Project Training

Session 3

- Basic
  - database/table
  - SELECT/INSERT/UPDATE/DELETE
- Database relationships
  - ManyToMany
  - OneToMany (aka ForeignKey)
  - OneToOne
- Project I
  - Modelisation of the final project database regarding user, posts, comments relationships between all the items.

Database

This command will create a database from PostgreSQL shell prompt, but you should have appropriate privilege to create a database. By default, the new database will be created by cloning the standard system database template1.

The basic syntax of CREATE DATABASE statement is as follows

CREATE DATABASE dbname;

Databased can also be created by cli:

$> createdb [OPTIONS] dbname

Tables

CREATE TABLE is a keyword, telling the database system to create a new table. The unique name or identifier for the table follows the CREATE TABLE statement. Initially, the empty table in the current database is owned by the user issuing the command.

Then, in brackets, comes the list, defining each column in the table and what sort of data type it is.

Tables

```
CREATE TABLE table_name(
    column1 datatype,
    column2 datatype,
    column3 datatype,
    .....
    columnN datatype,
    PRIMARY KEY( one or more columns )
);
```

Tables - Data types

While creating table, for each column, you specify a data type, i.e., what kind of data you want to store in the table fields. This enables several benefits:

- **Consistency** – Operations against columns of same data type give consistent results and are usually the fastest.
- **Validation** – Proper use of data types implies format validation of data and rejection of data outside the scope of data type.
- **Compactness** – As a column can store a single type of value, it is stored in a compact way.
- **Performance** – Proper use of data types gives the most efficient storage of data. The values stored can be processed quickly, which enhances the performance.

Tables - Data types

PostgreSQL supports a wide set of Data Types. Besides, users can create their own custom data type using CREATE TYPE SQL command. There are different categories of data types in PostgreSQL.

- Numeric Types
- Monetary Types
- Character Types
- Binary Data Types
- Boolean Type
- Enumerated Type
- Date/Time Types
- Geometric Type

Tables - Data types

- Network Address Type
- Bit String Type
- Text Search Type
- UUID Type
- UUID Type
- JSON Type
- Array Type
- Range Types
- Object Identifier Types
- Pseudo Types

Tables - Constraints

Constraints are the rules enforced on data columns on table. These are used to prevent invalid data from being entered into the database. This ensures the accuracy and reliability of the data in the database.

Constraints could be column level or table level. Column level constraints are applied only to one column whereas table level constraints are applied to the whole table. Defining a data type for a column is a constraint in itself. For example, a column of type DATE constrains the column to valid dates.

Tables - Constraints

The following are commonly used constraints available in PostgreSQL.

- **NOT NULL** Constraint − Ensures that a column cannot have NULL value.
- **DEFAULT** Constraint − Ensures that a default value is set for a given type.
- **UNIQUE** Constraint − Ensures that all values in a column are different.
- **PRIMARY KEY** − Uniquely identifies each row/record in a database table.
- **FOREIGN KEY** − Constrains data based on columns in other tables.
- CHECK Constraint − The CHECK constraint ensures that all values in a column satisfy certain conditions.
- **EXCLUSION** Constraint − The EXCLUDE constraint ensures that if any two rows are compared on the specified column(s) or expression(s) using the specified operator(s), not all of these comparisons will return TRUE.

Tables - Constraints

NOT NULL Constraint:

By default, a column can hold NULL values. If you do not want a column to have a NULL value, then you need to define such constraint on this column specifying that NULL is now not allowed for that column. A NOT NULL constraint is always written as a column constraint.

A NULL is not the same as no data; rather, it represents unknown data.

Tables - Constraints

DEFAULT Constraint:

A column can be assigned a default value. When a new row is created and no values are specified for some of the columns, those columns will be filled with their respective default values. A data manipulation command can also request explicitly that a column be set to its default value, without having to know what that value is.

Tables - Constraints

UNIQUE Constraint

The UNIQUE Constraint prevents two records from having identical values in a particular column. In the COMPANY table, for example, you might want to prevent two or more people from having identical age.

Tables - Constraints

PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a database table. There can be more UNIQUE columns, but only one primary key in a table. Primary keys are important when designing the database tables. Primary keys are unique ids.

We use them to refer to table rows. Primary keys become foreign keys in other tables, when creating relations among tables. Due to a 'longstanding coding oversight', primary keys can be NULL in SQLite. This is not the case with other databases.

A primary key is a field in a table, which uniquely identifies each row/record in a database table. Primary keys must contain unique values. A primary key column cannot have NULL values.

A table can have only one primary key, which may consist of single or multiple fields. When multiple fields are used as a primary key, they are called a composite key.

If a table has a primary key defined on any field(s), then you cannot have two records having the same value of that field(s).

Tables - Constraints

FOREIGN KEY Constraint

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table. We say this maintains the referential integrity between two related tables. They are called foreign keys because the constraints are foreign; that is, outside the table. Foreign keys are sometimes called a referencing key.

Tables - Constraints

CHECK Constraint

The CHECK Constraint enables a condition to check the value being entered into a record. If the condition evaluates to false, the record violates the constraint and is not entered into the table.

```
CREATE TABLE company (
    id INT PRIMARY KEY    NOT NULL,
    name        TEXT    NOT NULL,
    age         INT    NOT NULL,
    address      CHAR(50),
    salary       REAL    CHECK(SALARY > 0)
);
```

Tables - Constraints

EXCLUSION Constraint

Exclusion constraints ensure that if any two rows are compared on the specified columns or expressions using the specified operators, at least one of these operator comparisons will return false or null.

Playing with data

- SELECT
- INSERT
- UPDATE
- DELETE

SELECT

Examples:

SELECT * FROM table WHERE conditions ORDER BY fields;

SELECT id FROM table WHERE condition GROUP BY fields;

SELECT id FROM table LIMIT 1

SELECT NOW();

SELECT my_function();

SELECT

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ * | expression [ [ AS ] output_name ] [, ...] ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY grouping_element [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
```

INSERT

INSERT inserts new rows into a table. One can insert one or more rows specified by value expressions, or zero or more rows resulting from a query.

The target column names can be listed in any order. If no list of column names is given at all, the default is all the columns of the table in their declared order; or the first N column names, if there are only N columns supplied by the VALUES clause or query. The values supplied by the VALUES clause or query are associated with the explicit or implicit column list left-to-right.

INSERT

Examples:

INSERT INTO table (field [, field]) VALUES (value [, value]);

INSERT INTO table (id) VALUES (1), (2);

INSERT INTO table (id) VALUES (1) ON CONFLICT DO NOTHING;

INSERT

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
    [ ON CONFLICT [ conflict_target ] conflict_action ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

where conflict_target can be one of:

```
    ( { index_column_name | ( index_expression ) } [ COLLATE collation ] [ opclass ] [, ...] ) [ WHERE
index_predicate ]
    ON CONSTRAINT constraint_name
```

INSERT

and conflict_action is one of:

```
DO NOTHING
DO UPDATE SET { column_name = { expression | DEFAULT } |
            ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] ) |
            ( column_name [, ...] ) = ( sub-SELECT )
          } [, ...]
        [ WHERE condition ]
```

UPDATE

UPDATE changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need be mentioned in the SET clause; columns not explicitly modified retain their previous values.

There are two ways to modify a table using information contained in other tables in the database: using sub-selects, or specifying additional tables in the FROM clause. Which technique is more appropriate depends on the specific circumstances.

UPDATE

Examples:

```
UPDATE table SET id = 0 WHERE True;

UPDATE table SET updated_at = NOW() WHERE id = 1;

UPDATE table SET counter = counter + 1;
```

UPDATE

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    SET { column_name = { expression | DEFAULT } |
          ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] ) |
          ( column_name [, ...] ) = ( sub-SELECT )
        } [, ...]
    [ FROM from_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

DELETE

DELETE deletes rows that satisfy the WHERE clause from the specified table. If the WHERE clause is absent, the effect is to delete all rows in the table. The result is a valid, but empty table.

There are two ways to delete rows in a table using information contained in other tables in the database: using sub-selects, or specifying additional tables in the USING clause. Which technique is more appropriate depends on the specific circumstances.

DELETE

Examples:

DELETE FROM table WHERE id > 0;

DELETE FROM tag; // This one will remove ALL the rows in your table.

DELETE

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING using_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

Relationships

The database term "relational" or "relationship" describes the way that data in tables is connected.

Newcomers to the world of databases often have a hard time seeing the difference between a database and a spreadsheet. They see tables of data and recognize that databases allow you to organize and query data in new ways, but fail to grasp the significance of the relationships between data that give relational database technology its name.

What is it?

Relationships allow you to describe the connections between different database tables in powerful ways. These relationships can then be leveraged to perform powerful cross-table queries, known as joins.

Types of database relationships

There are three different types of database relationships, each named according to the number of table rows that may be involved in the relationship. Each of these three relationship types exists between two tables.

Types of database relationships

One-to-one relationships occur when each entry in the first table has one, and only one, counterpart in the second table. One-to-one relationships are rarely used because it is often more efficient to simply put all of the information in a single table. Some database designers do take advantage of this relationship by creating tables that contain a subset of the data from another table.

Types of database relationships

One-to-many relationships are the most common type of database relationship. They occur when each record in Table A corresponds to one or more records in Table B, but each record in the Table B corresponds to only one record in Table A. For example, the relationship between a Teachers table and a Students table in an elementary school database would likely be a one-to-many relationship, because each student has only one teacher, but each teacher has multiple students. This one-to-many design helps eliminate duplicated data.

Types of database relationships

Many-to-many relationships occur when each record in Table A corresponds to one or more records in Table B, and each record in Table B corresponds to one or more records in Table A. For example, the relationship between a Teachers and a Courses table would likely be many-to-many because each teacher may instruct more than one course, and each course may have more than one instructor.

Using postgres, this RDBMS doesn’t have any **ManyToMany** or **OneToOne** types built in.

To use those types, you will need to rely on ForeignKeys, and sometimes, an extra table that can hold the references between the items you want to connect.

IMPORTANT: Since you will have to deal with many tables and you might not be able to perform all the operation within one single query. It’s highly suggested to manage your relationships with transactions, triggers or functions.

Example

```
-- Create a table for blog entries.
CREATE TABLE entry (
    id      SERIAL PRIMARY KEY,
    title   TEXT,
    content TEXT
);
```

Example

```
-- Create a table for tag entries.
CREATE TABLE tag (
    id     SERIAL PRIMARY KEY,
    name   TEXT
);
```

Example

```
-- Create a table for entry_coll_tag entries.
CREATE TABLE entry_coll_tag (
    entry_id    INTEGER REFERENCES entry(id)
                    ON UPDATE CASCADE
                    ON DELETE CASCADE,
    tag_id      INTEGER REFERENCES tag(id)
                    ON UPDATE CASCADE
                    ON DELETE CASCADE,
    created_at  TIMESTAMPTZ,
    PRIMARY KEY (entry_id, tag_id)
);
```

Exercise 1

For the next session, you will have to modelize your full social network database structure.

You will have to stick to this model as much as you can during the development. However, if changes have to be made, don’t forget to update your documentation too since I will use it during the final examination!

You will have to modelize at least users, posts, comments and friendship. JSONB or Array types are not allowed for relationships.