PostgreSQL data types

Before creating our own database tables, we need to have a closer look at the different data types that are available in Postgres. In PostgreSQL, there are many data types available for defining columns in tables. Here are some of the commonly used data types (you can find a complete list here : https://www.postgresql.org/docs/current/datatype.html) :

1. **Numeric Types**:
    - **integer**: A signed four-byte integer.
    - **bigint**: A signed eight-byte integer.
    - **numeric(precision, scale)**: Arbitrary precision number with specified precision and scale.
    - **decimal(precision, scale)**: Alias for **numeric**.
2. **Character Types**:
    - **character varying(n)**: Variable-length character string with a maximum length of n characters.
    - **varchar(n)**: Alias for **character varying**.
    - **text**: Variable-length character string (no maximum length).
    - **char(n)**: Fixed-length character string with a length of n characters.
    - **character(n)**: Alias for **char**.
3. **Boolean Type**:
    - **boolean**: Logical Boolean (true/false).
4. **Date/Time Types**:
    - **timestamp**: Date and time (no time zone).
    - **timestamp with time zone**: Date and time with time zone.
    - **date**: Date only.
    - **time**: Time only (no time zone).
    - **time with time zone**: Time with time zone.
5. **Array Types**:
    - **integer[]**: Array of integers.
    - **varchar(n)[]**: Array of variable-length character strings.
    - **text[]**: Array of text.
    - **timestamp[]**: Array of timestamps.

Example

```
CREATE TABLE DataTypeExamples (
  ID SERIAL PRIMARY KEY,
  SampleInteger INT, -- Integer: A typical integer number, without decimals.
  SampleText TEXT, -- Text: Variable unlimited length string.
  SampleVarchar VARCHAR(255), -- Varchar(n): Variable length string with a limit.
  SampleBoolean BOOLEAN, -- Boolean: True or false value.
  SampleDate DATE, -- Date: Calendar date (year, month, day).
```

```
   SampleTimestamp TIMESTAMP, -- Timestamp: Date and time, without time
zone.
   SampleFloat FLOAT, -- Float: Floating-point number.
   SampleNumeric NUMERIC(10, 2), -- Numeric: Exact numeric values with a
specified precision (p) and scale (s).
   SampleUUID UUID -- UUID: Universally Unique Identifier.
);
```

Creating a new SQL database

CREATE DATABASE databasename;

Deleting an SQL database

DROP DATABASE databasename;

Creating tables

Now that we know more about data types, we can start creating our own database.

Creating tables with CREATE TABLE statement:

- The **CREATE TABLE** statement is like building a blueprint for storing data in a database.
- It allows you to define the structure of a new table by specifying the names and data types of its columns.
- Each column represents a piece of information (e.g., name, age, email) and has a specific data type (e.g., text, number, date).
- You can also set constraints, like primary keys and foreign keys, to ensure data integrity.
- If you want to specify a constraint for a field:

NOT NULL	**Cannot have NULL value**
UNIQUE	Each value is unique
PRIMARY KEY	A combination of NOT NULL and UNIQUE
FOREIGN KEY	Prevents destruction of links
CHECK	Satisfy a condition

DEFAULT	If a value is not specifies, the default value will be given
SERIAL (only in PostgresQL)	Generates a sequence of integers

Examples:

```
CREATE TABLE Cities (
   CityID SERIAL PRIMARY KEY,
   CityName varchar(255) NOT NULL UNIQUE
);

 CREATE TABLE Persons (
         -- SERIAL: auto-incremented integer, PRIMARY KEY ensures uniqueness
and non-nullability
  PersonID SERIAL PRIMARY KEY,
  -- NOT NULL: cannot be left blank
  LastName varchar(255) NOT NULL,
  FirstName varchar(255) NOT NULL,
  Address varchar(255) NOT NULL,
  CityID int NOT NULL,
  -- CHECK: Age must be greater than 0 and less than 150
  Age int CHECK (Age > 0 AND Age < 150),
  -- UNIQUE: No two persons can have the same email, DEFAULT: 'N/A' if not
provided
  Email varchar(255) UNIQUE DEFAULT 'N/A',
  -- FOREIGN KEY: References CityID from Cities table
  FOREIGN KEY (CityID) REFERENCES Cities(CityID)
);
```

To create a SQL table with references (foreign keys), you typically use the **CREATE TABLE** statement along with the **REFERENCES** clause to establish the relationship between tables. Here's an example:

Let's say we have two tables: **students** and **courses**. Each student can enroll in multiple courses, so there's a one-to-many relationship between **students** and **courses**.

```
-- Create the 'students' table
CREATE TABLE students (
   student_id INT PRIMARY KEY,
   student_name VARCHAR(50),
    date_of_birth DATE
);

-- Create the 'courses' table
CREATE TABLE courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(50),
    instructor VARCHAR(50)
);

-- Create a third table 'enrollments' to represent the many-to-many relationship
CREATE TABLE enrollments (
    enrollment_id SERIAL PRIMARY KEY,
    student_id INT,
    course_id INT,
    FOREIGN KEY (student_id) REFERENCES students(student_id),
    FOREIGN KEY (course_id) REFERENCES courses(course_id)
);
```

Explanation:

- In the **students** table, **student_id** is the primary key.
- In the **courses** table, **course_id** is the primary key.
- In the **enrollments** table, **enrollment_id** is a unique identifier for each enrollment record. **student_id** and **course_id** are foreign keys that reference the **student_id** in the **students** table and **course_id** in the **courses** table, respectively.
- By specifying **FOREIGN KEY (column_name) REFERENCES table_name(column_name)**, you establish referential integrity constraints, ensuring that every **student_id** and **course_id** in the **enrollments** table must exist in the **students** and **courses** tables, respectively.

This schema ensures that each enrollment record in the **enrollments** table corresponds to an existing student in the **students** table and an existing course in the **courses** table, thereby maintaining the integrity of the database relationships.

Modifying tables

If you made a mistake by creating a table, or if you structure needs to change you can always alter an existing table :

- The **ALTER TABLE** statement is like making changes to a building after it's been constructed.

- It allows you to modify the structure of an existing table by adding, modifying, or dropping columns.
- You can also add constraints or modify existing ones, like adding a primary key or changing the data type of a column.

ADD column

```
ALTER TABLE table_name
ADD column_name datatype;

ALTER TABLE Customers
ADD Email varchar(255);
```

DROP column

```
ALTER TABLE table_name
DROP COLUMN column_name;

ALTER TABLE Customers
DROP COLUMN Email;
```

RENAME column

```
ALTER TABLE table_name
RENAME COLUMN old_name to new_name;
```

CHANGE column **TYPE**

```
ALTER TABLE event
ALTER COLUMN start_datetime TYPE TIMESTAMP USING
start_datetime::TIMESTAMP;
```

Insert data inside a table

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);

INSERT INTO Customers (CustomerName, City, Country)
VALUES ('Cardinal', 'Stavanger', 'Norway');
-- or multiple inserts
INSERT INTO Customers (CustomerName, ContactName, Address, City,
PostalCode, Country)
VALUES
```

('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway'),
('Greasy Burger', 'Per Olsen', 'Gateveien 15', 'Sandnes', '4306', 'Norway'),
('Tasty Tee', 'Finn Egan', 'Streetroad 19B', 'Liverpool', 'L1 0AA', 'UK');

Deleting tables

Deleting tables with DROP TABLE statement:

- The **DROP TABLE** statement is like demolishing a building to remove it completely.
- It allows you to delete an entire table from the database, along with all its data and structure.
- Once a table is dropped, it cannot be recovered, so use this statement with caution.

Example:

DROP TABLE table_name

Delete all the data inside a table

TRUNCATE TABLE table_name;

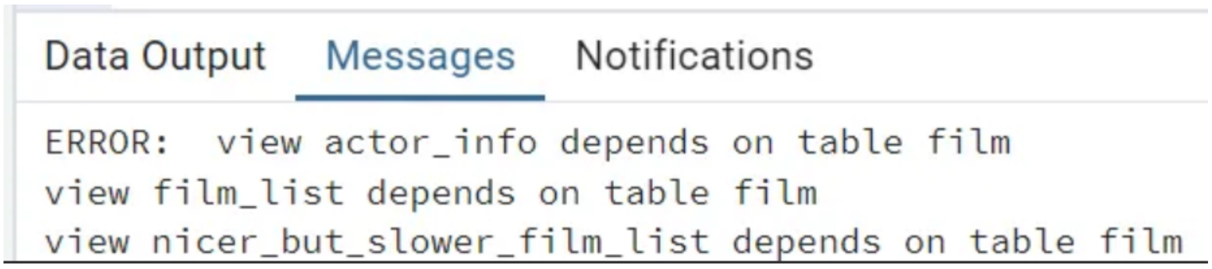
-- deletes the data and restarts the SERIAL counter
TRUNCATE TABLE table_name RESTART IDENTITY;
-- deletes the data and restart SERIAL for tables with references
TRUNCATE table_name restart IDENTITY CASCADE;

DELETE CASCADE

When trying to drop a table that has references to other tables, Postgres won't let you.

For example, if we try to drop the table film in our dvdrental database, we will get an error :

DROP table film;



```
ERROR:  view actor_info depends on table film
view film_list depends on table film
view nicer_but_slower_film_list depends on table film
```

This is due to the fact, that you can't "break" a reference to other tables, as you would first need to remove the data in the tables that are being referenced.

This is where the **DELETE CASCADE** is being useful :

1. If DELETE CASCADE is specified in the foreign key constraint, PostgreSQL automatically deletes all rows in the child tables that reference the deleted row in the parent table.
2. The cascading delete operation is performed recursively, meaning that if child tables themselves have foreign key constraints with DELETE CASCADE enabled, their referenced rows will also be deleted, and so on.
3. DELETE CASCADE can help maintain referential integrity by ensuring that all related rows are deleted when a parent row is deleted, preventing orphaned records in the database.

The ON DELETE CASCADE option needs to be specified when creating the table :

```
-- Create the orders table
CREATE TABLE orders (
  order_id SERIAL PRIMARY KEY,
  order_date DATE,
  total_amount NUMERIC
);

-- Create the order_items table with a foreign key constraint referencing orders
CREATE TABLE order_items (
  item_id SERIAL PRIMARY KEY,
  order_id INT REFERENCES orders(order_id) **ON DELETE CASCADE**,
  product_name VARCHAR(100),
  quantity INT,
  unit_price NUMERIC
);
```