# PosgreSQL database

## Installation

To install PostgreSQL, first refresh your server’s local package index:

sudo apt update

Then, install the Postgres package along with a -contrib package that adds some additional utilities and functionality:

sudo apt install postgresql postgresql-contrib

#### Change root password

Ubuntu current user can connect to the database without a password if PostgreSql was installed correctly. Just type the command **psql** in the terminal. If you can't log in, then follow this procedure on terminal:

1. Type command **whoami**. It will report your login name. Let's say that your login is "**JohnTrevolta**".
2. Run the psql command line interface in full admin mode.

sudo -u postgres psql

1. Create database and postgreSQL user for the JohnTrevolta. You will need to press **Enter** after every command:

CREATE DATABASE JohnTrevolta;

CREATE USER JohnTrevolta;

GRANT ALL PRIVILEGES ON DATABASE JohnTrevolta TO JohnTrevolta;

1. Verify and correct permissions of the current user entering command **\du**. JohnTrevolta would gave the same permissions as **postgres** user (**Superuser**, **Create role**, **Replication**, **Bypass RLS**).

ALTER USER JohnTrevolta [ WITH ] CREATEROLE CREATEDB;

It is possible to create user with enhanced permissionsbut standard security rules do not recommend it:

ALTER USER JohnTrevolta [ WITH ] SUPERUSER CREATEROLE

CREATEDB REPLICATION;

There are two concepts in standard SQL: **role** and **user**. **User** is **role** with **LOGIN** attribute in the PostgreSQL database. **CREATE USER** command creates a role with the **LOGIN** attribute. You need to add the **LOGIN** attribute yourself in the **CREATE ROLE** command. Roles without the **LOGIN** attribute are used for creating groups. Group attributes may be inherited by users belonging to that group.

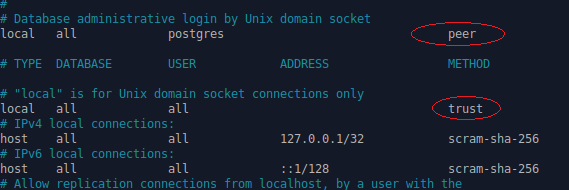
1. Set password if you are going to use the same login from remote computer:

ALTER USER JohnTrevolta WITH ENCRYPTED PASSWORD 'YourPassword';

1. Close psql entering **\q**.
2. Open file /*etc*/postgresql/14/main/pg\_hba.conf in nano or any other text editor:

sudo nano /etc/postgresql/14/main/pg\_hba.conf

1. Verify settings fo **local all**. They must have **peer** and **trust** values (look at picture below):



1. Correct values if they are different, save the file and restart postgreSQL server:

sudo systemctl restart postgresql

1. Now you will be able to connect to postgreSQL by typing the **psql** command on local computer. You would enter command

psql -h host\_name -d database\_name -U JohnTrevolta -W

for connecting from remote computer.

PostgreSql has many GUI tools for a management of the database. You can install [pgAdmin](https://www.pgadmin.org/) if you have a powerful enough computer. Here I will describe **psql**, which works in the terminal and does not require a lot of resources from the computer.

## SQL Dump

The idea behind this dump method is to generate a file with SQL commands that, when fed back to the server, will recreate the database in the same state as it was at the time of the dump. PostgreSQL provides the utility program pg\_dump for this purpose. The basic usage of this command is:

pg\_dump dbname > dumpfile

This command writes its result to the standard output. While the above command creates a text file, **pg\_dump** can create files in other formats that allow for parallelism and more fine-grained control of object restoration.

**pg\_dump** is a regular PostgreSQL client application (albeit a particularly clever one). This means that you can perform this backup procedure from any remote host that has access to the database. But remember that pg\_dump does not operate with special permissions. In particular, it must have read access to all tables that you want to back up, so in order to back up the entire database you almost always have to run it as a database superuser. (If you do not have sufficient privileges to back up the entire database, you can still back up portions of the database to which you do have access using options such as -n schema or -t table.)

To specify which database server pg\_dump should contact, use the command line options **-h host** and **-p port**. The default host is the local host or whatever your **PGHOST** environment variable specifies. Similarly, the default port is indicated by the **PGPORT** environment variable or, failing that, by the compiled-in default. (Conveniently, the server will normally have the same compiled-in default.)

### Restoring the Dump

Text files created by pg\_dump are intended to be read in by the psql program. The general command form to restore a dump is

psql dbname < dumpfile

where dumpfile is the file output by the pg\_dump command. The database dbname will not be created by this command, so you must create it yourself from template0 before executing psql (e.g., with **createdb -T template0 dbname**). **psql** supports options similar to **pg\_dump** for specifying the database server to connect to and the user name to use. See the psql reference page for more information. Non-text file dumps are restored using the **pg\_restore** utility.

Before restoring an SQL dump, all the users who own objects or were granted permissions on objects in the dumped database must already exist. If they do not, the restore will fail to recreate the objects with the original ownership and/or permissions. (Sometimes this is what you want, but usually it is not.)

Like any other PostgreSQL client application, pg\_dump will by default connect with the database user name that is equal to the current operating system user name. To override this, either specify the -U option or set the environment variable PGUSER.

Dumps created by pg\_dump are internally consistent, meaning, the dump represents a snapshot of the database at the time pg\_dump began running.

pg\_dump does not block other operations on the database while it is working. (Exceptions are those operations that need to operate with an exclusive lock, such as most forms of ALTER TABLE.)

The ability of pg\_dump and psql to write to or read from pipes makes it possible to dump a database directly from one server to another, for example:

pg\_dump -h host1 dbname | psql -h host2 dbname

## PSQL

**psql** is a terminal-based front-end to PostgreSQL. It enables you to type in queries interactively, issue them to PostgreSQL, and see the query results. Alternatively, input can be from a file or from command line arguments. In addition, psql provides a number of meta-commands and various shell-like features to facilitate writing scripts and automating a wide variety of tasks.

**PostgreSQL installer** for Windows installs this utility but not registers it in the PATH environment variable. You need to open Environment Variables dialog and register this path

C:\Program Files\PostgreSQL\16\bin

in User or System variables. Restart Command prompt after registering and vereify registration:

psql –version

### Get help on psql commands

1. To know all available psql commands, you use the **\?** command.
2. \?

### Connect to PostgreSQL database

psql

This command will connect to the same computer and install the database under the name you connected to the computer. Read paragraph **Change root password** if login fails.

The following command connects to a database under a specific user. After pressing Enter PostgreSQL will ask for the password of the user.

psql -d database -U user -W

If you want to connect to a database that resides on another host, you add the -h option as follows:

psql -h host -d database -U user -W

This command will only work if psql is installed on your computer. If you don't have **psql**, use **ssh** and connect your terminal to the remote computer where the **PostgreSql** database is installed.

ssh remote\_username@remote\_server\_ip

ssh remote\_username@remote\_server\_name

In case you want to use SSL mode for the connection, just specify it as shown in the following command:

psql -U user -h host "dbname=db sslmode=require"

This command also requires psql installed on your computer.

### List available databases

To list all databases in the current PostgreSQL database server, you use **\l** command:

\l

### Switch connection to a new database

Once you are connected to a database, you can switch the connection to a new database under a user-specified by user. The previous connection will be closed. If you omit the user parameter, the current user is assumed.

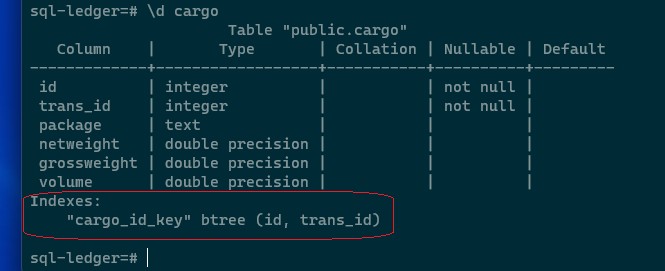
\c dbname username

### Describe a table

To describe a table such as a column, type, or modifiers of columns, you use the following command:

\d table\_name

This command displays information about [indexes](https://www.postgresql.org/docs/current/indexes.html) in addition to columns:



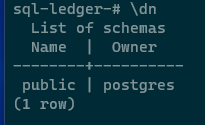
### List commands

#### Available schemas

A database contains one or more named **schemas**, which in turn contain tables. Schemas also contain other kinds of named objects, including data types, functions, and operators. The same object name can be used in different schemas without conflict; for example, both schema1 and myschema can contain tables named mytable. Unlike databases, schemas are not rigidly separated: a user can access objects in any of the schemas in the database they are connected to, if they have privileges to do so. **Schemas** are analogous to directories at the operating system level, except that schemas cannot be nested.

**psql** can only to display a list of schemas. **Use** SQL for all other operations with schemas. To list all schemas of the currently connected database, you use the **\dn** command.

\dn



#### Available tables

1. To list all tables in the current database, you use the \dt command:
2. \dt [table\_name]
3. \dt+[table\_name]
4. The second command displays extended table. Both commands will show only one table if you write the parameter. The parameter wild cards:
5. \dt ac\*
6. will list all tables with names starting with ac.

#### Available functions

To list available functions in the current database, you use the **\df** command.

\df

#### Available views

To list available views in the current database, you use the **\dv** command.

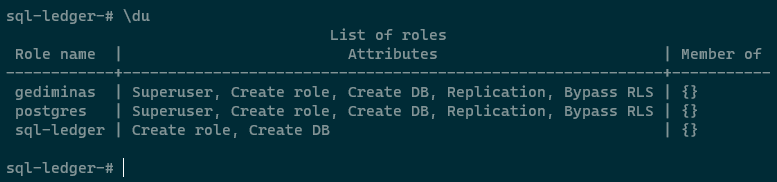
\dv

#### Users and their roles

To list all users and their assigned roles, you use **\du** command:

\du

\du+



### History

#### Execute the previous command

The **\q** command allows you to repeat a previously executed SQL statement.

\q

If you want to save the command history to a file, you need to specify the file name followed the \s command as follows:

#### Command history

To display command history, you use the \s command.

\s

If you want to save the command history to a file, you need to specify the file name followed the \s command as follows:

\s filename

Working with SSH it is important to remember that the file will be saved on the remote computer (the one you connected to with SSH). Working in a Windows environment, you can connect to the remote host with WinScp and copy the file to your computer. Use scp command working from Linux environment (see SSH topics in 01\_OperatingSystem.docx).

### Indexes

Suppose we have a table similar to this:

CREATE TABLE test1 (

id integer,

content varchar

);

and the application issues many queries of the form:

SELECT content FROM test1 WHERE id = constant;

With no advance preparation, the system would have to scan the entire test1 table, row by row, to find all matching entries. If there are many rows in test1 and only a few rows (perhaps zero or one) that would be returned by such a query, this is clearly an inefficient method. But if the system has been instructed to maintain an **index** on the id column, it can use a more efficient method for locating matching rows.

The following command can be used to create an **index** on the id column, as discussed:

CREATE INDEX test1\_id\_index ON test1 (id);

The name test1\_id\_index can be chosen freely, but you should pick something that enables you to remember later what the index was for. Once an index is created, no further intervention is required: the system will update the index when the table is modified, and it will use the index in queries when it thinks doing so would be more efficient than a sequential table scan.

PostgreSQL provides several index types: **B-tree**, **Hash**, **GiST**, **SP-GiST**, **GIN**, **BRIN**, and the extension bloom. Each index type uses a different algorithm that is best suited to different types of queries. By default, the **CREATE INDEX** command creates **B-tree** indexes, which fit the most common situations. The other index types are selected by writing the keyword USING followed by the index type name. For example, to create a Hash index:

CREATE INDEX name ON table USING HASH (column);

#### B-Tree

[B-trees](https://www.postgresql.org/docs/current/indexes-types.html" \l "INDEXES-TYPES-BTREE) can handle equality and range queries on data that can be sorted into some ordering. In particular, the PostgreSQL query planner will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators:

< <= = >= >

Constructs equivalent to combinations of these operators, such as BETWEEN and IN, can also be implemented with a B-tree index search. Also, an IS NULL or IS NOT NULL condition on an index column can be used with a B-tree index.

#### Hash

[Hash](https://www.postgresql.org/docs/current/indexes-types.html" \l "INDEXES-TYPES-HASH) indexes store a 32-bit hash code derived from the value of the indexed column. Hence, such indexes can only handle simple equality comparisons. The query planner will consider using a hash index whenever an indexed column is involved in a comparison using the equal operator:

=

#### GiST

[GiST](https://www.postgresql.org/docs/current/indexes-types.html" \l "INDEXES-TYPE-GIST) indexes are not a single kind of index, but rather an infrastructure within which many different indexing strategies can be implemented. Accordingly, the particular operators with which a GiST index can be used vary depending on the indexing strategy (the operator class). As an example, the standard distribution of PostgreSQL includes GiST operator classes for several two-dimensional geometric data types, which support indexed queries using these operators:

<< &< &> >> <<| &<| |&> |>> @> <@ ~= &&

PostgreSQL supports [geometric data types](https://www.postgresql.org/docs/6.3/c0806.htm) and wide list of [geometric operators](https://www.postgresql.org/docs/current/functions-geometry.html) on these types. GiST indexes are also capable of optimizing “nearest-neighbor” searches, such as

SELECT \* FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;

which finds the ten places closest to a given target point.

#### SP-GiST

[SP-GiST](https://www.postgresql.org/docs/current/spgist.html) indexes, like GiST indexes, offer an infrastructure that supports various kinds of searches. SP-GiST permits implementation of a wide range of different non-balanced disk-based data structures, such as quadtrees, k-d trees, and radix trees (tries). As an example, the standard distribution of PostgreSQL includes SP-GiST operator classes for two-dimensional points, which support indexed queries using these operators:

<< >> ~= <@ <<| |>>

The SP-GiST operator classes included in the standard distribution are documented in this [table](https://www.postgresql.org/docs/current/spgist-builtin-opclasses.html" \l "SPGIST-BUILTIN-OPCLASSES-TABLE).

#### GIN

GIN indexes are “inverted indexes” which are appropriate for data values that contain multiple component values, such as arrays. An inverted index contains a separate entry for each component value, and can efficiently handle queries that test for the presence of specific component values.

Like GiST and SP-GiST, GIN can support many different user-defined indexing strategies, and the particular operators with which a GIN index can be used vary depending on the indexing strategy. As an example, the standard distribution of PostgreSQL includes a GIN operator class for arrays, which supports indexed queries using these operators:

<@ @> = &&

The GIN operator classes included in the standard distribution are documented in this [table](https://www.postgresql.org/docs/current/gin-builtin-opclasses.html" \l "GIN-BUILTIN-OPCLASSES-TABLE). For more information see [Chapter 70](https://www.postgresql.org/docs/current/gin.html) in the documentation.

#### BRIN

[BRIN](https://www.postgresql.org/docs/current/indexes-types.html" \l "INDEXES-TYPES-BRIN) indexes (a shorthand for Block Range INdexes) store summaries about the values stored in consecutive physical block ranges of a table. Thus, they are most effective for columns whose values are well-correlated with the physical order of the table rows. Like GiST, SP-GiST and GIN, BRIN can support many different indexing strategies, and the particular operators with which a BRIN index can be used vary depending on the indexing strategy. For data types that have a linear sort order, the indexed data corresponds to the minimum and maximum values of the values in the column for each block range. This supports indexed queries using these operators:

< <= = >= >

The BRIN operator classes included in the standard distribution are documented in this [t](https://www.postgresql.org/docs/current/brin-builtin-opclasses.html" \l "BRIN-BUILTIN-OPCLASSES-TABLE)able. For more information see [Chapter 71](https://www.postgresql.org/docs/current/brin.html) in the documentation.

### Sequences

In PostgreSQL, a sequence is a database object that allows you to generate a sequence of unique integers.

Typically, you use a sequence to generate a **unique identifier** for a **primary key** in a table. Additionally, you can use a sequence to generate unique numbers across tables.

To create a new sequence, you use the **CREATE SEQUENCE** statement.

CREATE SEQUENCE [ IF NOT EXISTS ] sequence\_name

[ AS { SMALLINT | INT | BIGINT } ]

[ INCREMENT [ BY ] increment ]

[ MINVALUE minvalue | NO MINVALUE ]

[ MAXVALUE maxvalue | NO MAXVALUE ]

[ START [ WITH ] start ]

[ CACHE cache ]

[ [ NO ] CYCLE ]

[ OWNED BY { table\_name.column\_name | NONE } ]

To get the next value from the sequence, you use the nextval() function:

SELECT nextval('mysequence');

#### sequence\_name

Specify the name of the sequence after the **CREATE SEQUENCE** clause. The I**F NOT EXISTS** conditionally creates a new sequence only if it does not exist. The sequence name must be distinct from any other sequences, tables, indexes, views, or foreign tables in the same schema.

#### AS { SMALLINT | INT | BIGINT }

Specify the data type of the sequence. The valid data type is **SMALLINT**, **INT**, and **BIGINT**. The default data type is **BIGINT** if you skip it.

The data type of the sequence which determines the sequence’s minimum and maximum values.

#### INCREMENT [ BY ] increment

The increment specifies which value to add to the current sequence value. A positive number will make an ascending sequence whereas a negative number will form a descending sequence. The default increment value is **1**.

#### MINVALUE MAXVALUE

Define the minimum value and maximum value of the sequence. If you use **NO MINVALUE** and **NO MAXVALUE**, the sequence will use the default value. For an ascending sequence, the default maximum value is the maximum value of the data type of the sequence and the default minimum value is 1. In the case of a descending sequence, the default maximum value is -1 and the default minimum value is the minimum value of the data type of the sequence.

#### cache

The CACHE determines how many sequence numbers are preallocated and stored in memory for faster access. One value can be generated at a time. By default, the sequence generates one value at a time i.e., no cache.

#### CYCLE | NO CYCLE

The **CYCLE** allows you to restart the value if the limit is reached. The next number will be the minimum value for the ascending sequence and the maximum value for the descending sequence. If you use **NO CYCLE**, when the limit is reached, attempting to get the next value will result in an error. The NO CYCLE is the default if you don’t explicitly specify CYCLE or NO CYCLE.

### OWNED BY table\_name.column\_name

The OWNED BY clause allows you to associate the table column with the sequence so that when you drop the column or table, PostgreSQL will automatically drop the associated sequence. Note that when you use the **SERIAL** pseudo-type for a column of a table, behind the scenes, PostgreSQL automatically creates a sequence associated with the column.

[17 Practical psql Commands That You Don't Want To Miss (postgresqltutorial.com)](https://www.postgresqltutorial.com/postgresql-administration/psql-commands/)

[Run PostgreSQL and pgAdmin in docker for local development using docker compose - Blogs, Ideas, Train of Thoughts (belowthemalt.com)](https://belowthemalt.com/2021/06/09/run-postgresql-and-pgadmin-in-docker-for-local-development-using-docker-compose/)

https://medium.com/coding-blocks/creating-user-database-and-adding-access-on-postgresql-8bfcd2f4a91e

## Backup a Single PostgreSQL Database

## SqlLedger admin page

Connect to the site entering URL

http://localhost/sql-ledger/admin.pl

The initial password is "pass123"/ I chnged it to „rublis“.