Database Theory

Why do we need databases?

Until now, we have stored the “data” from our Programs into variables, which are created and accessed only when running the program. The data is gone if we restart the program (there is no ***data persistence***).

In theory, we could, for example, write the variables into a text file (.txt) and not lose the values when exiting the program, but that would quickly get out of hand, as the data will not be very organized.

Databases, however offer several advantages :

1. **Data Organization and Storage:** Databases provide a ***structured*** way to organize and store large volumes of data. Instead of saving data in unstructured files or spreadsheets, databases allow us to categorize information into tables, rows, and columns, making it easier to manage and access.
2. **Data Retrieval and Manipulation:** Databases offer powerful querying capabilities, allowing users to retrieve specific data based on criteria, perform complex calculations, and manipulate data as needed. This enables efficient data analysis, reporting, and decision-making.
3. **Data Integrity and Consistency:** Databases enforce data integrity constraints, such as unique keys, to ensure the accuracy and consistency of stored data. This helps prevent errors, duplication, and inconsistencies within the database.
4. **Concurrency Control:** In multi-user environments, databases manage simultaneous access to data by multiple users or processes. Concurrency control mechanisms ensure that transactions are executed in a controlled manner to maintain data consistency and prevent conflicts.
5. **Data Security:** Databases offer security features to control access to data and protect sensitive information from unauthorized users. Authentication, authorisation, and encryption mechanisms help safeguard data privacy and confidentiality.
6. **Data Scalability:** As data volumes grow, databases can scale to accommodate increasing storage and processing requirements. They support features like partitioning, replication, and clustering to distribute data across multiple servers and handle high workloads efficiently.
7. **Data Recovery and Backup:** Databases provide mechanisms for data backup and recovery, enabling organizations to restore data in the event of hardware failures, software errors, or disasters. Regular backups ensure data resilience and minimize the risk of data loss.

General structure of a database

A database can contain several tables, where the tables can be linked together.

Each table has columns (Header) & rows (data). The columns define the names and the data types that are being stored in the table.

There are 2 different types of column(s) that have special caracteristics :

Primary Key :

- A primary key is a column or a set of columns that uniquely identifies each record in a table.
- It must contain unique values for each row, and it cannot contain NULL values.
- There can be only one primary key in a table.
- Primary keys are typically used as the basis for relationships with other tables.

Foreign Key :

- A foreign key is a column or a set of columns in one table that refers to the primary key in another table.
- It establishes a link between two tables, enforcing referential integrity.
- The foreign key constraint ensures that the values in the foreign key column(s) match values in the primary key column(s) of the referenced table or are NULL.
- It helps maintain consistency and integrity in the database by preventing actions that would cause orphaned records or invalid references.

⇒ In this course, we will use **PostgreSQL** as a relational database management system (**RDBMS**).

To access and manipulate databases, we use the language **SQL** (Structured Query Language).

FYI : There are also **NoSQL** Databases, but this is outside the scope of this course.

SQL Commands

Some SQL Commands to retrieve and manipulate data :

SELECT statement:

- The **SELECT** statement is used to retrieve data from one or more tables in a database.
- It allows you to specify which columns you want to retrieve and from which tables.
- You can also use aggregate functions (e.g., **COUNT**, **SUM**, **AVG**, **MAX**, **MIN**) within the **SELECT** statement to perform calculations on the data.

The basic syntax of the **SELECT** statement is:

```
SELECT column1, column2, ... FROM table_name;
```

Filtering data with WHERE clause:

- The **WHERE** clause is used to filter records based on specified conditions.
- It allows you to specify criteria that must be met for a row to be included in the result set.
- Conditions can include comparisons (e.g., **=**, **<>**, **<**, **>**, **<=**, **>=**), logical operators (**AND**, **OR**, **NOT**), and other functions.
- The **WHERE** clause follows the **FROM** clause in a **SELECT** statement.

Example:

```
SELECT * FROM employees WHERE department = 'Sales';
```

Sorting data with ORDER BY clause:

- The **ORDER BY** clause is used to sort the result set of a query in ascending or descending order based on one or more columns.
- By default, it sorts in ascending order, but you can specify **ASC** (ascending) or **DESC** (descending) explicitly.

- You can specify multiple columns for sorting, in which case the order of sorting is determined by the order of the columns listed.

- The **`ORDER BY`** clause comes after the **`WHERE`** clause (if used) in a **`SELECT`** statement.

Example:

```
SELECT * FROM products ORDER BY price DESC;
```

Aggregating data with GROUP BY clause:

- The **`GROUP BY`** clause is used to group rows that have the same values into summary rows, often to perform aggregate functions on those groups.
- It divides the result set into groups based on the specified columns or expressions.
- Aggregate functions (e.g., **`COUNT`**, **`SUM`**, **`AVG`**, **`MAX`**, **`MIN`**) can be used in conjunction with the **`GROUP BY`** clause to perform calculations on each group.
- Columns listed in the **`SELECT`** statement that are not part of an aggregate function must be included in the **`GROUP BY`** clause.
- The **`GROUP BY`** clause comes after the **`WHERE`** clause (if used) and before the **`ORDER BY`** clause (if used) in a **`SELECT`** statement.
- Example:

`COUNT()`

```
SELECT department, COUNT(employee) AS employee_count FROM employees
GROUP BY
department;
```

`AVG()`

```
SELECT AVG(release_year) FROM film
# the average year 2006.0000000
```

`DISTINCT()`

```
SELECT DISTINCT(title) FROM film
# list of all distinct titles
```

`MAX()`

```
SELECT MAX(rental_rate) FROM film
```

`MIN()`

```
SELECT MIN(rental_rate) FROM film
```

`SUM()`

```
SELECT SUM(rental_rate) FROM film
```

Use "**SELECT DISTINCT**" if you want unique rows.

Database to practice :

Before creating our own database, we will work on a sample database to practice **`SELECT`** statements.

The database represents a **DVD Rental database**. Here is a short description of the tables within the rental database :

- actor – stores actor data including first name and last name.
- film – stores film data such as title, release year, length, rating, etc.
- film_actor – stores the relationships between films and actors.
- category – stores film's categories data.
- film_category- stores the relationships between films and categories.
- store – contains the store data including manager staff and address.
- inventory – stores inventory data.
- rental – stores rental data.
- payment – stores customer's payments.
- staff – stores staff data.
- customer – stores customer data.

- address – stores address data for staff and customers
- city – stores city names.
- country – stores country names.

Connection to Java

- **Java Database Connectivity (JDBC):** An API that enables Java programs to execute SQL statements. This allows Java applications to interact with any SQL-compliant database.
- **PostgreSQL JDBC Driver:** A specific driver that implements the JDBC API for PostgreSQL databases. It converts requests from Java programs into requests that the PostgreSQL database understands.

Setting Up the Connection

URL Format: The JDBC URL for PostgreSQL is formatted as `jdbc:postgresql://[host]:[port]/[database]`, where `host` is the server address, `port` is the port number (default is 5432), and `database` is the database name.

```
private String url = "jdbc:postgresql://localhost:5432/event_manager";
private String user = "postgres";
private String password = "admin";
```

Properties Object: A `Properties` object can be used to set various options for the connection. Common properties include:

- `user`: The database user's login name.
- `password`: The database user's password.

```
Properties props = new Properties();
props.setProperty("user", "postgres");
props.setProperty("password", "admin");
```

Establishing Connection

- **Try-with-resources Statement:** Ensures that each resource is closed at the end of the statement. This is crucial for avoiding memory leaks.
- **Connection Interface:** Represents a connection to the database. It is used to execute SQL queries and retrieve results.

```
// Try-with-resources statement to ensure that resources are closed
```

```
try (Connection conn = DriverManager.getConnection(url, props)) {
    System.out.println("Connected to the PostgreSQL servers successfully.");

} catch (SQLException e) {
    System.out.println("Connection to the PostgreSQL server failed: " +
e.getMessage());
}
```

Executing a Query

- **Statement Interface:** Used to send SQL statements to the database.
- **ResultSet Interface:** Represents the result set of a SQL query. It is used to navigate and read the data returned from the database.
- **Example Query:** An example SQL query might retrieve names and descriptions from an `event` table. The query `"SELECT name, description FROM event LIMIT 10;"` fetches the first 10 events.

```
// Example query
String query = "SELECT name, description FROM event LIMIT 10;";

// Try-with-resources statement to ensure that resources are closed
try (Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(query)) {

    while (rs.next()) {
        String title = rs.getString("name");
        String description = rs.getString("description");
        System.out.println(title + " - " + description);
    }
} catch (SQLException e) {
    System.out.println("Query execution failed: " + e.getMessage());
}
```

Handling Exceptions

- **SQLException:** Catches and handles SQL related errors. It provides detailed information about the error including a message, SQL state, and an error code.
- **Common Error Messages:** Include connection failures, bad SQL grammar, missing database or permissions issues.

Closing the connection

Releases resources and terminates the database connection.

conn.close();

ERD Schemas

ERD stands for Entity-Relationship Diagram, which is a visual representation of the entities (or tables), attributes (or columns), and relationships between entities in a database schema. ERD schemas for Postgres are essentially diagrams that depict the structure of a database designed to be implemented in a PostgreSQL database management system.

We already saw many of the below definitions, but here is a Summary of all the key concepts that are relevant for designing an ERD Schema :

1. **Entities (Tables)**:
    - Entities represent the tables in the database schema. Each entity corresponds to a table, and its name typically describes the type of data it holds.
    - Attributes of an entity are represented as columns in the table. For example, in a table representing "Customers," attributes could include "Customer_ID," "Name," "Email," etc.
2. **Attributes (Columns)**:
    - Attributes are the properties or characteristics of entities. In PostgreSQL, attributes are represented as columns within a table.
    - Each column represents a specific type of data that can be stored in the database. For instance, a column named "Customer_ID" might store unique identifiers for customers, while a column named "Name" might store their names.
3. **Relationships**:
    - Relationships depict how entities are related to each other. In a database schema, relationships are defined by foreign key constraints.
    - Common types of relationships include one-to-one, one-to-many, and many-to-many. These relationships help maintain the integrity of the data and ensure consistency.
    - In an ERD schema, relationships are depicted using lines connecting related entities. The lines are labeled to indicate the nature of the relationship, such as "1" for one-to-one, "1-M" for one-to-many, and "M-M" for many-to-many.
4. **Primary Keys**:

    - Primary keys uniquely identify each record (or row) within a table. In PostgreSQL, primary keys are typically represented by columns that have unique values for each record.
    - Primary keys are essential for enforcing entity integrity and ensuring that each record can be uniquely identified.
    - In an ERD schema, primary keys are denoted by underlining the attribute that serves as the primary key within an entity.
5. **Foreign Keys**:
    - Foreign keys establish relationships between tables in a database schema. They are columns that reference the primary key of another table.
    - Foreign keys help maintain referential integrity by ensuring that values in one table's foreign key column match values in another table's primary key column.
    - In an ERD schema, foreign keys are depicted as lines connecting related entities. The line originates from the foreign key attribute in one entity and points to the primary key attribute in another entity.

ERD schemas provide a visual representation of the database structure, making it easier to understand the relationships between entities and design an efficient database schema for PostgreSQL.

⇒ in PgAdmin, it is possible to see the ERD Schema of the existing Database (dvdrentals). To do so, right click on your database, and select “ERD For Database”. Postgres will create the Schema automatically.