

FreeCodeCamp - Learn PostgreSQL

Tutorial - Full Course for Beginners

1. Setting Up Tools & Introduction:

Download PostgreSQL | Install PostgreSQL | How to connect to a database | Database GUI clients | connect to database using SQL shell command line | connect to database using PgAdmin 4 GUI client

2. Basic Database Commands:

Terminal Help | PSQL mode | list databases command | create database command | connect to remote database | Change Database command | Drop database | drop database with force

3. Basic Table Commands:

create table without constraints | create table with constraints | primary key | auto increment | drop table | describe table | describe database

4. Inserting data into Tables:

insert into command | generate test sql data using mockaroo | \i command (import from file) | SELECT * FROM table

5. Select from Queries with clauses:

order by desc and asc | distinct | where | and | or | comparison operators >, >=, <, <= | limit | offset | fetch | in keyword | between keyword | like operator with wildcard & underscore matching | ilike matching

6. Aggregate functions:

Group by | count | having keyword | min, max, avg and sum functions

7. Mathematical & Arithmetic Operators:

mathematical operators (addition, subtraction, division, multiplication) | arithmetic Operators (power of, factorial, modulus) | alias as keyword | Handling null values with coalesce, nullif

8. Timestamp & Date:

datetime | date | now function | casting now function to date or time | using interval to perform calculations on datetime | extract part of datetime | age function

9. Primary Key & Constraints:

Primary key | drop pkey | add pkey | unique constraints | check constraints

10. Deleting and Updating Rows:

delete from table | delete by primary key | delete multiple rows by non primary key | delete with where clause | on conflict do nothing | upsert |

11. Joins:

joins and relationship theory | adding FK column | updating FK column | inner joins theory | inner joins query | left join theory | left join query examples | Deletion where a Join FK exists

12. Export a Query to CSV:

export a select query to csv

13. Serial and Sequences:

explanation of sequences | reset a sequence

14. Extensions:

about PostgreSQL extensions | understanding UUID Data Type | install UUID-OSSP extension | generate globally unique ID using UUIDv4 | build SQL tables & insert queries Using UUID | import from SQL file with UUID's | update FK Columns with UUIDs | join using keyword

1. PostgreSQL Setting Up & Introduction

[1.1 Download postgreSQL](#)

[1.2 Install postgreSQL](#)

[1.3 Launch PostgreSQL](#)

[1.4 Database GUI clients](#)

[1.5 Connect to Database using Command Line Terminal](#)


[1.6 Connect to Database using PgAdmin 4 GUI Client](#)

1.1 Download postgreSQL

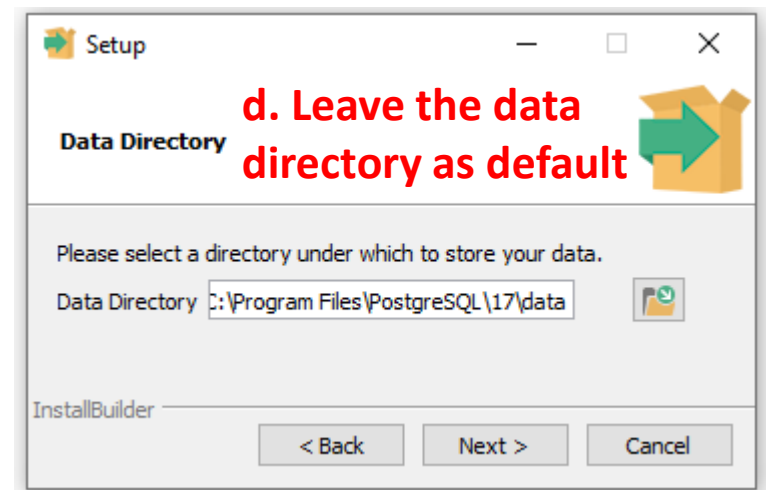
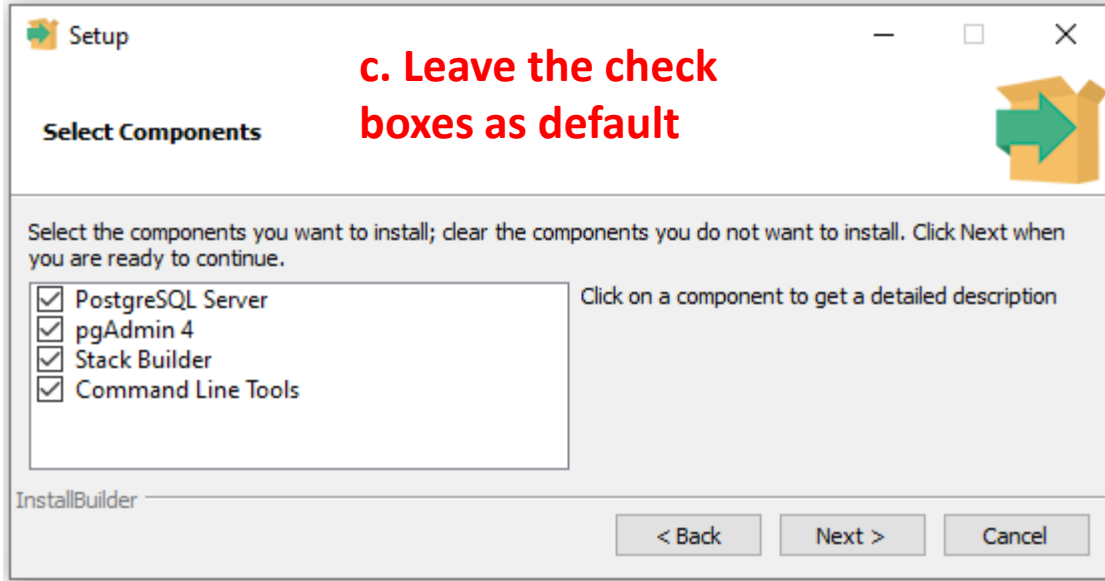
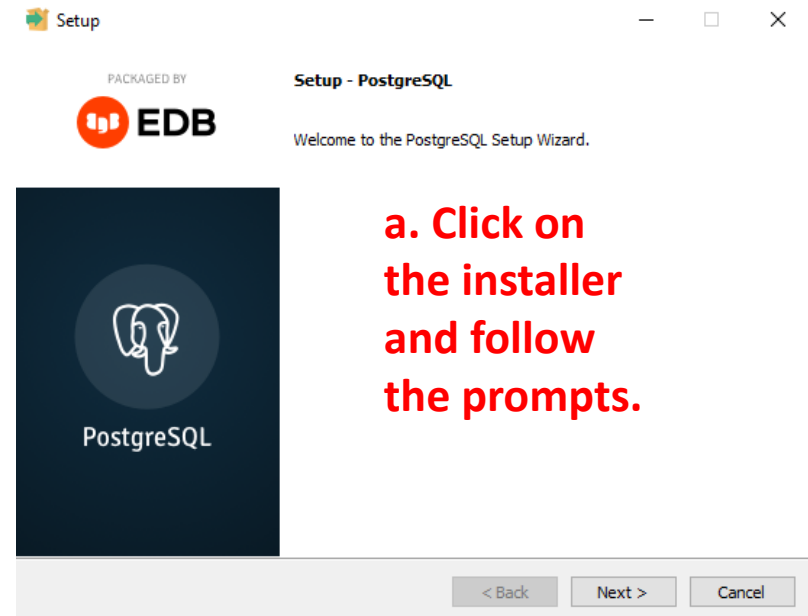
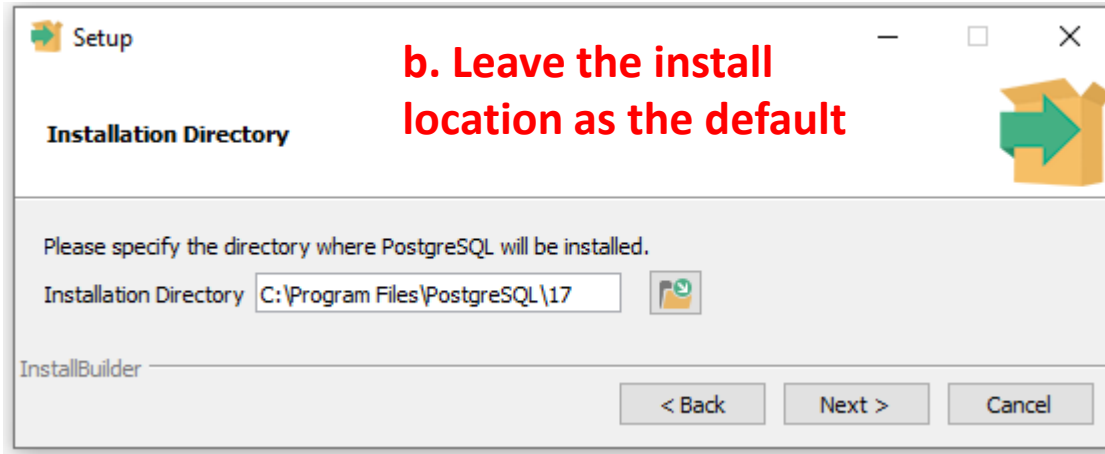
Go to <https://www.postgresql.org> and select download then select the operating system family.



Under **Interactive installer by EDB** is a link <https://www.enterprisedb.com/downloads/postgres-postgresql-downloads> to the downloads page. Click on the download icon for the latest version.

Download PostgreSQL					
Open source PostgreSQL packages and installers from EDB					
PostgreSQL Version	Linux x86-64	Linux x86-32	Mac OS X	Windows x86-64	Windows x86-32
17.2	postgresql.org 	postgresql.org 			Not supported
16.6	postgresql.org 	postgresql.org 			Not supported

1.2 Install postgreSQL



Setup

Password

Please provide a password for the database superuser (postgres).

Password

Retype password

e. Define and remember a superuser password that will be used to connect to the database

InstallBuilder

< Back Next > Cancel

Setup

Port

Please select the port number the server should listen on.

Port

f. Leave the port as default

InstallBuilder

< Back Next > Cancel

Setup

Advanced Options

Select the locale to be used by the new database cluster.

Locale

g. Leave the locale as default

InstallBuilder

< Back Next > Cancel

Setup

Pre Installation Summary

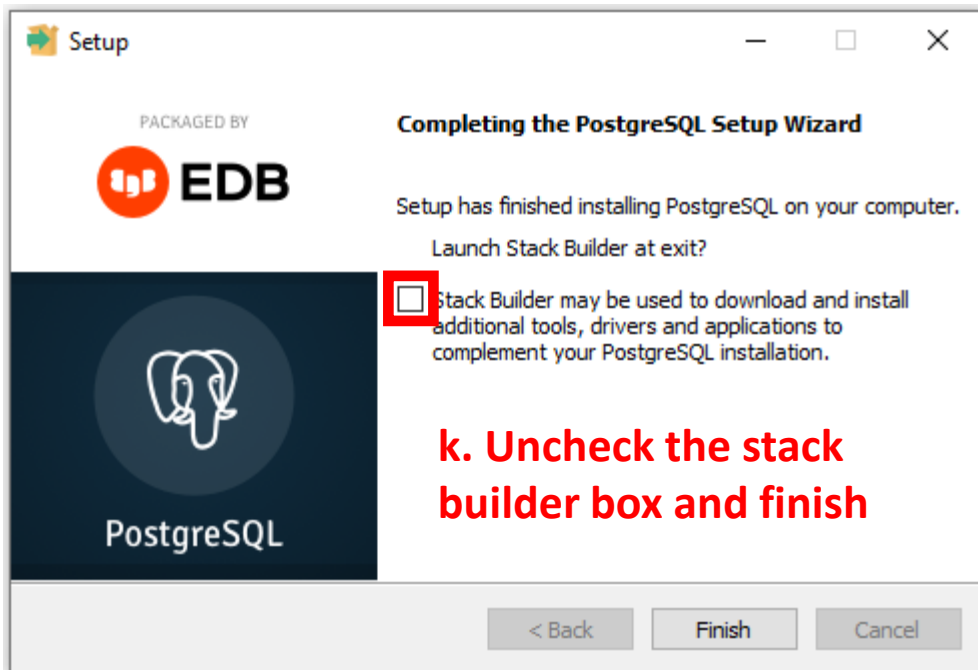
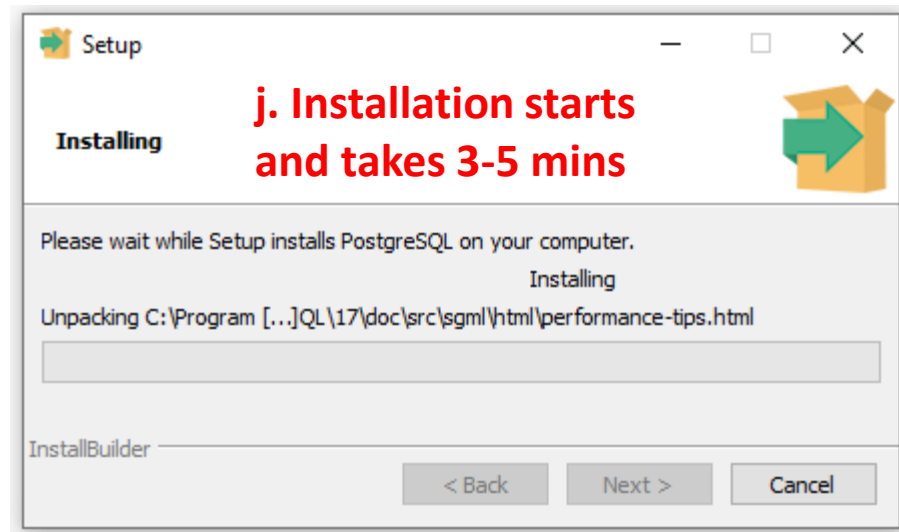
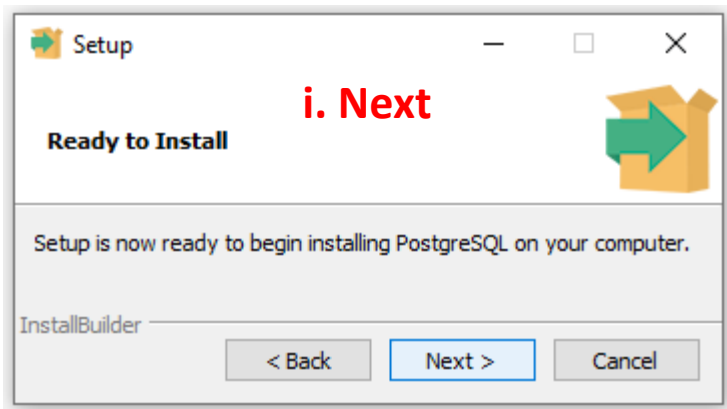
The following settings will be used for the installation::

- Installation Directory: C:\Program Files\PostgreSQL\17
- Server Installation Directory: C:\Program Files\PostgreSQL\17
- Data Directory: C:\Program Files\PostgreSQL\17\data
- Database Port: 5432
- Database Superuser: postgres
- Operating System Account: NT AUTHORITY\NetworkService
- Database Service: postgresql-x64-17
- Command Line Tools Installation Directory: C:\Program Files\PostgreSQL\17
- pgAdmin 4 Installation Directory: C:\Program Files\PostgreSQL\17\pgAdmin 4
- Stack Builder Installation Directory: C:\Program Files\PostgreSQL\17
- Installation Log: C:\Users\jello\AppData\Local\Temp\install-postgresql.log

h. Review installation settings and select next

InstallBuilder

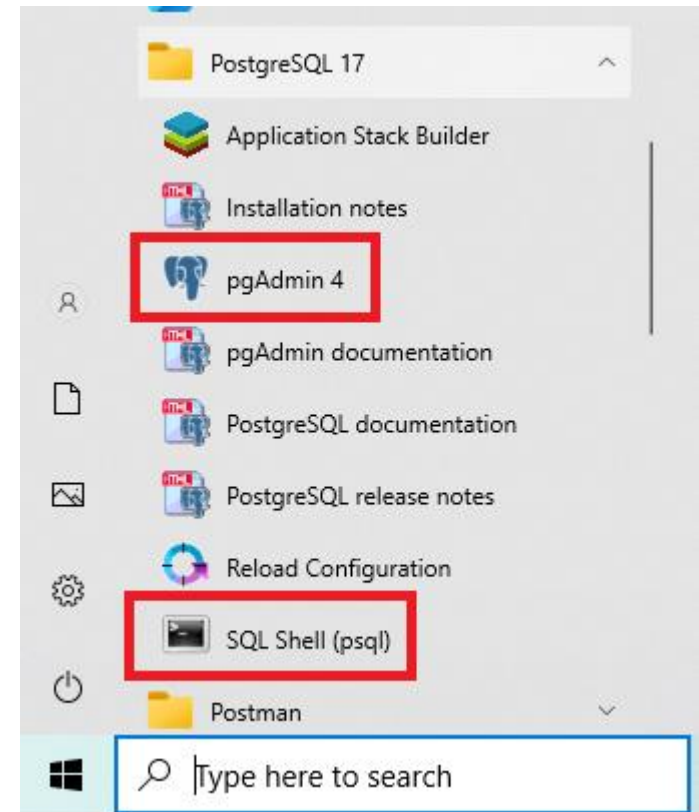
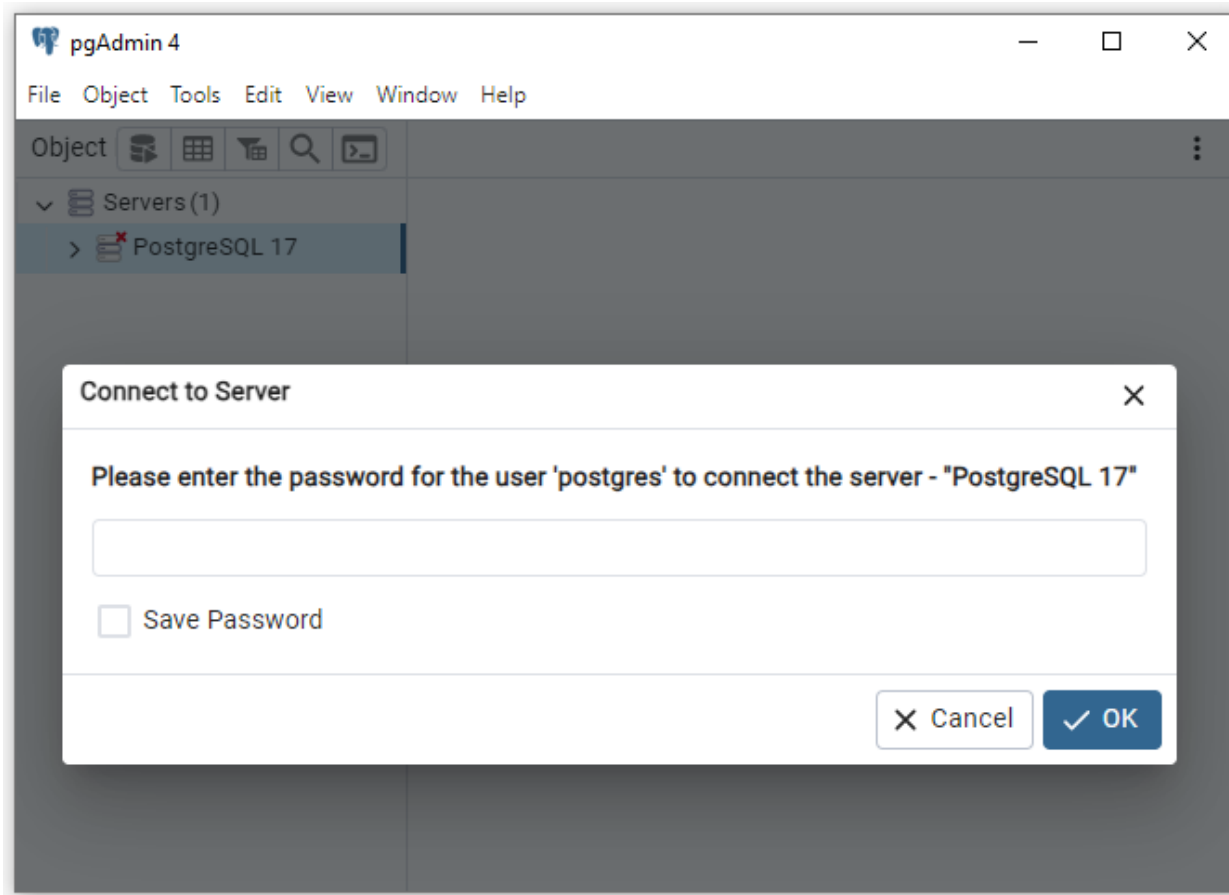
< Back Next > Cancel



1.3 Launch PostGreSQL

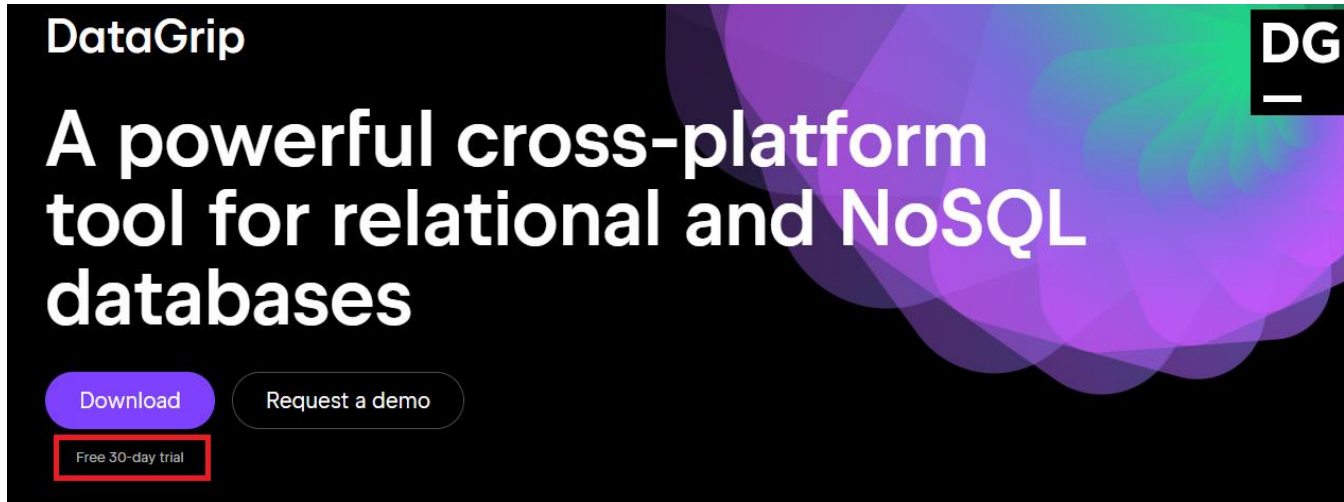
In windows start there is now the PostGreSQL folder with the applications that we are going to use:

1. **pgAdmin 4** – the graphical user interface (GUI Client)
2. **SQL Shell** – the command line interface



When connecting to pgAdmin 4 GUI client, we are prompted for the superuser password defined in the setup.

1.4 Database GUI clients



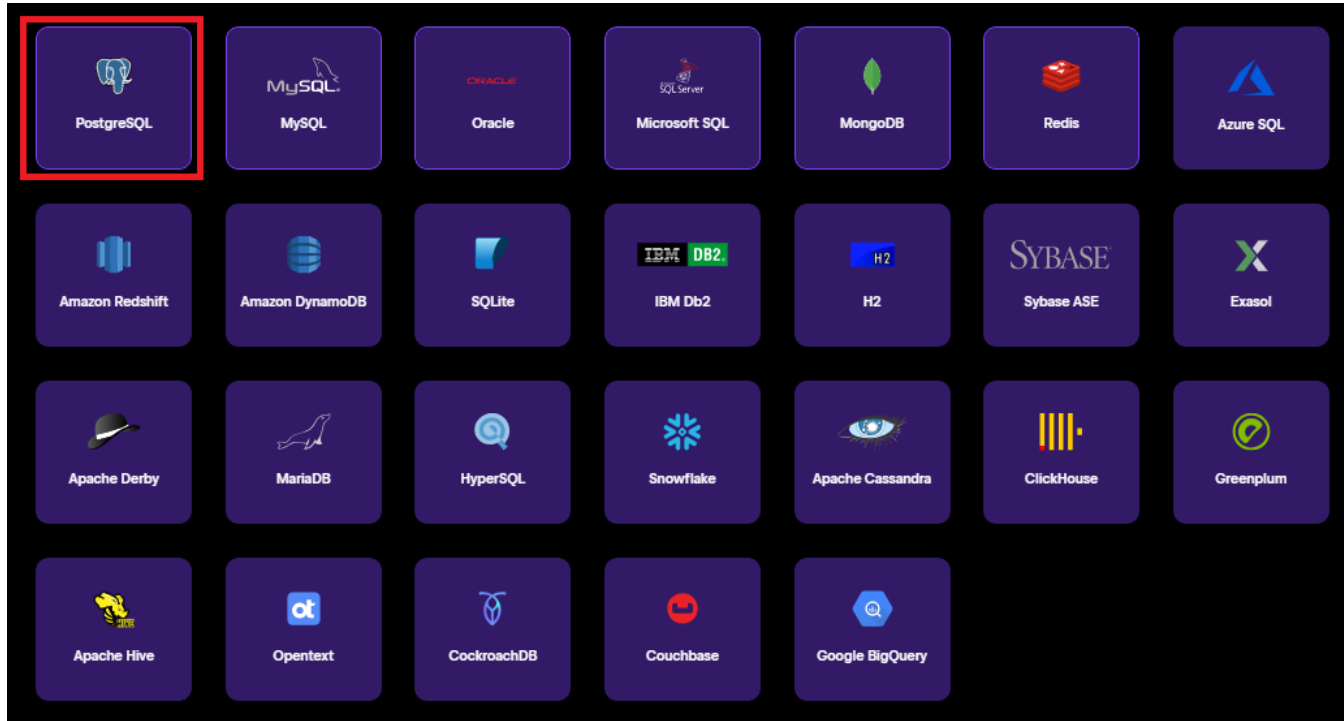
DataGrip

A powerful cross-platform tool for relational and NoSQL databases

Download Request a demo

Free 30-day trial

DG



PostgreSQL	MySQL	Oracle	Microsoft SQL Server	MongoDB	Redis	Azure SQL
Amazon Redshift	Amazon DynamoDB	SQLite	IBM Db2	H2	Sybase ASE	Exasol
Apache Derby	MariaDB	HypersQL	Snowflake	Apache Cassandra	ClickHouse	Greenplum
Apache Hive	Opentext	CockroachDB	Couchbase	Google BigQuery		

PgAdmin 4 is a GUI client for connecting to the database and is not the best out there. Other examples of a GUI client include **DataGrep** which is a **licensed service** costing from 99 euros per year depending on the use. DataGrep comes with a **30 day trial**.

DataGrep can connect to any of the database programs show to the left including postGreSQL.

Postico is a **Free GUI Client** for postgreSQL but only runs on MAC.

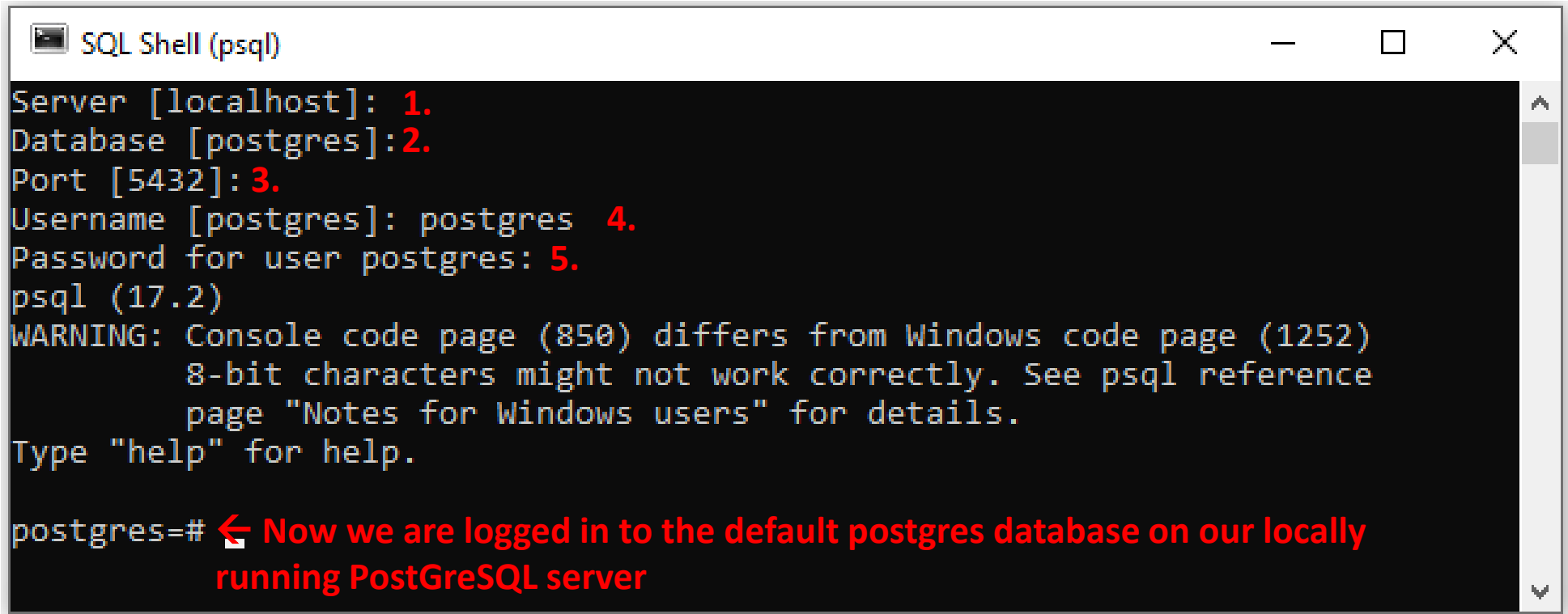


Postico 2

The native Mac app for PostgreSQL
all new again

PgAdmin 4 is the **free windows GUI Client** but is not as nice as DataGrep

1.5 Connect to Database using Command Line Terminal

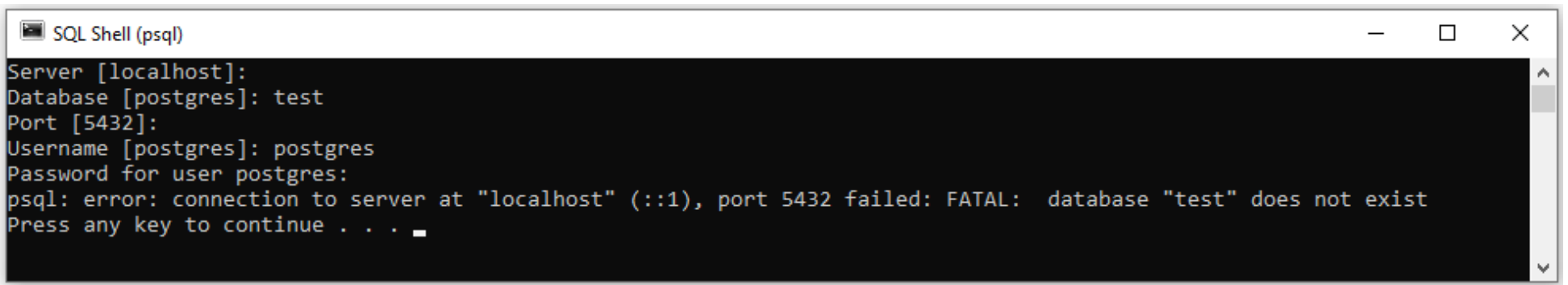


```
SQL Shell (psql)
Server [localhost]: 1.
Database [postgres]: 2.
Port [5432]: 3.
Username [postgres]: postgres 4.
Password for user postgres: 5.
psql (17.2)
WARNING: Console code page (850) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

postgres=#
```

← Now we are logged in to the default postgres database on our locally running PostgreSQL server

1. From the SQL shell terminal, the prompt **Server [localhost]:** is where we would enter the IP address of the remote server. The server is local on our machine so we can just press enter.
2. PostgreSQL has a default database called postgres so we can just press enter. Here we would enter the database name if we wanted to connect to another database.
3. Specify the port. We can just press enter to leave this as the default of 5432
4. Specify the username. PostgreSQL default username is postgres
5. Password prompt. This is the superuser password defined during setup



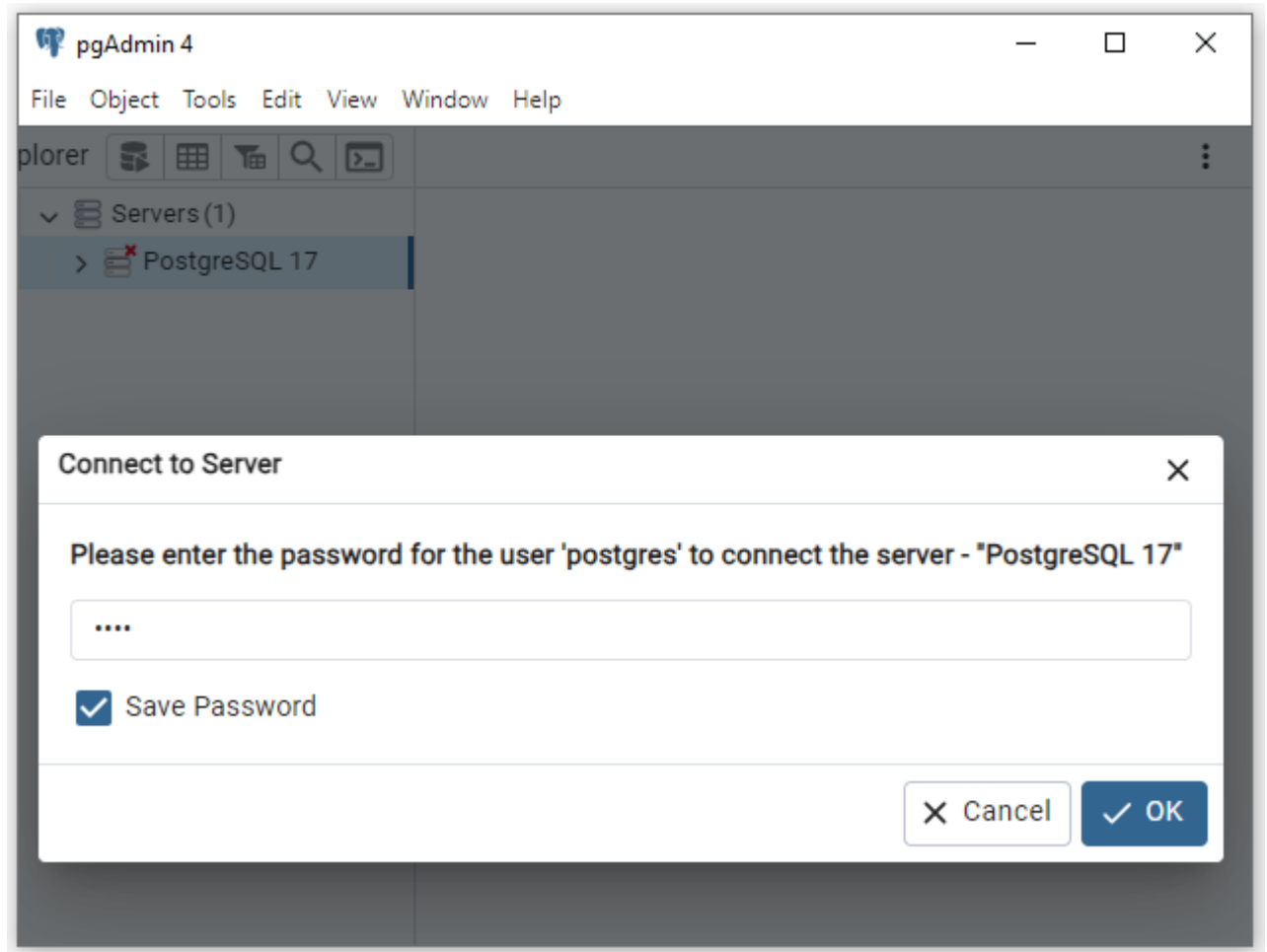
```
SQL Shell (psql)
Server [localhost]:
Database [postgres]: test
Port [5432]:
Username [postgres]: postgres
Password for user postgres:
psql: error: connection to server at "localhost" (:::1), port 5432 failed: FATAL:  database "test" does not exist
Press any key to continue . . .
```

If I try to connect to a database that does not exist, i.e test, I get a fatal error saying that the database does not exist.

1.6 Connect to Database using PgAdmin 4 GUI Client

When connecting to pgAdmin 4 GUI client, we are prompted for the superuser password defined in the setup.

I can use the **save password checkbox** so that I do not have to enter the password each time I open PgAdmin 4.



pgAdmin 4

File Object Tools Edit View Window Help

Object Explorer

Servers (1)

PostgreSQL 17 1.

Databases (1)

postgres 2.

Casts

Catalogs

Event Triggers

Extensions

Foreign Data Wrappers

Languages

Publications

Schemas

Subscriptions

Login/Group Roles

Tablespaces

1. PgAdmin 4 is showing the default server of PostgreSQL 17

2. The PostgreSQL 17 server has the one database called postgres

2. Basic Database Commands

[2.1 Help & PSQL mode](#)

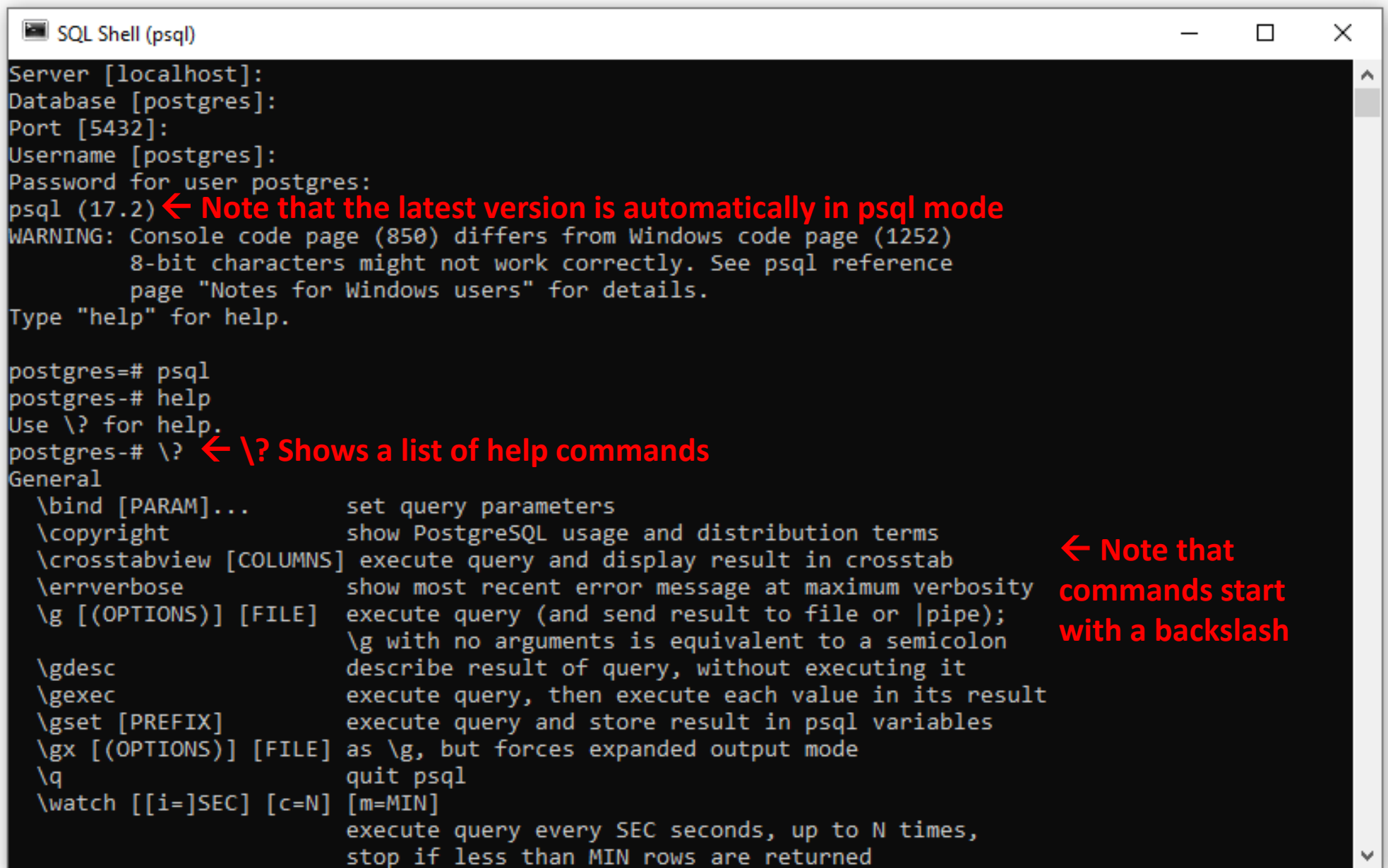
[2.2 List Databases & Create database](#)

[2.3 Connect to Database & Change Database](#)

[2.4 Drop Database & Drop Database with force](#)

[2.5 PostgreSQL Basic Database Commands review](#)

2.1 Help & PSQL mode



```
SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Password for user postgres:
psql (17.2) ← Note that the latest version is automatically in psql mode
WARNING: Console code page (850) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

postgres=# psql
postgres=# help
Use \? for help.
postgres=# \? ← \? Shows a list of help commands
General
 \bind [PARAM]...      set query parameters
 \copyright             show PostgreSQL usage and distribution terms
 \crosstabview [COLUMNS] execute query and display result in crosstab
 \errverbose           show most recent error message at maximum verbosity
 \g [(OPTIONS)] [FILE] execute query (and send result to file or |pipe);
                       \g with no arguments is equivalent to a semicolon
 \gdesc               describe result of query, without executing it
 \gexec              execute query, then execute each value in its result
 \gset [PREFIX]       execute query and store result in psql variables
 \gx [(OPTIONS)] [FILE] as \g, but forces expanded output mode
 \q                  quit psql
 \watch [[i=]SEC] [c=N] [m=MIN]
                       execute query every SEC seconds, up to N times,
                       stop if less than MIN rows are returned
```

← Note that commands start with a backslash

2.2 List Databases & Create database

A `\l` (forward slash l) command lists all the databases on the server. note that we have three default databases already.

```
postgres=# \l
```

Name	Owner	Encoding	Locale Provider	Collate	Ctype	Locale	ICU Rules	Access privileges
postgres	postgres	UTF8	libc	English_United Kingdom.1252	English_United Kingdom.1252			
template0	postgres	UTF8	libc	English_United Kingdom.1252	English_United Kingdom.1252			=c/postgres + postgres=CTc/postgres
template1	postgres	UTF8	libc	English_United Kingdom.1252	English_United Kingdom.1252			=c/postgres + postgres=CTc/postgres

(3 rows)

```
postgres=#
```

CREATE DATABASE <database name>; command will create a new database. Note that lowercase can be used but it is better to use uppercase to differentiate the command part. End with semi-colon or the command will not execute.

```
postgres=# CREATE DATABASE test;
CREATE DATABASE
postgres=# \l
```

Name	Owner	Encoding	Locale Provider	Collate	Ctype	Locale	ICU Rules	Access privileges
postgres	postgres	UTF8	libc	English_United Kingdom.1252	English_United Kingdom.1252			
template0	postgres	UTF8	libc	English_United Kingdom.1252	English_United Kingdom.1252			=c/postgres + postgres=CTc/postgres
template1	postgres	UTF8	libc	English_United Kingdom.1252	English_United Kingdom.1252			=c/postgres + postgres=CTc/postgres
test	postgres	UTF8	libc	English_United Kingdom.1252	English_United Kingdom.1252			

(4 rows)

```
postgres=#
```

2.3 Connect to Database & Change Database

```
Server [localhost]:
Database [postgres]: test
Port [5432]:
Username [postgres]:
Password for user postgres:
psql (17.2)
WARNING: Console code page (850) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

test=#
```

I can connect to the new database from SQL shell (psql)

```
C:\Program Files\PostgreSQL\17\bin>psql -h localhost -p 5432 -d postgres -U postgres
Password for user postgres:
psql (17.2)
WARNING: Console code page (850) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

postgres=#
```

I can connect to the new database from a command prompt specifying the -h (host), -p (port), -d (database) and -U (username)

```
test-# \c postgres
You are now connected to database "postgres" as user "postgres".
postgres-#
```

`\c` command can be used to change from one database to another on the server.

2.4 Drop Database & Drop Database with force

```
test=# DROP DATABASE test; 1.  
ERROR:  cannot drop the currently open database  
  
test=# \c postgres  
You are now connected to database "postgres" as user "postgres".  
  
postgres=# DROP DATABASE test; 2.  
ERROR:  database "test" is being accessed by other users  
DETAIL:  There is 1 other session using the database.  
postgres=#  
  
postgres=# DROP DATABASE test with (force); 3.  
DROP DATABASE  
postgres=#
```

1. DROP DATABASE command is used to **permanently delete** a database in an **irreversible manner**.

2. Note that I cannot delete a database when I am currently connected to it or there are any other sessions connected to that database.

3. DROP DATABASE <database_name> with (force) command is used to **forcefully delete** a database.

2.5 PostGreSQL Basic Database Commands review

\l List all databases on the PostgreSQL server.

\c <database_name>; Used to switch between databases on the PostgreSQL server

CREATE DATABASE <database_name>;

CREATE DATABASE animal_shelter;

CREATE DATABASE DogApp;

CREATE DATABASE Dog App; ❌

Database names cannot contain spaces but underscores are acceptable.

DROP database <name>;

Dropping a database will completely remove it, i.e. delete it and is irreversible.

DROP database <name> with (force);

Forcefully drop a database even if there are sessions connected.

3. Basic Table Commands

[3.1 Create a table & describe table](#)

[3.2 Create a table with Constraints](#)

[3.3 PostgreSQL Basic Table Commands review](#)

3.1 Create a table & describe table

The SQL command to create a table is identical to MySQL and MariaDB. For a list of datatypes see the PostgreSQL documentation

<https://www.postgresql.org/docs/current/datatype.html>

Note how the response is CREATE TABLE.

```
test=# CREATE TABLE person (  
test(# id INT,  
test(# first_name  
VARCHAR(50),  
test(# last_name  
VARCHAR(50),  
test(# gender VARCHAR(6),  
test(# date_of_birth  
DATE);CREATE TABLE  
test=#
```

The CREATE TABLE command can be written on multiple lines. Note how the prompt on the second line has an opening parenthesis and on the last line we close the parenthesis and add a semicolon to execute the command.

`\d` command will describe the current database, i.e. it will list all tables.

```
test=# \d  
List of relations  
Schema | Name | Type | Owner  
-----+-----+-----+-----  
public | person | table | postgres  
(1 row)
```

```
test=# \d person  
Table "public.person"  
Column | Type | Collation | Nullable | Default  
-----+-----+-----+-----+-----  
id | integer | | | |  
first_name | character varying(50) | | | |  
last_name | character varying(50) | | | |  
gender | character varying(6) | | | |  
date_of_birth | date | | | |  
  
test=#
```

`\d <table_name>` will describe the columns in a table.

```
test=# CREATE TABLE person  
(id INT, first_name  
VARCHAR(50), last_name  
VARCHAR(50), gender  
VARCHAR(6), date_of_birth  
DATE);  
CREATE TABLE  
test=#
```

3.2 Create a table with Constraints

DROP TABLE <table_name>; command is used to permanently delete a table and its contents.

```
test=# DROP TABLE person;
DROP TABLE
test=#
```

```
test=# CREATE TABLE person (
test(# Id BIGSERIAL NOT NULL PRIMARY KEY,
test(# first_name VARCHAR(50) NOT NULL,
test(# last_name VARCHAR(50) NOT NULL,
test(# gender VARCHAR(6) NOT NULL,
test(# date_of_birth DATE NOT NULL,
test(# email VARCHAR(50) );
CREATE TABLE
test=#
```

Now I recreate the table but this time I add some constraints to the columns. Note that [BIGSERIAL](#) is an autoincrementing integer 8 bytes long with a range of 1 to 9223372036854775807.

Note that because I used BIGSERIAL which is auto-incrementing the describe database shows two rows because the **id column for person table is auto_incrementing**

```
test=# \dt
      List of relations
Schema | Name   | Type  | Owner
-----+-----+-----+-----
public | person | table | postgres
(1 row)
test=#
```

```
test=# \d
      List of relations
Schema | Name           | Type      | Owner
-----+-----+-----+-----
public | person         | table     | postgres
public | person_id_seq  | sequence  | postgres
(2 rows)
test=#
```

To only show the table use **\dt** (describe tables)

```
test=# \d person
```

Table "public.person"				
Column	Type	Collation	Nullable	Default
id	bigint		not null	nextval('person_id_seq'::regclass)
first_name	character varying(50)		not null	
last_name	character varying(50)		not null	
gender	character varying(6)		not null	
date_of_birth	date		not null	
email	character varying(50)			

```
Indexes:
```

```
    "person_pkey" PRIMARY KEY, btree (id)
```


3.3 PostGreSQL Basic Table Commands review

`\d` Describe current database, i.e. list all tables

```
CREATE TABLE <table_name> (  
  <column1_name> DATA_TYPE,  
  column2_name DATATYPE,  
  column3_name DATATYPE  
);
```

Create table (without constraints) command syntax

```
CREATE TABLE <table_name> (  
  <column1_name> DATA_TYPE CONSTRAINT,  
  column2_name DATATYPE CONSTRAINT,  
  column3_name DATATYPE CONSTRAINT  
);
```

Create table (with constraints) command syntax

```
DROP TABLE <table_name>;
```

Permanently delete a table and it's contents.

4. Inserting Data into Tables

[4.1 Insert Records into Tables](#)

[4.2 Generate Bulk Test Data with Mockaroo](#)

[4.3 Modify SQL file in VsCode to add the constraints](#)

[4.4 Import an SQL file](#)

[4.5 Verify Contents of Table](#)

[4.6 PostgreSQL Basic insert Records Commands review](#)

4.1 Insert Records into Tables

```
test=# INSERT INTO person (  
test(# first_name,  
test(# last_name,  
test(# gender,  
test(# date_of_birth)  
test=# VALUES ('Anne', 'Smith', 'FEMALE', DATE '1988-01-09');  
INSERT 0 1  
test=#
```

```
test=# INSERT INTO person (  
test(# first_name,  
test(# last_name,  
test(# gender,  
test(# date_of_birth,  
test(# email)  
test=# VALUES ('Jake', 'Smith', 'MALE', DATE '1990-01-10',  
'jake@gmail.com');  
INSERT 0 1  
test=#
```

```
test=# INSERT INTO person (  
test(# first_name,  
test(# last_name,  
test(# gender,  
test(# date_of_birth)  
test=# VALUES ('Andrew', 'Jacob', 'MALE', DATE '1977-12-17');
```

```
test=# INSERT INTO person (  
test(# first_name,  
test(# last_name,  
test(# gender,  
test(# date_of_birth)  
test=# VALUES ('Julia', 'Bravo', 'FEMALE', DATE '2012-01-12');
```

The INSERT INTO command can be used to add a row of data. Note that because email column does not have a constraint we do not need to specify it. In this scenario Anne Smith does not have an email address.

The DOB column is a **DATE type** and not a string so we have to specify this and ensure we enter the date in the correct format of **YYYY-MM-DD**.

The id column is not specified because it is auto-increment.

Adding more test data

4.2 Generate Bulk Test Data with Mockaroo

The screenshot shows the Mockaroo website interface for generating bulk test data. The top navigation bar includes links for SCHEMAS, DATASETS, APIS, DATABASES, SCENARIOS, PROJECTS, FUNCTIONS, and a SIGN IN button. The main area displays a table of field configurations for a schema named 'person'.

Field Name	Type	Options
first_name	First Name	blank: 0 %
last_name	Last Name	blank: 0 %
email	Email Address	blank: 30 %
gender	Gender	blank: 0 %
date_of_birth	Datetime	06/01/2020 to 01/08/2025 format: yyyy-mm-dd blank: 0 %
country_of_birth	Country	blank: 0 %

Annotations on the image:

- 30% of the records will have a blank email address** (pointing to the 'email' row's blank percentage).
- Add another column called country_of_birth** (pointing to the 'country_of_birth' row).

At the bottom, the configuration for generating data is shown:

- # Rows: 1000
- Format: SQL
- Table Name: person
- ☒ include CREATE TABLE

Buttons at the bottom include: GENERATE DATA, PREVIEW, SAVE AS..., DERIVE FROM EXAMPLE..., and MORE.

www.mockaroo.com Search Ecosia

mockaroo SCHEMAS DATASETS APIS DATABASES NEW SCENARIOS PROJECTS FUNCTIONS SIGN IN UPGRADE NOW

Preview

```
create table person (  
  first_name VARCHAR(50),  
  last_name VARCHAR(50),  
  email VARCHAR(50),  
  gender VARCHAR(50),  
  date_of_birth DATE,  
  country_of_birth VARCHAR(50)  
);  
  
insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values ('Chase', 'Gaine of England', 'cgaineofengland', 'Male', '2024-01-10', 'England')  
insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values ('Wyatt', 'Ebi', null, 'Male', '2024-01-10', 'England')  
insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values ('Caryn', 'Oswal', 'coswal2@wisc.edu', 'Female', '2024-01-10', 'England')  
insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values ('Derrik', 'Witherington', null, 'Male', '2020-01-10', 'England')  
insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values ('Emylee', 'Gergolet', null, 'Female', '2022-01-10', 'England')  
insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values ('Harrietta', 'De Cristoforo', 'hdecristoforo5@amazon.de', 'Female', '2024-01-10', 'England')  
insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values ('Benjy', 'Rosingdall', 'brosingdall6@amazon.de', 'Male', '2024-01-10', 'England')  
insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values ('Bernardine', 'Normanville', 'bnormanville7@amazon.de', 'Female', '2024-01-10', 'England')  
insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values ('Pammie', 'Dwelly', 'pdwelly8@gravatar.com', 'Female', '2024-01-10', 'England')  
insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values ('Shanie', 'Sitwell', 'ssitwell9@indiegogo.com', 'Female', '2024-01-10', 'England')  
insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values ('Griffie', 'Androck', 'gandrocka@desdev.cn', 'Male', '2024-01-10', 'England')  
insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values ('Isaiah', 'Pude', 'ipudeb@desdev.cn', 'Male', '2024-01-10', 'England')  
insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values ('Orlan', 'Van der Kruis', 'ovanderkruisc@blogspot.com', 'Male', '2024-01-10', 'England')
```

The preview shows us the SQL to create the table then each line of Insert commands to add each row

showing first 100 rows

Rows: 1000

GENERATE DATA CLOSE

4.3 Modify SQL file in VsCode to add the constraints

```
1 create table person (  
2     id BIGSERIAL NOT NULL PRIMARY KEY,  
3     first_name VARCHAR(50) NOT NULL,  
4     last_name VARCHAR(50) NOT NULL,  
5     email VARCHAR(150),  
6     gender VARCHAR(50) NOT NULL,  
7     date_of_birth DATE,  
8     country_of_birth VARCHAR(50) NOT NULL  
9 );  
10 insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values ('Trstram', 'Germann',  
11 'tgermann0@photobucket.com', 'Male', '2022-02-20', 'Luxembourg');  
12 insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values ('Hinze', 'Gaize', null,  
13 'Male', '2021-08-18', 'Russia');  
14 insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values ('Tomasina', 'Georgius',  
15 'tgeorgius2@spiegel.de', 'Female', '2021-01-16', 'Russia');
```

Sql file is modified to include the id field with its constraints and add constraints to all the other fields except email

```
test=# DROP TABLE person;  
DROP TABLE  
test=#
```

Permantly delete the person table and it's contents.

4.4 Import an SQL file

```
test=# \?
...
Input/Output
  \copy ...           perform SQL COPY with data stream to the client host
  \echo [-n] [STRING] write string to standard output (-n for no newline)
  \i FILE             execute commands from file
  \ir FILE            as \i, but relative to location of current script
  \o [FILE]           send all query results to file or |pipe
  \qecho [-n] [STRING] write string to \o output stream (-n for no newline)
  \warn [-n] [STRING] write string to standard error (-n for no newline)
```

From the help menu under the imports and exports section, note that the \i command can be used to execute commands from file.

```
test=# \i C:/Users/ellio/Downloads/person.sql
CREATE TABLE
INSERT 0 1
...
INSERT 0 1
```

Using the `\i <full_path/file_name.sql>` command I have created a table and inserted 1000 rows of data.

4.5 Verify Contents of Table

```
test=# SELECT * FROM person;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
--						
1	Trstram	Germann	tgermann0@photobucket.com	Male	2022-02-20	Luxembourg
2	Hinze	Gaize		Male	2021-08-18	Russia
3	Tomasina	Georgius	tgeorgius2@spiegel.de	Female	2021-01-16	Russia
4	Benyamin	Duprey		Male	2020-08-15	Indonesia
5	Oran	Friel	ofriel4@sakura.ne.jp	Male	2024-07-07	Denmark
6	Dita	Schrader	dschrader5@feedburner.com	Female	2023-10-04	Cuba
7	Hailey	Ghiron		Non-binary	2020-07-12	Indonesia
8	Keen	Tarrier	ktarrier7@admin.ch	Agender	2023-06-25	Indonesia
9	Natalie	Cockrill	ncockrill18@archive.org	Female	2021-01-27	Venezuela
10	Catlee	Roskruge		Female	2022-04-06	China
11	Egon	Howells	ehowellsa@theguardian.com	Male	2022-06-11	Colombia
12	Simon	Chadband	schadbandb@163.com	Male	2024-01-26	Indonesia
13	Nike	Kinsella	nkinsellac@hubpages.com	Female	2022-03-25	United States
14	De witt	Balassi	dbalassid@yellowbook.com	Male	2024-05-07	Portugal
15	Bond	Dominik	bdominike@time.com	Male	2021-03-20	United States
16	Dane	Giorgielli		Male	2021-07-20	Russia
17	Sella	Prettyjohns	sprettyjohnsg@timesonline.co.uk	Female	2022-04-20	Slovenia
18	Debra	Beininck		Female	2022-09-20	Russia
19	Leda	Farres	lfarresi@reverbnation.com	Female	2020-06-07	Vietnam
20	Charlton	Everil	ceverilj@nature.com	Male	2022-03-08	Rwanda
21	Gertruda	Surcomb	gsurcombk@mtv.com	Female	2021-12-18	Canada
22	Hube	Iacovaccio		Male	2023-01-07	Thailand
23	Brianne	Greenslade	bgreensladem@usatoday.com	Female	2023-01-27	Poland
24	Micky	Langabeer	mlangabeern@macromedia.com	Male	2021-04-30	China
25	Rosy	Giuron	rgiurono@hubpages.com	Female	2023-09-10	Sweden
26	Giraud	Koch		Male	2021-09-14	Sierra Leone
27	Sofia	Midford	smidfordq@spotify.com	Female	2020-12-13	China
■ More	--					
■ 1000	Darcy	Suller		Male	2022-04-10	Uganda
■ (1000	rows)					

SELECT * FROM person (same as MariaDb). Use space bar to show more.

4.6 PostGreSQL Basic insert Records Commands review

```
INSERT INTO <table_name> (  
Column1_name,  
Column2_name,  
Column3_name)  
VALUES ('string', 'string', DATE 'YYYY-MM-DD');
```

INSERT INTO syntax. Specify data type if it is not a string

```
DROP TABLE <table_name>;
```

 Permanently delete a table and it's contents.

```
\i <path_to_file>/<file_name>.sql;
```

 Import an SQL file into the database

```
SELECT * FROM <table_name>;
```

 Show all rows and columns of a table

5. Basic Selecting Queries

[5.1 Select FROM table](#)

[5.2 Select FROM table ORDER BY](#)

[5.3 Select with Distinct clause](#)

[5.4 Select with where clause](#)

[5.5 Select with where clause and or clause](#)

[5.6 Select with Comparison operators](#)

[5.7 Select with Limit, offset and fetch](#)

[5.8 In keyword](#)

[5.9 Between keyword](#)

[5.10 Like Operator with wildcard & underscore matching](#)

5.1 Select FROM table

Select * FROM <table_name> will give an output of all data in all columns because star is a wildcard meaning all whereas if I just **SELECT FROM <table_name>** I just get a row count. This can be useful to verify that all rows have been imported when the table is very large with thousands of rows.

Select <column_name> FROM <table_name> will give an output of all rows of that specific column.

```
test=# SELECT FROM person;
--
(1000 rows)
```

```
test=#
```

```
test=# SELECT first_name
from person;
 first_name
-----
Trstram
...
Elsy
-- More --
```

More examples of SELECT FROM (syntax is the same as Maria Db or MySQL)

```
test=# SELECT * FROM person WHERE first_name = 'Egon';
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
11	Egon	Howells	ehowellsa@theguardian.com	Male	2022-06-11	Colombia
701	Egon	Gibson		Male	2023-05-12	France

(2 rows)

```
test=# SELECT * FROM person WHERE email IS NULL;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
2	Hinze	Gaize		Male	2021-08-18	Russia
...						
1000	Darcy	Suller		Male	2022-04-10	Uganda

(311 rows)

```
test=#
```

```
test=# SELECT * FROM person WHERE country_of_birth = 'Spain';
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
154	Elvin	Woodington		Male	2020-11-01	Spain
244	Had	Ellcome		Male	2022-01-07	Spain
780	Stafford	Bootell	sbootelln@army.mil	Male	2023-09-18	Spain

```
(3 rows)
```

```
test=#
```

5.2 Select FROM table ORDER BY

```
test=# SELECT * FROM person ORDER BY country_of_birth;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
261	Barn	Melbourne		Male	2022-07-10	Afghanistan
417	Torrey	Fisby	tfisbybk@constantcontact.com	Male	2021-11-26	Afghanistan
277	Livvy	Silliman	lsilliman7o@booking.com	Female	2020-06-15	Albania
57	Domenico	Tuck	dtuck1k@state.tx.us	Male	2024-03-30	Angola
896	Victor	Piscopiello	vpiscopielloov@china.com.cn	Male	2022-08-20	Antigua and Barbuda
536	Kyle	Bezemer	kbezemerev@amazon.co.uk	Female	2023-09-07	Argentina
...						
44	Rustie	Letten	rletten17@hao123.com	Male	2020-08-13	Yemen
596	Goldina	Road	groadgj@twitpic.com	Female	2023-03-04	Zambia
923	Gris	Jarrelt	gjarreltpm@ucoz.com	Male	2023-06-09	Zambia
371	Adolphus	Puig	apuigaa@rediff.com	Male	2021-06-25	Zambia
587	Nixie	Foskew	nfoskewga@theguardian.com	Female	2023-11-07	Zimbabwe

(1000 rows)

By default ORDER BY will sort ASCENDING alphabetically or numerically depending on the column data type we are ordering on. i.e A to Z or 0 to infinity. I can also specify ASCENDING with the ASC parameter. Use DESCENDING, DESC to order Z-A or highest to lowest numerically.

```
test=# SELECT * FROM person ORDER BY country_of_birth ASC;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
261	Barn	Melbourne		Male	2022-07-10	Afghanistan
417	Torrey	Fisby	tfisbybk@constantcontact.com	Male	2021-11-26	Afghanistan
277	Livvy	Silliman	lsilliman7o@booking.com	Female	2020-06-15	Albania

```
test=# SELECT * FROM person ORDER BY country_of_birth DESC;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
587	Nixie	Foskew	nfoskewga@theguardian.com	Female	2023-11-07	Zimbabwe
596	Goldina	Road	groadgj@twitpic.com	Female	2023-03-04	Zambia
371	Adolphus	Puig	apuigaa@rediff.com	Male	2021-06-25	Zambia

We can ORDER BY a DATE column

```
test=# SELECT * FROM person ORDER BY date_of_birth DESC;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
249	Adham	Heald	aheald6w@paginegialle.it	Male	2025-01-07	Brazil
565	Yardley	Gason	ygasonfo@booking.com	Male	2025-01-07	Russia
427	Daron	Brockwell	dbrockwellbu@accuweather.com	Female	2025-01-04	Ecuador
717	Ulrica	Minney	uminneyjw@baidu.com	Female	2025-01-01	Philippines

We can ORDER BY the email column. Note how it gives the null (or empty) values first. If this query was ORDER BY ASC then the null values would come last.

```
test=# SELECT * FROM person ORDER BY email DESC;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
-						
1000	Darcy	Suller		Male	2022-04-10	Uganda
2	Hinze	Gaize		Male	2021-08-18	Russia
4	Benyamin	Duprey		Male	2020-08-15	Indonesia
7	Hailey	Ghiron		Non-binary	2020-07-12	Indonesia
10	Catlee	Roskruge		Female	2022-04-06	China
16	Dane	Giorgielli		Male	2021-07-20	Russia
18	Debra	Beininck		Female	2022-09-20	Russia
22	Hube	Iacovaccio		Male	2023-01-07	Thailand
...						
256	Adair	Biggadike	abiggadike73@businesswire.com	Male	2021-04-01	Moldova
82	Alec	Baillies	abailles29@aol.com	Male	2021-05-01	China
246	Adel	Absolem	aabsolem6t@hubpages.com	Female	2021-05-12	China
(1000 rows)						

We can ORDER BY multiple columns. It will order by the first specified column then order by the second column for each first column entry.

```
test=# SELECT * FROM person ORDER BY first_name, date_of_birth ASC;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
153	Abbey	MacPherson	amacpherson48@parallels.com	Male	2022-01-31	Philippines
166	Abbey	Weddeburn	aweddeburn41@webnode.com	Male	2024-07-25	China
783	Abdel	Iannuzzelli	aiannuzzellilq@chronoengine.com	Male	2020-11-12	Japan
421	Abe	Peacop	apeacopbo@ucoz.com	Male	2021-03-02	Indonesia
256	Adair	Biggadike	abiggadike73@businesswire.com	Male	2021-04-01	Moldova
764	Addie	Jecks	ajecks17@eepurl.com	Male	2021-12-31	Indonesia
990	Ade	Lapidus		Male	2024-03-01	South Africa

5.3 Select with Distinct clause

```
test=# SELECT country_of_birth FROM person ORDER BY country_of_Birth;  
country_of_birth
```

1.

```
-----  
Afghanistan  
Afghanistan  
Albania  
Angola  
Antigua and Barbuda  
Argentina  
Argentina  
Argentina  
Argentina  
Argentina  
Argentina  
Argentina  
Armenia  
...  
Yemen  
Zambia  
Zambia  
Zambia  
Zimbabwe  
(1000 rows)
```

```
test=#
```

```
test=# SELECT DISTINCT country_of_birth FROM person ORDER BY country_of_Birth;  
country_of_birth
```

2.

```
-----  
Afghanistan  
Albania  
Angola  
Antigua and Barbuda  
Argentina  
Armenia  
Azerbaijan  
Bahamas  
Bahrain  
...  
Uganda  
Ukraine  
United Kingdom  
United States  
Uruguay  
Uzbekistan  
Venezuela  
Vietnam  
Yemen  
Zambia  
Zimbabwe  
(119 rows)
```

```
test=#
```

1. Note that the result has multiple entries for each country

2. Using the **DISTINCT** keyword will only show each same row once. Note that there are now **119 rows**, not **1000 rows**.

```
test=# SELECT DISTINCT country_of_birth FROM person ORDER BY country_of_Birth DESC;  
country_of_birth
```

3.

```
-----  
Zimbabwe  
Zambia  
Yemen  
Vietnam  
Venezuela  
Uzbekistan  
...  
Bahamas  
Azerbaijan  
Armenia  
Argentina  
Antigua and Barbuda  
Angola  
Albania  
Afghanistan  
(119 rows)
```

```
test=#
```

3. **DISTINCT** and **ORDER BY** can be combined with **DESC**

5.5 Select with where clause and or clause

```
test=# SELECT * FROM person WHERE gender='Female';
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
3	Tomasina	Georgius	tgeorgius2@spiegel.de	Female	2021-01-16	Russia
6	Dita	Schrader	dschrader5@feedburner.com	Female	2023-10-04	Cuba
9	Natalie	Cockrill	ncockrill18@archive.org	Female	2021-01-27	Venezuela
10	Catlee	Roskruge		Female	2022-04-06	China
13	Nike	Kinsella	nkinsellac@hubpages.com	Female	2022-03-25	United States
...						
996	Natalie	Schimke		Female	2023-04-28	Japan
997	Francine	Bohl	fbohlro@umich.edu	Female	2020-06-26	Ukraine
998	Basia	Sackett	bsackettrp@mapquest.com	Female	2024-02-04	Russia
999	Lizbeth	Wisam	lwisamrq@loc.gov	Female	2023-09-21	Indonesia

(440 rows)

```
test=# SELECT * FROM person WHERE gender='Female' AND email IS NOT NULL;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
3	Tomasina	Georgius	tgeorgius2@spiegel.de	Female	2021-01-16	Russia
6	Dita	Schrader	dschrader5@feedburner.com	Female	2023-10-04	Cuba
9	Natalie	Cockrill	ncockrill18@archive.org	Female	2021-01-27	Venezuela
13	Nike	Kinsella	nkinsellac@hubpages.com	Female	2022-03-25	United States
17	Sella	Prettyjohns	sprettyjohnsg@timesonline.co.uk	Female	2022-04-20	Slovenia
19	Leda	Farres	lfarresi@reverbnation.com	Female	2020-06-07	Vietnam
21	Gertruda	Surcomb	gsurcombk@mtv.com	Female	2021-12-18	Canada
23	Brianne	Greenslade	bgreensladem@usatoday.com	Female	2023-01-27	Poland
...						
995	Isadora	Trouncer	itrouncerrm@cpanel.net	Female	2020-11-21	Brazil
997	Francine	Bohl	fbohlro@umich.edu	Female	2020-06-26	Ukraine
998	Basia	Sackett	bsackettrp@mapquest.com	Female	2024-02-04	Russia
999	Lizbeth	Wisam	lwisamrq@loc.gov	Female	2023-09-21	Indonesia

(312 rows)

Syntax is the same as MariaDb and MySQL.

```
test=# SELECT * FROM person WHERE gender='Female' AND email IS NULL;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
10	Catlee	Roskruge		Female	2022-04-06	China
18	Debra	Beininck		Female	2022-09-20	Russia
31	Lin	Cambling		Female	2021-01-12	Brazil
43	Jere	McLeod		Female	2022-06-21	Russia
...						
949	Chrissie	Benzie		Female	2021-05-26	Brazil
966	Helga	Offer		Female	2021-10-10	Russia
981	Phyllis	Cridlon		Female	2021-07-04	Vietnam
996	Natalie	Schimke		Female	2023-04-28	Japan

(128 rows)

Syntax is the same as MariaDb and MySQL.

Note the brackets surrounding the OR clause in the second query. This is to specify that I want from either country and I want only females. Without the brackets males would be included from Cuba also.

```
test=# SELECT * FROM person WHERE gender='Female' AND (country_of_birth='Poland' OR country_of_birth='Cuba') ORDER BY country_of_birth;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
6	Dita	Schrader	dschrader5@feedburner.com	Female	2023-10-04	Cuba
224	Doralia	Kensitt		Female	2021-08-22	Cuba
295	Bess	Renfree	brenfree86@comsenz.com	Female	2020-08-28	Cuba
489	Ashleigh	O' Donohue	aodonohuedk@ow.ly	Female	2021-02-04	Cuba
736	Carlynn	Merriday		Female	2022-11-13	Cuba
883	Jacquelyn	Syme		Female	2022-06-27	Cuba
621	Erinn	Conachie	econachie8@indiatimes.com	Female	2022-06-28	Poland
278	Benedikta	Jaulmes		Female	2024-04-04	Poland
682	Kristine	McGraw	kmcgrawix@purevolume.com	Female	2023-01-21	Poland
389	Kaleena	Sans	ksansas@buzzfeed.com	Female	2021-03-22	Poland
419	Cicely	Bain	cbainbm@tripod.com	Female	2022-01-04	Poland
463	Meggi	Hinckley		Female	2021-02-06	Poland
752	Tybi	Tatlock	ttatlockkv@storify.com	Female	2022-10-19	Poland
564	Kore	Burchmore	kburchmorefn@guardian.co.uk	Female	2024-03-25	Poland
576	Cybil	Shilstone	cshilstonefz@sourceforge.net	Female	2024-06-12	Poland
586	Olimpia	Mearns	omearnsg9@springer.com	Female	2024-01-14	Poland
23	Brianne	Greenslade	bgreensladem@usatoday.com	Female	2023-01-27	Poland
56	Elsy	Bissell	ebissell1j@chicagotribune.com	Female	2020-11-13	Poland
87	Trina	Hampson	thampson2e@livejournal.com	Female	2020-10-25	Poland
173	Helaine	Duester		Female	2024-11-05	Poland
190	Jemima	Yakov	jyakov59@ebay.com	Female	2023-02-10	Poland

(21 rows)

5.6 Select with Comparison operators

```
test=# SELECT * FROM person WHERE date_of_birth >= DATE '2025-01-01' ORDER BY date_of_birth;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
717	Ulrica	Minney	uminneyjw@baidu.com	Female	2025-01-01	Philippines
427	Daron	Brockwell	dbrockwellbu@accuweather.com	Female	2025-01-04	Ecuador
249	Adham	Heald	aheald6w@paginegialle.it	Male	2025-01-07	Brazil
565	Yardley	Gason	ygasonfo@booking.com	Male	2025-01-07	Russia

(4 rows)

```
test=# SELECT * FROM person WHERE date_of_birth >= DATE '2023-01-01' AND gender = 'Female' AND email IS NOT NULL ORDER BY date_of_birth;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
193	Jobina	McAughtry	jmcaughtry5c@mac.com	Female	2023-01-01	China
958	Myrta	Leport	mleportql@joomla.org	Female	2023-01-02	Costa Rica
710	Dorice	Frugier	dfrugierjp@a8.net	Female	2023-01-02	Japan
613	Rosanne	Wilmington	rwilmingtonh0@baidu.com	Female	2023-01-03	Portugal
221	Hanni	De La Cote	hdelacote64@dailyemotion.com	Female	2023-01-14	Croatia
391	Drusy	Plowman	dplowmanau@mysql.com	Female	2023-01-18	Egypt
658	Rasia	Hearson	rhearsoni9@huffingtonpost.com	Female	2023-01-20	Slovenia
...						
975	Aurea	Lill	alillr2@eepurl.com	Female	2024-12-14	China
812	Kalina	Leile	kleilemj@theforest.net	Female	2024-12-25	China
717	Ulrica	Minney	uminneyjw@baidu.com	Female	2025-01-01	Philippines
427	Daron	Brockwell	dbrockwellbu@accuweather.com	Female	2025-01-04	Ecuador

(147 rows)

Comparison operators are the same in PostgreSQL as MariaDb and MySQL

> Greater than

< less than

>= Greater than or equal to

<= less than or equal to

5.7 Select with Limit, offset and fetch

```
test=# SELECT * FROM person WHERE date_of_birth >= DATE '2023-01-01' AND gender = 'Female' AND email IS NOT NULL ORDER BY date_of_birth;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
193	Jobina	McAughtry	jmcaughtry5c@mac.com	Female	2023-01-01	China
958	Myrta	Leport	mleportql@joomla.org	Female	2023-01-02	Costa Rica
...						
717	Ulrica	Minney	uminneyjw@baidu.com	Female	2025-01-01	Philippines
427	Daron	Brockwell	dbrockwellbu@accuweather.com	Female	2025-01-04	Ecuador

(147 rows)

The LIMIT keyword can be used to restrict the results to the first x number of rows.

```
test=# SELECT * FROM person WHERE date_of_birth >= DATE '2023-01-01' AND gender = 'Female' AND email IS NOT NULL ORDER BY date_of_birth LIMIT 5;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
193	Jobina	McAughtry	jmcaughtry5c@mac.com	Female	2023-01-01	China
710	Dorice	Frugier	dfrugierjp@a8.net	Female	2023-01-02	Japan
958	Myrta	Leport	mleportql@joomla.org	Female	2023-01-02	Costa Rica
613	Rosanne	Wilmington	rwilmingtonh0@baidu.com	Female	2023-01-03	Portugal
221	Hanni	De La Cote	hdelacote64@dailyemotion.com	Female	2023-01-14	Croatia

(5 rows)

The OFFSET keyword is used to start showing rows after the offset value. This can be combined with LIMIT to show the results in defined batches of fixed size which is usefull when performing pagination of results in an app.

```
test=# SELECT * FROM person WHERE date_of_birth >= DATE '2023-01-01' AND gender = 'Female' AND email IS NOT NULL ORDER BY date_of_birth OFFSET 5 LIMIT 5;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
391	Drusy	Plowman	dplowmanau@mysql.com	Female	2023-01-18	Egypt
658	Rasia	Hearson	rhearsoni9@huffingtonpost.com	Female	2023-01-20	Slovenia
682	Kristine	McGraw	kmcgrawix@purevolume.com	Female	2023-01-21	Poland
23	Brianne	Greenslade	bgreensladem@usatoday.com	Female	2023-01-27	Poland
130	Bonni	Delgardillo	bdelgardillo3l@domainmarket.com	Female	2023-02-04	Indonesia

(5 rows)

The LIMIT keyword is not an 'official' SQL keyword. FETCH is the official keyword and we can use FETCH by specifying what we want to fetch i.e. **FETCH FIRST 5 ROW ONLY**.

```
test=# SELECT * FROM person WHERE date_of_birth >= DATE '2023-01-01' AND gender = 'Female' AND email IS NOT NULL ORDER BY date_of_birth  
FETCH FIRST 5 ROW ONLY;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
193	Jobina	McAughtry	jmcaughtry5c@mac.com	Female	2023-01-01	China
710	Dorice	Frugier	dfrugierjp@a8.net	Female	2023-01-02	Japan
958	Myrta	Leport	mleportql@joomla.org	Female	2023-01-02	Costa Rica
613	Rosanne	Wilmington	rwilmingtonh0@baidu.com	Female	2023-01-03	Portugal
221	Hanni	De La Cote	hdelacote64@dailymotion.com	Female	2023-01-14	Croatia

(5 rows)

```
test=# SELECT * FROM person WHERE date_of_birth >= DATE '2023-01-01' AND gender = 'Female' AND email IS NOT NULL ORDER BY date_of_birth  
FETCH FIRST ROW ONLY;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
193	Jobina	McAughtry	jmcaughtry5c@mac.com	Female	2023-01-01	China

(1 row)

5.8 IN keyword

```
test=# SELECT * FROM person WHERE gender='Female' AND country_of_birth IN ('Poland', 'Cuba');
 id | first_name | last_name | email | gender | date_of_birth | country_of_birth
-----+-----+-----+-----+-----+-----+-----
  6 | Dita       | Schrader | dschrader5@feedburner.com | Female | 2023-10-04 | Cuba
 23 | Brianne   | Greenslade | bgreensladem@usatoday.com | Female | 2023-01-27 | Poland
 56 | Elsy      | Bissell   | ebissell1j@chicagotribune.com | Female | 2020-11-13 | Poland
 87 | Trina     | Hampson   | thampson2e@livejournal.com | Female | 2020-10-25 | Poland
173 | Helaine   | Duester   | | Female | 2024-11-05 | Poland
190 | Jemima    | Yaknov    | jyaknov59@ebay.com | Female | 2023-02-10 | Poland
224 | Doralia   | Kensitt   | | Female | 2021-08-22 | Cuba
278 | Benedikta | Jaulmes   | | Female | 2024-04-04 | Poland
295 | Bess      | Renfree   | brenfree86@comsenz.com | Female | 2020-08-28 | Cuba
389 | Kaleena   | Sans      | ksansas@buzzfeed.com | Female | 2021-03-22 | Poland
419 | Cicely    | Bain      | cbainbm@tripod.com | Female | 2022-01-04 | Poland
463 | Meggi     | Hinckley  | | Female | 2021-02-06 | Poland
489 | Ashleigh  | O' Donohue | aodonohuedk@ow.ly | Female | 2021-02-04 | Cuba
564 | Kore      | Burchmore | kburchmorefn@guardian.co.uk | Female | 2024-03-25 | Poland
576 | Cybil     | Shilstone | cshilstonefz@sourceforge.net | Female | 2024-06-12 | Poland
586 | Olimpia   | Mearns    | omearnsg9@springer.com | Female | 2024-01-14 | Poland
621 | Erinn     | Conachie  | econachieh8@indiatimes.com | Female | 2022-06-28 | Poland
682 | Kristine  | McGraw    | kmcgrawix@purevolume.com | Female | 2023-01-21 | Poland
736 | Carlynne  | Merriday  | | Female | 2022-11-13 | Cuba
752 | Tybi      | Tatlock   | ttatlockkv@storify.com | Female | 2022-10-19 | Poland
883 | Jacquelyn  | Syme      | | Female | 2022-06-27 | Cuba
(21 rows)
```

IN is a keyword that takes an array of values as a coma separated string and can be used to tidy up a query replacing many OR statements. The syntax is IN ('value1', 'value2', 'value3')

5.9 Between keyword

The between keyword can be used to find rows where the results are within a range of two defined values.

```
test=# SELECT * FROM person WHERE gender='Female' AND country_of_birth IN ('Poland', 'Cuba') AND date_of_birth BETWEEN DATE '2023-01-01' AND DATE '2023-12-31' ORDER BY date_of_birth;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
682	Kristine	McGraw	kmcgrawix@purevolume.com	Female	2023-01-21	Poland
23	Brianne	Greenslade	bgreensladem@usatoday.com	Female	2023-01-27	Poland
190	Jemima	Yaknov	jyaknov59@ebay.com	Female	2023-02-10	Poland
6	Dita	Schrader	dschrader5@feedburner.com	Female	2023-10-04	Cuba

(4 rows)

```
test=# SELECT * FROM person WHERE gender='Female' AND country_of_birth IN ('Poland', 'Cuba', 'Spain') AND date_of_birth BETWEEN DATE '2023-01-01' AND DATE '2024-12-31' ORDER BY date_of_birth;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
682	Kristine	McGraw	kmcgrawix@purevolume.com	Female	2023-01-21	Poland
23	Brianne	Greenslade	bgreensladem@usatoday.com	Female	2023-01-27	Poland
190	Jemima	Yaknov	jyaknov59@ebay.com	Female	2023-02-10	Poland
6	Dita	Schrader	dschrader5@feedburner.com	Female	2023-10-04	Cuba
586	Olimpia	Mearns	omearns9@springer.com	Female	2024-01-14	Poland
564	Kore	Burchmore	kburchmorefn@guardian.co.uk	Female	2024-03-25	Poland
278	Benedikta	Jaulmes		Female	2024-04-04	Poland
576	Cybil	Shilstone	cshilstonefz@sourceforge.net	Female	2024-06-12	Poland
173	Helaine	Duester		Female	2024-11-05	Poland

(9 rows)

5.10 Like Operator with wildcard & underscore matching

The Like operator can be used to match any part of a string in a column. For example if I want to find all emails that end .com I can use the wildcard followed by .com to find any matches where only the .com part has to match.

```
test=# SELECT * FROM person WHERE email LIKE '%.com' AND gender = 'Female';
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
6	Dita	Schrader	dschrader5@feedburner.com	Female	2023-10-04	Cuba
13	Nike	Kinsella	nkinsellac@hubpages.com	Female	2022-03-25	United States
19	Leda	Farres	lfarresi@reverbnation.com	Female	2020-06-07	Vietnam
21	Gertruda	Surcomb	gsurcombk@mtv.com	Female	2021-12-18	Canada
23	Brianne	Greenslade	bgreensladem@usatoday.com	Female	2023-01-27	Poland
25	Rosy	Giuron	rgiurono@hubpages.com	Female	2023-09-10	Sweden
...						
992	Brinna	Hursthouse	bhursthouserj@adobe.com	Female	2024-05-10	Morocco
994	Betsey	Flook	bflooksr1@blogger.com	Female	2021-05-21	Russia
998	Basia	Sackett	bsackettrp@mapquest.com	Female	2024-02-04	Russia

(194 rows)

```
test=# SELECT * FROM person WHERE email LIKE '%gmail.com';
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
----	------------	-----------	-------	--------	---------------	------------------

(0 rows)

```
test=# SELECT * FROM person WHERE email LIKE '%@skype.com';
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
127	Bonnee	Snazel	bsnazel3i@skype.com	Female	2021-07-16	Ukraine
946	Clemence	Klagge	cklaggeq9@skype.com	Female	2023-02-21	China

(2 rows)

I can use a wildcard either side of the desired string so that it can be found anywhere in the string. For example I want to find people that have the .gov part in the domain name.

```
test=# SELECT * FROM person WHERE email LIKE '%.gov%';
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
50	Tomi	Cowell	tcowell1d@oaic.gov.au	Female	2021-07-13	Philippines
70	Timothy	Goodwill	tgoodwill1x@house.gov	Male	2020-10-28	China
81	Justinn	Luttgert	jluttgert28@census.gov	Female	2024-11-15	United States
...						
735	Nicko	Sokell	nsokellke@census.gov	Male	2021-12-06	Indonesia
738	Boy	Belsey	bbelseykh@privacy.gov.au	Male	2021-01-15	Czech Republic
753	Elbert	Bartunek	ebartunekkw@ed.gov	Male	2024-10-30	Mexico
771	Sharla	Mullin	smullinle@ca.gov	Female	2022-01-22	Philippines
803	Brett	Ryrie	bryriema@oaic.gov.au	Male	2021-01-04	France
846	Joyan	Agett	jagetttnh@whitehouse.gov	Female	2021-04-03	Indonesia
999	Lizabeth	Wisam	lwisamrq@loc.gov	Female	2023-09-21	Indonesia

(33 rows)

Wild card string matching in PostgreSQL is the same as MariaDB and MySQL

Underscores can be used to match a single character. In the below example I am using six underscores to match any email that has six characters before the at symbol.

```
test=# SELECT * FROM person WHERE email LIKE '_____@%';
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
33	Leonardo	Noot	lnootw@arstechnica.com	Male	2022-08-14	Belarus
35	Daniel	Vant	dvanty@omniture.com	Male	2021-01-06	Belarus
439	Marcile	Esp	mespc6@yandex.ru	Female	2023-03-13	Yemen
570	Pavia	Kos	pkosft@psu.edu	Female	2021-11-30	Yemen
779	Arlan	Wyd	awydlm@berkeley.edu	Male	2021-02-07	Peru

(5 rows)

I can see that there are 4 people who were born in countries starting with the letter Z. Note that I have to use an uppercase Z because the Like query is case sensitive.

```
test=# SELECT * FROM person WHERE country_of_birth LIKE 'Z%';
id | first_name | last_name | email | gender | date_of_birth | country_of_birth
-----+-----+-----+-----+-----+-----+-----
371 | Adolphus | Puig | apuigaa@rediff.com | Male | 2021-06-25 | Zambia
587 | Nixie | Foskew | nfoskewga@theguardian.com | Female | 2023-11-07 | Zimbabwe
596 | Goldina | Road | groadgj@twitpic.com | Female | 2023-03-04 | Zambia
923 | Gris | Jarrelt | gjarreltpm@ucoz.com | Male | 2023-06-09 | Zambia
(4 rows)

test=# SELECT * FROM person WHERE country_of_birth LIKE 'z%';
id | first_name | last_name | email | gender | date_of_birth | country_of_birth
-----+-----+-----+-----+-----+-----+-----
(0 rows)
```

The ILIKE keyword is not case sensitive.

```
test=# SELECT * FROM person WHERE country_of_birth ILIKE 'z%';
id | first_name | last_name | email | gender | date_of_birth | country_of_birth
-----+-----+-----+-----+-----+-----+-----
371 | Adolphus | Puig | apuigaa@rediff.com | Male | 2021-06-25 | Zambia
587 | Nixie | Foskew | nfoskewga@theguardian.com | Female | 2023-11-07 | Zimbabwe
596 | Goldina | Road | groadgj@twitpic.com | Female | 2023-03-04 | Zambia
923 | Gris | Jarrelt | gjarreltpm@ucoz.com | Male | 2023-06-09 | Zambia
(4 rows)
```

6. Aggregate Functions

[6.1 Selecting with Group By and Count](#)

[6.2 having keyword](#)

[6.3 Generate new table of numeric values](#)

[6.4 Max, Min, Avg and Sum functions](#)

6.1 Selecting with Group By and Count

```
test=# SELECT DISTINCT country_of_birth from Person;
country_of_birth
```

1.

```
-----
Indonesia
Venezuela
Kiribati
...
Palestinian Territory
Poland
Costa Rica
Antigua and Barbuda
(119 rows)
```

1. In the person table I have people that were born in 119 different countries.

```
test=# SELECT country_of_birth, COUNT(*)
FROM person GROUP BY country_of_birth;
```

2.

```
country_of_birth | count
-----+-----
Indonesia        |    99
Venezuela        |     8
Kiribati          |     1
Cameroon         |     2
Luxembourg       |     1
Czech Republic   |    11
...
Palestinian Territory |     1
Poland           |    35
Costa Rica       |     4
Antigua and Barbuda |     1
(119 rows)
```

2. But if I want a tally of the number of people from each country I have to include the COUNT keyword

```
test=# SELECT country_of_birth, COUNT(*) FROM person
GROUP BY country_of_birth ORDER BY country_of_birth;
```

3.

```
country_of_birth | count
-----+-----
Afghanistan      |     2
Albania          |     1
Angola           |     1
Antigua and Barbuda |     1
Argentina        |     8
Armenia          |     3
Azerbaijan       |     3
...
Venezuela        |     8
Vietnam          |     9
Yemen            |     5
Zambia           |     3
Zimbabwe         |     1
(119 rows)
```

3. These results can be ordered alphabetically

```
test=# SELECT country_of_birth,
COUNT(*) FROM person GROUP BY
country_of_birth ORDER BY COUNT DESC;
```

4.

```
country_of_birth | count
-----+-----
China            |   180
Indonesia        |    99
Philippines      |    60
...
Uruguay          |     1
Bahamas          |     1
Laos             |     1
El Salvador      |     1
Chile            |     1
(119 rows)
```

4. These results can be ordered by the COUNT

6.2 having keyword

```
test=# SELECT country_of_birth, COUNT(*) FROM person GROUP
BY country_of_birth HAVING COUNT(*) > 10 ORDER BY COUNT;
```

country_of_birth	count
Mexico	11
Czech Republic	11
Canada	11
Greece	12
Peru	16
Ukraine	17
United States	21
Sweden	25
Japan	26
Poland	35
France	35
Brazil	40
Portugal	48
Russia	56
Philippines	60
Indonesia	99
China	180

(17 rows)

1.

1. The having keyword takes a function to apply a conditional statement to the query. Below I am counting countries of birth where the count is greater than 10.

```
test=# SELECT country_of_birth, COUNT(*) FROM person GROUP BY
country_of_birth HAVING COUNT(*) BETWEEN 10 AND 100 ORDER BY COUNT;
```

country_of_birth	count
Colombia	10
Ecuador	10
Czech Republic	11
Canada	11
Mexico	11
Greece	12
Peru	16
Ukraine	17
United States	21
Sweden	25
Japan	26
Poland	35
France	35
Brazil	40
Portugal	48
Russia	56
Philippines	60
Indonesia	99

(18 rows)

2.

2. I can use the having keyword with between to find all countries that have between 10 and 100 rows of people born there.

All of these aggregate functions and many more can be found in the Post GreSQL documentation <https://www.postgresql.org/docs/9.5/functions-aggregate.html>

6.3 Generate new table of numeric values

Use Mockeroo to create a table of test data for car.

```
create table car (  
  id BIGSERIAL NOT NULL PRIMARY KEY,  
  make VARCHAR(100) NOT NULL,  
  model VARCHAR(100) NOT NULL,  
  price NUMERIC(19,2) NOT NULL  
);  
insert into car (id, make, model, price) values (1, 'Dodge', 'Caravan', '85534.98');  
insert into car (id, make, model, price) values (2, 'Jaguar', 'S-Type', '95099.14');  
insert into car (id, make, model, price) values (3, 'Lexus', 'GX', '70495.11');  
insert into car (id, make, model, price) values (4, 'Kia', 'Rio', '92358.99');  
insert into car (id, make, model, price) values (5, 'Citroën', '2CV', '46889.98');  
insert into car (id, make, model, price) values (6, 'Audi', 'S4', '16919.84');  
insert into car (id, make, model, price) values (7, 'Isuzu', 'Rodeo', '71714.42');  
insert into car (id, make, model, price) values (8, 'Mercedes-Benz', 'SLK-Class', '83014.90');  
insert into car (id, make, model, price) values (9, 'Mercedes-Benz', 'CL-Class', '83900.54');  
insert into car (id, make, model, price) values (10, 'Volkswagen', 'Passat', '17984.87');
```

Note that Mockeroo now does not have the option to specify the currency as none so when I got my SQL table all the prices came with dollar sign which did not insert as a numeric data type obviously. I had to use find and replace all in vscode to remove the dollar.

```
test=# \i C:/Users/ellio/Downloads/car.sql  
INSERT 0 1  
...  
INSERT 0 1  
  
test=# SELECT COUNT(*) FROM car;  
count  
-----  
1000  
(1 row)
```

Use the \i command to import the data from the car.sql file.

Verify that the car table now has 1000 rows of data.

6.4 Max, Min, AVG and Sum functions

```
test=# SELECT MAX(price) FROM car;
      max
-----
 99747.21
(1 row)
```

```
test=# SELECT MIN(price) FROM car;
      min
-----
  1030.82
(1 row)
```

```
test=# SELECT AVG(price) FROM car;
      avg
-----
50611.667380000000
(1 row)
```

```
test=# SELECT ROUND(AVG(price)) FROM car;
      round
-----
    50612
(1 row)
```

So if I wanted to find the cheapest car model for each manufacturer I can use the MIN function and GROUP BY.

The MAX and MIN functions can be used to find the maximum and minimum values of a particular column where the function takes the column name as a parameter.

The AVG function finds the average of all the column values. The AVG function can be wrapped inside the ROUND function to get the average as an integer.

```
test=# SELECT make, model, MIN(price) FROM car GROUP BY make, model;
      make |      model      |      min
-----+-----+-----
 Oldsmobile | Silhouette      | 37689.86
      Kia  | Amanti          | 99233.42
   Daewoo  | Lanos           | 66123.83
   Subaru  | Forester        | 24009.23
      Ford  | Econoline E350  |  8221.26
      Jeep  | Grand Cherokee  | 19821.44
   Lincoln | Aviator         | 98559.22
   Hyundai | Accent          | 39831.72
      ...
 Mercedes-Benz | G-Class      | 29395.47
   Subaru    | Alcyone SVX   |  1914.50
      BMW    | 525           | 33278.45
   Pontiac   | GTO           | 27170.27
      Toyota | Land Cruiser   | 28555.56
      GMC    | Yukon Denali   | 53683.22
 Mercedes-Benz | S-Class      | 58751.87
(520 rows)
```

```
test=# SELECT make, model, MAX(price) FROM car GROUP BY make, model;
```

make	model	max
Oldsmobile	Silhouette	37689.86
Kia	Amanti	99233.42
Daewoo	Lanos	66123.83
Subaru	Forester	90711.57
Ford	Econoline E350	74437.77
Jeep	Grand Cherokee	48684.81
Lincoln	Aviator	98559.22
...		
Subaru	B9 Tribeca	39032.49
Cadillac	CTS-V	67359.21
Mercedes-Benz	G-Class	29395.47
Subaru	Alcyone SVX	33871.79
BMW	525	88009.50
Pontiac	GT0	88120.80
Toyota	Land Cruiser	99499.11
GMC	Yukon Denali	53683.22
Mercedes-Benz	S-Class	91378.02

(520 rows)

I can find the most expensive car model for each manufacturer using the MAX function and GROUP BY.

So which manufacturer makes the cheapest cars?

```
test=# SELECT make, model, round(AVG(price)) FROM car GROUP BY make, model ORDER BY round;
```

make	model	round
Volkswagen	Jetta	1190
Suzuki	Swift	1190
GMC	Jimmy	1239
Mitsubishi	Montero	1856
Chevrolet	Astro	2278
Volvo	C70	2954
Mitsubishi	Challenger	3102
GMC	1500	3419
...		
Buick	Roadmaster	98904
Spyker	C8 Spyder	99104
Toyota	Camry Hybrid	99111
Kia	Amanti	99233

(520 rows)

One way to do this might be to find the average price for model per manufacturer then order the results by the average rounded price.


```
test=# SELECT make, model, MIN(price) FROM car GROUP BY make, model ORDER BY min;
```

make	model	min
GMC	Yukon XL 1500	1030.82
Mazda	B-Series	1129.06
Toyota	4Runner	1150.74
Suzuki	Swift	1189.83
Volkswagen	Jetta	1190.43
GMC	Jimmy	1238.92
GMC	Safari	1270.36
Saab	9000	1435.80
...		
Maserati	430	98415.89
Lincoln	Aviator	98559.22
Buick	Roadmaster	98903.88
Spyker	C8 Spyder	99103.59
Toyota	Camry Hybrid	99111.13
Kia	Amanti	99233.42

(520 rows)

What is the cheapest car offered by each manufacturer?

Use the Min function combined with GROUP BY.

```
test=# SELECT SUM(price) FROM car;
```

```
-----
50611667.38
(1 row)
```

What is the total price of all the models of cars? – SUM function.

```
test=# SELECT make, SUM(price) FROM car GROUP BY make;
```

make	sum
McLaren	12722.59
Ford	4619394.91
Maserati	402289.91
Dodge	2924428.05
Infiniti	742134.39
MINI	198795.36
...	
Hummer	110089.38
Toyota	2571964.80

(59 rows)

What is the total price of all the models of cars per manufacturer? – SUM function with GROUP BY.

Note how McLaren's total price for all models is low. This is because McLaren only has one model in the table (SELECT * FROM car WHERE make='McLaren';)

7. Arithmetic Operators

[7.1 Types of mathematic and Arithmetic Operators](#)

[7.1 Round function](#)

[7.3 Alias a Column name](#)

[7.4 Handling nulls with coalesce keyword](#)

[7.5 Handling division by zero with NullIf](#)

7.1 Types of mathematic & Arithmetic Operators

```
test=# SELECT 10^2;
?column?
-----
      100
(1 row)
```

```
test=# SELECT 10^3;
?column?
-----
     1000
(1 row)
```

```
test=# SELECT 10 % 3;
?column?
-----
      1
(1 row)
```

Use the ^ to perform to the power of

```
test=# SELECT 5!;
ERROR:  syntax error at or near ";"
LINE 1: SELECT 5!;
                  ^

test=# SELECT factorial(5);
?column?
-----
      120
(1 row)
```

The factorial of a number is the multiplication of all integers upto that number. i.e. $5 \times 4 \times 3 \times 2 \times 1$

Note that from [Postgres 14](https://www.postgresql.org/docs/8.2/functions-math.html) the factorial operator has been removed.

Addition
Subtraction
Multiplication
Division

```
test=# SELECT 10 + 2;
?column?
-----
      12
(1 row)
```

```
test=# SELECT 10 - 2;
?column?
-----
      8
(1 row)
```

```
test=# SELECT 10 * 2;
?column?
-----
     20
(1 row)
```

```
test=# SELECT 10 / 2;
?column?
-----
      5
(1 row)
```

```
test=# SELECT 10 / 2 + 5;
?column?
-----
     10
(1 row)
```

The modulus (%) allows us to get the remainder after a division.

For the complete list of mathematic and Arithmetic operators
<https://www.postgresql.org/docs/8.2/functions-math.html>

7.1 Round function

```
test=# SELECT id, make, model, price, price*0.10 FROM car;
```

id	make	model	price	?column?
1	Dodge	Caravan	85534.98	8553.4980
2	Jaguar	S-Type	95099.14	9509.9140
3	Lexus	GX	70495.11	7049.5110
4	Kia	Rio	92358.99	9235.8990
5	Citro %	2CV	46889.98	4688.9980
6	Audi	S4	16919.84	1691.9840
7	Isuzu	Rodeo	71714.42	7171.4420
...				
996	Toyota	Matrix	7743.68	774.37
997	Mercedes-Benz	SL-Class	9727.03	972.70
998	MINI	Clubman	70272.22	7027.22
999	Mazda	B-Series Plus	69992.58	6999.26
1000	BMW	3 Series	35109.33	3510.93

(1000 rows)

I am using the multiply mathematical operator to calculate 10% of each car's price and round it to two decimal places because it is currency.

Note how this column does not have a name but says ?column?

Now we can add another column that is the price minus 10% rounded to two decimal places.

So now we have the list price of the car, a 10% discount value and the price of the car with the discount applied.

```
test=# SELECT id, make, model, price, ROUND(price*0.10,2), ROUND(price - (price * 0.10),2) FROM car;
```

id	make	model	price	round	round
1	Dodge	Caravan	85534.98	8553.50	76981.48
2	Jaguar	S-Type	95099.14	9509.91	85589.23
3	Lexus	GX	70495.11	7049.51	63445.60
4	Kia	Rio	92358.99	9235.90	83123.09
5	Citro %	2CV	46889.98	4689.00	42200.98
6	Audi	S4	16919.84	1691.98	15227.86
7	Isuzu	Rodeo	71714.42	7171.44	64542.98
...					
996	Toyota	Matrix	7743.68	774.37	6969.31
997	Mercedes-Benz	SL-Class	9727.03	972.70	8754.33
998	MINI	Clubman	70272.22	7027.22	63245.00
999	Mazda	B-Series Plus	69992.58	6999.26	62993.32
1000	BMW	3 Series	35109.33	3510.93	31598.40

(1000 rows)

7.3 Alias a Column name

```
test=# SELECT id, make, model, price as list_price, ROUND(price*0.10,2) AS ten_pc_discount,
ROUND(price - (price * 0.10),2) AS new_price FROM car;
```

id	make	model	list_price	ten_pc_discount	new_price
1	Dodge	Caravan	85534.98	8553.50	76981.48
2	Jaguar	S-Type	95099.14	9509.91	85589.23
3	Lexus	GX	70495.11	7049.51	63445.60
4	Kia	Rio	92358.99	9235.90	83123.09
5	Citro ½n	2CV	46889.98	4689.00	42200.98
6	Audi	S4	16919.84	1691.98	15227.86
7	Isuzu	Rodeo	71714.42	7171.44	64542.98
8	Mercedes-Benz	SLK-Class	83014.90	8301.49	74713.41
9	Mercedes-Benz	CL-Class	83900.54	8390.05	75510.49
...					
996	Toyota	Matrix	7743.68	774.37	6969.31
997	Mercedes-Benz	SL-Class	9727.03	972.70	8754.33
998	MINI	Clubman	70272.22	7027.22	63245.00
999	Mazda	B-Series Plus	69992.58	6999.26	62993.32
1000	BMW	3 Series	35109.33	3510.93	31598.40

(1000 rows)

I can use the AS keyword to rename a column in the SQL query output

Note how:
price => list_price,
Round => ten_pc_discount,
Round => new_price

7.4 Handling nulls with coalesce keyword

The coalesce function's role is to return the first non-null value it encounters when reading from left to right. In addition, it can replace null values with a specified non-null value.

```
test=# SELECT COALESCE(1);
 coalesce
```

```
-----
          1
(1 row)
```

```
test=# SELECT COALESCE(1) AS number;
 number
```

```
-----
          1
(1 row)
```

```
test=# SELECT COALESCE(null, 1) AS number;
 number
```

```
-----
          1
(1 row)
```

```
test=# SELECT COALESCE(null, null, 1) AS number;
 number
```

```
-----
          1
(1 row)
```

Note the null (or empty) values in some of the email addresses in the people table

```
test=# SELECT * FROM person;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
1	Trstram	Germann	tgermann0@photobucket.com	Male	2022-02-20	Luxembourg
2	Hinze	Gaize		Male	2021-08-18	Russia
3	Tomasina	Georgius	tgeorgius2@spiegel.de	Female	2021-01-16	Russia
4	Benyamin	Duprey		Male	2020-08-15	Indonesia
5	Oran	Friel	ofriel14@sakura.ne.jp	Male	2024-07-07	Denmark
6	Dita	Schrader	dschrader5@feedburner.com	Female	2023-10-04	Cuba
7	Hailey	Ghiron		Non-binary	2020-07-12	Indonesia
8	Keen	Tarrier	ktarrier7@admin.ch	Agender	2023-06-25	Indonesia
...						
997	Francine	Bohl	fbohlro@umich.edu	Female	2020-06-26	Ukraine
998	Basia	Sackett	bsackettrp@mapquest.com	Female	2024-02-04	Russia
999	Lizbeth	Wisam	lwisamrq@loc.gov	Female	2023-09-21	Indonesia
1000	Darcy	Suller		Male	2022-04-10	Uganda

(1000 rows)

```
test=# SELECT COALESCE(email, 'No email provided') FROM person;  
          coalesce
```

```
-----  
tgermann0@photobucket.com  
No email provided  
tgeorgius2@spiegel.de  
No email provided  
ofriel4@sakura.ne.jp  
dschrader5@feedburner.com  
No email provided  
ktarrier7@admin.ch  
ncockrill8@archive.org  
...  
itrouncerrm@cpanel.net  
No email provided  
fbohlro@umich.edu  
bsackettrp@mapquest.com  
lwisamrq@loc.gov  
No email provided  
(1000 rows)
```

I can alias the coalesce column to make a prettier table.

COALESCE is used to replace the null values of email with the string 'no email provided'

```
test=# SELECT COALESCE(email, 'No email provided') AS email FROM person;  
          email
```

```
-----  
tgermann0@photobucket.com  
No email provided  
tgeorgius2@spiegel.de  
No email provided  
ofriel4@sakura.ne.jp  
dschrader5@feedburner.com  
No email provided  
ktarrier7@admin.ch  
ncockrill8@archive.org  
...  
itrouncerrm@cpanel.net  
No email provided  
fbohlro@umich.edu  
bsackettrp@mapquest.com  
lwisamrq@loc.gov  
No email provided  
(1000 rows)
```

7.5 Handling division by zero with Nullif

```
test=# SELECT 10 / 0;  
ERROR:  division by zero
```

If you try and perform a division by zero it will blow and throw an error.

NULLIF takes two arguments and returns the first argument if the second argument is not equal to the first argument.

For example 10 and 10 are equal so the NULLIF returns NULL
But 10 and 1 are NOT EQUAL so NULLIF returns the first argument.

```
test=# SELECT 10 / NULL;  
?column?  
-----  
(1 row)
```

What we can do with this is that PostgreSQL does not throw an error when dividing by NULL.

```
test=# SELECT 10 / NULLIF(2, 9);  
?column?  
-----  
5  
(1 row)
```

So now we can use NULLIF in division calculations

Now when we try and divide by a value that is zero NULLIF will return a NULL making the SQL query valid

```
test=# SELECT COALESCE(10 / NULLIF(0,0), 0);  
coalesce  
-----  
0  
(1 row)
```

And combine it with COALESCE to return 0 rather than null

```
test=# SELECT NULLIF(10, 10);  
nullif  
-----
```

(1 row)

```
test=# SELECT NULLIF(10, 1);  
nullif  
-----  
10  
(1 row)
```

```
test=# SELECT NULLIF(1, 10);  
nullif  
-----  
1  
(1 row)
```

```
test=# SELECT 10 / NULLIF(0, 0);  
?column?  
-----  
(1 row)
```


8. TimeStamp & Date

[8.1 Timestamp & Date Calculations](#)

[8.2 Extracting Part of a Timestamp or Date](#)

[8.3 Age Function](#)

8.1 Timestamp & Date Calculations

```
test=# SELECT NOW();
           now
-----
2025-01-09 00:56:47.339493+01
(1 row)
```

The NOW() function will return the current timestamp consisting of the date, hours, mins, seconds and milliseconds and the timezone from UTC

```
test=# SELECT NOW()::DATE;
           now
-----
2025-01-09
(1 row)
```

The NOW() function can be cast to get a component of it such as date. Use double semi-colon to cast.

```
test=# SELECT NOW()::TIME;
           now
-----
01:01:43.01079
(1 row)
```

Current time can also be cast from NOW() function.

It is also possible to perform calculations on a datetime. For example using the INTERVAL keyword I can subtract 10 years from the current datetime, or add 10 days.

```
test=# SELECT NOW();
           now
-----
2025-01-09 01:07:11.36177+01
(1 row)

test=# SELECT NOW() - INTERVAL '10 YEAR';
           ?column?
-----
2015-01-09 01:07:32.177497+01
(1 row)

test=# SELECT NOW() + INTERVAL '10 DAY';
           ?column?
-----
2025-01-19 01:12:12.056532+01
(1 row)

test=# SELECT NOW() + INTERVAL '10 MONTH';
           ?column?
-----
2025-11-09 01:13:41.720448+01
(1 row)
```

```
test=# SELECT NOW()::DATE + INTERVAL '10 MONTH';
       ?column?
-----
2025-11-09 00:00:00
(1 row)

test=# SELECT (NOW()::DATE + INTERVAL '10 MONTH')::DATE;
       date
-----
2025-11-09
(1 row)
```

It is also possible to cast NOW() to date then add 10 months. This will return a timestamp as YYYY-MM-DD hh:mm:ss.

If I only want the date then I can wrap the whole thing in brackets and cast it again to date.

A complete list of timestamp and date types can be found on the documentation <https://www.postgresql.org/docs/17/datatype-datetime.html>

8.2 Extracting Part of a Timestamp or Date

```
test=# SELECT NOW();
           now
-----
2025-01-09 01:22:37.534131+01
(1 row)

test=# SELECT EXTRACT (YEAR FROM NOW());
      extract
-----
         2025
(1 row)

test=# SELECT EXTRACT (MONTH FROM NOW());
      extract
-----
           1
(1 row)

test=# SELECT EXTRACT (DAY FROM NOW());
      extract
-----
           9
(1 row)
```

```
test=# SELECT EXTRACT (DOW FROM NOW());
      extract
-----
           4
(1 row)

test=# SELECT EXTRACT (CENTURY FROM NOW());
      extract
-----
          21
(1 row)
```

Using the EXTRACT keyword it is possible to get the Year, Month, Day, Time, Day of Week (DOW) and Century

8.3 Age Function

```
test=# SELECT first_name, last_name, gender, country_of_birth, date_of_birth from person;
```

first_name	last_name	gender	country_of_birth	date_of_birth
Trstram	Germann	Male	Luxembourg	2022-02-20
Hinze	Gaize	Male	Russia	2021-08-18
Tomasina	Georgius	Female	Russia	2021-01-16
Benyamin	Duprey	Male	Indonesia	2020-08-15
...				
Basia	Sackett	Female	Russia	2024-02-04
Lizbeth	Wisam	Female	Indonesia	2023-09-21
Darcy	Suller	Male	Uganda	2022-04-10

(1000 rows)

The age function can be used to calculate the age from a datetime value. This can also be combined with EXTRACT to get part of a datetime

```
test=# SELECT first_name, last_name, gender, country_of_birth, date_of_birth, AGE(date_of_birth) from person;
```

first_name	last_name	gender	country_of_birth	date_of_birth	age
Trstram	Germann	Male	Luxembourg	2022-02-20	2 years 10 mons 17 days
Hinze	Gaize	Male	Russia	2021-08-18	3 years 4 mons 22 days
Tomasina	Georgius	Female	Russia	2021-01-16	3 years 11 mons 24 days
Benyamin	Duprey	Male	Indonesia	2020-08-15	4 years 4 mons 25 days
...					
Basia	Sackett	Female	Russia	2024-02-04	11 mons 5 days
Lizbeth	Wisam	Female	Indonesia	2023-09-21	1 year 3 mons 18 days
Darcy	Suller	Male	Uganda	2022-04-10	2 years 8 mons 29 days

(1000 rows)

```
test=# SELECT first_name, last_name, gender, country_of_birth, date_of_birth, EXTRACT(YEAR FROM AGE(date_of_birth)) from person;
```

first_name	last_name	gender	country_of_birth	date_of_birth	extract
Trstram	Germann	Male	Luxembourg	2022-02-20	2
Hinze	Gaize	Male	Russia	2021-08-18	3
Tomasina	Georgius	Female	Russia	2021-01-16	3
Benyamin	Duprey	Male	Indonesia	2020-08-15	4
...					
Basia	Sackett	Female	Russia	2024-02-04	0
Lizbeth	Wisam	Female	Indonesia	2023-09-21	1
Darcy	Suller	Male	Uganda	2022-04-10	2

(1000 rows)

9. Primary Key & Constraints

[9.1 Why Primary keys?](#)

[9.2 Identify the Primary Key of a Table](#)

[9.3 Alter Table to Drop pkey Constraint](#)

[9.4 Alter Table to Add pkey Constraint](#)

[9.5 Unique Constraints](#)

[9.6 Check Constraints](#)

9.1 Why Primary keys?

person				
first_name	last_name	gender	date_of_birth	email
Anne	Smith	FEMALE	09/01/88	anne@gmail.com
Anne	Smith	FEMALE	09/01/88	<u>anne@hotmail.com</u>

1. Uniquely identify rows in a table where all the columns of data could be the same.
2. Primary key cannot be duplicated and is unique in that table. Two rows can not have the same primary key.
3. Primary key can not be NULL (empty)
4. It is typical that primary key is an auto-incrementing number, i.e. each time a new row is added the highest primary key number is incremented by 1.
5. Primary key allows us to uniquely identify a record in a table.

9.2 Identify the Primary Key of a Table

When describing a table the output will identify all indexes including the primary key.

```
test=# \d person
```

Table "public.person"				
Column	Type	Collation	Nullable	Default
id	bigint		not null	nextval('person_id_seq'::regclass)
first_name	character varying(50)		not null	
last_name	character varying(50)		not null	
email	character varying(150)			
gender	character varying(50)		not null	
date_of_birth	date			
country_of_birth	character varying(50)		not null	

Indexes:

"person_pkey" PRIMARY KEY, btree (id)

Note how the default value of the Id field says next value person_id_seq. this means that we do not change the value of ID because it is automatically sequenced.

```
test=# INSERT INTO person (id, first_name, last_name, gender, email, date_of_birth, country_of_birth)
VALUES (1, 'John', 'Smith', 'Male', 'js@gmail.com', '1953-01-10', 'England');
ERROR:  duplicate key value violates unique constraint "person_pkey"
DETAIL:  Key (id)=(1) already exists.
test=#
```

If I manually try and insert a row with a duplicate primary key it will throw an error because the pkey value already exists.

9.3 Alter Table to Drop pkey Constraint

```
test=# ALTER TABLE person DROP CONSTRAINT person_pkey;
ALTER TABLE
test=#
```

The person table is altered to drop the named primary key constraint.

```
test=# \d person
```

Table "public.person"					
Column	Type	Collation	Nullable	Default	
id	bigint		not null	nextval('person_id_seq'::regclass)	
first_name	character varying(50)		not null		
last_name	character varying(50)		not null		
email	character varying(150)				
gender	character varying(50)		not null		
date_of_birth	date				
country_of_birth	character varying(50)		not null		

Now the person table does not have a primary key.

```
test=#
```

Now it is possible to insert a new row with a duplicate id value.

```
test=# INSERT INTO person (id, first_name, last_name, gender, email, date_of_birth, country_of_birth)
VALUES (1, 'John', 'Smith', 'Male', 'js@gmail.com', '1953-01-10', 'England');
INSERT 0 1
test=#
```

```
test=# SELECT * FROM person WHERE id=1;
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
1	Trstram	Germann	tgermann0@photobucket.com	Male	2022-02-20	Luxembourg
1	John	Smith	js@gmail.com	Male	1953-01-10	England

(2 rows)

9.4 Alter Table to Add pkey Constraint

```
test=# ALTER TABLE person ADD PRIMARY KEY(id);
ERROR:  could not create unique index "person_pkey"
DETAIL:  Key (id)=(1) is duplicated.
```

Use alter table to add a primary key to a table.

Note that if there are duplicate rows with the same value in the column that I want to add the primary key, then it will fail because this **column does not contain unique values**.

```
test=# DELETE FROM person where id=1;
DELETE 2

test=# SELECT * FROM person WHERE id=1;
 id | first_name | last_name | email | gender | date_of_birth | country_of_birth
-----+-----+-----+-----+-----+-----+-----
(0 rows)
```

To solve this I have to delete all rows with the same id.

This **causes data loss** of two rows of person. These rows would have to manually inserted into the database ensuring not to duplicate the primary key.

```
test=# INSERT INTO person (id, first_name, last_name, gender, email, date_of_birth, country_of_birth) VALUES
(1, 'John', 'Smith', 'Male', 'js@gmail.com', '1953-01-10', 'England');
INSERT 0 1
```

```
test=# ALTER TABLE person ADD PRIMARY KEY(id);
ERROR:  could not create unique index "person_pkey"
DETAIL:  Key (id)=(1) is duplicated.
```

The primary key can now be successfully added.

When I describe the table the primary key is now added.

```
test-# \d person
```

Table "public.person"				
Column	Type	Collation	Nullable	Default
id	bigint		not null	nextval('person_id_seq'::regclass)
first_name	character varying(50)		not null	
last_name	character varying(50)		not null	
email	character varying(150)			
gender	character varying(50)		not null	
date_of_birth	date			
country_of_birth	character varying(50)		not null	

Indexes:

```
"person_pkey" PRIMARY KEY, btree (id)
```

9.5 Unique Constraints

```
test=# SELECT email, COUNT(*) FROM person GROUP BY email;
```

email	count
lmacneicnq@webeden.co.uk	1
lscandred8z@ed.gov	1
civannikovgfv@1688.com	1
	311
sdeortega3o@bbb.org	1
tgoodwill1x@house.gov	1
...	
eadnamnj@squarespace.com	
estoven79@xrea.com	
jhuffer6c@cdbaby.com	

(690 rows)

In the person table I can see that 60 people have an email address and 311 have NULL email addresses.

```
test=# SELECT email, COUNT(*) FROM person GROUP BY email HAVING COUNT(*) > 1;
email | count
-----+-----
      | 311
(1 row)
```

Now I insert two people with the same email address.

```
test=# insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values
('Trstram', 'Germann', 'tgermann0@photobucket.com', 'Male', '2022-02-20', 'Luxembourg');
INSERT 0 1
test=# insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values
('Geoff', 'Pullman', 'tgermann0@photobucket.com', 'Male', '1998-02-20', 'Spain');
INSERT 0 1
```

```
test=# SELECT email, COUNT(*) FROM person GROUP BY email HAVING COUNT(*) > 1;
email | count
-----+-----
      | 311
tgermann0@photobucket.com | 2
(2 rows)
```

I can see that I have two duplicate email addresses.

Logically, two people cannot have the same email address so if I wanted to send an email to Geoff it would also go to Tristram.

```
test=# SELECT * FROM person WHERE email = 'tgermann0@photobucket.com';
```

id	first_name	last_name	email	gender	date_of_birth	country_of_birth
1001	Trstram	Germann	tgermann0@photobucket.com	Male	2022-02-20	Luxembourg
1002	Geoff	Pullman	tgermann0@photobucket.com	Male	1998-02-20	Spain

(2 rows)

So I would want to add a unique constraint to not allow duplicate emails in the email column. Unique Constraint is not the same as a primary key.

```
test=# ALTER TABLE person ADD CONSTRAINT unique_email_address UNIQUE (email);
ERROR:  could not create unique index "unique_email_address"
DETAIL:  Key (email)=(tgermann0@photobucket.com) is duplicated.
```

But of course when I try and add the UNIQUE constraint to email it fails.

```
test=# DELETE FROM person where id=1002;
DELETE 1
```

I could modify the email of Geoff, or nullify it but for now I will just delete it.

```
test=# ALTER TABLE person ADD CONSTRAINT unique_email_address UNIQUE (email);
ALTER TABLE
```

Now the UNIQUE constraint can be added.

Now if I describe the table I can see the named UNIQUE index on the email column. Note that when defining the constraint I gave it the descriptive name of “unique_email_address” so that I can identify what it is doing easily.

```
test=# \d person
Table "public.person"
  Column      |      Type      | Collation | Nullable |      Default
-----+-----+-----+-----+-----
 id           | bigint         |           | not null | nextval('person_id_seq'::regclass)
 first_name   | character varying(50) |           | not null |
 last_name    | character varying(50) |           | not null |
 email        | character varying(150) |           |          |
 gender       | character varying(50) |           | not null |
 date_of_birth | date           |           |          |
 country_of_birth | character varying(50) |           | not null |
Indexes:
    "person_pkey" PRIMARY KEY, btree (id)
    "unique_email_address" UNIQUE CONSTRAINT, btree (email)

test=#
```

And if I now try and insert a duplicate email address it thros an error.

```
test=# insert into person (first_name, last_name, email, gender, date_of_birth, country_of_birth) values
('Geoff', 'Pullman', 'tgermann0@photobucket.com', 'Male', '1998-02-20', 'Spain');
ERROR:  duplicate key value violates unique constraint "unique_email_address"
DETAIL:  Key (email)=(tgermann0@photobucket.com) already exists.

test=#
```

```
test=# ALTER TABLE person DROP CONSTRAINT unique_email_address;
ALTER TABLE
```

Now I will drop the constraint

```
test=# \d person
```

Table "public.person"				
Column	Type	Collation	Nullable	Default
id	bigint		not null	nextval('person_id_seq'::regclass)
first_name	character varying(50)		not null	
last_name	character varying(50)		not null	
email	character varying(150)			
gender	character varying(50)		not null	
date_of_birth	date			
country_of_birth	character varying(50)		not null	

Indexes:

"person_pkey" PRIMARY KEY, btree (id)

```
test=# ALTER TABLE person ADD UNIQUE(email);
ALTER TABLE
```

I can ADD a UNIQUE constraint without specifying a name and allow postgreSQL to define the name.

```
test=# \d person
```

Table "public.person"				
Column	Type	Collation	Nullable	Default
id	bigint		not null	nextval('person_id_seq'::regclass)
first_name	character varying(50)		not null	
last_name	character varying(50)		not null	
email	character varying(150)			
gender	character varying(50)		not null	
date_of_birth	date			
country_of_birth	character varying(50)		not null	

Indexes:

"person_pkey" PRIMARY KEY, btree (id)

"person_email_key" UNIQUE CONSTRAINT, btree (email)

```
test=#
```

9.6 Check Constraints

Check constraint allows us to specify that the value in a certain column must satisfy a Boolean (truth-value) expression.

Lets say that we want to have biological gender as Male or Femail and identifies_as_gender the same values as shown.

```
test=# SELECT DISTINCT gender FROM person;
gender
-----
Genderqueer
Bigender
Genderfluid
Male
Polygender
Non-binary
Female
Agender
(8 rows)
```

The CHECK() constraint takes a condition where I define the two possible values for that column.

```
test=# ALTER TABLE person ADD CONSTRAINT biological_gender_constraint CHECK (gender = 'Male' OR gender = 'Female');
ERROR:  check constraint "biological_gender_constraint" of relation "person" is violated by some row
test=#
```

But in this example the creation of the check constraint fails because there are rows where gender is not male or female.

```
create table person (
    id BIGSERIAL NOT NULL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(150),
    bio_gender VARCHAR(2) NOT NULL,
    id_gender VARCHAR(20) NOT NULL,
    date_of_birth DATE NOT NULL,
    country_of_birth VARCHAR(50) NOT NULL
```

I dropped the table and generated a new set of person data with bio_gender as 'M' or 'F' then id_gender. I added in the email UNIQUE constraint.

```
);
insert into person (id, first_name, last_name, email, bio_gender, id_gender, date_of_birth, country_of_birth)
values (1, 'Haleigh', 'Feldklein', null, 'M', 'Non-binary', '1909-03-19', 'Brazil');
```



```
test=# \d person
```

Table "public.person"				
Column	Type	Collation	Nullable	Default
id	bigint		not null	nextval('person_id_seq'::regclass)
...				
bio_gender	character varying(2)		not null	
id_gender	character varying(20)		not null	
...				
"person_pkey" PRIMARY KEY, btree (id)				
"unique_email_address" UNIQUE CONSTRAINT, btree (email)				

```
test=# ALTER TABLE person ADD CONSTRAINT biological_gender_constraint CHECK (bio_gender = 'M' OR
bio_gender = 'F');
```

```
ALTER TABLE
```

```
test=#
```

```
test=# \d person
```

Table "public.person"				
Column	Type	Collation	Nullable	Default
id	bigint		not null	nextval('person_id_seq'::regclass)
first_name	character varying(50)		not null	
last_name	character varying(50)		not null	
email	character varying(150)			
bio_gender	character varying(2)		not null	
id_gender	character varying(20)		not null	
date_of_birth	date		not null	
country_of_birth	character varying(50)		not null	

```
Indexes:
```

```
    "person_pkey" PRIMARY KEY, btree (id)
```

```
    "unique_email_address" UNIQUE CONSTRAINT, btree (email)
```

```
Check constraints:
```

```
    "biological_gender_constraint" CHECK (bio_gender::text = 'M'::text OR bio_gender::text = 'F'::text)
```

**Now I can add the
CHECK constraint on
bio_gender.**

If I try and add in a new person with a bio_gener value that is not M or F the insert query fails because of the check constraint.

```
test=# INSERT INTO person (id, first_name, last_name, email, bio_gender, id_gender, date_of_birth,
country_of_birth) VALUES (1, 'Haleigh', 'Feldklein', null, 'N', 'Non-binary', '1909-03-19', 'Brazil');
ERROR:  new row for relation "person" violates check constraint "biological_gender_constraint"
DETAIL:  Failing row contains (1, Haleigh, Feldklein, null, N, Non-binary, 1909-03-19, Brazil).
```

Check constraints are really powerful and means of enforcing data integrity within the database.

Another example would be that a numerical price value is always a positive number i.e. $\text{price} > 0$.

10. Deleting & Updating Rows

[10.1 Delete From table](#)

[10.2 Delete From table by primary key](#)

[10.3 Delete From table by non primary key](#)

[10.4 Update a record to set a Column to Value](#)

[10.5 On Conflict Do Nothing](#)

[10.6 Upsert \(On Conflict do Update\)](#)

10.1 Delete From table

```
test=# DELETE FROM person;  
DELETE 1000
```

```
test=# SELECT * FROM person;  
 id | first_name | last_name | email | bio_gender | id_gender | date_of_birth | country_of_birth  
----+-----+-----+-----+-----+-----+-----+-----  
(0 rows)
```

```
test=#
```

A badly typed SQL query can erase all data from a table. Note that **'DELETE FROM <table_name>;'** without an asterick to denote 'all' will also delete all rows from a table. Note that only the data is deleted, not the structure of the table.

```
test=# \d person
```

Table "public.person"				
Column	Type	Collation	Nullable	Default
id	bigint		not null	nextval('person_id_seq'::regclass)
first_name	character varying(50)		not null	
last_name	character varying(50)		not null	
email	character varying(150)			
bio_gender	character varying(2)		not null	
id_gender	character varying(20)		not null	
date_of_birth	date		not null	
country_of_birth	character varying(50)		not null	

Indexes:

```
"person_pkey" PRIMARY KEY, btree (id)
```

```
"unique_email_address" UNIQUE CONSTRAINT, btree (email)
```

Check constraints:

```
"biological_gender_constraint" CHECK (bio_gender::text = 'M'::text OR bio_gender::text = 'F'::text)
```

10.2 Delete From table by primary key

I have restored the table importing the SQL query as done before.

Conclusion is that it is best practice to identify a row for deletion with the primary key to ensure that only the specified unique row is deleted.

```
test=# DELETE FROM person WHERE id=1;
DELETE 1
```

```
test=# SELECT from person WHERE id=1;
--
(0 rows)
```

```
test=# SELECT * FROM person;
```

	id	first_name	last_name	email	bio_gender	id_gender	date_of_birth	country_of_birth
2	Ameline	Gittoes		agittoes1@parallels.com	F	Female	2017-04-19	Philippines
3	Mortie	Heck			F	Male	1996-02-10	Thailand
4	Rogers	Woofinden			M	Male	2016-01-12	China
5	Tamma	Jemison			M	Female	2022-12-15	Indonesia
6	Bree	Ashbolt		bashbolt5@sakura.ne.jp	M	Female	2017-12-26	Indonesia
7	Holly	Simnor			F	Female	1957-01-29	China
8	Paulina	Di Ruggero			M	Bigender	2021-02-26	Israel
9	Moore	Emmines		memmines8@dmoz.org	M	Male	2023-03-15	Indonesia

10.3 Delete From table by non primary key

```
test=# DELETE FROM person WHERE bio_gender='F';  
DELETE 504
```

I can delete multiple rows from a table with a WHERE clause. E.G. delete all females from the person table.

```
test=# DELETE FROM person WHERE bio_gender = 'M' AND country_of_birth = 'China';  
DELETE 102
```

I can delete multiple rows using WHERE, AND clauses

Using the where clause can be used to filter rows for deleting and can be combined with an AND clause to select multiple rows for deletion.

For the next exercise I deleted all rows of person and restored all 1000 rows from the SQL file by importing the queries.

10.4 Update a record to set a Column value

At the moment person with id of 1 has a null email address.

```
test=# SELECT * FROM person WHERE id=1;
 id | first_name | last_name | email | bio_gender | id_gender | date_of_birth | country_of_birth
-----+-----+-----+-----+-----+-----+-----+-----
  1 | Haleigh   | Feldklein |      | M          | Non-binary | 1909-03-19    | Brazil
(1 row)
```

```
test=# UPDATE person SET email = 'haleigh44@gmail.com' WHERE id=1;
UPDATE 1
```

Using an UPDATE query I can SET new values for multiple columns WHERE id matches.

```
test=# SELECT * FROM person WHERE id=1;
 id | first_name | last_name | email | bio_gender | id_gender | date_of_birth | country_of_birth
-----+-----+-----+-----+-----+-----+-----+-----
  1 | Haleigh   | Feldklein | haleigh44@gmail.com | M          | Non-binary | 1909-03-19    | Brazil
(1 row)
```

```
test=# UPDATE person SET first_name = 'Hally', Last_name = 'Felder' WHERE id=1;
UPDATE 1
```

Multiple columns can be changed within the same update query

```
test=# SELECT * FROM person WHERE id=1;
 id | first_name | last_name | email | bio_gender | id_gender | date_of_birth | country_of_birth
-----+-----+-----+-----+-----+-----+-----+-----
  1 | Hally      | Felder   | haleigh44@gmail.com | M          | Non-binary | 1909-03-19    | Brazil
(1 row)
```

Without the where clause, all rows would get updated, except for email column which has a unique constraint.

10.5 On Conflict Do Nothing

```
test=# SELECT * FROM person WHERE id=1;
 id | first_name | last_name | email | bio_gender | id_gender | date_of_birth | country_of_birth
-----+-----+-----+-----+-----+-----+-----+-----
  1 | Hally      | Felder    | haleigh44@gmail.com | M          | Non-binary | 1909-03-19    | Brazil
(1 row)
```

The id field is what uniquely identifies each person on the table.

```
test=# INSERT INTO person (id, first_name, last_name, email, bio_gender, id_gender, date_of_birth,
country_of_birth) VALUES ('1', 'Hally', 'Felder', 'haleigh44@gmail.com', 'M', 'Non-binary', '1909-03-19',
'Brazil');
ERROR:  duplicate key value violates unique constraint "person_pkey"
DETAIL:  Key (id)=(1) already exists.
```

If I try to perform a query which might cause a conflict I get an error. Note how it says that the id column with a value of 1 already exists.

```
test=# INSERT INTO person (id, first_name, last_name, email, bio_gender, id_gender, date_of_birth,
country_of_birth) VALUES ('1', 'Hally', 'Felder', 'haleigh44@gmail.com', 'M', 'Non-binary', '1909-03-19',
'Brazil') ON CONFLICT(id) DO NOTHING;
INSERT 0 0
```

Now I can add to the query to specify the column that might conflict and I can specify to do nothing. Note how PostgreSQL does not throw an error message but simply does not perform the query.


```
test=# INSERT INTO person (id, first_name, last_name, email, bio_gender, id_gender, date_of_birth,
country_of_birth) VALUES ('1', 'Hally', 'Felder', 'haleigh44@gmail.com', 'M', 'Non-binary', '1909-03-19',
'Brazil') ON CONFLICT(email) DO NOTHING;
ERROR:  duplicate key value violates unique constraint "person_pkey"
DETAIL:  Key (id)=(1) already exists.
```

Note that if I try and update with an on conflict do nothing against the email column I get an error message because the primary key is also unique.

```
test=# INSERT INTO person (id, first_name, last_name, email, bio_gender, id_gender, date_of_birth,
country_of_birth) VALUES ('1001', 'Hally', 'Felder', 'haleigh44@gmail.com', 'M', 'Non-binary', '1909-03-19',
'Brazil') ON CONFLICT(email) DO NOTHING;
INSERT 0 0
```

```
test=# INSERT INTO person (id, first_name, last_name, email, bio_gender, id_gender, date_of_birth,
country_of_birth) VALUES ('1001', 'Hally', 'Felder', 'haleigh44@gmail.com', 'M', 'Non-binary', '1909-03-19',
'Brazil') ON CONFLICT(first_name) DO NOTHING;
ERROR:  there is no unique or exclusion constraint matching the ON CONFLICT specification
```

If I try and set an on conflict do nothing against a column which is not unique I get an error message saying ‘there is no unique or exclusion constraint matching the ON CONFLICT specification’. The takeaway from this is that the on conflict defined column must be a unique column for this to work.

10.6 Upsert (On Conflict do Update)

Upsert is a combination of update and insert. The upsert allows you to update an existing row or insert a new one if it doesn't exist. For example a new user registers with the database entering personal details such as first name, surname, email gender etc...

Upsert is typically used in a distributed system where one or more SQL servers are running behind a load balancer.

Lets say that shortly the same user registers again but with a different email what would be the outcome?

```
test=# SELECT * FROM person WHERE id=1;
```

id	first_name	last_name	email	bio_gender	id_gender	date_of_birth	country_of_birth
1	Hally	Felder	haleigh44@gmail.com	M	Non-binary	1909-03-19	Brazil

(1 row)

```
test=# INSERT INTO person (id, first_name, last_name, email, bio_gender, id_gender, date_of_birth,
country_of_birth) VALUES ('1', 'Hally', 'Felder', 'hally@hotmail.com', 'M', 'Non-binary', '1909-03-19',
'Brazil') ON CONFLICT(email) DO NOTHING;
ERROR:  duplicate key value violates unique constraint "person_pkey"
DETAIL:  Key (id)=(1) already exists.
```

```
test=# INSERT INTO person (id, first_name, last_name, email, bio_gender, id_gender, date_of_birth,
country_of_birth) VALUES ('1', 'Hally', 'Felder', 'hally@hotmail.com', 'M', 'Non-binary', '1909-03-19',
'Brazil') ON CONFLICT(id) DO NOTHING;
INSERT 0 0
```

The net result is that we cannot insert a row into the table where any of the unique fields are duplicated.

```
test=# INSERT INTO person (id, first_name, last_name, email, bio_gender, id_gender, date_of_birth,
country_of_birth) VALUES ('1', 'Hally', 'Felder', 'hally@hotmail.com', 'M', 'Non-binary', '1909-03-19',
'Brazil') ON CONFLICT(id) DO UPDATE SET email = EXCLUDED.email;
INSERT 0 1
```

Now I modify the on conflict(id) to say DO UPDATE and then SET the email to the EXCLUDED.email. Basically it is overriding the conflict on the ID parameter to run an update query to set the email.

```
test=# SELECT * FROM person WHERE id=1;
```

id	first_name	last_name	email	bio_gender	id_gender	date_of_birth	country_of_birth
1	Hally	Felder	hally@hotmail.com	M	Non-binary	1909-03-19	Brazil

(1 row)

The update query of the on conflict part has successfully run and updated the email.

```
test=# INSERT INTO person (id, first_name, last_name, email, bio_gender, id_gender, date_of_birth,
country_of_birth) VALUES ('1', 'Harry', 'Frederick', 'harry@hotmail.com', 'M', 'Non-binary', '1909-03-19',
'Brazil') ON CONFLICT(id) DO UPDATE SET email = EXCLUDED.email, last_name = EXCLUDED.last_name,
first_name=EXCLUDED.first_name;
INSERT 0 1
```

I can use this method to update multiple columns to the excluded values and it does not matter what order the excluded values are written in.

```
test=# SELECT * FROM person WHERE id=1;
```

id	first_name	last_name	email	bio_gender	id_gender	date_of_birth	country_of_birth
1	Harry	Frederick	harry@hotmail.com	M	Non-binary	1909-03-19	Brazil

(1 row)

11. Joins

[11.1 Joins and Relationships Theory](#)

[11.2 Joins and Relationships lab setup](#)

[11.3 Updating FK columns](#)

[11.4 Inner Joins Theory](#)

[11.5 Inner Joins Query](#)

[11.6 Left Join Theory](#)

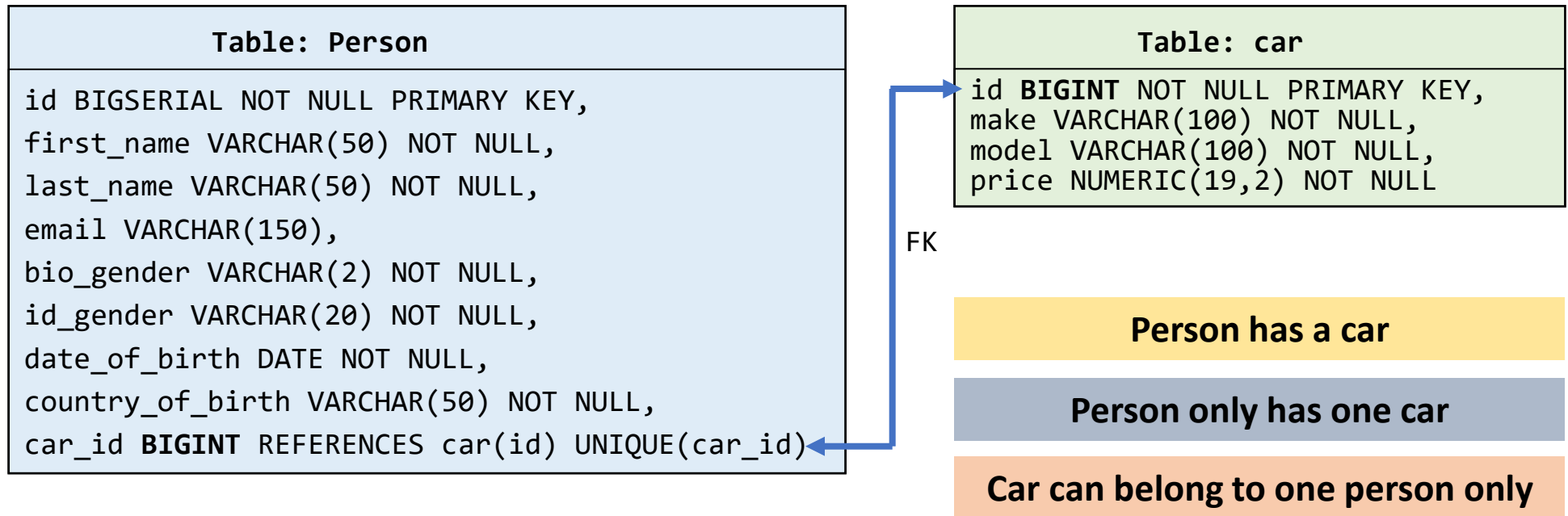
[11.7 Deleting Rows With a Join](#)

11.1 Joins and Relationships Theory

So we have an abstract table which contains details about people and an abstract table of data about cars. If we wanted to relate people to cars then we could just expand the person table to include extra columns about their cars but this would be bad practice.

The correct solution is to define a foreign key relationship (FK) between the two tables.

Relationship



For the FK relationship to work the types have to be identical, in this case **bigint**.

11.2 Joins and Relationships lab setup

For this exercise I dropped the people and car table and imported the below two tables and data. Note that for this exercise a person can only have one car and a car can only belong to one person.

```
create table car (  
  id BIGSERIAL NOT NULL PRIMARY KEY,  
  make VARCHAR(100) NOT NULL,  
  model VARCHAR(100) NOT NULL,  
  price numeric(19, 1) NOT NULL  
);  
  
create table person (  
  id BIGSERIAL NOT NULL PRIMARY KEY,  
  first_name VARCHAR(50) NOT NULL,  
  last_name VARCHAR(50) NOT NULL,  
  gender VARCHAR(1) NOT NULL,  
  email VARCHAR(100),  
  date_of_birth DATE NOT NULL,  
  country_of_birth VARCHAR(50) NOT NULL,  
  car_id BIGINT REFERENCES car (id),  
  UNIQUE (car_id)  
);
```

Note the Car_id column which references car (id). This is the foreign key.

Because each person can only have one or no car, the car_id is UNIQUE.

Note that table car does not exist at the time of executing the SQL to create table person and because this references car table it will throw an error so the car table has to be created first.

```
insert into person (id, first_name, last_name, gender, email, date_of_birth, country_of_birth) values (1,  
'Haleigh', 'Feldklein', 'M', 'haleighfeldklein@hotmail.com', '1909-03-19', 'Brazil');  
insert into person (id, first_name, last_name, gender, email, date_of_birth, country_of_birth) values (2,  
'Ameline', 'Gittoes', 'F', 'agittoes1@parallels.com', '2017-04-19', 'Philippines');  
insert into person (id, first_name, last_name, gender, email, date_of_birth, country_of_birth) values (3,  
'Mortie', 'Heck', 'M', 'mortieHeck@gmail.com', '1996-02-10', 'Thailand');  
  
insert into car (id, make, model, price) values (1, 'Dodge', 'Caravan', '85534.98');  
insert into car (id, make, model, price) values (2, 'Jaguar', 'S-Type', '95099.14');
```

11.3 Updating FK columns

```
test=# SELECT * FROM person;
```

id	first_name	last_name	gender	email	date_of_birth	country_of_birth	car_id
1	Haleigh	Feldklein	M	haleighfeldklein@hotmail.com	1909-03-19	Brazil	
2	Ameline	Gittoes	F	agittoes1@parallels.com	2017-04-19	Philippines	
3	Mortie	Heck	M	mortieHeck@gmail.com	1996-02-10	Thailand	

```
(3 rows)
```

```
test=# \d person
```

Table "public.person"				
Column	Type	Collation	Nullable	Default
id	bigint		not null	nextval('person_id_seq'::regclass)
first_name	character varying(50)		not null	
last_name	character varying(50)		not null	
gender	character varying(1)		not null	
email	character varying(100)			
date_of_birth	date		not null	
country_of_birth	character varying(50)		not null	
car_id	bigint			

```
Indexes:
```

```
    "person_pkey" PRIMARY KEY, btree (id)
```

```
    "person_car_id_key" UNIQUE CONSTRAINT, btree (car_id)
```

```
Foreign-key constraints:
```

```
    "person_car_id_fkey" FOREIGN KEY (car_id) REFERENCES car(id)
```

Note that the car_id column of person table is currently empty. Also note the Foreign-key-constraint on the person table car_id column.

```
test=# UPDATE person SET car_id = 2 WHERE id = 1;  
UPDATE 1
```

Use an update query to SET the car_id column

```
test=# SELECT * FROM person;
```

id	first_name	last_name	gender	email	date_of_birth	country_of_birth	car_id
2	Ameline	Gittoes	F	agittoes1@parallels.com	2017-04-19	Philippines	
3	Mortie	Heck	M	mortieHeck@gmail.com	1996-02-10	Thailand	
1	Haleigh	Feldklein	M	haleighfeldklein@hotmail.com	1909-03-19	Brazil	2

(3 rows)

If I try and update car_id of another person by setting the same car(id) it throws an error because of the unique constraint on the car_id column.

```
test=# UPDATE person SET car_id = 2 WHERE id = 2;  
ERROR:  duplicate key value violates unique constraint "person_car_id_key"  
DETAIL:  Key (car_id)=(2) already exists.
```

```
test=# UPDATE person SET car_id = 1 WHERE id = 2;  
UPDATE 1
```

```
test=# SELECT * FROM person;
```

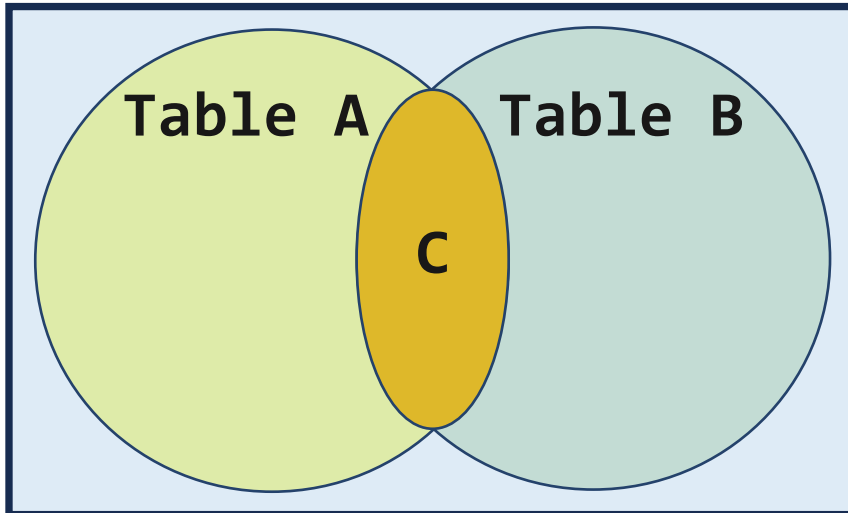
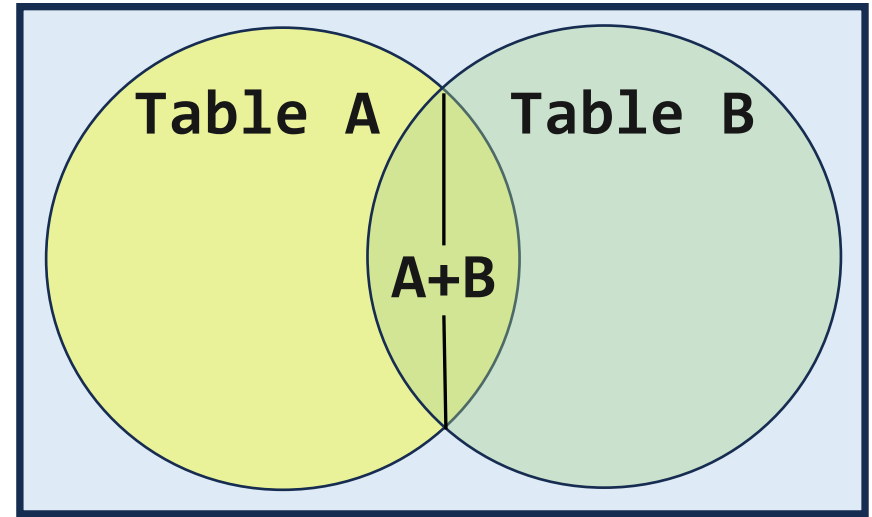
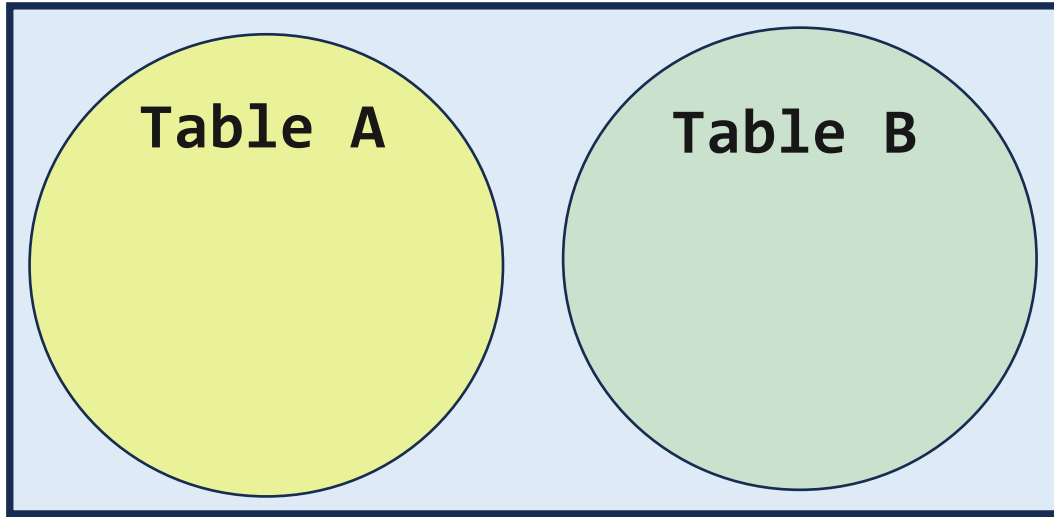
id	first_name	last_name	gender	email	date_of_birth	country_of_birth	car_id
3	Mortie	Heck	M	mortieHeck@gmail.com	1996-02-10	Thailand	
1	Haleigh	Feldklein	M	haleighfeldklein@hotmail.com	1909-03-19	Brazil	2
2	Ameline	Gittoes	F	agittoes1@parallels.com	2017-04-19	Philippines	1

(3 rows)

If I try and update car_id of a person with a car(id) that does not exist I get an error because of the Foreign Key constraint.

```
test=# UPDATE person SET car_id = 4 WHERE id = 2;  
ERROR:  insert or update on table "person" violates foreign key constraint "person_car_id_fkey"  
DETAIL:  Key (car_id)=(4) is not present in table "car".
```


11.4 Inner Joins Theory



Inner join works by taking what is common in table A and Table B and gives the result of what is common in both tables as a new record, C.

11.5 Inner Joins Query

```
test=# SELECT * FROM person;
 id | first_name | last_name | gender |          email          | date_of_birth | country_of_birth | car_id
-----+-----+-----+-----+-----+-----+-----+-----
  3 | Mortie     | Heck      | M      | mortieHeck@gmail.com   | 1996-02-10    | Thailand         | 
  1 | Haleigh    | Feldklein | M      | haleighfeldklein@hotmail.com | 1909-03-19    | Brazil           | 2
  2 | Ameline    | Gittoes   | F      | agittoes1@parallels.com | 2017-04-19    | Philippines      | 1
(3 rows)
```

Table A

```
test=# SELECT * FROM car;
 id | make  | model  | price
-----+-----+-----+-----
  1 | Dodge | Caravan | 85535.0
  2 | Jaguar | S-Type | 95099.1
(2 rows)
```

Table B

```
test=# SELECT * FROM person JOIN car ON person.car_id = car.id;
 id | first_name | last_name | gender |          email          | date_of_birth | country_of_birth | car_id | id | make  | model  | price
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
  2 | Ameline    | Gittoes   | F      | agittoes1@parallels.com | 2017-04-19    | Philippines      | 1      | 1 | Dodge | Caravan | 85535.0
  1 | Haleigh    | Feldklein | M      | haleighfeldklein@hotmail.com | 1909-03-19    | Brazil           | 2      | 2 | Jaguar | S-Type | 95099.1
(2 rows)
```

Table A + Table B = Table C

```
test=# SELECT * FROM person JOIN car ON person.car_id = car.id;
-[ RECORD 1 ]-----+-----
 id              | 2
 first_name      | Ameline
 last_name       | Gittoes
 gender          | F
 email           | agittoes1@parallels.com
 date_of_birth   | 2017-04-19
 country_of_birth | Philippines
 car_id          | 1
 id              | 1
 make            | Dodge
 model           | Caravan
 price           | 85535.0
-[ RECORD 2 ]-----+-----
 id              | 1
 first_name      | Haleigh
 last_name       | Feldklein
 gender          | M
 email           | haleighfeldklein@hotmail.com
 date_of_birth   | 1909-03-19
 country_of_birth | Brazil
 car_id          | 2
 id              | 2
 make            | Jaguar
 model           | S-Type
 price           | 95099.1
```

I can combine the columns of table A and Table B to make new Table C by using an inner Join.

SELECT * FROM <table_a> JOIN <table_b> ON
<table_a>.<table_a_column_FK> =
<table_b>.<column>

```
test=# \x
Expanded display is on.
```

When joining tables with many columns often the screen will not be wide enough to fit the whole table in so \x can be used to turn on expanded display and the table will output as shown.

```
test=# SELECT person.first_name, car.make, car,model, car,price FROM person JOIN car ON person.car_id = car.id;
-[ RECORD 1 ]-----
first_name | Ameline
make       | Dodge
car        | (1,Dodge,Caravan,85535.0)
model      | Caravan
car        | (1,Dodge,Caravan,85535.0)
price      | 85535.0
-[ RECORD 2 ]-----
first_name | Haleigh
make       | Jaguar
car        | (2,Jaguar,S-Type,95099.1)
model      | S-Type
car        | (2,Jaguar,S-Type,95099.1)
price      | 95099.1
```

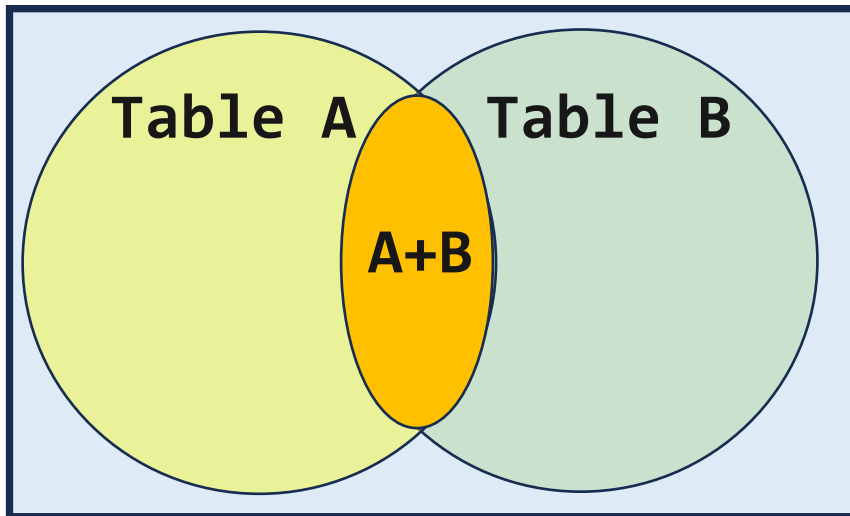
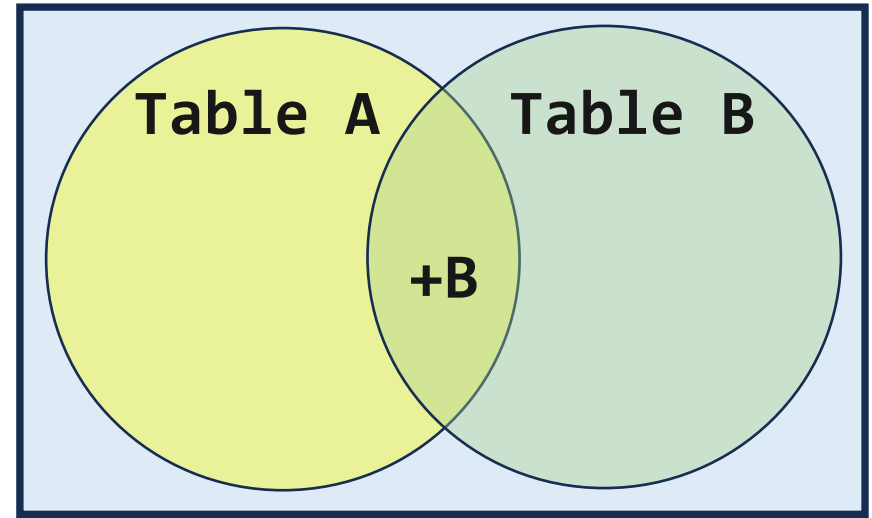
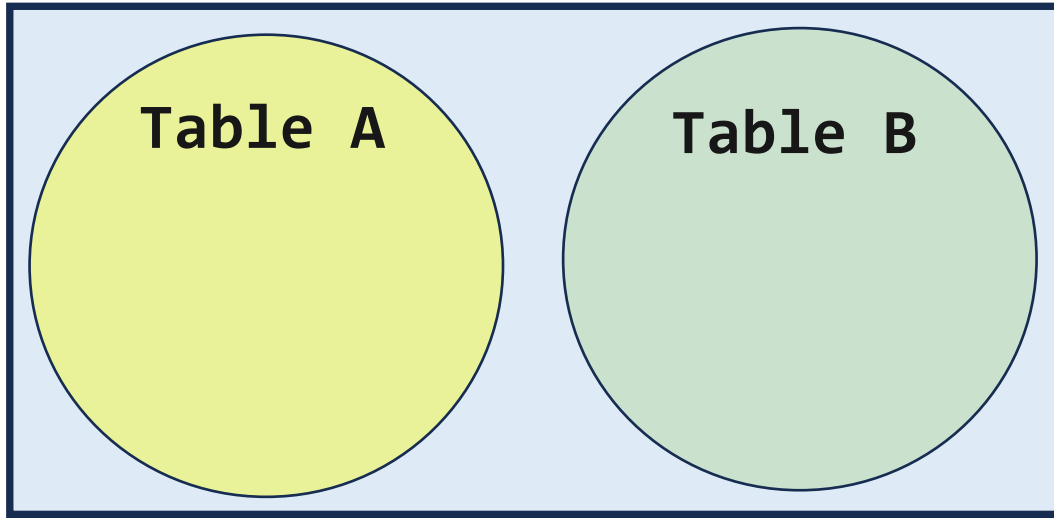
I can also use the table.column notation to target specific columns from both tables to return in the table C query results.

```
test=# \x
Expanded display is off.
```

\x is also used to turn off expanded display. It functions as a toggle.

```
test=# SELECT person.first_name, car.make, car,model, car,price FROM person JOIN car ON person.car_id = car.id;
 first_name | make | car | model | car | price
-----+-----+-----+-----+-----+-----
 Ameline   | Dodge | (1,Dodge,Caravan,85535.0) | Caravan | (1,Dodge,Caravan,85535.0) | 85535.0
 Haleigh   | Jaguar | (2,Jaguar,S-Type,95099.1) | S-Type | (2,Jaguar,S-Type,95099.1) | 95099.1
(2 rows)
```

11.6 Left Join Theory



A left join includes all the rows from the left table (table A) as well as the records from table B that have a corresponding relationship as well as the records that do not have a corresponding relationship to make table C.

11.6 Left Join Query

```
test=# SELECT * FROM person JOIN car ON person.car_id = car.id;
```

id	first_name	last_name	gender	email	date_of_birth	country_of_birth	car_id	id	make	model	price
2	Ameline	Gittoes	F	agittoes1@parallels.com	2017-04-19	Philippines	1	1	Dodge	Caravan	85535.0
1	Haleigh	Feldklein	M	haleighfeldklein@hotmail.com	1909-03-19	Brazil	2	2	Jaguar	S-Type	95099.1

(2 rows)

From the inner join above we can see that from table A, person we only have two columns because only two rows have a relationship with table B, car.

```
test=# SELECT * FROM person LEFT JOIN car ON person.car_id = car.id;
```

id	first_name	last_name	gender	email	date_of_birth	country_of_birth	car_id	id	make	model	price
2	Ameline	Gittoes	F	agittoes1@parallels.com	2017-04-19	Philippines	1	1	Dodge	Caravan	85535.0
1	Haleigh	Feldklein	M	haleighfeldklein@hotmail.com	1909-03-19	Brazil	2	2	Jaguar	S-Type	95099.1
3	Mortie	Heck	M	mortieHeck@gmail.com	1996-02-10	Thailand					

(3 rows)

If I run the same query as a left Join, see that all records from table A are included as well as corresponding records from table B.

```
test=# SELECT person.first_name, car.make, car,model, car,price FROM person LEFT JOIN car ON person.car_id = car.id;
```

first_name	make	car	model	car	price
Ameline	Dodge	(1,Dodge,Caravan,85535.0)	Caravan	(1,Dodge,Caravan,85535.0)	85535.0
Haleigh	Jaguar	(2,Jaguar,S-Type,95099.1)	S-Type	(2,Jaguar,S-Type,95099.1)	95099.1
Mortie					

(3 rows)

With a left join I can seleect all people even if they do not have a car.

```
test=# SELECT * FROM person LEFT JOIN car ON person.car_id = car.id;
```

id	first_name	last_name	gender	email	date_of_birth	country_of_birth	car_id	id	make	model	price
2	Ameline	Gittoes	F	agittoes1@parallels.com	2017-04-19	Philippines	1	1	Dodge	Caravan	85535.0
1	Haleigh	Feldklein	M	haleighfeldklein@hotmail.com	1909-03-19	Brazil	2	2	Jaguar	S-Type	95099.1
3	Mortie	Heck	M	mortieHeck@gmail.com	1996-02-10	Thailand					

(3 rows)

```
test=# SELECT * FROM PERSON WHERE car_id IS NULL;
```

id	first_name	last_name	gender	email	date_of_birth	country_of_birth	car_id
3	Mortie	Heck	M	mortieHeck@gmail.com	1996-02-10	Thailand	

(1 row)

Lets say I want to find all people without a car. The Obvious solution would be to select all people where the car_id column is NULL (empty).

```
test=# SELECT * FROM person LEFT JOIN car ON person.car_id = car.id WHERE car.* IS NULL;
```

id	first_name	last_name	gender	email	date_of_birth	country_of_birth	car_id	id	make	model	price
3	Mortie	Heck	M	mortieHeck@gmail.com	1996-02-10	Thailand					

(1 row)

But to see all columns of person and car I can amend the left join query with a WHERE clause to select all from car when all columns are NULL.

11.7 Deleting Rows With a Join

```
test=# insert into car (id, make, model, price) values (3, 'Lexus', 'GX', '70495.11');
INSERT 0 1
test=# insert into person (id, first_name, last_name, gender, email, date_of_birth,
country_of_birth) values (4000, 'john', 'Smith', 'M', null, '2016-01-12', 'Spain');
INSERT 0 1
```

Now I have inserted another person and another car but with no FK relationship between the two tables.

```
test=# SELECT person.id, person.first_name, car.make, car,model, car,price FROM person LEFT JOIN car ON
person.car_id = car.id;
```

id	first_name	make	car	model	car	price
2	Ameline	Dodge	(1,Dodge,Caravan,85535.0)	Caravan	(1,Dodge,Caravan,85535.0)	85535.0
1	Haleigh	Jaguar	(2,Jaguar,S-Type,95099.1)	S-Type	(2,Jaguar,S-Type,95099.1)	95099.1
4000	john					
3	Mortie					

(4 rows)

```
test=# SELECT * FROM car;
```

id	make	model	price
1	Dodge	Caravan	85535.0
2	Jaguar	S-Type	95099.1
3	Lexus	GX	70495.1

(3 rows)

And I can see that there are three cars now in the car table

```
test=# UPDATE person SET car_id = 3 WHERE person.id = 4000;
UPDATE 1
```

Now I can assign car with Id of 3 to John with id of 4000 using an UPDATE query I can see that John now has a car.

```
test=# SELECT person.id, person.first_name, car.make, car,model, car,price FROM person LEFT JOIN car ON
person.car_id = car.id;
```

id	first_name	make	car	model	car	price
2	Ameline	Dodge	(1,Dodge,Caravan,85535.0)	Caravan	(1,Dodge,Caravan,85535.0)	85535.0
1	Haleigh	Jaguar	(2,Jaguar,S-Type,95099.1)	S-Type	(2,Jaguar,S-Type,95099.1)	95099.1
4000	john	Lexus	(3,Lexus,GX,70495.1)	GX	(3,Lexus,GX,70495.1)	70495.1
3	Mortie					

(4 rows)

```
test=# DELETE from car WHERE id = 3;
ERROR:  update or delete on table "car" violates foreign key constraint "person_car_id_fkey" on table "person"
DETAIL:  Key (id)=(3) is still referenced from table "person".
```

If I try and delete car 3 (assigned to John with FK) I get an error telling me this.

Default SQL behavoiur is to not permit deletion of a row that is referenced in another table.

```
test=# UPDATE person SET car_id = null WHERE person.id = 4000;
```

I could update the person table to remove the FK relation.

```
test=# DELETE FROM person WHERE id = 4000;
DELETE 1
```

Note that if I delete a person that has a FK relationship this does not throw an error.

```
test=# SELECT * FROM person WHERE id = 4000;
```

id	first_name	last_name	gender	email	date_of_birth	country_of_birht	car_id
----	------------	-----------	--------	-------	---------------	------------------	--------

(0 rows)


```
test=# DELETE from car WHERE id = 3;  
DELETE 1
```

```
test=# SELECT * FROM car WHERE id = 3;  
 id | make | model | price  
----+-----+-----+-----  
(0 rows)
```

Now I can delete car 3 that was previously assigned to John.

We can have a cascade on our tables where a cascade simply ignores a FK relationship and goes ahead and deletes data regardless. This is bad SQL practice because you always want to have full control of your data and always know what you are deleting. Doing so without knowing what you are doing can be very costly.

12. Export Query Results To CSV

[12.1 Export a Select Query to CSV](#)

12.1 Export a Select Query to CSV

```
test=# SELECT * FROM person LEFT JOIN car ON person.car_id = car.id;
```

id	first_name	last_name	gender	email	date_of_birth	country_of_birth	car_id	id	make	model	price
2	Ameline	Gittoes	F	agittoes1@parallels.com	2017-04-19	Philippines	1	1	Dodge	Caravan	85535.0
1	Haleigh	Feldklein	M	haleighfeldklein@hotmail.com	1909-03-19	Brazil	2	2	Jaguar	S-Type	95099.1
3	Mortie	Heck	M	mortieHeck@gmail.com	1996-02-10	Thailand					

(3 rows)

Lets say I want to output the above query as a CSV file. I need to use the copy command.

test=# \?

...

Input/Output

`\copy ...`

\echo [-n] [STRING]

\i FILE

• • •

```
perform SQL COPY with data stream to the client host
write string to standard output (-n for no newline)
execute commands from file
```

```
test=# \copy (SELECT * FROM person LEFT JOIN car ON person.car_id = car.id) TO
```

COPY 3 ← NOTE: it says we have copied three rows

Use the **\copy** command and **within brackets specify the SQL query** then say **TO** and **specify in quotes the destination path** and **filename with extension**. Finally **DELIMITER** followed by a **coma in quotes** and **specify the file type as CSV**. And that is what I get. I have manually beutified the CSV.

[illegible]

13. Serial and Sequences

[13.1 Explanation of sequences](#)

[13.2 Reset a sequence](#)

13.1 Explanation of sequences

bigserial is a keyword which auto-increments a number but is not actually a datatype. the datatype is **bigint**. The sequencing is controlled by the **nextval default statement**.

```
create table car (  
  id BIGSERIAL NOT NULL PRIMARY KEY,  
  make VARCHAR(100) NOT NULL,  
  model VARCHAR(100) NOT NULL,  
  price numeric(19, 1) NOT NULL  
);
```

```
create table person (  
  id BIGSERIAL NOT NULL PRIMARY KEY,  
  first_name VARCHAR(50) NOT NULL,  
  last_name VARCHAR(50) NOT NULL,  
  gender VARCHAR(1) NOT NULL,  
  email VARCHAR(100),  
  date_of_birth DATE NOT NULL,  
  country_of_birth VARCHAR(50),  
  car_id BIGINT REFERENCES car (id)  
  UNIQUE (car_id)  
);
```

```
insert into person (id, first_name, last_name, gender, email, date_of_birth, country_of_birth)  
values ('Feldklein', 'M', 'haleighfeldklein', 'M', 'haleighfeldklein@gmail.com', '1990-01-01', 'USA');  
insert into person (id, first_name, last_name, gender, email, date_of_birth, country_of_birth)  
values ('Gittoes', 'F', 'agittoes1@para', 'F', 'agittoes1@para.com', '1990-01-01', 'USA');  
insert into person (id, first_name, last_name, gender, email, date_of_birth, country_of_birth)  
values ('Heck', 'M', 'mortieHeck@gmail.com', 'M', 'mortieHeck@gmail.com', '1990-01-01', 'USA');  
  
insert into car (id, make, model, price)  
values (1, 'Ford', 'Mustang', 25000);  
insert into car (id, make, model, price)  
values (2, 'Ford', 'Mustang', 25000);
```

test=# \d car

Table "public.car"				
Column	Type	Collation	Nullable	Default
id	bigint		not null	nextval('car_id_seq'::regclass)
make	character varying(100)		not null	
model	character varying(100)		not null	
price	numeric(19,1)		not null	

Indexes:

"car_pkey" PRIMARY KEY, btree (id)

Referenced by:

TABLE "person" CONSTRAINT "person_car_id_fkey" FOREIGN KEY (car_id) REFERENCES car(id)

test=#

```
test=# \d car
```

Table "public.car"				
Column	Type	Collation	Nullable	Default
id	bigint		not null	nextval('car_id_seq'::regclass)
make	character varying(100)		not null	
model	character varying(100)		not null	
price	numeric(19,1)		not null	

Indexes:

"car_pkey" PRIMARY KEY, btree (id)

Referenced by:

TABLE "person" CONSTRAINT "person_car_id_fkey" FOREIGN KEY (car_id) REFERENCES car(id)

```
test=#
```

```
test=# SELECT * FROM car;
```

id	make	model	price
1	Dodge	Caravan	85535.0
2	Jaguar	S-Type	95099.1

(2 rows)

```
test=# SELECT * FROM car_id_seq;
```

last_value	log_cnt	is_called
1	0	f

(1 row)

nextval actually refers to a table which we can see contains the last value of car table.

```
test=# \d person
```

Table "public.person"				
Column	Type	Collation	Nullable	Default
id	bigint		not null	nextval('person_id_seq'::regclass)
first_name	character varying(50)		not null	
last_name	character varying(50)		not null	
gender	character varying(1)		not null	
email	character varying(100)			
date_of_birth	date		not null	
country_of_birth	character varying(50)		not null	
car_id	bigint			

Indexes:

- "person_pkey" PRIMARY KEY, btree (id)
- "person_car_id_key" UNIQUE CONSTRAINT, btree (car_id)

Foreign-key constraints:

- "person_car_id_fkey" FOREIGN KEY (car_id) REFERENCES car(id)

But this does not seem intuitive because surely the last value would be the highest id value in the table.

```
test=# SELECT * FROM person_id_seq;
last_value | log_cnt | is_called
-----+-----+-----
1          | 0       | f
(1 row)
```

```
test=# SELECT * FROM PERSON;
id | first_name | last_name | gender | email | date_of_birth | country_of_birth | car_id
---+-----+-----+-----+-----+-----+-----+-----
3 | Mortie | Heck | M | mortieHeck@gmail.com | 1996-02-10 | Thailand | 
1 | Haleigh | Feldklein | M | haleighfeldklein@hotmail.com | 1909-03-19 | Brazil | 2
2 | Ameline | Gittoes | F | agittoes1@parallels.com | 2017-04-19 | Philippines | 1
(3 rows)
```

```

test=# SELECT nextval('person_id_seq'::regclass);
nextval
-----
1
(1 row)

test=# SELECT nextval('person_id_seq'::regclass);
nextval
-----
2
(1 row)

test=# SELECT nextval('person_id_seq'::regclass);
nextval
-----
3
(1 row)

test=# SELECT nextval('person_id_seq'::regclass);
nextval
-----
4
(1 row)

test=# SELECT nextval('person_id_seq'::regclass);
nextval
-----
5
(1 row)

```

This is simply how to use sequences. So...
 a type of **Serial** => **int** and a
 type of **bigserial** => **bigint**.

But `nextval('person_id_seq'::regclass)` is simply a function that increments by one.

```

test=# SELECT * FROM person_id_seq;
last_value | log_cnt | is_called
-----+-----+-----
5          |      28 | t
(1 row)

```

At the moment I have three rows in person. If I now add a new row what will be the id?

```

test=# SELECT id, first_name FROM person;
id | first_name
---+-----
3  | Mortie
1  | Haleigh
2  | Ameline
(3 rows)

```

```

test=# insert into person (first_name, last_name,
gender, email, date_of_birth, country_of_birth) values
('john', 'Smith', 'M', null, '2016-01-12', 'Spain');
INSERT 0 1

```

```

test=# SELECT id, first_name FROM person;
id | first_name
---+-----
3  | Mortie
1  | Haleigh
2  | Ameline
6  | john
(4 rows)

```

The answer is six because the last_value is 5.

13.2 Reset a sequence

If I invoke the function a few times more the next_val is now 11

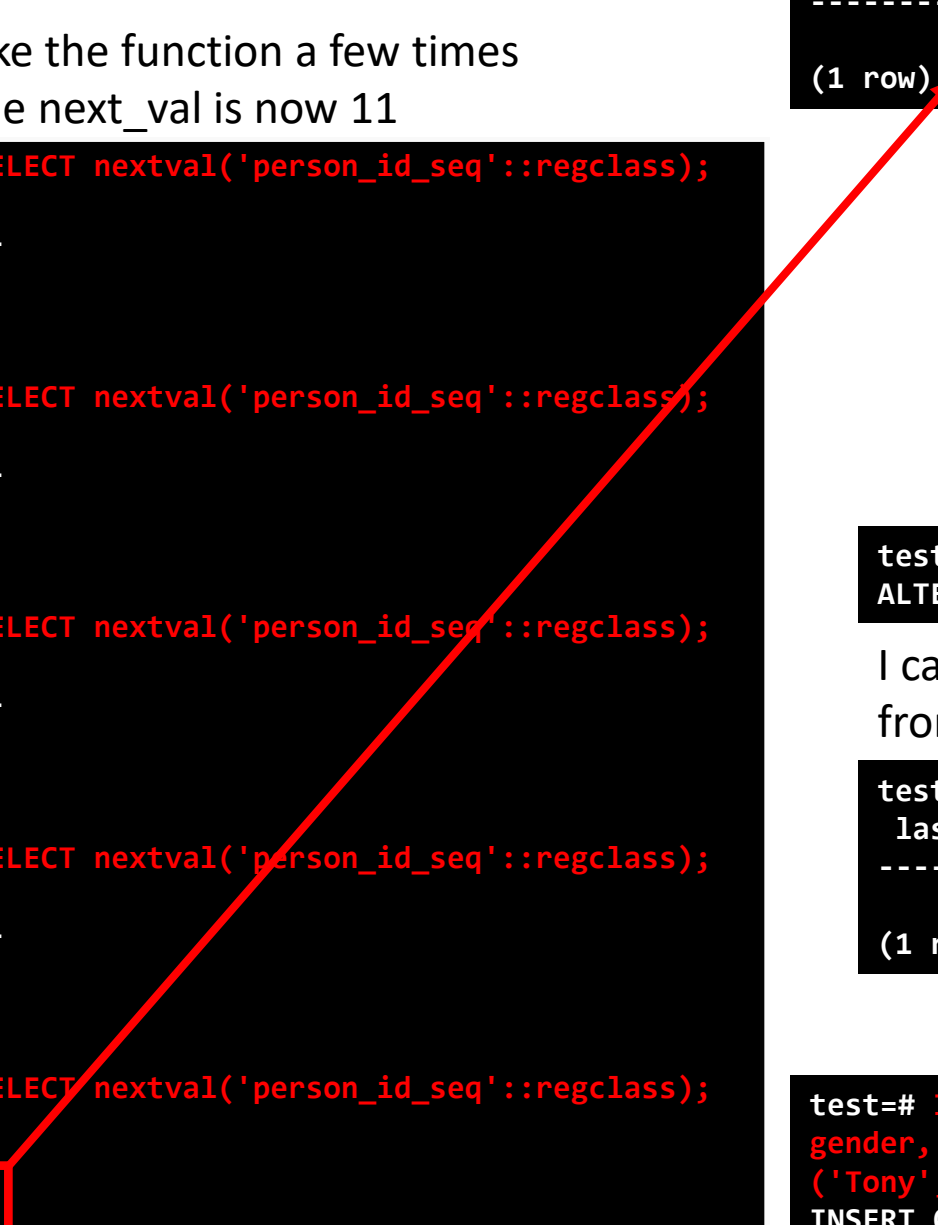
```
test=# SELECT nextval('person_id_seq'::regclass);
nextval
-----
       7
(1 row)

test=# SELECT nextval('person_id_seq'::regclass);
nextval
-----
       8
(1 row)

test=# SELECT nextval('person_id_seq'::regclass);
nextval
-----
       9
(1 row)

test=# SELECT nextval('person_id_seq'::regclass);
nextval
-----
      10
(1 row)

test=# SELECT nextval('person_id_seq'::regclass);
nextval
-----
      11
(1 row)
```



```
test=# SELECT * FROM person_id_seq;
last_value | log_cnt | is_called
-----+-----+-----
          11 |      28 | t
(1 row)
```

```
test=# SELECT id, first_name FROM person;
id | first_name
---+-----
 3 | Mortie
 1 | Haleigh
 2 | Ameline
 6 | John
(4 rows)
```

I want to restart the sequence from 7, not from 11.

```
test=# ALTER SEQUENCE person_id_seq RESTART WITH 6;
ALTER SEQUENCE
```

I can manually alter the sequence to restart from the next highest value in the table.

```
test=# SELECT * FROM person_id_seq;
last_value | log_cnt | is_called
-----+-----+-----
          6 |        0 | f
(1 row)
```

```
test=# INSERT INTO person (first_name, last_name,
gender, email, date_of_birth, country_of_birth) VALUES
('Tony', 'Smith', 'M', null, '2016-01-12', 'Spain');
INSERT 0 1
```

```
test=# SELECT id, first_name FROM person;
```

id	first_name
----	------------

3	Mortie
1	Haleigh
2	Ameline
6	john
7	Tony

(5 rows)

I now see that the next row is inserted with an id value of 7

14. Extensions

[14.1 About PostGreSQL extensions](#)

[14.2 Understanding UUID Data Type](#)

[14.3 Install UUID-OSSP extension](#)

[14.4 Genrate Globally unique ID using UUIDv4](#)

[14.5 Build SQL Tables Using UUID](#)

[14.6 Import from SQL file with UUID's](#)

[14.7 Update FK Columns with UUIDs](#)

[14.8 Join Using keyword](#)

14.1 About PostGreSQL extensions

PostGresSQL is designed to be easily extensible so extensions that add extra functionality to PostGreSQL can be loaded into the database and work like the pre-installed functions.

A list of compatible extension can be found in the documentation.

<https://www.postgresql.org/download/products/6-postgresql-extensions/>

Use **SELECT * FROM pg_available_extensions;** to see the complete list of available extensions.

```
test=# SELECT * FROM pg_available_extensions;
```

name	default_version	installed_version	comment
amcheck	1.4		functions for verifying relation integrity
autoinc	1.0		functions for autoincrementing fields
bloom	1.0		bloom access method - signature file based index
...			
uuid-oss	1.1		generate universally unique identifiers (UUIDs)
xml2	1.1		XPath querying and XSLT

(61 rows)

Note the installed version column that shows what version if any of each extension is installed.

Uuid-oss is a good extension for generating universally unique identifiers which makes it good for use with primary keys.

14.2 Understanding UUID Data Type

A **Universally Unique Identifier (UUID)** is a 128-bit label used to uniquely identify objects in computer systems. The term Globally Unique Identifier (GUID) is also used, mostly in Microsoft systems.

Format

A UUID is 128 bits in size, in which 2 to 4 bits are used to indicate the format's variant. The most common variant in use today, OSF DCE, additionally defines 4 bits for its version. The use of the remaining bits is governed by the variant/version selected.

- https://en.wikipedia.org/wiki/Universally_unique_identifier

The use of UUID pretty much guaranties collisions by using a number of variables (depending on version of UUID) such as mac-address, datetime, DCE security version and namespace hashing to algorithmically generate a unique ID.

Versions 1 and 6 (date-time and MAC address)

Version 2 (date-time and MAC address, DCE security version)

Versions 3 and 5 (namespace name-based)

Version 4 (random)

Version 7 (timestamp and random)

Version 8 (custom)

14.3 Install UUID-OSSP extension

```
test=# CREATE EXTENSION IF NOT EXISTS "uuid-oss";
CREATE EXTENSION
```

To install an extension we can use the CREATE EXTENSION command.

IF NOT EXISTS means that we can run the command as many times and only install the extension once. Finally the name of the extension needs to be in double quotes. Single quotes will not work.

```
test=# SELECT * FROM pg_available_extensions;
```

name	default_version	installed_version	comment
amcheck	1.4		functions for verifying relation integrity
autoinc	1.0		functions for autoincrementing fields
bloom	1.0		bloom access method - signature file based index
...			
uuid-oss	1.1	1.1	generate universally unique identifiers (UUIDs)
xml2	1.1		XPath querying and XSLT

(61 rows)

```
test=# \df
```

List of functions				
Schema	Name	Result data type	Argument data types	Type
public	uuid_generate_v1	uuid		func
public	uuid_generate_v1mc	uuid		func
public	uuid_generate_v3	uuid	namespace uuid, name text	func
public	uuid_generate_v4	uuid		func
public	uuid_generate_v5	uuid	namespace uuid, name text	func
public	uuid_nil	uuid		func
public	uuid_ns_dns	uuid		func
public	uuid_ns_oid	uuid		func
public	uuid_ns_url	uuid		func
public	uuid_ns_x500	uuid		func

(10 rows)

/df command will list all the available functions and we can see the UUID functions that are available to use including UUID version 4 which will generate a random UUID.

14.4 Generate Globally unique ID using UUIDv4

```
test=# SELECT uuid_generate_v4();
          uuid_generate_v4
-----
 11821dde-763a-445b-aea9-17cb516e0645
(1 row)

test=# SELECT uuid_generate_v4();
          uuid_generate_v4
-----
 4a07d204-533b-4863-9c2e-9b3541600dc7
(1 row)
```

To generate a globally unique random number I just invoke the `uuid_generate_v4` function.

Note that each time I invoke the function it generates a different but globally unique number.

I can run this a million times and the number generated will never be the same. This makes it a good feature for generating ID column values.

An advantage of using UUIDs is that it makes it very hard for attackers to mine our database. For example if you had an API for users an attacker could exploit this by updating or deleting off the ID. UUIDs make this very difficult because of the randomness of the UUID. Additionally when migrating data from one database to another using UUIDs can avoid possible conflicts with the primary key because the primary key is normally auto-incremented so they would clash between database A and database B.

PostgreSQL documentation defines the UUID datatype.

<https://www.postgresql.org/docs/current/datatype-uuid.html>

14.5 Build SQL Tables Using UUID

```
create table car (  
  car_uid UUID NOT NULL PRIMARY KEY, 1.  
  make VARCHAR(100) NOT NULL,  
  model VARCHAR(100) NOT NULL,  
  price numeric(19, 1) NOT NULL  
);
```

```
create table person (  
  person_uid UUID NOT NULL PRIMARY KEY, 1.  
  first_name VARCHAR(50) NOT NULL,  
  last_name VARCHAR(50) NOT NULL,  
  gender VARCHAR(1) NOT NULL,  
  email VARCHAR(100),  
  date_of_birth DATE NOT NULL,  
  country_of_birth VARCHAR(50) NOT NULL,  
  car_uid UUID REFERENCES car (car_uid), 2.  
  UNIQUE (car_uid),  
  UNIQUE (email)  
);
```

3a.

```
insert into person (person_uid, first_name, last_name, gender, email, date_of_birth, country_of_birth) values  
(uuid_generate_v4(), 'Haleigh', 'Feldklein', 'M', 'haleighfeldklein@hotmail.com', '1909-03-19', 'Brazil');
```

```
insert into person (person_uid, first_name, last_name, gender, email, date_of_birth, country_of_birth) values  
(uuid_generate_v4(), 'Ameline', 'Gittoes', 'F', 'agittoes1@parallels.com', '2017-04-19', 'Philippines');
```

3b.

```
insert into person (person_uid, first_name, last_name, gender, email, date_of_birth, country_of_birth) values  
(uuid_generate_v4(), 'Mortie', 'Heck', 'M', 'mortieHeck@gmail.com', '1996-02-10', 'Thailand');
```

3a.

3b.

```
insert into car (car_uid, make, model, price) values (uuid_generate_v4(), 'Dodge', 'Caravan', '85534.98');  
insert into car (car_uid, make, model, price) values (uuid_generate_v4(), 'Jaguar', 'S-Type', '95099.14');
```

1. Create a SQL script defining the ID datatype as UUID.

2. Create any foreign keys that reference a datatype of UUID

3a. For an insert query specify the UUID column so not using auto-increment.

3b. Define the UUID value by invoking the uuid_auto_generate function.

14.6 Import from SQL file with UUID's

Remove old tables

```
test=# DROP TABLE person;  
DROP TABLE  
test=# DROP TABLE car;  
DROP TABLE
```

```
test=# \i C:/Users/ellio/Documents/CODING-LESSONS/14-PostGreSQL/person-car2.sql;  
CREATE TABLE  
CREATE TABLE  
INSERT 0 1  
INSERT 0 1  
INSERT 0 1  
INSERT 0 1  
INSERT 0 1
```

Import the SQL file with the UUIDs

```
test=# SELECT * FROM person;  
      person_uid      | first_name | last_name | gender |      email      | date_of_birth | country_of_birth | car_uid  
-----+-----+-----+-----+-----+-----+-----+-----  
dd7513a4-3a33-4695-a1a0-a97cafcef031 | Haleigh   | Feldklein | M      | haleighfeldklein@hotmail.com | 1909-03-19    | Brazil           |  
f6870c25-f03c-4c5f-bc68-5870a918411f | Ameline   | Gittoes   | F      | agittoes1@parallels.com      | 2017-04-19    | Philippines      |  
81343d51-c1d9-4d2f-8cac-12898b9bd839 | Mortie    | Heck      | M      | mortieHeck@gmail.com         | 1996-02-10    | Thailand          |  
(3 rows)
```

```
test=# SELECT * from CAR;  
      car_uid      | make  | model  | price  
-----+-----+-----+-----  
9c5d7d90-282d-46b1-9252-d87ab614988b | Dodge | Caravan | 85535.0  
c6b31d99-4c81-4829-b572-8bd403cb128c | Jaguar | S-Type  | 95099.1
```

Now in both tables the uid columns are globally unique randomly generated UUID numbers.

Note that I can use `\x` to view the tables in expanded screen mode if the content is too wide for the display.

14.7 Update FK Columns with UUIDs

```
test=# UPDATE person SET car_uid = '9c5d7d90-282d-46b1-9252-d87ab614988b' WHERE person_uid = 'dd7513a4-3a33-4695-a1a0-a97cafcef031';
UPDATE 1
test=# UPDATE person SET car_uid = 'c6b31d99-4c81-4829-b572-8bd403cb128c' WHERE person_uid = 'f6870c25-f03c-4c5f-bc68-5870a918411f';
UPDATE 1
```

I can use an update query exactly as before to SET the car_uid for each person. Note that because UUID is a hexz number and not a numeric data type it has to be in quotes.

I can also run join queries with UUID.

```
test=# \x
Expanded display is on.

test=# SELECT * FROM person JOIN car on person.car_uid = car.car_uid;
-[ RECORD 1 ]-----+-----
person_uid          | dd7513a4-3a33-4695-a1a0-a97cafcef031
first_name          | Haleigh
last_name           | Feldklein
gender              | M
email               | haleighfeldklein@hotmail.com
date_of_birth       | 1909-03-19
country_of_birth    | Brazil
car_uid             | 9c5d7d90-282d-46b1-9252-d87ab614988b
car_uid             | 9c5d7d90-282d-46b1-9252-d87ab614988b
make                | Dodge
model               | Caravan
price               | 85535.0
-[ RECORD 2 ]-----+-----
person_uid          | f6870c25-f03c-4c5f-bc68-5870a918411f
first_name          | Ameline
last_name           | Gittoes
gender              | F
email               | agittoes1@parallels.com
date_of_birth       | 2017-04-19
country_of_birth    | Philippines
car_uid             | c6b31d99-4c81-4829-b572-8bd403cb128c
car_uid             | c6b31d99-4c81-4829-b572-8bd403cb128c
make                | Jaguar
model               | S-Type
price               | 95099.1
```

14.8 Join Using keyword

```
test=# SELECT * FROM person JOIN car on person.car_uid = car.car_uid;
```

Because both the column names are the same this query can be shortened.

Note how the car table can be joined to person **using car_uid** and that the **column name must be in brackets**.

```
test=# SELECT * FROM person JOIN car USING (car_uid);
-[ RECORD 1 ]-----+-----
car_uid       | 9c5d7d90-282d-46b1-9252-d87ab614988b
person_uid    | dd7513a4-3a33-4695-a1a0-a97cafcef031
first_name    | Haleigh
last_name     | Feldklein
gender        | M
email         | haleighfeldklein@hotmail.com
date_of_birth | 1909-03-19
country_of_birth | Brazil
make          | Dodge
model         | Caravan
price         | 85535.0
-[ RECORD 2 ]-----+-----
car_uid       | c6b31d99-4c81-4829-b572-8bd403cb128c
person_uid    | f6870c25-f03c-4c5f-bc68-5870a918411f
first_name    | Ameline
last_name     | Gittoes
gender        | F
email         | agittoes1@parallels.com
date_of_birth | 2017-04-19
country_of_birth | Philippines
make          | Jaguar
model         | S-Type
price         | 95099.1
```

The using keyword with Join also works for a left join.

```
test=# SELECT * FROM person LEFT JOIN car USING (car_uid);
-[ RECORD 1 ]-----+-----
car_uid      | 9c5d7d90-282d-46b1-9252-d87ab614988b
person_uid   | dd7513a4-3a33-4695-a1a0-a97cafcef031
first_name   | Haleigh
last_name    | Feldklein
gender       | M
email        | haleighfeldklein@hotmail.com
date_of_birth | 1909-03-19
country_of_birth | Brazil
make         | Dodge
model        | Caravan
price        | 85535.0
-[ RECORD 2 ]-----+-----
car_uid      | c6b31d99-4c81-4829-b572-8bd403cb128c
person_uid   | f6870c25-f03c-4c5f-bc68-5870a918411f
first_name   | Ameline
last_name    | Gittoes
gender       | F
email        | agittoes1@parallels.com
date_of_birth | 2017-04-19
country_of_birth | Philippines
make         | Jaguar
model        | S-Type
price        | 95099.1
-[ RECORD 3 ]-----+-----
car_uid      | 
person_uid   | 81343d51-c1d9-4d2f-8cac-12898b9bd839
first_name   | Mortie
last_name    | Heck
gender       | M
email        | mortieHeck@gmail.com
date_of_birth | 1996-02-10
country_of_birth | Thailand
make         | 
model        | 
price        | 
```

Course Content

Download PostGreSQL | Install PostGreSQL | How to connect to a postGreSQL database | Database GUI clients | connect to database using SQL shell command line | connect to database using PgAdmin 4 GUI client | Terminal Help | PSQL mode | list databaes command | create database command | connect to remote database | Change Database command | Drop database | drop database with force | create table without constraints | create table with constraints | primary key | auto increment | drop table | describe table | describe database | insert into command | generate test sql data using mockaroo | \i command (import from file) | SELECT * FROM table | order by | select distinct | order by desc and asc | distinct | where | and | or | comparison operators >, >=, <, <= | limit | offset | fetch | in keyword | betweed keyword | like operator with wildcard & undersocre matching | ilike matching | group by | count | having keyword | min, max, avg and sum functions | mathmatical operators (addition, subtraction, division, multplication) | arithmetic Operators (power of, factorial, modulus) | alias as keyword | Handling null values with coalesce, nullif | datetime | date | now function | casting now function to date or time | using interval to perform calculations on datetime | extract part of datetime | age function | Primary key | drop pkey | add pkey | unique constraints | check constraints | delete from table | delete by primary key | delete multiple rows by non primary key | delete with where clause | on conflict do nothing | upsert | joins and relationship theory | adding FK column | updating FK column | inner joins theory | inner joins query | left join theory | left join query examples | Deletion where a Join FK exists | export a select query to csv | explanation of sequences | reset a sequence | about PostGreSQL extensions | understanding UUID Data Type | install UUID-OSSP extension | genrate globally unique ID using UUIDv4 | build SQL tables & insert queries Using UUID | import from SQL file with UUID's | update FK Columns with UUIDs | join using keyword