

Introduction to database using PostgreSQL

There are two definitions of a database. The first definition is that a database is an organized collection of information or data. A database also gives us a method of accessing and manipulating the data. So a database would allow us to order the cocktails, for instance, by page number and they would also allow us to only see the cocktails containing tequila.

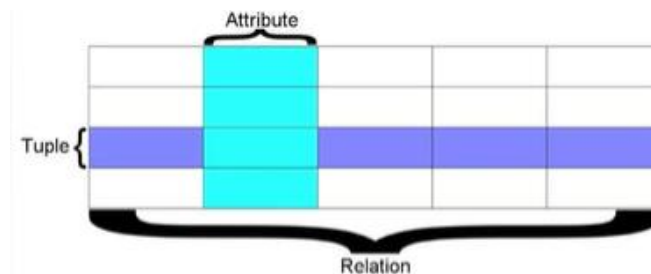
SQL

Structured Query Language is the language we use to issue commands to the database. With it, we can create/insert data, read/select some data, update data, delete data, etc.

Terminology

- Database: contains one or more tables.
- Relation (or table): contains tuples and attributes.
- Tuple (or row): a set of fields which generally represents an “object” like a person or a music track.
- Attribute (also column or field) – one of possibly many elements of data corresponding to the object represented by the row.

A relation is defined as a set of tuples that have the same attributes. A tuple usually represents an object and information about that object. Objects are typically physical objects or concepts. A relation is usually describes as a table, which is organized into rows and columns. All the data referenced by an attribute are in the same domain and conform to be the same constraints. (Wikipedia)



Common Database Systems

- Major Database management systems is wide use:
 - PostgreSQL: 100% open source, feature rich
 - Oracle: large commercial, enterprise-scale, very tweakable
 - MySQL: Fast and scalable: commercial open source
 - SqlServer: very nice – from Microsoft (also Access)
- Smaller projects: SQLite, HSQL...

Database Tables

Tables contain columns (fields) and rows of data (records). Each column has a defined data type which defines what type of data can be contained within that column. Each row of data should be unique. Each column should contain only one value per row.

In a relational database, tables can be linked together. Two tables are linked through primary keys and foreign keys.

This is an example of database tables.

id integer	first_name character varying (30)	last_name character varying (30)	city character varying (30)	state character (2)
1	Samuel	Smith	Boston	MA
2	Emma	Johnson	Seattle	WA
3	John	Oliver	New York	NY
4	Olivia	Brown	San Francisco	CA
5	Simon	Smith	Dallas	TX

PostgreSQL data type

Data Types PostgreSQL has a rich set of native data types available to users. Users can add new types to PostgreSQL using the CREATE TYPE command.

Numeric Types

Numeric types consist of two-, four-, and eight-byte integers, four- and eight-byte floating-point numbers, and selectable-precision decimals:

Data Type	Description	Example Columns
INT	Whole Numbers	Age, Quantity
NUMERIC(P,S)	Decimal Numbers	Height, Price
SERIAL	Auto incrementing Whole Number	Id (primary key)

Name	Storage Size	Description	Range
smallint	2 bytes	small-range integer	-32768 to +32767
integer	4 bytes	typical choice for integer	-2147483648 to +2147483647
bigint	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807

Name	Storage Size	Description	Range
<code>decimal</code>	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
<code>numeric</code>	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
<code>real</code>	4 bytes	variable-precision, inexact	6 decimal digits precision
<code>double precision</code>	8 bytes	variable-precision, inexact	15 decimal digits precision
<code>smallserial</code>	2 bytes	small autoincrementing integer	1 to 32767
<code>serial</code>	4 bytes	autoincrementing integer	1 to 2147483647
<code>bigserial</code>	8 bytes	large autoincrementing integer	1 to 9223372036854775807

Integer Types

The types `smallint`, `integer`, and `bigint` store whole numbers, that is, numbers without fractional components, of various ranges. Attempts to store values outside of the allowed range will result in an error. The type `integer` is the common choice, as it offers the best balance between range, storage size, and performance. The `smallint` type is generally only used if disk space is at a premium. The `bigint` type is designed to be used when the range of the `integer` type is insufficient. SQL only specifies the integer types `integer` (or `int`), `smallint`, and `bigint`. The type names `int2`, `int4`, and `int8` are extensions, which are also used by some other SQL database systems

Arbitrary Precision Numbers

The type `numeric` can store numbers with a very large number of digits. It is especially recommended for storing monetary amounts and other quantities where exactness is required. Calculations with `numeric` values yield exact results where possible, e.g., addition, subtraction, multiplication. However, calculations on `numeric` values are very slow compared to the integer types, or to the floating-point types

We use the following terms below: The precision of a `numeric` is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. The scale of a `numeric` is the count of decimal digits in the fractional part, to the right of the decimal point. So the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero. Both the maximum precision and the maximum scale of a `numeric` column can be configured.

The types `decimal` and `numeric` are equivalent. Both types are part of the SQL standard. When rounding values, the `numeric` type rounds ties away from zero, while (on most machines) the `real` and `double precision` types round ties to the nearest even number.

.Floating-Point Types

The data types real and double precision are inexact, variable-precision numeric types. Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and retrieving a value might show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire branch of mathematics and computer science and will not be discussed here, except for the following points:

- If you require exact storage and calculations (such as for monetary amounts), use the numeric type instead.
- If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), you should evaluate the implementation carefully.
- Comparing two floating-point values for equality might not always work as expected.

By default, floating point values are output in text form in their shortest precise decimal representation; the decimal value produced is closer to the true stored binary value than to any other value representable in the same binary precision. (However, the output value is currently never exactly midway between two representable values, in order to avoid a widespread bug where input routines do not properly respect the round-to-nearest-even rule.) This value will use at most 17 significant decimal digits for float8 values, and at most 9 digits for float4 values.

Character or string Types

Data Type	Description	Example Columns
CHAR(N)	Fixed length string of length N	Gender, State
VARCHAR(N)	Varying length string of maximum length N	Name, Email
TEXT	Varying length string with no maximum length	Comments, Reviews

SQL defines two primary character types: character varying(n) and character(n), where n is a positive integer. Both of these types can store strings up to n characters (not bytes) in length. An attempt to store a longer string into a column of these types will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length. (This somewhat bizarre exception is required by the SQL standard.) If the string to be stored is shorter than the declared length, values of type character will be space-padded; values of type character varying will simply store the shorter string.

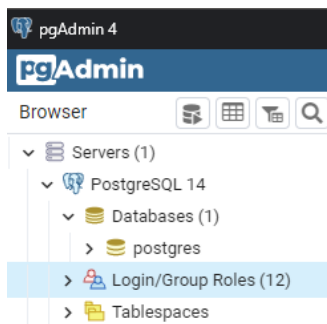
Data Type	Description	Example Columns
TIME	HH:MM:SS	
DATE	YYYY-MM-DD	Date of Birth
TIMESTAMP	YYYY-MM-DD HH:MM:SS	Order Time

Data Type	Description	Example Columns
BOOLEAN	True or False	In Stock
ENUM	A list of values input by the user	Gender

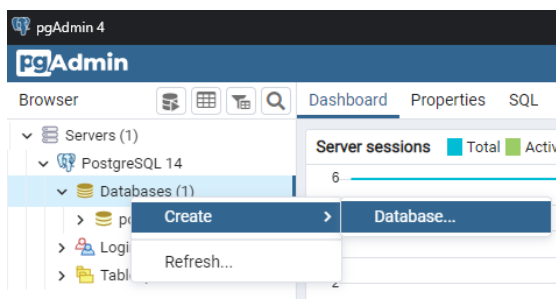
Creating database

The first test to see whether you can access the database server is to try to create a database. A running PostgreSQL server can manage many databases. Typically, a separate database is used for each project or for each user.

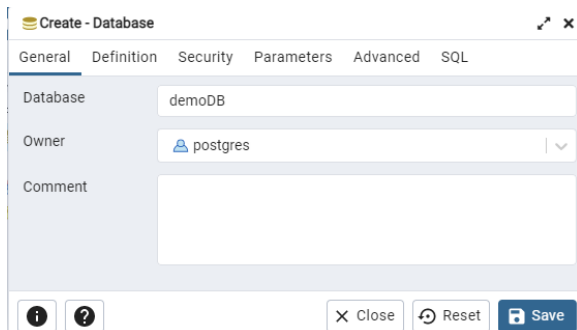
To create a new database, in this example named **demoDB**, we can create by login-in to PostgreSQL



From the left of the Browser column, go and right-click on Databases icon, move the mouse to hover the Create link and click on Database...

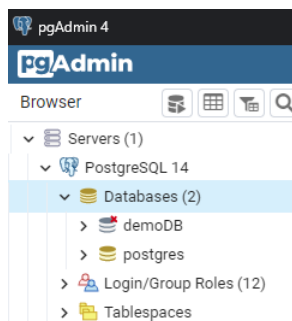


The Create-Database window appears, on General, you can fill up with the following info:

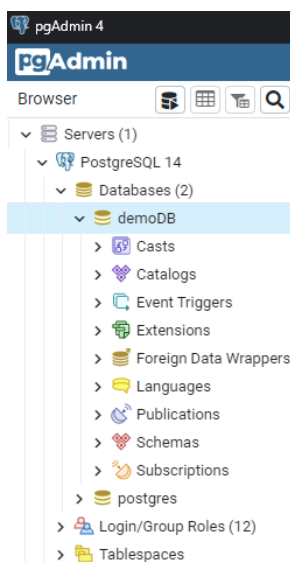


The image shows the 'Create - Database' dialog box in pgAdmin 4. It has tabs for General, Definition, Security, Parameters, Advanced, and SQL. The General tab is active. The 'Database' field contains 'demoDB'. The 'Owner' field shows a user icon and 'postgres'. The 'Comment' field is empty. At the bottom, there are buttons for 'Close', 'Reset', and 'Save'.

And click on Save. When you clicked on Save, the new database **demoDB** appears on the left column as a database:

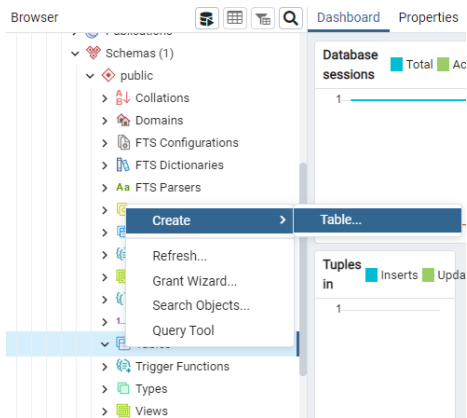


If you click on the new database demoDB, it expands and show the items that we can work with the database:

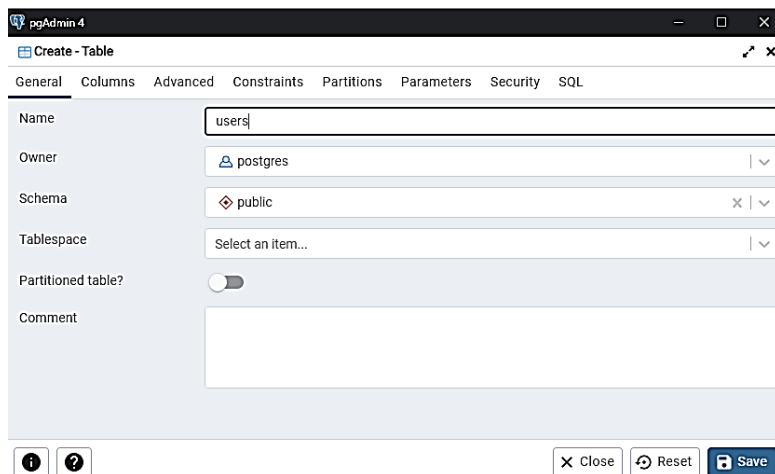


Create a table in PostgreSQL

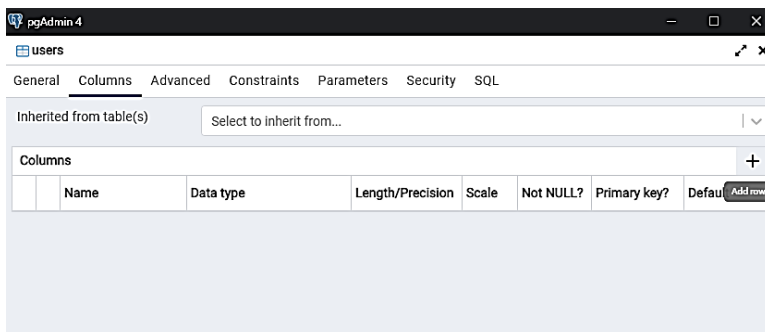
Click on the Schema icon to open and dropdown the Schema list. Once in Schema, go to Tables and right-click and select new table:



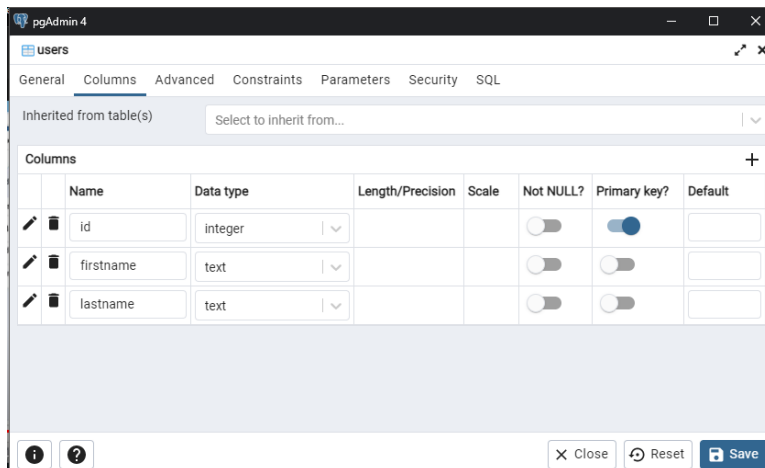
On the table, on General, we can name our table **users**:



We can also go the Columns tab and click on the plus icon to add columns to our table users. We create an ID column, as an integer value, and set as the primary key:

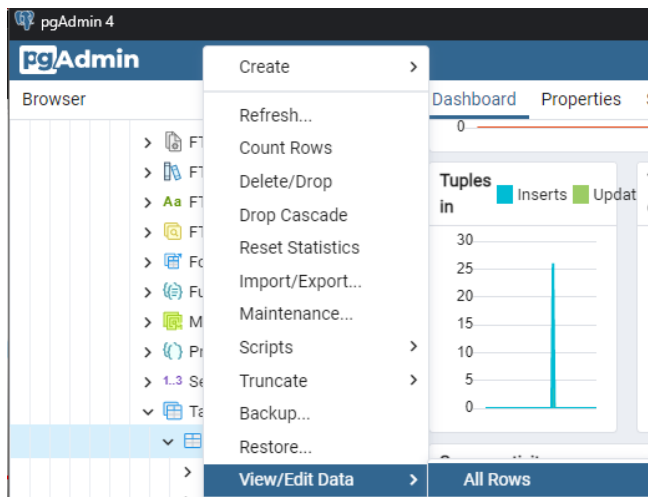


Let us to create those three column:

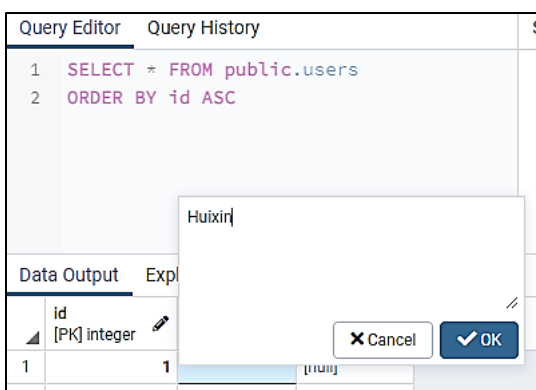


Once we finish we our table users, we can click on Save.

We can add data manually or import data from outside the database. To add data manually, we can right click on table users, go to View/Edit data, and click on All Rows:



When you do so, the table appears and on the bottom of the table, we can add data manually. To input the data to **firstname** and **lastname**, we double-click on the field and type a text in the dialog box :



Using SQL statement

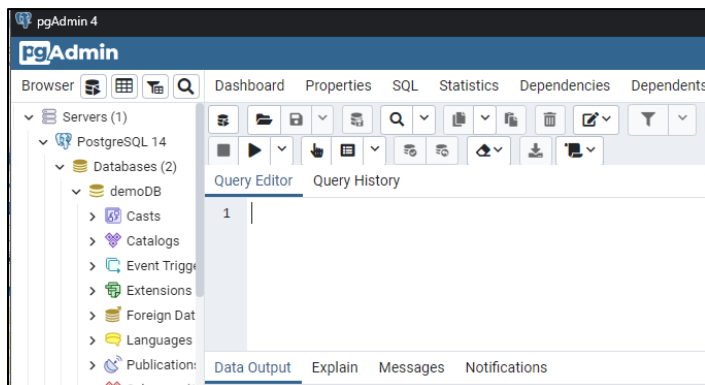
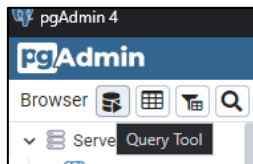
SQL CREATE TABLE

Create a table into our database

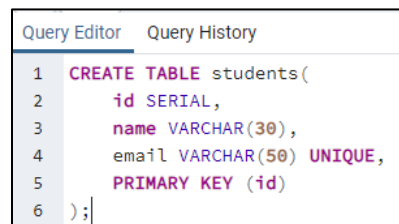
Syntax:

```
CREATE TABLE table's name(  
    ...  
);
```

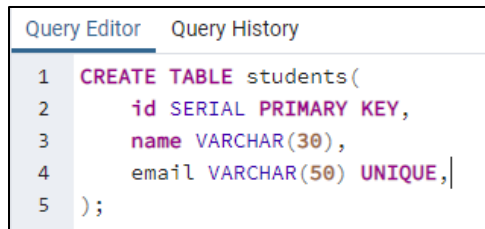
We can create a table using SQL statement. For this, we can go to the Query Tool to open the Query Editor



Let us create a table *students* with three columns: id, name, and email. The **id** is going to be the primary key and it will automatically increment; the name accepts strings up to 30 characters; and an email that will set as unique entry up to 50 characters.



An alternative to create set the PRIMARY KEY is:



```
1 CREATE TABLE students(  
2     id SERIAL PRIMARY KEY,  
3     name VARCHAR(30),  
4     email VARCHAR(50) UNIQUE,  
5 );
```

SQL INSERT

The INSERT statement inserts a row into a table

Syntax

INSERT INTO *table's name* (*column(s)'s name*) **VALUES** (*values to columns*);

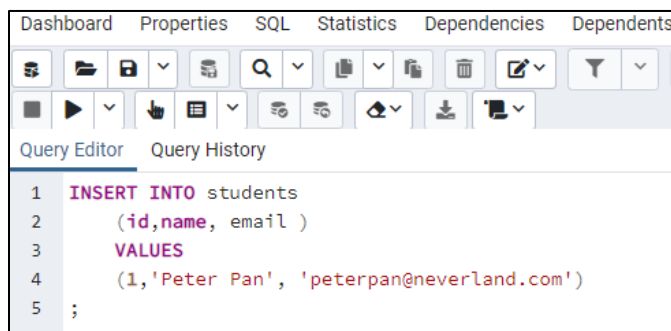
At the Query Editor, we can create manipulate our database data using SQL statement.

Let us create a table named **students** with **id**, **name**, and **email address**, as:

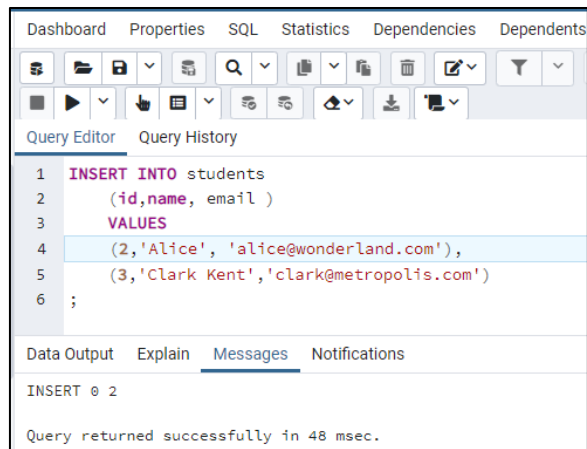
```
INSERT INTO students(id,name, email ) VALUES (1,'Peter Pan', peterpan@neverland.com');
```

Note: VARCHAR uses single quote for the values

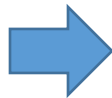
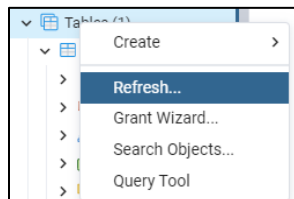
To add multiple values, we have separate each set of values using a comma.



```
1 INSERT INTO students  
2     (id,name, email )  
3     VALUES  
4     (1,'Peter Pan', 'peterpan@neverland.com')  
5 ;
```



To view the table in our database, we can go to the left column, right-click on **Table** icons, and click on **Refresh**:



Data Output	Explain	Messages	Notifications
id	[PK] integer	name	email
1	1	Peter Pan	peterpan@neverland.com
2	2	Alice	alice@wonderland.com
3	3	Clark Kent	clark@metropolis.com

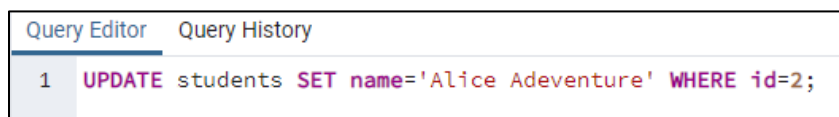
SQL UPDATE...SET... WHERE

UPDATE allows the updating of a field with a WHERE clause

Syntax

UPDATE *table's name* **SET** *column's name* = 'values' **WHERE** *column = value*

If we want to update one data from the table, we can use the statement below. For example, if we want to change/update the name Alice to Alice Adventure, we can write the line as:



Refreshing the table we will have:

id	name	email
[PK] integer	character varying (128)	character varying (128)
1	Peter Pan	peterpan@neverland.com
2	Alice Adeventure	alice@wonderland.com
3	Clark Kent	clark@metropolis.com

SQL DELETE

DELETE deletes a row in a table based on selection criteria

Syntax

DELETE FROM *table's name* **WHERE** *column_name = value;*

For example, let us delete the third student of our table *students*

[Query Editor](#) [Query History](#)

```
1 DELETE FROM students WHERE id=3;
```

If we refresh the table, this is we have:

	id [PK] integer	name character varying (128)	email character varying (128)
1	1	Peter Pan	peterpan@neverland.com
2	2	Alice Adeventure	alice@wonderland.com

SQL DROP

DROP keyword is used to delete different items in a database. DROP can be used to delete a column, a unique constraint, a primary key, foreign key, a check constraint, a default constraint, an index, a database, a table, or a view.

DROP COLUMN

The **DROP COLUMN** command is used to delete a column in an existing table. The following SQL deletes the "ContactName" column from the "Customers" table:

```
ALTER TABLE Customers  
DROP COLUMN ContactName;
```

DROP CONSTRAINT

To drop a UNIQUE constraint, use the following SQL:

```
ALTER TABLE Persons  
DROP INDEX UC_Person;
```

DROP PRIMARY KEY

```
ALTER TABLE Persons  
DROP PRIMARY KEY;
```

DROP FOREIGN KEY

```
ALTER TABLE Orders  
DROP FOREIGN KEY FK_PersonOrder;
```

DROP A CHECK CONSTRAINT

```
ALTER TABLE Persons  
DROP CHECK CHK_PersonAge;
```

DROP DEFAULT

The **DROP DEFAULT** command is used to delete a DEFAULT constraint. To drop a DEFAULT constraint, use the following SQL:

```
ALTER TABLE Persons  
ALTER City DROP DEFAULT;
```

DROP INDEX

The **DROP INDEX** command is used to delete an index in a table.

```
DROP INDEX table_name.index_name;
```

DROP DATABASE

The **DROP DATABASE** command is used to delete an existing SQL database. The following SQL drops a database named "testDB":

```
DROP DATABASE testDB;
```

DROP TABLE

The **DROP TABLE** command deletes a table in the database. The following SQL deletes the table "Shippers":

```
DROP TABLE Shippers;
```

DROP VIEW

The **DROP VIEW** command deletes a view. The following SQL drops the "Brazil Customers" view:

```
DROP VIEW [Brazil Customers];
```


Relational Database

These are by far the most popular type of databases around today, the most popular databases such as Oracle, MySQL or SQL server, these are all relational databases. PostgreSQL is also a relational database.

In a relational database, data is stored in tables which are referred to as relations. The tables are just like tables in excel. They contain columns and rows of data these tables can be linked together in relationships.

It is these relationships between different tables in the database that makes relational databases so powerful they allow us to identify a piece of data in relation to another piece of data within the database. For example, there are two tables as below:

PETS

id integer	species character varying (30)	full_name character varying (30)	age integer	owner_id integer
1	Dog	Rex	6	1
2	Rabbit	Fluffy	2	5
3	Cat	Tom	8	2
4	Mouse	Jerry	2	2
5	Dog	Biggles	4	1
6	Tortoise	Squirtle	42	3

OWNERS

id integer	first_name character varying (30)	last_name character varying (30)	city character varying (30)	state character (2)
1	Samuel	Smith	Boston	MA
2	Emma	Johnson	Seattle	WA
3	John	Oliver	New York	NY
4	Olivia	Brown	San Francisco	CA
5	Simon	Smith	Dallas	TX

These two tables are linked and are set to be in a database relationship from these two columns. We can match the data and see who owns which pets.

Relational Database Management Systems (RDBMS)

When people talk about databases they are normally talking about relational database management systems, the popular databases such as Oracle, MySQL, SQL Server and PostgreSQL are actually relational database management systems or RDBMS. RDBMS allows us to interact with the relational database they provide tools and a gooey or graphical user interface to interact with the database PostgreSQL is the relational database management system we will be using for this course and it is the most powerful and popular open source database in the world.

SQL stands for Structured Query Language and it is the language used to talk or interact with relational databases. We can write queries using SQL to create tables within the database as well as insert and modify data and even retrieve data from a database and much more as well.

The syntax for SQL is very similar across the different database systems as well. What we learn in this course won't just apply to PostgreSQL, but can also be used for MySQL, Oracle and other relational databases.

Three kinds of keys

Primary key: generally an integer auto-increment field

Foreign key: a `foreign key` is a field (or collection of fields) in one table, that refers to the [PRIMARY KEY](#) in another table.

Logical key: logical keys are anything that define relationships between data and tables. For example, a logical key could be a **pet_species**.

Primary key rules

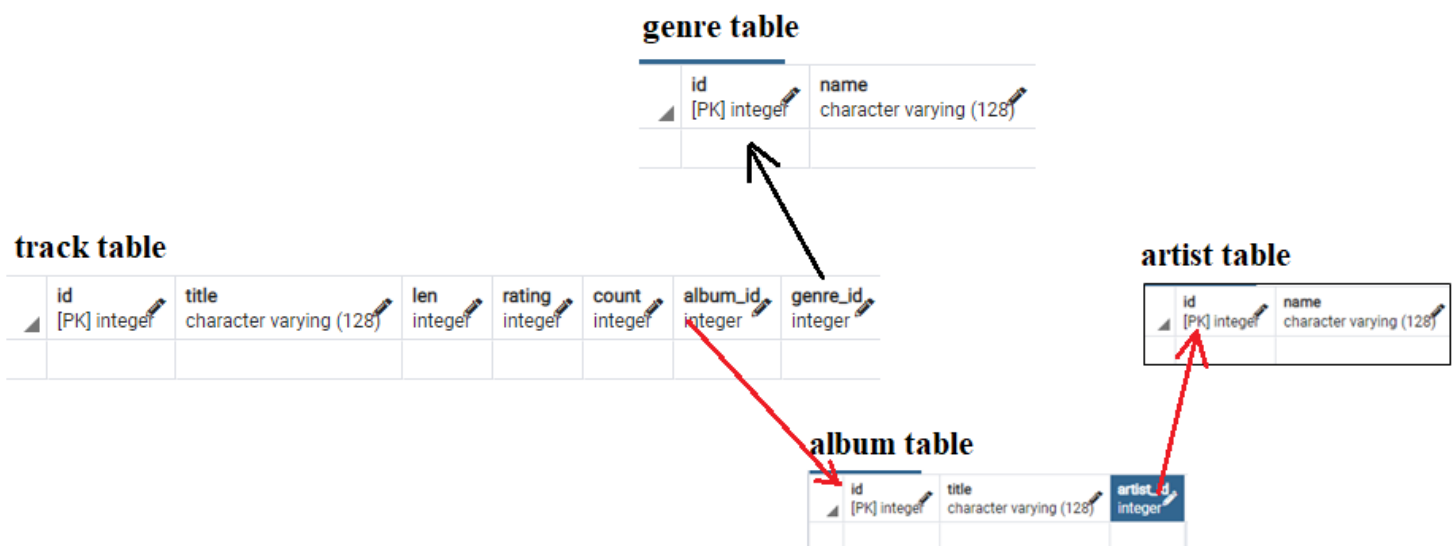
A primary key is the column that contains values that are uniquely identified in each row in a table.

Best practices:

- Never use your logical key as the primary key
- Logical keys can do change, slowly!
- Relationships that are based on matching string fields are less efficient than integers.

Building tables

We can create a relationship between two tables by REFERENCES to the next table. REFERENCES refers to a foreign key. We have to create the tables first in order to make it into foreign key.



artist table

id	name
[PK] integer	character varying (128)

genre table

id	name
[PK] integer	character varying (128)

In order to make a connection, between the album and the artist table, we use the REFERENCES keyword. The **references** keyword is used to define which table and column is used in a foreign key relationship.

Syntax:

Name_Foreign_key key_value REFERENCES name_foreign_table(primary_key_name);

For our schema, let us say that we want to link **album** table to the **artist** table:

album table

id	title	artist_id
[PK] integer	character varying (128)	integer

artist table

id	name
[PK] integer	character varying (128)

for this, after creating the table artist, we can create the album table and have a foreign key to artist table:

```
Query Editor  Query History
1  CREATE TABLE album(
2      id SERIAL PRIMARY KEY,
3      title VARCHAR(128),
4      artist_id INTEGER REFERENCES artist(id) ON DELETE CASCADE
5  );
```

Now, let us create a **track** table with a foreign key to **album** and **genre** table

```
5  CREATE TABLE track(
6      id SERIAL PRIMARY KEY,
7      title VARCHAR(128),
8      len INTEGER,
9      rating INTEGER,
10     count INTEGER,
11     album_id INTEGER REFERENCES album(id) ON DELETE CASCADE,
12     genre_id INTEGER REFERENCES genre(id) ON DELETE CASCADE,
13 );
```

Refreshing the **track** table we have:

	id [PK] integer	title character varying (128)	len integer	rating integer	count integer	album_id integer	genre_id integer

INSERTING DATA

Query Editor Query History

```
1 INSERT INTO genre(id,name) VALUES(1,'Rock');
2 INSERT INTO genre(id,name) VALUES(2,'Metal');
3
4 INSERT INTO artist(id, name) VALUES(1, 'Led Zeppelin');
5 INSERT INTO artist(id, name) VALUES(2, 'AC/DC');
6
7 INSERT INTO album (id, title, artist_id) VALUES(1, 'Who made who',2);
8 INSERT INTO album (id, title, artist_id) VALUES(2, 'IV', 1);
```

Refreshing our tables, we have:

genre table

	id [PK] integer	name character varying (128)
1	1	Rock
2	2	Metal

artist table

	id [PK] integer	name character varying (128)
1	1	Led Zeppelin
2	2	AC/DC









album table

	id [PK] integer	title character varying (128)	artist_id integer
1	1	Who made who	2
2	2	IV	1

Now, let us insert some values to **track** album

```
Query Editor  Query History
1  INSERT INTO track(title, rating, len, count, album_id, genre_id)
2    VALUES ('Black Dog', 5, 297,0,2,1);
3  INSERT INTO track(title, rating, len, count, album_id, genre_id)
4    VALUES ('Stairway',5,482,0,2,1);
5
6  INSERT INTO track(title, rating, len, count, album_id, genre_id)
7    VALUES ('About to Rock',5,313,0,1,2);
8  INSERT INTO track(title, rating, len, count, album_id, genre_id)
9    VALUES ('Who MAde who',5,207,0,1,2);
10
```

Refreshing the **track** table

Data Output		Explain	Messages	Notifications			
	id [PK] integer 	title character varying (128) 	len integer 	rating integer 	count integer 	album_id integer 	genre_id integer 
1	4	Black Dog	297	5	0	2	1
2	5	Stairway	482	5	0	2	1
3	6	About to Rock	313	5	0	1	2
4	7	Who MAde who	207	5	0	1	2

JOIN clause

The SQL JOIN clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

Syntax:

SELECT table_column_name, table_column_name **FROM** table_name **JOIN** foreign_table_name **ON** table_foreign_key_name = foreign_table_primary_key_name

For example, we can display the name of the album title with the artist name in one table using JOIN as:

```
Query Editor  Query History
1  SELECT album.title, artist.name FROM album JOIN artist ON album.artist_id = artist.id;
```

The resulting table will be:

	title character varying (128)	name character varying (128)
1	Who made who	AC/DC
2	IV	Led Zeppelin