Install local postgresql then add the PostgreSQL bin directory path to the PATH environmental variable.

Then:

>psql -U postgres

Password is: password

**create database**

```

CREATE DATABASE employees;

```

**list databases**

```

\l

```

**connect to a database**

```

\c <database name>

```

**switch back to postgres database**

```

\c postgres

```

**see current user**

```

SELECT current\_user;

```

**see current database**

```

SELECT current\_database();

```

**drop (remove, delete) database**

```

DROP DATABASE <database name>;

```

**create table**

```

CREATE TABLE employees (

ID INT PRIMARY KEY NOT NULL,

NAME TEXT NOT NULL,

RANK INT NOT NULL,

ADDRESS CHAR(50),

SALARY REAL DEFAULT 25500.00,

BDAY DATE DEFAULT '1900-01-01'

);

```

**show tables in a database** (list down)

```

\d

```

**show details of a table**

```

\d <table name>

```

**drop a table**

```

DROP TABLE <table name>;

```

**schema**

Schemas allow us to organize our database and database code.

A schema is like a folder.

Into this folder, you can put tables, views, indexes, sequences, data types, operators, and functions.

Unlike folders, however, schemas can't be nested.

Schemas provide namespacing.

[Read more about schemas](<https://www.tutorialspoint.com/postgresql/postgresql_schema.htm>)

**insert a record**

```

INSERT INTO employees (ID,NAME,RANK,ADDRESS,SALARY,BDAY) VALUES (1, 'Mark', 7, '1212 E. Lane, Someville, AK, 57483', 43000.00 ,'1992-01-13');

```

**list records in a table**

```

SELECT \* FROM <table name>;

```

**insert a record - variations**

omitted values will have the [default value](https://www.postgresql.org/docs/9.3/static/ddl-default.html):

```

INSERT INTO employees (ID,NAME,RANK,ADDRESS,BDAY) VALUES (2, 'Marian', 8, '7214 Wonderlust Ave, Lost Lake, KS, 22897', '1989-11-21');

```

we can use DEFAULT rather leaving a field blank or specifying a value:

```

INSERT INTO employees (ID,NAME,RANK,ADDRESS,SALARY,BDAY) VALUES (3, 'Maxwell', 6, '7215 Jasmine Place, Corinda, CA 98743', 87500.00, DEFAULT);

```

we can insert multiple rows:

```

INSERT INTO employees (ID,NAME,RANK,ADDRESS,SALARY,BDAY) VALUES (4, 'Jasmine', 5, '983 Star Ave., Brooklyn, NY, 00912 ', 55700.00, '1997-12-13' ), (5, 'Orranda', 9, '745 Hammer Lane, Hammerfield, Texas, 75839', 65350.00 , '1992-12-13');

```

**auto increment key field**

Instead of creating a unique ID number ourselves, we can have postgres automatically increment this ID field.

To do this we use the data types **smallserial**, **serial** or **bigserial** (not true types but for convenience).

This is like AUTO\_INCREMENT in other databases.

```

CREATE TABLE phonenumbers(

ID **SERIAL** PRIMARY KEY,

PHONE TEXT NOT NULL

);

```

```

INSERT INTO phonenumbers (PHONE) VALUES ( '234-432-5234'), ('543-534-6543'), ('312-123-5432');

```

```

\d phonenumbers

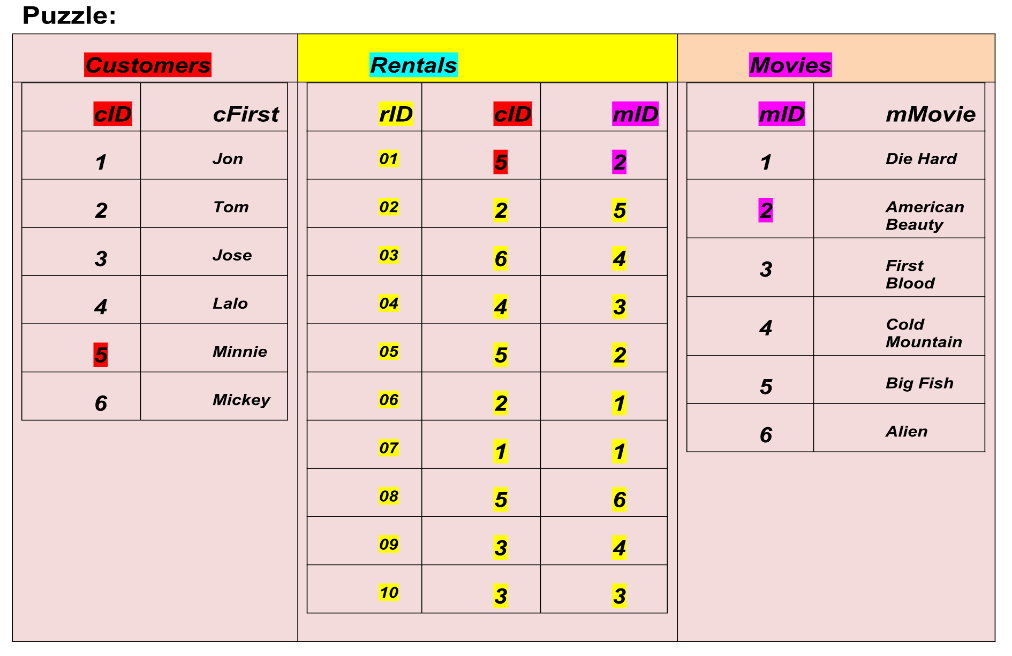
```

```

SELECT \* FROM phonenumbers;

```

**relational databases**



```

CREATE TABLE phonenumbers (

ID SERIAL PRIMARY KEY NOT NULL,

PHONE CHAR(50) NOT NULL,

EMP\_ID INT references employees(ID)

);

```

```

INSERT INTO phonenumbers (PHONE,EMP\_ID) VALUES ('555-777-8888', 4), ('555-222-3345', 4), ('777-543-3451', 1), ('544-756-2334', 2);

```

**queries**

**querying from more than one table**

This query is an \*\*inner join\*\*.

```

SELECT employees.NAME, phonenumbers.PHONE FROM employees INNER JOIN phonenumbers ON employees.ID = phonenumbers.EMP\_ID;

```

**join queries**

Join queries allow us to select records from two or more tables.

A join query combines columns from one or more tables - it joins a bunch of columns from different tables together.

There are five types of joins in postgres:

**cross join**

A cross join returns the \*\*Cartesian product\*\* of rows from tables in the join. In other words, it will produce rows which combine each row from the first table with each row from the second table.

```

CREATE TABLE person (

ID SERIAL PRIMARY KEY NOT NULL,

NAME CHAR(50) NOT NULL

);

```

```

INSERT INTO person (NAME) VALUES ('Shen'), ('Daniel'), ('Juan'), ('Arin'), ('McLeod');

```

```

CREATE TABLE sport (

ID SERIAL PRIMARY KEY NOT NULL,

NAME CHAR(50) NOT NULL,

P\_ID INT references person(ID)

);

```

```

INSERT INTO sport (NAME, P\_ID) VALUES ('Surf',1),('Soccer',3),('Ski',3),('Sail',3),('Bike',3);

```

```

SELECT person.NAME, sport.NAME FROM person CROSS JOIN sport;

```

**create a database**

```

create database blockbuster;

```

switch into the database

```

\c blockbuster

```

create three tables

```

create table customers (cid serial primary key not null, cfirst char(50) not null);

```

```

create table movies (mid serial primary key not null, mmovie char(50) not null);

```

```

create table rentals (rid serial primary key not null, cid int references customers(cid), mid int references movies(mid));

```

populate tables

```

insert into customers (cfirst) values ('James Bond'), ('Miss Moneypenny'), ('Q'), ('M'), ('Fleming');

```

```

insert into movies (mmovie) values ('Jaws'), ('Alien'), ('Never Say Never'), ('Skyfall'), ('Highlander');

```

```

insert into rentals (cid, mid) values (1,3), (2,5), (4,1), (3,2), (5,4), (3,2), (1,3), (2,4), (5,4), (2,1), (2,3), (4,5), (5,2), (2,1), (3,2), (3,3), (2,3), (1,4), (3,2), (2,3), (3,3), (2,4), (2,3), (1,2), (3,5), (3,4), (1,5);

```

inner join query

```

select customers.cfirst, movies.mmovie from customers inner join rentals on customers.cid = rentals.cid inner join movies on rentals.mid = movies.mid;

```

How this works

```

select \* from

tableA inner join tableB

on tableA.common = tableB.common

inner join TableC

on tableB.common = TableC.common

```

you might also see alias use

```

select \* from

tableA a inner join tableB b

on a.common = b.common

inner join TableC c

on b.common = c.common

```

**general syntax**

select

```

SELECT <fields> FROM <table>;

```

```

SELECT \* FROM employees;

```

```

SELECT name, score FROM employees;

```

**cross join**

```

SELECT <fields> FROM <table1> CROSS JOIN <table2>;

```

```

SELECT person.NAME, sport.NAME FROM person CROSS JOIN sport;

```

**inner join**

```

SELECT <fields> FROM <table> INNER JOIN <table>

ON <pkey> = <fkey>;

```

```

SELECT person.NAME, sport.NAME FROM person INNER JOIN sport

ON person.ID = sport.P\_ID;

```

**inner join**

An inner join allows us to select records from two tables.

We used an inner join above when we asked for the phone numbers associated with an employee:

```

SELECT employees.NAME, phonenumbers.PHONE FROM employees INNER JOIN phonenumbers ON employees.ID = phonenumbers.EMP\_ID;

```

We can use one with our ```people``` and ```sports``` table too, if we wanted, as these tables are connected (remember ```P\_ID INT references person(ID)```).

```

SELECT person.NAME, sport.NAME FROM person INNER JOIN sport ON person.ID = sport.P\_ID;

```

Here is how wikipedia explains an inner join:

An inner join requires each row in the two joined tables to have matching rows, and is a commonly used join operation in applications but should not be assumed to be the best choice in all situations.

Inner join creates a new result table by combining column values of two tables (A and B) based upon the join-predicate.

The query compares each row of A with each row of B to find all pairs of rows which satisfy the join-predicate.

When the join-predicate is satisfied by matching non-NULL values, column values for each matched pair of rows of A and B are combined into a result row.

The result of the join can be defined as the outcome of first taking the Cartesian product (or Cross join) of all rows in the tables (combining every row in table A with every row in table B) and then returning all rows which satisfy the join predicate. Actual SQL implementations normally use other approaches, such as hash joins or sort-merge joins, since computing the Cartesian product is slower and would often require a prohibitively large amount of memory to store.

SQL specifies two different syntactical ways to express joins: the "explicit join notation" and the "implicit join notation". The "implicit join notation" is no longer considered a best practice, although database systems still support it. The \*\*"explicit join notation"\*\* uses the JOIN keyword, optionally preceded by the INNER keyword, to specify the table to join, and the ON keyword to specify the predicates for the join.

**outer join**

**left outer join**

A left outer join gives you everything in one table, and also the matching records in another table.

For tables A and B a left outer join would give you all rows of the "left" table (A), even if the join-condition does not find any matching row in the "right" table (B).

This means that if the ON clause matches 0 (zero) rows in B (for a given row in A), the join will still return a row in the result (for that row)—but with NULL in each column from B.

```

SELECT person.NAME, sport.NAME FROM person LEFT OUTER JOIN sport ON person.ID = sport.P\_ID;

```

**right outer join**

A right outer join is like a left outer join, but for the table on the right.

```

INSERT INTO sport (NAME) VALUES ('Squirrel Suit Flying');

```

```

SELECT person.NAME, sport.NAME FROM person RIGHT OUTER JOIN sport ON person.ID = sport.P\_ID;

```

**full outer join**

A full outer join is like running both a left outer join and a right outer join at the same time. It gives you everything from all tables, and matches what matches.

```

SELECT person.NAME, sport.NAME FROM person FULL OUTER JOIN sport ON person.ID = sport.P\_ID;

```

**clauses**

**where**

Adding \*\*WHERE\*\* to a SQL query allows you to filter results.

```

SELECT \* FROM employees WHERE salary > 60000;

```

**and**

```

SELECT \* FROM employees WHERE salary > 60000 AND score = 26;

```

**in**

```

SELECT \* FROM employees WHERE score IN (25,26);

```

**not**

```

SELECT \* FROM employees WHERE score NOT IN (25,26);

```

**between**

```

SELECT \* FROM employees WHERE score BETWEEN 23 AND 26;

```

**is not null**

```

SELECT \* FROM employees WHERE score IS NOT NULL;

```

**like**

```

SELECT \* FROM employees WHERE name LIKE '%an%';

```

**0r**

```

SELECT \* FROM employees WHERE score <= 24 OR salary < 50000;

```

**limit**

Limit the number of records returned

```

SELECT \* FROM employees LIMIT 4;

```

```

SELECT \* FROM employees ORDER BY id LIMIT 4;

```

**update**

syntax

```

UPDATE table

SET col1 = val1, col2 = val2, ..., colN = valN

WHERE <condition>;

```

```

SELECT \* FROM employees;

```

```

UPDATE employees SET score = 99 WHERE ID = 3;

```

**order by**

```

SELECT \* FROM employees ORDER BY id;

```

**delete**

syntax

```

DELETE FROM table

WHERE <condition>;

```

```

SELECT \* FROM sport;

```

```

DELETE FROM sport WHERE id = 6;

```

\*\*WARNING: this deletes all records:\*\*

```

DELETE FROM sport;

```

**users & privileges**

see current user

```

SELECT current\_user;

```

**details of users**

```

\du

```

**create user**

```

CREATE USER james WITH PASSWORD 'password';

```

**grant privileges**

privileges: SELECT, INSERT, UPDATE, DELETE, RULE, ALL

```

GRANT ALL PRIVILEGES ON DATABASE company to james;

```

**revoke privileges**

```

REVOKE ALL PRIVILEGES ON DATABASE company from james;

```

**alter**

```

ALTER USER james WITH SUPERUSER;

```

```

ALTER USER james WITH NOSUPERUSER;

```

**remove**

```

DROP USER james;

```

**Great reference**

[Alex Edwards Code Organization Article](<http://www.alexedwards.net/blog/organising-database-access>)

**Code organization**

How you organize your code depends upon the project and your abilities.

Do not sacrifice simplicity & readability for brevity & cleverness.

Your goal should be to write code which is maintainable. This means that an intermediate developer should be able to sit down, read your code, understand it, and work with it.

**Different Approaches**

**One package**

When needed we can put variables in the package scope

**Two packages**

We will import the code from the second package when needed

**Three+ packages with package for config variables**

We will import the code from the config package when needed. The config package will hold variables like DB and TPL

FYI: there is no global scope in Go

