

after install postgresql on ubuntu via apt

After you install PostgreSQL on Ubuntu using **apt**, there are a few important things you should do next (to confirm it works and to start using it).

1. Check PostgreSQL service status

bash

 Copy code

```
sudo systemctl status postgresql
```

If it's running, you should see **active (running)**.

If not, start it:

bash

 Copy code

```
sudo systemctl start postgresql
sudo systemctl enable postgresql
```

2. Check PostgreSQL version

bash

 Copy code

```
psql --version
```

3. Switch to the default postgres admin user

PostgreSQL creates a Linux user called `postgres`.

bash

 Copy code

```
sudo -i -u postgres
```

Then open PostgreSQL shell:

```
bash
```

 Copy code

```
psql
```

Exit psql:

```
sql
```

 Copy code

```
\q
```

Exit postgres Linux user:

```
bash
```

 Copy code

```
exit
```

4. Set password for postgres database user (important)

Run:

```
bash
```

 Copy code

```
sudo -u postgres psql
```

Then inside psql:

```
sql
```

 Copy code

```
ALTER USER postgres PASSWORD 'YourStrongPassword';
```

Exit:

```
sql
```

 Copy code

```
\q
```

5. Create your own database user and database (recommended)

Example:

bash

 Copy code

```
sudo -u postgres createuser --interactive
```

Then create a database:

bash

 Copy code

```
sudo -u postgres createdb yourdbname
```

6. Test login with your user

bash

 Copy code

```
psql -U youruser -d yourdbname
```

If password login fails, it may be because Ubuntu defaults to **peer authentication**.

7. (Optional) Allow remote connections

Edit config file:

bash

 Copy code

```
sudo nano /etc/postgresql/*main/postgresql.conf
```

Find:

conf

```
#listen_addresses = 'localhost'
```

Change to:

conf

 Copy code

```
listen_addresses = '*'
```

Then edit:

bash

 Copy code

```
sudo nano /etc/postgresql/*/main/pg_hba.conf
```

Add:

conf

 Copy code

```
host      all      all      0.0.0.0/0      md5
```

Restart PostgreSQL:

bash

 Copy code

```
sudo systemctl restart postgresql
```

Also open firewall (if UFW enabled):

bash

 Copy code

```
sudo ufw allow 5432/tcp
```

If you want, I can give you the **exact commands** for creating a database + user for an app (best practice setup).



login postgre via pgadmin4

To login to PostgreSQL using **pgAdmin4**, you need to make sure PostgreSQL allows **password login** (not only local peer auth).

Step 1: Install pgAdmin4 (if not installed)

bash

 Copy code

```
sudo apt install pgadmin4
```

Step 2: Make sure PostgreSQL listens on the correct IP

Check this file:

bash

 Copy code

```
sudo nano /etc/postgresql/*main/postgresql.conf
```

Find:

conf

 Copy code

```
#listen_addresses = 'localhost'
```

Change to:

If pgAdmin is on same Ubuntu machine

conf

 Copy code

```
listen_addresses = 'localhost'
```

If pgAdmin is on another PC (remote access)

conf

 Copy code

```
listen_addresses = '*'

---


```

Step 3: Allow password login in pg_hba.conf

Edit:

bash

 Copy code

```
sudo nano /etc/postgresql/*/main/pg_hba.conf
```

Make sure you have this line:

For local login (pgAdmin on same machine)

conf

 Copy code

```
host      all      all      127.0.0.1/32      md5
```

For remote login (pgAdmin from another machine)

conf

 Copy code

```
host      all      all      0.0.0.0/0      md5
```

 If you use remote access, it's better to restrict IP like:

conf

 Copy code

```
host      all      all      192.168.1.50/32      md5
```

Step 4: Set password for your PostgreSQL user

Run:

bash

 Copy code

```
sudo -u postgres psql
```

Then:

```
sql
```

 Copy code

```
ALTER USER postgres PASSWORD 'YourPassword';
```

Exit:

```
\q
```

 Copy code

Step 5: Restart PostgreSQL

```
bash
```

 Copy code

```
sudo systemctl restart postgresql
```

Step 6: Login using pgAdmin4 GUI

Open pgAdmin4 and:

Add New Server

- **Name:** anything (ex: MyPostgres)

Connection tab:

- **Host name/address:**
 - If pgAdmin is on same machine: localhost
 - If remote: server IP address (ex: 192.168.1.100)
- **Port:** 5432
- **Maintenance database:** postgres
- **Username:** postgres (or your DB user)
- **Password:** the password you set

Click **Save**.

Step 7: If remote, open firewall port 5432

If UFW enabled:

bash

 Copy code

```
sudo ufw allow 5432/tcp
```

Common Problem

 "no pg_hba.conf entry for host"

Means your pg_hba.conf does not allow your IP.

 "password authentication failed"

Means wrong password OR user is not using md5/scram.

If you tell me whether pgAdmin4 is on **same Ubuntu server or on your Windows PC**, I can give you the exact best config.

      ...

[create a simple pg database](#)

Sure. Here is the simplest way to create a PostgreSQL database on Ubuntu.

1. Login as postgres user

bash

 Copy code

```
sudo -i -u postgres
```

2. Open PostgreSQL shell

bash

 Copy code

```
psql
```

3. Create a new database

Example database name: testdb

sql

 Copy code

```
CREATE DATABASE testdb;
```

4. Verify database created

sql

 Copy code

```
\l
```

5. Connect to the database

sql

 Copy code

```
\c testdb
```

6. Create a simple table

sql

 Copy code

```
CREATE TABLE students (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    age INT
);
```

7. Insert sample data

sql

 Copy code

```
INSERT INTO students (name, age)
VALUES ('David', 25),
       ('Alice', 22);
```

8. Query the table

sql

 Copy code

```
SELECT * FROM students;
```

9. Exit PostgreSQL

sql

 Copy code

```
\q
```

Exit postgres Linux user:

bash

 Copy code

```
exit
```

If you want, I can also show you how to create a **database user + password** so pgAdmin4 can connect cleanly.

      ...

padmin4 create a table

To create a table using pgAdmin4, do this:

Method 1 (Easiest): Use Query Tool (SQL)

1. Open pgAdmin4

Login → expand your server:

Servers → YourServer → Databases → testdb

2. Click your database (example: testdb)

Then click:

Tools → Query Tool

3. Run SQL to create a table

Paste this:

sql

 Copy code

```
CREATE TABLE students (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    age INT
);
```

Click the  Execute button.

4. Verify the table exists

Run:

sql

 Copy code

```
SELECT * FROM students;
```

It will return empty rows (because no data yet).

Method 2: Create table using GUI

1. Expand database tree

Databases → testdb → Schemas → public → Tables

2. Right click Tables

Select:

Create → Table

3. Fill in table name

- Name: students
- Schema: public

Click Save (not done yet)

4. Add columns

Go to Columns tab → click +

Add:

Column 1

- Name: id
- Data type: serial
- Check: Primary Key

Column 2

- Name: name
- Data type: varchar(50)

Column 3

- Name: age
- Data type: integer

Click Save.

Insert data (SQL way)

Query Tool:

sql

 Copy code

```
INSERT INTO students (name, age)
VALUES ('David', 25),
       ('Alice', 22);
```

Check:

sql

 Copy code

```
SELECT * FROM students;
```

If you want, I can show you how to insert rows using pgAdmin GUI (no SQL).



[insert data using pgadmin](#)

Here's how you can **insert data into a table using pgAdmin4 GUI** without writing SQL:

1. Open pgAdmin4 and select your table

- Servers → YourServer → Databases → testdb → Schemas → public → Tables →

students

- Right-click **students** → **View/Edit Data** → **All Rows**
-

2. Insert new rows

- A grid opens (like a spreadsheet) showing your table columns.
 - Scroll to the bottom (it shows a blank row with * in the first column).
 - Enter data for each column:
 - **id**: leave blank (if `SERIAL`, PostgreSQL auto-generates)
 - **name**: e.g., Bob
 - **age**: e.g., 30
-

3. Save the data

- Click the  **Save** button on the top toolbar (or press **Ctrl+S**) to commit the new row.
 - The row is now inserted into the database.
-

4. Verify data

- You should see your new row in the grid.
- Or run SQL in **Query Tool**:

sql

 Copy code

```
SELECT * FROM students;
```

If you want, I can make a **step-by-step screenshot-style guide** for inserting multiple rows in pgAdmin4—it's very visual and easier for beginners.

Do you want me to do that?



I see a lock on the table

Ah! That **lock icon** in pgAdmin4 is normal—it shows the **table is currently being viewed or edited**, or it reflects **row-level locks** from other sessions. Here's what it means and what you can do:

1. Why the lock appears

- **Viewing/Edit Mode:** When you open **View/Edit Data → All Rows**, pgAdmin automatically locks the table to prevent conflicting edits.
 - **Another session:** If someone else (or another pgAdmin window) is editing the table, PostgreSQL locks it to avoid conflicts.
 - **Transactions not committed:** If you started inserting or updating rows but didn't **save/commit**, the table stays locked for that session.
-

2. How to release the lock

- **Save/Commit your changes** in pgAdmin:
 - Click the  Save button in the data grid.
- **Close the data grid tab** in pgAdmin.
- **Disconnect idle sessions** if needed:
 - Right-click server → **Disconnect** and reconnect.
- **Check locks (optional):** Run SQL in **Query Tool**:

sql

 Copy code

```
SELECT * FROM pg_locks l
JOIN pg_class t ON l.relation = t.oid
WHERE t.relname = 'students';
```

This shows active locks on the table.

3. Tip

- Don't leave **Edit Data** grids open for too long; PostgreSQL keeps a lock until you commit or close.
 - For normal SELECT queries, no lock appears.
-

If you want, I can show a **quick way to insert multiple rows in pgAdmin without locking issues**—so you won't see the lock each time.

Do you want me to show that?



postgres comment out sQL

In PostgreSQL / SQL, you can comment out lines in two ways:

1. Single-line comment

Use `--`

sql

Copy code

```
-- This is a single-line comment
SELECT * FROM students; -- This comment is after a statement
```

Everything after `--` on the same line is ignored by PostgreSQL.

2. Multi-line (block) comment

Use `/* ... */`

sql

Copy code

```
/*
This is a
multi-line comment
*/
SELECT * FROM students;
```

You can also put it **inside** a line:

sql

 Copy code

```
SELECT /* column list */ * FROM students;
```



- -- is simplest for quick commenting.
 - /* */ is good for temporarily disabling multiple lines of SQL code.
-

If you want, I can show a **trick to quickly comment/uncomment multiple lines in pgAdmin Query Tool**—very handy when testing SQL. Do you want me to?



create a simple database with these 2 requirements "

Select six recent popular movies (or television episodes or books or songs or ...)

Ask at least five people to rate each movie they have seen on a 1–5 scale."

Got it! You want a **simple PostgreSQL database** that can store:

1. A list of 6 items (movies, for example).

2. Ratings from at least 5 people for each item.

Here's a clean way to design it.

Step 1: Create the database

sql

 Copy code

```
-- Create the database
CREATE DATABASE movie_ratings;
```

Connect to it:

sql

 Copy code

```
\c movie_ratings
```

Step 2: Create tables

We need two tables: `movies` and `ratings`.

Table 1: movies

sql

 Copy code

```
CREATE TABLE movies (
    movie_id SERIAL PRIMARY KEY,
    title VARCHAR(100) NOT NULL
);
```

Table 2: ratings

sql

 Copy code

```
CREATE TABLE ratings (
    rating_id SERIAL PRIMARY KEY,
    movie_id INT REFERENCES movies(movie_id),
    rater_name VARCHAR(50) NOT NULL,
```

```
rating INT CHECK (rating BETWEEN 1 AND 5)
);
```

- `movies` stores movie titles.
 - `ratings` stores the rating (1–5) for each movie by each person.
-

Step 3: Insert sample movies

sql

 Copy code

```
INSERT INTO movies (title) VALUES
('Avatar: The Way of Water'),
('Oppenheimer'),
('Barbie'),
('Stranger Things S5'),
('The Marvels'),
('Killers of the Flower Moon');
```

Step 4: Insert sample ratings from 5 people

sql

 Copy code

```
INSERT INTO ratings (movie_id, rater_name, rating) VALUES
(1, 'Alice', 5),
(1, 'Bob', 4),
(1, 'Charlie', 3),
(1, 'David', 4),
(1, 'Eve', 5),

(2, 'Alice', 4),
(2, 'Bob', 5),
(2, 'Charlie', 4),
(2, 'David', 3),
(2, 'Eve', 4),

(3, 'Alice', 3),
(3, 'Bob', 4),
```

```
(3, 'Charlie', 5),  
(3, 'David', 3),  
(3, 'Eve', 4),  
  
(4, 'Alice', 5),  
(4, 'Bob', 5),  
(4, 'Charlie', 4),  
(4, 'David', 4),  
(4, 'Eve', 5),  
  
(5, 'Alice', 3),  
(5, 'Bob', 4),  
(5, 'Charlie', 3),  
(5, 'David', 4),  
(5, 'Eve', 3),  
  
(6, 'Alice', 4),  
(6, 'Bob', 4),  
(6, 'Charlie', 5),  
(6, 'David', 5),  
(6, 'Eve', 4);
```

Step 5: Query the data

List all movies and ratings:

sql Copy code

```
SELECT m.title, r.rater_name, r.rating  
FROM movies m  
JOIN ratings r ON m.movie_id = r.movie_id  
ORDER BY m.movie_id, r.rater_name;
```

Average rating per movie:

sql Copy code

```
SELECT m.title, ROUND(AVG(r.rating), 2) AS avg_rating  
FROM movies m
```

```
JOIN ratings r ON m.movie_id = r.movie_id
GROUP BY m.title
ORDER BY avg_rating DESC;
```

This setup is **simple**, **normalized**, and fully supports your requirements:

- 6 movies
 - At least 5 ratings per movie
 - Easy to query averages, top-rated movies, etc.
-

If you want, I can also **show how to do the same thing entirely in pgAdmin4 using the GUI**, so you can create tables and insert ratings without typing SQL.

Do you want me to do that?



now switch rstudio to connect to this postgres database

Perfect! To connect RStudio to your PostgreSQL database, you'll need the **RPostgres** package (or **RPostgreSQL**). Here's the step-by-step guide:

1 Install necessary R packages

Open RStudio and run:

r

Copy code

```
install.packages("DBI")      # Generic database interface
install.packages("RPostgres") # PostgreSQL driver
```

2 Load packages

r

 Copy code

```
library(DBI)  
library(RPostgres)
```

3 Connect to your PostgreSQL database

Replace placeholders with your setup:

r

 Copy code

```
con <- dbConnect(  
  RPostgres::Postgres(),  
  dbname = "movie_ratings",    # your database name  
  host = "localhost",         # or server IP  
  port = 5432,                # default PostgreSQL port  
  user = "postgres",          # your DB username  
  password = "YourPassword"   # your DB password  
)
```

4 Test the connection

List tables:

r

 Copy code

```
dbListTables(con)
```

You should see:

csharp

 Copy code

```
[1] "movies" "ratings"
```

5 Query the data

Example: get all movies and ratings:

r

 Copy code

```
query <- "SELECT m.title, r.rater_name, r.rating  
FROM movies m  
JOIN ratings r ON m.movie_id = r.movie_id  
ORDER BY m.movie_id, r.rater_name;"  
  
data <- dbGetQuery(con, query)  
print(data)
```

6 Close the connection

r

 Copy code

```
dbDisconnect(con)
```

⚠️ Tips

- 1. Local connection:** If PostgreSQL is on the same machine as RStudio, `host = "localhost"` is fine.
- 2. Remote connection:** Use the server's IP, and make sure firewall/pg_hba.conf allows your RStudio client.
- 3. pgAdmin password:** Use the same DB username/password you set in PostgreSQL.

If you want, I can write a **full R script** that:

- Connects to PostgreSQL
- Creates tables
- Inserts the movie ratings data
- Queries average ratings

This way you can do everything directly from RStudio.

Do you want me to do that?



how to dealing with NA value in the table

In PostgreSQL and R, **NA values (missing values)** can appear in tables or query results, and how you handle them depends on whether you want to **ignore, replace, or filter** them. Here's a thorough guide:

1 In PostgreSQL (SQL)

a) Check for NULLs

In PostgreSQL, missing values are stored as `NULL` (not NA).

sql

Copy code

```
SELECT * FROM ratings  
WHERE rating IS NULL;
```

b) Filter out NULLs

sql

Copy code

```
SELECT * FROM ratings  
WHERE rating IS NOT NULL;
```

c) Replace NULLs with a default value

Use `COALESCE()` to replace NULLs:

sql

```
SELECT rater_name, COALESCE(rating, 0) AS rating
FROM ratings;
```

Here, any missing rating is replaced with 0.

d) Average while ignoring NULLs

PostgreSQL automatically ignores NULLs in aggregation:

sql

 Copy code

```
SELECT movie_id, AVG(rating) AS avg_rating
FROM ratings
GROUP BY movie_id;
```

No need to filter manually; NULLs are skipped in AVG, SUM, etc.

2 In R (using DBI or dplyr)

Suppose you read data from PostgreSQL:

r

 Copy code

```
library(DBI)
library(dplyr)

data <- dbGetQuery(con, "SELECT * FROM ratings")
```

a) Check for NA values

r

 Copy code

```
summary(data)          # shows NAs per column
any(is.na(data))      # TRUE if any NA exists
```

b) Filter out NAs

r

Copy code

```
clean_data <- data %>%
  filter(!is.na(rating))
```

c) Replace NAs

r

Copy code

```
data$rating[is.na(data$rating)] <- 0 # replace NAs with 0
```

or using `mutate`:

r

Copy code

```
data <- data %>%
  mutate(rating = ifelse(is.na(rating), 0, rating))
```

d) Aggregate while handling NAs

r

Copy code

```
avg_ratings <- data %>%
  group_by(movie_id) %>%
  summarise(avg_rating = mean(rating, na.rm = TRUE))
```

`na.rm = TRUE` ignores NAs during the calculation.

Tips / Best Practices

1. Decide what NA means:

- NA = missing rating → you can leave as NULL in DB.
- NA = "not applicable" → may replace with 0 or other default.

2. Avoid inserting empty strings for numeric columns; use `NULL` instead.
 3. Always handle NA in queries for aggregation or plotting.
-

If you want, I can show a **full example** for your movie ratings table where some ratings are missing, and we handle NA in **both PostgreSQL and R**, including computing average ratings correctly.

Do you want me to do that?



Count ratings per user and movie

Got it! You want to **count how many ratings each user gave and/or how many ratings each movie received**. Here's how to do it in **PostgreSQL** and in **R**.

1 In PostgreSQL

a) Count ratings per user (`rater_name`)

sql

Copy code

```
SELECT rater_name, COUNT(*) AS ratings_count
FROM ratings
GROUP BY rater_name
ORDER BY ratings_count DESC;
```

This will show how many ratings each person gave.

b) Count ratings per movie

sql

Copy code

```
SELECT m.title, COUNT(r.rating) AS ratings_count
FROM movies m
LEFT JOIN ratings r ON m.movie_id = r.movie_id
GROUP BY m.title
ORDER BY ratings_count DESC;
```

- LEFT JOIN ensures movies with **no ratings** still appear (count = 0).
 - COUNT(r.rating) ignores NULLs.
-

c) Count ratings per user per movie

sql Copy code

```
SELECT m.title, r.rater_name, COUNT(r.rating) AS ratings_count
FROM movies m
JOIN ratings r ON m.movie_id = r.movie_id
GROUP BY m.title, r.rater_name
ORDER BY m.title, r.rater_name;
```

This shows, for each movie, how many ratings each user submitted (usually 1 per user).

2 In R (after fetching data)

r Copy code

```
library(dplyr)

# Fetch ratings
data <- dbGetQuery(con, "SELECT * FROM ratings JOIN movies USING(movie_id)")

# Count per user
data %>%
  group_by(rater_name) %>%
  summarise(ratings_count = n())

# Count per movie
data %>%
```

```
group_by(title) %>%  
  summarise(ratings_count = n())  
  
# Count per user per movie  
data %>%  
  group_by(title, rater_name) %>%  
  summarise(ratings_count = n())
```

Tips:

- Usually, each user has **one rating per movie**, so `COUNT(*)` is 1 per row.
 - If you allowed **multiple ratings per user/movie**, the above queries still work to aggregate.
 - Use `COUNT(rating)` instead of `COUNT(*)` to ignore missing ratings (`NULL`).
-

If you want, I can write a **single query** that gives a nice table showing for all movies: **total ratings, average rating, and number of users who rated**, ready for R or reporting.

Do you want me to do that?



Compute average ratings

Perfect! To **compute average ratings** for your movies, you can do it in **PostgreSQL** or R. I'll show both.

1 In PostgreSQL

a) Average rating per movie

sql

 Copy code

```
SELECT m.title, ROUND(AVG(r.rating), 2) AS avg_rating
FROM movies m
JOIN ratings r ON m.movie_id = r.movie_id
GROUP BY m.title
ORDER BY avg_rating DESC;
```

Explanation:

- `AVG(r.rating)` computes the average rating.
 - `ROUND(..., 2)` rounds to 2 decimal places.
 - `JOIN` ensures we get movie titles with their ratings.
 - Movies with no ratings won't appear (use `LEFT JOIN` if you want them with `NULL`).
-

b) Average rating per user

sql

 Copy code

```
SELECT r.rater_name, ROUND(AVG(r.rating), 2) AS avg_rating
FROM ratings r
GROUP BY r.rater_name
ORDER BY avg_rating DESC;
```

This shows each user's average rating given.

c) Optional: Average rating per movie ignoring missing ratings

sql

 Copy code

```
SELECT m.title, ROUND(AVG(r.rating), 2) AS avg_rating
FROM movies m
LEFT JOIN ratings r ON m.movie_id = r.movie_id AND r.rating IS NOT NULL
GROUP BY m.title
ORDER BY avg_rating DESC;
```

2 In R (after fetching data)

r

Copy code

```
library(dplyr)

# Fetch ratings with movie titles
data <- dbGetQuery(con, "SELECT * FROM ratings JOIN movies USING(movie_id)")

# Average rating per movie
data %>%
  group_by(title) %>%
  summarise(avg_rating = round(mean(rating, na.rm = TRUE), 2))

# Average rating per user
data %>%
  group_by(rater_name) %>%
  summarise(avg_rating = round(mean(rating, na.rm = TRUE), 2))
```

Notes:

- `na.rm = TRUE` ensures missing ratings (`NA`) don't affect the average.
- `round(..., 2)` gives a clean 2-decimal average.

If you want, I can make a **single table** showing **movie title, number of ratings, and average rating**, ready for reporting or plotting in R.

Do you want me to do that?



based on the average rating do a col plot

Perfect! Once you have **average ratings per movie**, you can make a **column/bar plot** in R easily using `ggplot2`. Here's a complete example:

1 Install and load ggplot2 (if not already)

r

Copy code

```
install.packages("ggplot2") # only once
library(ggplot2)
library(dplyr)
```

2 Fetch data and compute average ratings

r

Copy code

```
library(DBI)
library(RPostgres)

# Connect to PostgreSQL
con <- dbConnect(
  RPostgres::Postgres(),
  dbname = "movie_ratings",
  host = "localhost",
  port = 5432,
  user = "postgres",
  password = "YourPassword"
)

# Fetch ratings with movie titles
data <- dbGetQuery(con, "SELECT * FROM ratings JOIN movies USING(movie_id)")

# Compute average rating per movie
avg_ratings <- data %>%
  group_by(title) %>%
  summarise(avg_rating = round(mean(rating, na.rm = TRUE), 2))
```

3 Make a column/bar plot

r

Copy code

```
ggplot(avg_ratings, aes(x = reorder(title, -avg_rating), y = avg_rating)) +  
  geom_col(fill = "steelblue") +          # create the bars  
  geom_text(aes(label = avg_rating), vjust = -0.5) +  # show value on top  
  labs(title = "Average Ratings per Movie",  
       x = "Movie",  
       y = "Average Rating") +  
  ylim(0, 5) +                         # rating scale 1-5  
  theme_minimal() +  
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```

✓ Explanation:

- `reorder(title, -avg_rating)` → sorts bars descending by average rating.
 - `geom_col()` → creates vertical bars (columns).
 - `geom_text()` → labels each bar with the rating value.
 - `ylim(0, 5)` → keeps y-axis consistent with 1–5 rating scale.
 - `theme(axis.text.x = element_text(angle = 45))` → rotates movie names to avoid overlap.
-

This will give a **clean column plot of movies vs average ratings**.

If you want, I can make a **fully annotated plot with colors for high vs low ratings**, so you can visually see which movies are rated best.

Do you want me to do that?

