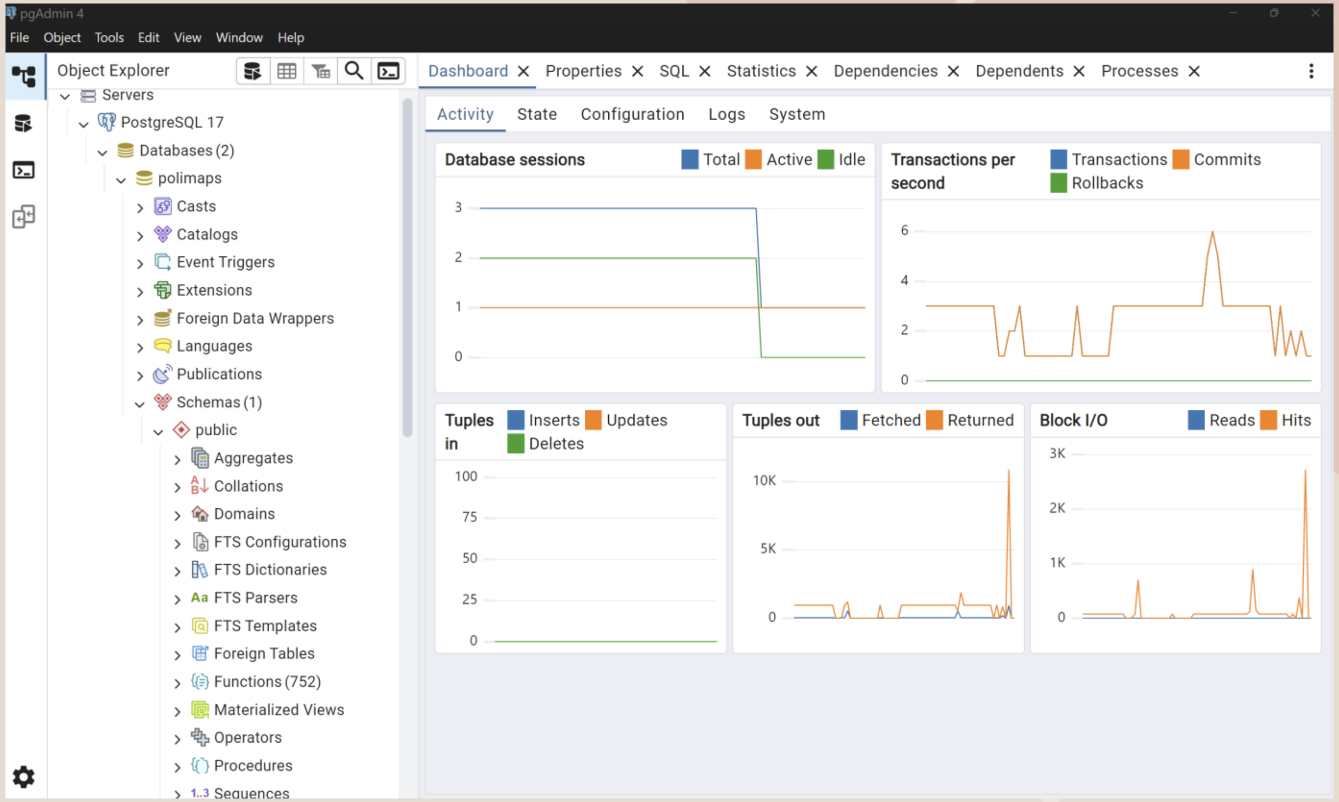
SQL is functional (a single construct performs specified operations), not imperative (it doesn't involve variables or lists of operations), although its extension, PL-SQL, allows for imperative function declarations.

While SQL provides a common foundation, **there are some variations between different database systems**, particularly for spatial data.

PostgreSQL & PostGIS

Is a free database with well-developed spatial support (**PostGIS**).

The database will be accessed through the standard interface, **pgAdmin IV**, although there are more advanced interfaces available.



SQL – Data definition

Let's start by discussing the SQL commands used to define the structure of data (i.e., the shape of tables).

There are three SQL commands for defining data:

- **CREATE TABLE**: creates a table.
- **DROP TABLE**: deletes a table.
- **ALTER TABLE**: modifies the structure of a table.

SQL – Data definition - CREATE

```
CREATE TABLE table_name
(
  column_name1 TYPE1,
  column_name2 TYPE2,
  ...
  column_n_name TYPEn
);
```

```
CREATE TABLE street
(
  name CHARACTER VARYING PRIMARY KEY,
  classification CHARACTER(2) NOT NULL,
  width REAL
);
```

SQL – Data definition – DROP/ALTER

```
DROP TABLE street;

ALTER TABLE street
ADD num_lanes INTEGER;
```

SQL – Data Manipulation

The main data manipulation commands are:

- **INSERT**: adds new rows to a table.
- **DELETE**: removes rows from a table.
- **UPDATE**: modifies existing data.

```
INSERT INTO street
(name, classification, width, num_lanes)
VALUES ('A1', '01', 16, 4);
```

SQL – Data Manipulation

The main data manipulation commands are:

- **INSERT**: adds new rows to a table.
- **DELETE**: removes rows from a table.
- **UPDATE**: modifies existing data.

```
DELETE FROM street;  -- delete all rows
DELETE FROM street
WHERE width > 20;  -- delete specific rows based on a condition
```

SQL – Data Manipulation

The main data manipulation commands are:

- **INSERT**: adds new rows to a table.
- **DELETE**: removes rows from a table.
- **UPDATE**: modifies existing data.

```
UPDATE street
SET num_lanes = 2
WHERE name = 'A23';
```

SQL – Relationships

A database is not only made of entities (tables) but also of relationships. Two objects are related if there is data linking them. For instance, roads are related to the classification table, as each road has a classification code.

```
ALTER TABLE street
ADD CONSTRAINT street_class_fk
FOREIGN KEY (class) REFERENCES
classification(code);
```

SQL – Indexes

Indexes are used to speed up search operations on one or more columns in a table. For instance, to speed up searches on the "width" of streets in a table with many records, we can create an index:

```
CREATE INDEX street_width_idx ON street(width);
```

Indexes can improve search speed but slightly slow down modification operations, as the index needs to be updated. They also require additional disk space.

SQL – Select

The **SELECT** command is the most complex SQL operation. It allows for querying data with various options

```
SELECT column1, column2, ... FROM table WHERE
conditions ORDER BY column1, column2;
```

```
SELECT * FROM street;
```

This will display all rows from the "street" table. To order by name and show only name and width:

```
SELECT name, width FROM street ORDER BY name;
```

SQL – Select

Aggregation combines multiple rows into summary values like minimum, maximum, average, sum, or count:

```
SELECT MIN(width), MAX(width), AVG(width) FROM
street;
```

To **group data** by classification:

```
SELECT class, MIN(width), MAX(width), COUNT(width)
FROM street GROUP BY class;
```

SQL – Select

To display the name of each street along with the description of its classification, a **JOIN** is used. For example:

```
SELECT street.name, classification.description
FROM street, classification
WHERE street.class = classification.code;
```

This query joins the "street" and "classification" tables based on the classification code. Without the "WHERE" clause, the query would return a Cartesian product of all possible combinations.

PostGIS

Includes several essential components for spatial data management:

- **GEOMETRY data type**: Allows the storage of geo-referenced geometries, with additional types like Box and Geography.
- **Support tables**:
  - "spatial_ref_sys": Lists supported reference systems.
  - "geometry_columns": Contains metadata of spatial columns.
- **Spatial functions**: Approximately 700 functions for spatial operations.
- **Spatial indices**: For efficient spatial queries.
- **External import/export tools**.

PostGIS does not include a built-in graphical viewer but works seamlessly with open-source GIS software like QGIS.

PostGIS - Geometry

PostGIS introduces **the GEOMETRY data type**, which is complex type that holds the geometry of a geographical object, **potentially with its associated reference system (SRID)**. The data can be in 2, 3, or 4 dimensions (x, y, z, and M) and can represent various geometric shapes like points, lines, areas, and curves.

Just like numbers and strings, the GEOMETRY type can handle literal values (constants) in **Well-Known Text (WKT)** format. These can be used to define geometries directly in SQL queries. For example: 'POINT(6.1 43.2)’ → A 2D point.

PostGIS - Functions

PostGIS offers about 700 spatial functions, which can be grouped into various categories, including:

- **Management:** e.g., "AddGeometryColumn" adds a geometry column to a table.
- **Construction**: e.g., "ST_MakePoint" creates a point geometry.
- **Access**: e.g., "ST_NDims" returns the number of dimensions of a geometry.
- **Modification**: e.g., "ST_Transform" projects a geometry into a new reference system.
- **Output**: Functions like "ST_AsSVG" convert geometries to formats like SVG, KML, and GML.
- **Measurement**: e.g., "ST_Length" and "ST_Area" measure the length and area of geometries.
- **Relations**: e.g., "ST_Intersects" checks if two geometries intersect.
- **Miscellaneous**: e.g., "ST_YMin" returns the lowest latitude of a geometry.

PostGIS – Spatial Analysis

Simple Measurements

Suppose we want to know the area of a buildings. We can use the "ST_Area" function:

SELECT id, descr, ST_Area(shape) FROM building;

However, this function returns the result in the same unit of measurement as the geometry, which is degrees squared—an unusual unit for area. To get the area in square meters, we need to project the data into a system that uses plane coordinates in meters (e.g., UTM-WGS84, zone 32 North). For this, we use the "ST_Transform" function:

SELECT id, descr, ST_Area(ST_Transform(shape, 32632)) FROM building;

PostGIS – Spatial Analysis

Spatial Aggregating Functions

Just like with alphanumeric data, the spatial component also supports aggregate functions. For example, to find the total extent of all the geographical objects in the table, we can use the "ST_Extent" function:

SELECT ST_Extent(shape) FROM edificio;

The result will be: "BOX(6 42,13 46)"

This represents the maximum extent (bounding box) of all the geometries combined, covering the two buildings in this case.

PostGIS – Spatial Analysis

Elementary Operations

Append: The "INSERT INTO" statement can add objects to an existing feature, provided that the attributes match in number and type.

Add and Calculate Fields: New attributes can be added and calculated with SQL commands. For instance, to add a "surface" attribute to a feature and calculate its area, "ST_Area" is used.

Add XY(Z) Coordinates: Geometric coordinates can be extracted using "ST_X", "ST_Y", and "ST_Z" functions to create new fields containing the x, y, and z coordinates.

Check Geometry: PostGIS provides functions for validating geometries (e.g., "ST_IsValid", "ST_IsEmpty").

PostGIS – Spatial Analysis

Basic geometric operations

Change of Reference System (Project): The "ST_Transform" function is used to change the coordinate reference system (CRS) of geometries.

Buffer Zone: The "ST_Buffer" function creates a polygon that represents a buffer zone around a geometry at a specified distance, using the units of the CRS. Using views for buffer zones ensures the data dynamically updates with the original data.

Feature to Point: The "ST_Centroid" function transforms geometric features (e.g., lakes) into points by finding their center of mass.

PostGIS – Spatial Analysis

Basic geometric operations

Dissolve: The "ST_Union" function aggregates features with common attributes. For example, merging provincial boundaries into regions based on the "region code."

Merge: The "UNION" SQL operator merges multiple similar tables into one.

Clip: The "ST_Intersection" function cuts one geometry using another (e.g., extracting roads within a specific municipality like Florence).

Intersect: Similar to "Clip", but it allows for multiple features to intersect, and the result includes the attributes of both intersecting features.

Erase: The "ST_Difference" function removes part of one geometry that intersects another, which is the reverse of "Clip".

PostGIS – Spatial Analysis

More details:

https://postgis.net/docs/postgis_introduction.html

POSTGIS ROUTING EXTENSION (PGROUTING)

pgRouting

Routing is a widely used technology for numerous geospatial applications, from logistics management to urban planning.

pgRouting is an open-source extension of the PostGIS/PostgreSQL geospatial database that offers a wide range of routing functionalities.

pgRouting stands for “PostgreSQL Routing. It extends the capabilities of PostGIS/PostgreSQL and allows users for advanced routing operations.

With pgRouting, users can **solve complex network analysis tasks** such as finding the shortest path between two points or solving the Travelling Salesman Problem (TSP).

pgRouting

The library includes a wide range of features:

- **Shortest Path Algorithms**: Including A*, Dijkstra and bidirectional variants for efficient pathfinding.
- **All Pairs Shortest Path**: Floyd-Warshall and Johnson’s Algorithm for network-wide path computations.
- **Travelling Salesman (TSP):** To find the optimal route that visits a set of locations.
- **Turn Restriction Shortest Path (TRSP):** For real-world scenarios where turns are restricted.
- **Driving Distance**: To calculate reachable areas based on distance or time.

pgRouting

pgRouting comes with several advantages:

- **User interactivity** will be immediately reflected in the routing output. On the fly route calculation, routing is based on current data.
- **Cost** can be calculated using multiple fields and tables.
- **Community and organisation support**: Users can contribute to the project, report issues, or seek help through the mailing list and GIS StackExchange. Commercial support is also available for those requiring professional assistance.

pgRouting

More details:

https://www.spatialtech.org/pgrouting-geospatial-routing-network.html