POSTGRESQL

BY MOHAMMAD HASAN KHODDMI

WHAT IS POSTGRESQL

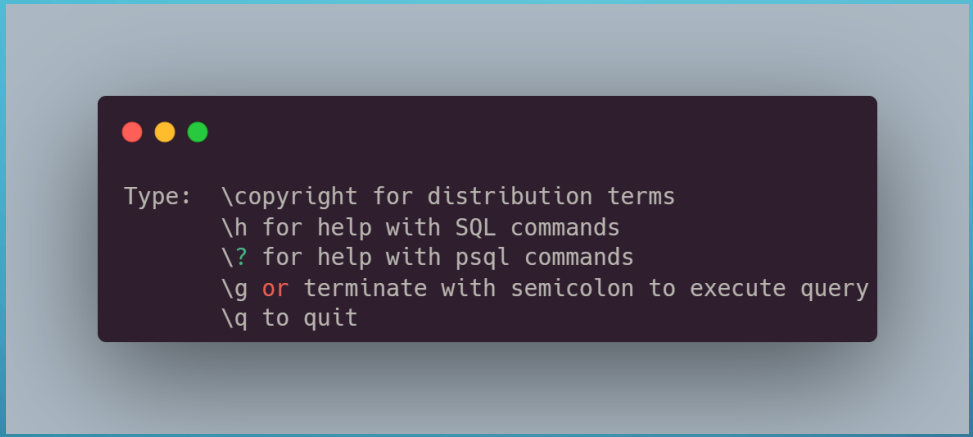
- Postgresql is a database that supported language is sql (sql based database)
- This database has a great match with python and Django
- For using postgresql in windows we can use pgadmin4 or using sqlshell(psql)\
- The next slide will be command of sql and postgresql

COMMANDS AND DETAILS

- First of all as we said postgresql using sql language wich means it has a temporary memory and it saved information before (;) so u can type many lines then run It (execute it) by (;)
- For many syntaxes and command we have to right it in capital form

BY TYPING HELP

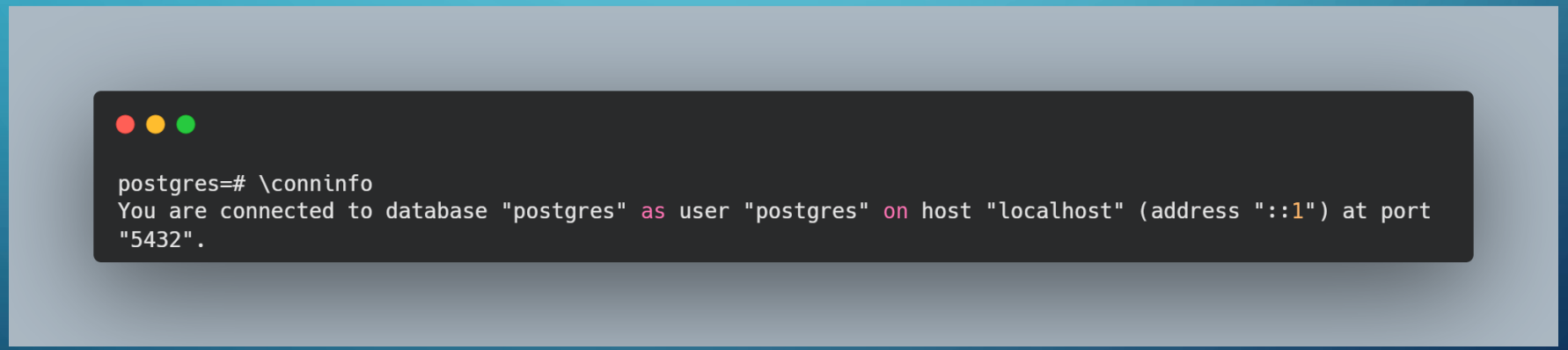
- We see this menu for getting help commands from posgresql



Also we can using \g instade of (;) but I prefer to use (;)

THE DEFAULT USERNAME AND PASSWORD

- Default username is postgres, default port is 5432 , and we can get this information by using \conninfo … the password that we have to write is the password that we set when we was installing the postgresql



COMMANDS

- for creating data base we using: CREATE DATABASE
- (Capital form)
- For getting list of databases using \l
- For delete database using :

DROP DATABASE (name)

```
postgres=# CREATE DATABASE name
postgres-# ;
CREATE DATABASE
```

```
postgres=# DROP DATABASE name
postgres-# ;
DROP DATABASE
```

```
postgres=# \l
                                                                    List of databases
   Name    |  Owner   | Encoding | Locale Provider |          Collate          |          Ctype           |  ICU Locale | ICU Rules |   Access privileges
-----------+----------+----------+-----------------+---------------------------+---------------------------+-------------+-----------+-----------------------
 hlweb     | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |             |           |
 name      | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |             |           |
 postgres  | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |             |           |
 template0 | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |             |           | =c/postgres          +
           |          |          |                 |                           |                           |             |           | postgres=CTc/postgres
 template1 | postgres | UTF8     | libc            | English_United States.1252 | English_United States.1252 |             |           | =c/postgres          +
           |          |          |                 |                           |                           |             |           | postgres=CTc/postgres
(5 rows)
```

QUERY BUFFER AND NOTES

- Query buffer is the temporary memory which save data before execute it by (;)
- > means you are normal member but # means you are superuser (in sql shell)

CONNECTING TO DATABASE

- For switch or connecting into the database we can use 2 specific commands

EXPANDED AND UNEXPANDED LIST

- For showing expanded or unexpanded databases we using \x and we can see in command lists that expanding show is on/off
- The expanded show is like this

CREATE TABLE

- For creating table we using CREATE TABLE name() in the parentheses we have to write the form and shape of the table and additional commands that we want. Actually we setting data types of our table

```
postgres=# CREATE TABLE users(
postgres(# id integer not null primary key,
postgres(# username varchar(100)
postgres(# );
CREATE TABLE
```

For knowing all data types please read the following link
https://www.w3schools.com/sql/sql_datatypes.asp

PRIMARY KEY

- Primary key sort the table , we define that which datatype is our primary key and it`s gonna be sorted by our defined primary key

Remember that after defining each data types using comma (,) is neccesary

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    PRIMARY KEY (ID)
);
```

WATCHING TABLES

- For watching all tables using \d then we can watch all tables
- For watching an specific or target table using \d name_of_table

```
postgres=# \d
List of relations
-[ RECORD 1 ]----
Schema | public
Name   | users
Type   | table
Owner  | postgres


postgres=# \d users
                    Table "public.users"
  Column  |          Type          | Collation | Nullable | Default
----------+------------------------+-----------+----------+---------
 id       | integer                |           | not null |
 username | character varying(100) |           |          |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)


postgres=#
```

DROP DATABASE EXCEPTION

- If currently we are connecting to database we can`t drop it
- First of all have to switch to another database then using drop database command

TEMPLATE0 AND TEMPLATE 1 DATABASES

- First of all don`t delete these two because when we command CREATE DATABASE name, postgresql using template 1 copy that and create another database with defined name every database that we create is a copy from template 1
- If we change template 1 , then every database we create has the same option and changes that we made for example:
- If we create a table on template 1 then crate another database which call it shop , the new database has the table

TEMPLATE 0

- Template 0 is like template 1 same option and function except that we can`t connect to template 0 and change the setting and options
- Like template 1 we can create database by template 0 for this command have to write code like this

```
postgres=# CREATE DATABASE example TEMPLATE template0;
CREATE DATABASE
```

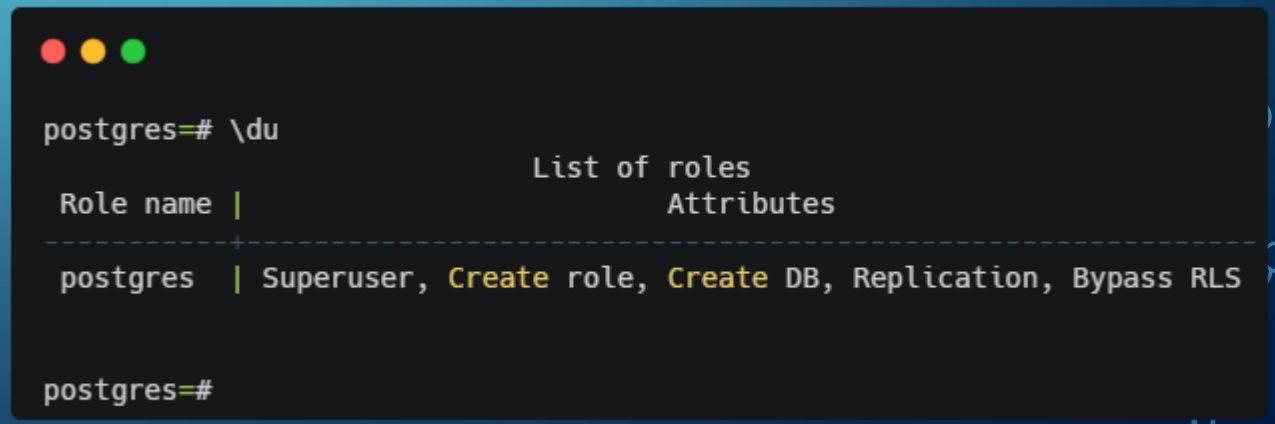
We can copy every database by using template … for example we can create another database by using (example) as template

ROLE

- The member of database are roles by default we have 1 rule known as postgres but we can make more roles.
- We shouldn`t give anyone access database by postgres user because postgres user is a superuser and it`s could be dangerous
- Can create role by CREATE ROLE name ATTRIBUTE
- For knowing all attributes read the following link:
- https://www.postgresql.org/docs/current/role-attributes.html

COMMANDS AND EXPLAINATION

- \du : showing lists of roles in our database (as we remember role means member)
- Attributes means permission which the role has it
- Member of Part means is the role joined a group or not (we will talk about it later)



COMMAND

- We can delete a role by DROP ROLE and can update(change) a role by ALTER ROLE
- Roles by default can`t login into our database if they want to login we have to give them login attribute

CREATE USER VS CREATE ROLE

- In create user , by default has the login attribute but for CREATE ROLE we have to define login attribute

CONNECTING TO USER

- For connecting to user first of all we have to define a password for a role or user
- Second of all we have to give him login attribute even if has superuser
- The third of all for connecting to database by user use this command :
- \c (or connect) databasename rolename
- And for checking the info use \conninfo
- Don`t forget to write password
- For create password use following command:
- CREATE ROLE name WITH PASSWORD ‘STRING’

ALTER ROLE

- ALTER ROLE use for changing (updating) role attributes
- For change attribute use this command :
- ALTER ROLE username ATTRIBUTE
- For taking an ATTRIBUTE from a ROLE write NO pre the ATTRIBUTE for example
- ALTER CHANGE Niklaus NOSUPERUSER NO LOGIN
- The next slide Is an example from these 3 slides

EXAMPLE

```
postgres=# \du
                                   List of roles
 Role name |                         Attributes
-----------+------------------------------------------------------------
 niklaus   | Superuser, Cannot login
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS


postgres=# ALTER RULE niklaus LOGIN
postgres-# ;
ERROR:  syntax error at or near "LOGIN"
LINE 1: ALTER RULE niklaus LOGIN
                           ^
postgres=# ALTER ROLE niklaus LOGIN;
ALTER ROLE
postgres=# \du
                                   List of roles
 Role name |                         Attributes
-----------+------------------------------------------------------------
 niklaus   | Superuser
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS


postgres=# \c postgres niklaus
Password for user niklaus:
connection to server at "localhost" (::1), port 5432 failed: fe_sendauth: no
password supplied
Previous connection kept
postgres=# ALTER ROLE niklaus PASSWORD '123'
postgres-# ;
ALTER ROLE
postgres=# \c niklaus
connection to server at "localhost" (::1), port 5432 failed: FATAL:  database
"niklaus" does not exist
Previous connection kept
postgres=# \c postgres niklaus
Password for user niklaus:
You are now connected to database "postgres" as user "niklaus".
postgres=# \conninfo
You are connected to database "postgres" as user "niklaus" on host "localhost"
(address "::1") at port "5432".
postgres=#
```

NOTE

- Remember that the user that we create it has a different password to postgres user (default user)
- The fastest and easier way to change password of a role we can use this command:
- \password rolename ‘string’;

GROUP

- We can create a group which can roles join it
- The group has specific attributes and when a role joins the group the role will gains the attributes and no need to define attribute for each role just define attribute for the group and join the role into the group

CREATING GROUP

- For creating group step by step
- Step 1 : create a role and give it NOLOGIN attributes
- Step 2 : create a role and use this command to add it into a group
- CREATE ROLE name IN ROLE groupname;
- Step 3: for adding a made user into group use this command:
- ALTER GROUP groupname ADD USER name;
- For seeing the roles who joins the group use \drg
- Examples in next slides

EXAMPLE

```
postgres=# CREATE ROLE developers;
CREATE ROLE
postgres=# CREATE ROLE shakib IN ROLE developers;
CREATE ROLE
postgres=# ALTER GROUP developers ADD USER niklaus
postgres-# ;
ALTER ROLE
postgres=# \drg
                List of role grants
 Role name | Member of  |   Options    | Grantor
-----------+------------+--------------+----------
 niklaus   | developers | INHERIT, SET | postgres
 shakib    | developers | INHERIT, SET | postgres
(2 rows)
postgres=# \du
                                   List of roles
 Role name  |                         Attributes
------------+------------------------------------------------------------
 developers | Cannot login
 niklaus    | Cannot login
 postgres   | Superuser, Create role, Create DB, Replication, Bypass RLS
 shakib     | Cannot login
```

DELETING ROLE FROM GROUP

- For deleting role from group use following command:
- ALTER GROUP groupname DROP ROLE rolename;

SYSTEM CATALOG

- Look don`t touch
- Every data saves on system catalog
- If hacker want to hack database he`s gonna attack system catalogs
- Pg_database = database details
- We can read the details by using sql commands:
- SELECT * FROM PG_DATABASE

COMMANDS

- PG_AUTHID → giving information about members , command:
- SELECT * FROM pg_authid (has to be admin to has access to this section)
- -1 → no limitation in connecting database
- T → True
- Pg_roles → giving information about members but don`t need to be an admin
- SELECT * FROM pg_roles by the difrrent of pg_authid it won`t show the passowrds or etc.

COMMANDS

- Pg_tables → giving us details and information about tables in databases
- SELECT * FROM PG_TABLES

ABOUT POSTGRESQL SECURITY

- Using md5 security to hash passowords
- And it`s safe !
- Postgresql asking a user or role , the password for login

ACL

- Access control list → we can give any member some access
- Imagine that we have a table and want to shows to members but can`t change it, we can do it by acl.
- The commands are GRANTS and REVOKE
- GRANTS give the member access and REVOKE take the member access

ACL PRIVILAGE

All the access that we can give the member is sorted in this table

Privilege	Abbreviation	Applicable Object Types
SELECT	r (“read”)	LARGE OBJECT, SEQUENCE, TABLE (and table-like objects), table column
INSERT	a (“append”)	TABLE, table column
UPDATE	w (“write”)	LARGE OBJECT, SEQUENCE, TABLE, table column
DELETE	d	TABLE
TRUNCATE	D	TABLE
REFERENCES	x	TABLE, table column
TRIGGER	t	TABLE
CREATE	C	DATABASE, SCHEMA, TABLESPACE
CONNECT	c	DATABASE
TEMPORARY	T	DATABASE
EXECUTE	X	FUNCTION, PROCEDURE
USAGE	U	DOMAIN, FOREIGN DATA WRAPPER, FOREIGN SERVER, LANGUAGE, SCHEMA, SEQUENCE, TYPE
SET	s	PARAMETER
ALTER SYSTEM	A	PARAMETER

\DT VS \DP VS \D

- All of these commands using for showing us the table
- But the difference is :
- \d[S+] list tables, views, and sequences
- \dp[S] [PATTERN] list table, view, and sequence access privileges
- \dt[S+] [PATTERN] list tables
- The next slide is a example of these commands and their function

EXAMPLE

Also we can command \dt or \dp or \d + name_of_table which gives us more details

But \d just gives us details of information about table

1

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


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


education=# \dp
                             Access privileges
 Schema | Name  | Type  | Access privileges | Column privileges | Policies
--------+-------+-------+-------------------+-------------------+----------
 public | users | table |                   |                   |
(1 row)
```

2

```
education=# \dp users
                             Access privileges
 Schema | Name  | Type  | Access privileges | Column privileges | Policies
--------+-------+-------+-------------------+-------------------+----------
 public | users | table |                   |                   |
(1 row)


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


education=# \d users
              Table "public.users"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           |          |
```

INSERT

- By INSERT command we can insert a value to the table by following command:
- INSERT INTO table_name(datatype_name) ATTRIBUTE_NAME(value that we want to insert);

For reading the table use Select by following command:
SELECT * FROM table_name;

```
education=# INSERT INTO users(id) VALUES (2);
INSERT 0 1
education=# SELECT * FROM users;
 id
----
  2
(1 row)
```

GRANT

- If we create a new role the new role hasn`t some access like SELECT , example or etc… for knowing all access read 33th slide
- So we have to give him access by grant command if a member wants to read table has to have SELECT access if we create role without access when role trying to read tables or etc… role gets error.

```
 CREATE ROLE jack LOGIN;
CREATE ROLE
 ALTER ROLE jack password '12345'
education-# ;
ALTER ROLE
\conninfo
You are connected to database "education" as user "jack" on host "localhost"
(address "::1") at port "5432".
education=> SELECT * FROM users;
ERROR:  permission denied for table users
```

GRANT

- So have to give him GRANT of SELECT (for reading the tables) by following command (as postgres user(superuser)):
- GRANT access_name ON table_name TO role_name

By using \dp we can see who has access

After (/) means who give access to role

```
education=# GRANT SELECT ON users TO jack;
GRANT
education=# \dp
                                Access privileges
 Schema | Name  | Type  |     Access privileges     | Column privileges | Policies
--------+-------+-------+---------------------------+-------------------+----------
 public | users | table | postgres=arwdDxt/postgres+|                   |
        |       |       | jack=r/postgres           |                   |
(1 row)
```

Jack = r
Means jack = select
Means jack just can read the informations

GRANT FINAL

- So we give jack the select now we can read the tables by jack:

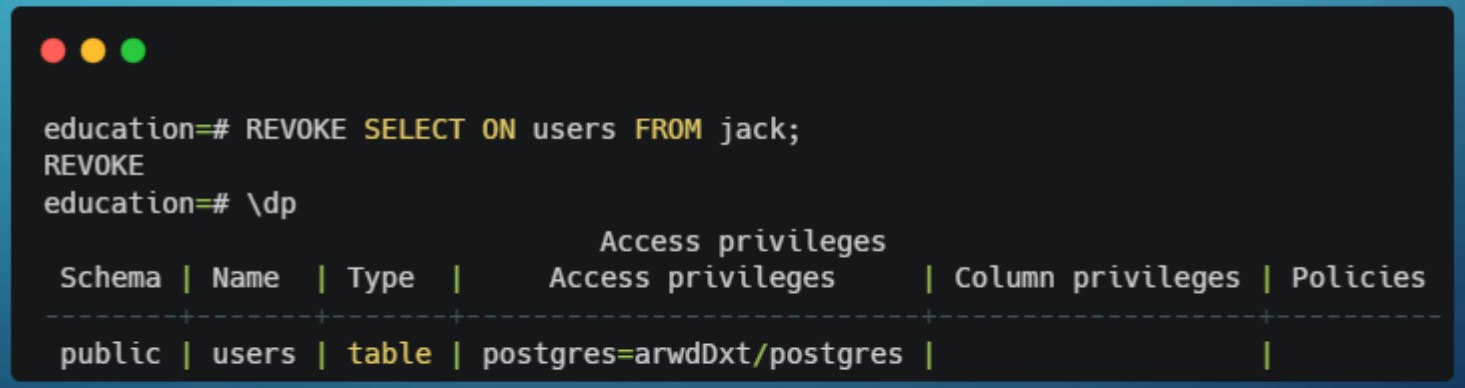
```
education=# \c education jack
Password for user jack:
You are now connected to database "education" as user "jack".
education=> \conninfo
You are connected to database "education" as user "jack" on host "localhost"
(address "::1") at port "5432".
education=> SELECT * FROM users;
 id
----
  2
(1 row)
```

But can`t use other access by jack

```
education=> INSERT INTO users(id) VALUES(9);
ERROR:  permission denied for table users
```

REVOKE

- Revoke is like GRANT by opposite side of it revoke take access from role or member but the following command is the same as GRANT with little diffrent:
- REVOKE access_name ON table_name FROM role_name;



NOTE

The postgres after (/) GRANT access to the postgres BEFORE (=)

Granted role

accesses

The role that grant

TYPES OF TABLES

- We have 3 types of table :
- Logged: the usual tables which created by CREATE TABLE name_of table(datatype_names)
- Unlogged
- Temporary

LOGGED TABLES

- Stability of logged tables is very high and there is a low chance to lose table information in logged tables
- But the weak point is these tables are slow and don`t have speed (probably in loading datas)

HOW TO FIND OUT A TYPE OF TABLE

- For getting type of table use sql command and read pg_class by following command:
- Imagine that we have table which called users now we want to find out the type of this table
- Step 1 : SELECT * FROM pg_class;
- Step 2 : find the target table In relnames
- Step 3 : find relpersistence if relpersistence is (p) means the table is logged

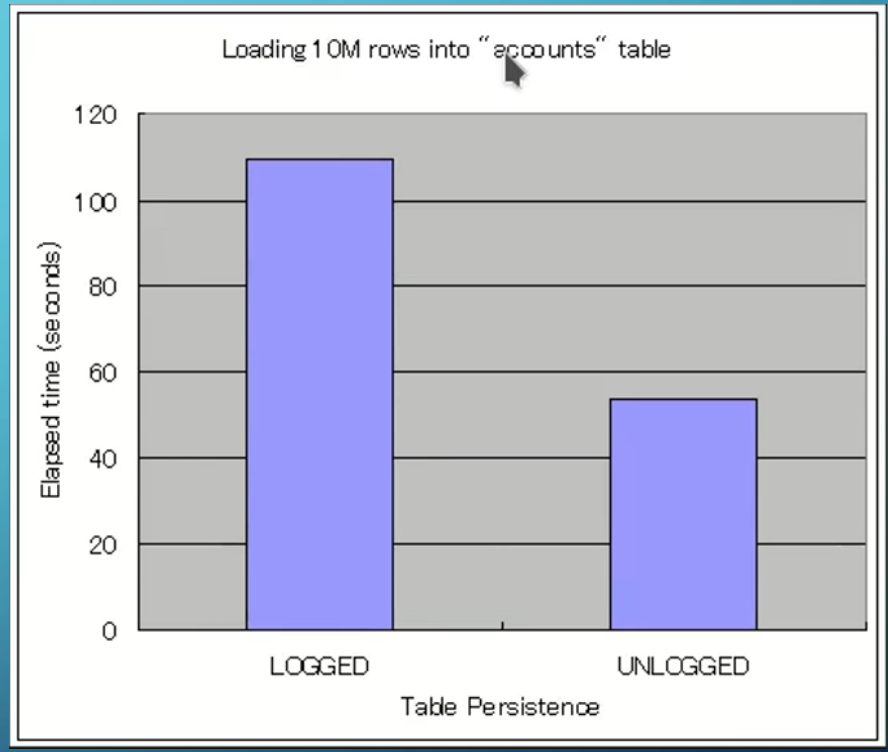
HOW TO FIND OUT A TYPE OF TABLE

```
postgres=# SELECT * FROM pg_class ;
-[ RECORD 1 ]-------+----------
oid                 | 16402
relname             | users
relnamespace        | 2200
reltype             | 16404
reloftype           | 0
relowner            | 10
relam               | 2
relfilenode         | 16402
reltablespace       | 0
relpages            | 0
reltuples           | -1
relallvisible       | 0
reltoastrelid       | 0
relhasindex         | t
relisshared         | f
relpersistence      | p
relkind             | r
relnatts            | 2
relchecks           | 0
relhasrules         | f
relhastriggers      | f
relhassubclass      | f
relrowsecurity      | f
relforcerowsecurity | f
relispopulated      | t
relreplident        | d
relispartition      | f
relrewrite          | 0
```

UNLOGGED TABLES

Unlogged tables has more speed than logged tables
In this picture 10Million rows gonna be add to tables
In logged table operation takes about 110 seconds
But in unlogged tables operation take about 50 seconds

If tables or information isn`t important to much we can use unlogged tables for example: sessions in web apllications



HOW TO CREATE UNLOGGED TABLES

- For creating unlogged table use the following command:
- CREATE UNLOGGED TABLE name_of_table(datatypes_name);
- How to find out it`s unlogged or not ? Again use SELECT * FROM pg_class

HOW TO FIND OUT A TABLE IS UNLOGGED

- Step 1 : SELECT * FROM pg_class;
- Step 2 : find table target
- Step 3: if relpersistence is U

Means table is unlogged

1

```
postgres=# CREATE UNLOGGED TABLE names(firstname varchar(255));
CREATE TABLE
postgres=# \d
List of relations
-[ RECORD 1 ]----
Schema | public
Name   | names
Type   | table
Owner  | postgres
```

2

```
postgres=# SELECT * FROM pg_class;
oid                 | 16429
relname             | names
relnamespace        | 2200
reltype             | 16431
reloftype           | 0
relowner            | 10
relam               | 2
relfilenode         | 16429
reltablespace       | 0
relpages            | 0
reltuples           | -1
relallvisible       | 0
reltoastrelid       | 0
relhasindex         | f
relisshared         | f
relpersistence      | u
relkind             | r
relnatts            | 1
relchecks           | 0
relhasrules         | f
relhastriggers      | f
relhassubclass      | f
relrowsecurity      | f
relforcerowsecurity | f
relispopulated      | t
relreplident        | d
relispartition      | f
relrewrite          | 0
relfrozenxid        | 785
relminmxid          | 1
relacl              |
reloptions          |
relpartbound        |
```

TEMPORARY TABLES

- As the name of those tables says they are temporary
- **1- if our connection to database getting interrupt the tables will be deleted automatically**
- **1.5 – if another user or role connect to database can`t see these types of tables**
- 2- for creating temporary tables using following command:
- CREATE TEMP/TEMPORARY TABLE name_of_table(datatypes_name)

HOW TO FIND OUT A TABLE IS TEMPORARY

- There is 2 ways to find out
- 1- using \dt when expanded show is off (\x)
- As we can see it`s pg_temp_3
- 2-using SELECT * FROM pg_class;
- Next slide is the image of and explanations

```
postgres=# CREATE TEMP TABLE test(email varchar(255));
CREATE TABLE
postgres=# \x
Expanded display is off.
postgres=# \dt
          List of relations
  Schema   | Name  | Type  |  Owner
-----------+-------+-------+----------
 pg_temp_3 | test  | table | postgres
 public    | names | table | postgres
 public    | users | table | postgres
(3 rows)
```

HOW TO FIND OUT A TABLE IS TEMPORARY

1- find the target table
2- if relpersistence is (t) that means the table is temproray

```
SELECT * FROM pg_class;
oid                 | 16434
relname             | test
relnamespace        | 16432
reltype             | 16436
reloftype           | 0
relowner            | 10
relam               | 2
relfilenode         | 16434
reltablespace       | 0
relpages            | 0
reltuples           | -1
relallvisible       | 0
reltoastrelid       | 0
relhasindex         | f
relisshared         | f
relpersistence      | t
relkind             | r
relnatts            | 1
relchecks           | 0
relhasrules         | f
relhastriggers      | f
relhassubclass      | f
relrowsecurity      | f
relforcerowsecurity | f
relispopulated      | t
relreplident        | d
relispartition      | f
relrewrite          | 0
relfrozenxid        | 786
relminmxid          | 1
relacl              |
reloptions          |
relpartbound        |
```

WHEN WE CREATE TEMPORARY TABLES

- 1- we change options and want to see what is the effects on tables
- 2- the information on tables are tests (for testing tables or informations)
- 3- temporary tables has more speed than usual tables
- How to delete temporary tables
- 1- DROP TABLE
- 2- disconnecting from database

FUNCTIONS

- We can define some operation in function and when we run function the operation will be execute
- Can create function in postgresql by SQL,PYTHON, JAVA OR etc.
- But the best option is sql and plpgsql → plpgsql is postgresql language
- At first we learn sql then plpgsql
- We using datatypes in functions so need to know them by following link you can learn postgresql datatypes:
- https://www.postgresql.org/docs/current/datatype.html

FUNCTION

- Can create a function by CREATE FUNCTION command

If I t has arguement

Our function name

1

```
CREATE FUNCTION name( )
RETRUN datatype
AS $$


$$
```

Has to write what datatype this function going to return

syntax

Betwwen $$ is the body of our function

The language that we write this function

2

```
CREATE FUNCTION name( )
RETRUN datatype
language sql
AS $$
    select "hello world";

$$;
```

(;) at the end of operation and $$

For printing hello world

Need tab in body part of function

FUNCTION SYNTAX EXPLANATION BY SQL

Step 1 : using CREATE FUNCTION name(argument)
Step 2 : write what datatype is going to return by this function
Step 3 : define the language which we writing this function by that
Step 4 : AS $$ → it`s a syntax
Step 5 : TAB !
Step 6 : code the operation in sql language SELECT = PRINT (in python)
Step 7: use (;) at the end of operation
Step 8 : $$ → it`s a syntax
Step 9: use (;) at the end of function after $$

```
CREATE FUNCTION name( )
RETRUN datatype
language sql
AS $$
    select "hello world";

$$;
```

FUNCTION SYNTAX

```
learning=# CREATE FUNCTION hello( )
learning-# RETURNS text
learning-# LANGUAGE sql
learning-# as $$
learning$#      SELECT 'hello world';
learning$# $$;
```

Notes about wirting code without error !!!!!!!
Step 1 : sql commands has to be capital (RETURNS , LANGUAGE , SELECT)
Step 2 : use RETURNS
Step 3 : don’t use “ “ it mean`s column use ‘ ‘ instead !

HOW TO SEE FUNCTION THAT WE MADE

```
learning=# \df
                              List of functions
 Schema | Name  | Result data type | Argument data types | Type
--------+-------+------------------+---------------------+------
 public | hello | text             |                     | func
(1 row)
```

By using \df command

HOW TO RUN FUNCTION

- By using following command :
- SELECT function_name(arg);

```
learning=# SELECT hello()
learning-# ;
    hello
-------------
 hello world
(1 row)


learning=#
```

Means the function is runing

CREATE OR REPLACE FUNCTION

- If a function has exist and we want to change (update it) we have to us CREATE OR REPLACE FUNCTION command If we don`t use OR REPLACE we will get error which is : hello() function is exist

The correct syntax

```
learning=# CREATE OR REPLACE FUNCTION hello()
learning-# RETURNS text
learning-# LANGUAGE sql
learning-# as $$
learning$#       SELECT 'hello world';
learning$# SELECT 'HI';
learning$# $$;
```

error

```
learning=# CREATE FUNCTION hello()
learning-# RETURNS text
learning-# LANGUAGE sql
learning-# as $$
learning$#       SELECT 'hello world';
learning$#     SELECT 'HI';
learning$# $$;
ERROR:  function "hello" already exists with same argument types
learning=#
```

RESULT

```
learning=# \df
                              List of functions
 Schema | Name  | Result data type | Argument data types | Type
--------+-------+------------------+---------------------+------
 public | hello | text             |                     | func
(1 row)


learning=# SELECT hello();
 hello
-------
 HI
(1 row)
```

Taaaadaaaa!

HOW TO DELETE FUNCTION

- By using DROP FUNCTION name_of_function() we can delete the function

```
learning=# DROP FUNCTION hello();
DROP FUNCTION
learning=# \df
                              List of functions
 Schema | Name | Result data type | Argument data types | Type
--------+------+------------------+---------------------+------
(0 rows)
```

ANOTHER FUNCTION

- In this function we want to add 2 integers so pay attention to agrs

And by \df we see our function this function need 2 args in next silde I will explain how to give this function args

```
learning=# SELECT add();
ERROR:  function add() does not exist
LINE 1: SELECT add();
               ^
HINT:  No function matches the given name and argument types. You might need to add explicit type casts.
```

```
learning=# CREATE OR REPLACE FUNCTION addnum(a integer, b integer)
learning-# RETURNS integer
learning-# LANGUAGE sql
learning-# AS $$
learning$#     SELECT a + b;
learning$# $$;
CREATE FUNCTION
learning=# \DF
invalid command \DF
Try \? for help.
learning=# \df
                                  List of functions
 Schema |  Name  | Result data type | Argument data types  | Type
--------+--------+------------------+----------------------+------
 public | addnum | integer          | a integer, b integer | func
(1 row)
```

GIVE ARGS TO FUNCTION IN SQL

- By using select command like this
- SELECT name_of_function (a,b,c …)

```
learning=# SELECT addnum(4, 5);
 addnum
--------
      9
(1 row)
```

FUNCTION FOR READING TABLE

- The first line is the same : CREATE OR REPLACE FUNCTION name()
- The second line is : RETURNS SETOF name_of_table (SETOF means that a part of that table)
- The third line is the same
- AS $$
  - SELECT * FROM name_of_table → using SELECT * FROM … for reading tables

  $$;

IN CODE

```
learning=# CREATE OR REPLACE FUNCTION read_users()
learning-# RETURNS SETOF users
learning-# LANGUAGE sql
learning-# AS $$
learning$#     SELECT * FROM users;
learning$# $$;
CREATE FUNCTION
```

By the way I created a database and call it learning

And run it by following command:
SELECT read_users();

```
learning=# SELECT read_users();
 read_users
------------
 (0,)
 (1,)
 (2,)
 (,2)
(4 rows)
```

NOTE

Because the function is SETOF we can use SELECT * FROM name_of_function() to gives us a full information about table

Can use column instead of * for example:

```
learning=# SELECT email FROM read_users();
 email
-------



 2
(4 rows
```

```
learning=# SELECT * FROM read_users();
 id | email
----+-------
  0 |
  1 |
  2 |
    | 2
(4 rows)
```

NOTE

- 1-If the function is complicated we don`t use sql , using plpgsql or python, java or etc… using plpgsql is recommended.
- 2- as u can see it shows us SETOF users as return

```
learning=# \df
                                List of functions
 Schema |    Name    | Result data type | Argument data types  | Type
--------+------------+------------------+----------------------+------
 public | add        | integer          | a integer, b integer | func
 public | addnum     | integer          | a integer, b integer | func
 public | read_users | SETOF users      |                      | func
(3 rows)
```

PGPLSQL

- PI → procedural languge
- For knowing all command read the following link
- https://www.postgresql.org/docs/current/plpgsql.html
- In sql we can`t write loop or etc. then we use pgplsql
- Recommend that code in vscode then copy and paste it on sql shell
- Plpgsql → procedural language postgres sql

SYNTAX

Opposite of sql language the body of function is start by BEGIN and finish by END
And language is plpgsql not sql
The other things are the same as sql

The body of the function

```
CRAETE OR REPLACE FUNCTION sum_num(n integer, m integer)
RETURNS integer
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN m + n ;
END
$$;
```

I wrote CREATE wrong in this picture….

SAME AS SQL

- Showing and run the function is same as sql

```
postgres=# \df
                                List of functions
 Schema |  Name   | Result data type | Argument data types  | Type
--------+---------+------------------+----------------------+------
 public | sum_num | integer          | n integer, m integer | func
(1 row)


postgres=# SELECT sum_num (4, 5);
 sum_num
---------
       9
(1 row)
```

NOTE

- If we want to define the arguments we use to angle bracket which means

=> This is angle bracket and we define args by this in plpgsql language

```
postgres=# SELECT sum_num(n=>4, m=>5);
 sum_num
---------
       9
(1 row)


postgres=#
```

NOTE

- We fill arguments blank in writing function but we have to write the datatypes and use $ in the body of function

Notice that on body part write RETURN not RETURNS!

No argument define just datatype

Means first argument

Means the second argument

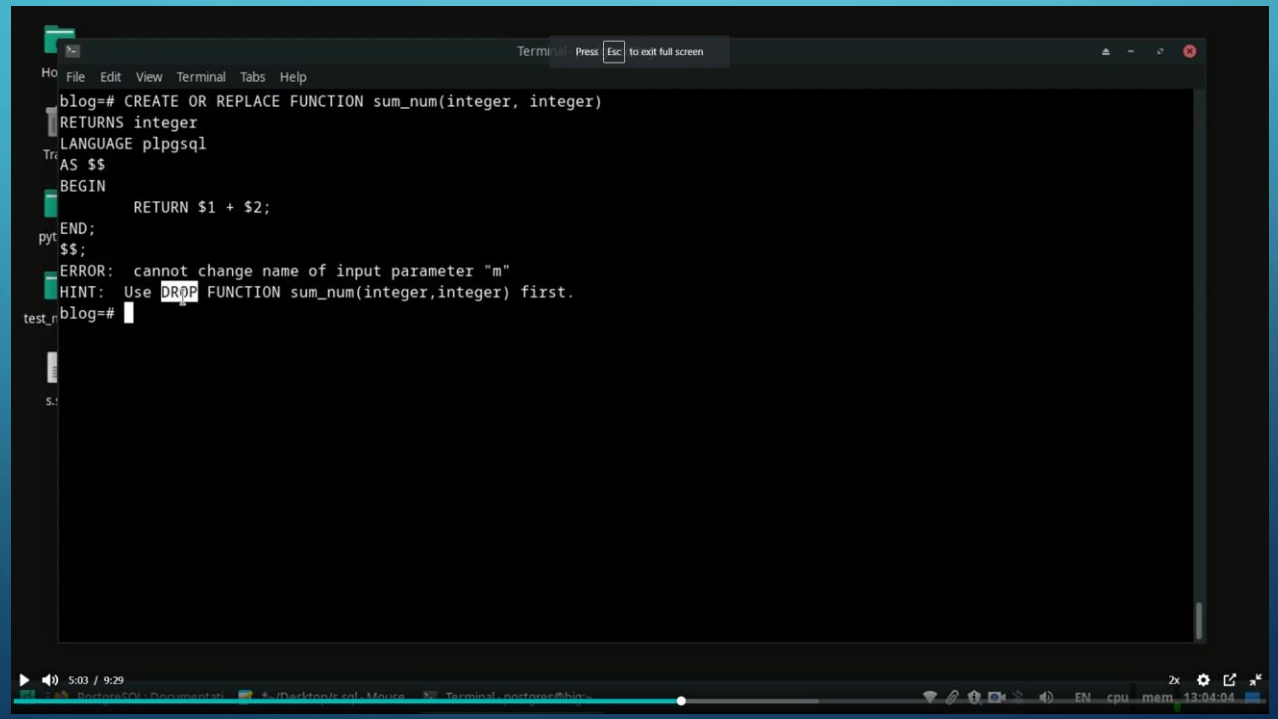
```
CREATE OR REPLACE FUNCTION sum_num(integer, integer)
RETURNS integer
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN $1 + $2 ;
END
$$;
```

NOTE

- Define arguments in body part has to be like this
- $1 + $2 + $3 + $4 + $5 + $6 + $7 …..
- $1 = first arg, $2 = second arg , $3 = third arg and etc.

CAN`T REPLACE

- If we change parameters or arguments we can`t replace function first of all we have to delete the function that we created then build a new function



RESULT

```
CREATE OR REPLACE FUNCTION sum_num(integer, integer)
RETURNS integer
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN $1 + $2 ;
END
$$;
```

FINDING BY MISTAKE

Can define 1 argument and get sum from that by $1 + $1

```
CREATE OR REPLACE FUNCTION sum_numm(integer)
RETURNS integer
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN $1 + $1 ;
END
$$;
```

```
postgres=# SELECT sum_numm(8);
 sum_numm
----------
       16
(1 row)
```

DEFINE VARIABLE INSIDE FUNCTION

- By using DECLARE we can define a variable into a function like this
- And there is specific syntax for it (writing datatype and := are important)

DECLARE (SYNTAX)
Define
variable
and
datatype

(:=) = (=)

```
CREATE OR REPLACE FUNCTION sum_num(integer, integer)
RETURNS integer
LANGUAGE plpgsql
AS $$
DECLARE
    x integer;
BEGIN
    x:= $1 + $2 ;
    RETURN x;
END
$$;
```

RESULT

```
postgres=# CREATE OR REPLACE FUNCTION sum_num(integer, integer)
postgres-# RETURNS integer
postgres-# LANGUAGE plpgsql
postgres-# AS $$
postgres$# DECLARE
postgres$# x integer;
postgres$# BEGIN
postgres$# x:= $1 + $2 ;
postgres$#     RETURN x;
postgres$# END
postgres$# $$;
CREATE FUNCTION
postgres=# SELECT sum_num(4, 5);
 sum_num
---------
       9
(1 row)


postgres=#
```

CREATE FUNCTION FOR CHANGING TABLES

If function don`t return anything use **VOID**

Using sql command to insert user into table

```
CREATE OR REPLACE FUNCTION add_user()
RETURNS VOID
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO users(id, email) VALUES(7, 'niklaus0021@gmail.com');
END;
$$;
```

BUG

- Imagine that you have table by 2 values (id, username)
- If into function you write (id, email) it won`t work!
- Your parameter has to be the same

```
postgres=# SELECT add_user()
postgres-# ;
ERROR:  column "email" of relation "users" does not exist
LINE 1: INSERT INTO users(id, email) VALUES(7, 'niklaus0021@gmail.co...
                          ^
QUERY:  INSERT INTO users(id, email) VALUES(7, 'niklaus0021@gmail.com')
CONTEXT:  PL/pgSQL function add_user() line 3 at SQL statement
postgres=#
```

DEBUG

I rebuild the function changing email parameter to username

```
CREATE OR REPLACE FUNCTION add_user()
RETURNS VOID
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO users(id, username) VALUES(7, 'niklaus0021@gmail.com');
END;
$$;
```

```
postgres=# SELECT add_user();
 add_user
----------

(1 row)

postgres=# SELECT * FROM users
postgres-# ;
 id |       username
----+-----------------------
  7 | niklaus0021@gmail.com
(1 row)
```

OLD - NEW

- These are 2 variables in postgresql
- We use old – new when we want to change a record inside the database
- Imagine that we have table and 2 records by old-new it`s like :

When we update a record the old record saves on old
And the new record saves on new
Notice that every column insert to old and all updated column goes to new

OLD - NEW

- Imagine that we create new record now the old variable gonna be blank and new is fill by new record

RULE

- Using rule is for sending signals for example if in our table x happened y have to happened in another place
- For better understanding:
- We can insert to table by this command

```
postgres=# INSERT INTO users(id, username) VALUES(8, 'mmdhsn@gmail.com');
INSERT 0 1
postgres=# SELECT * FROM users;
 id |       username
----+-----------------------
  7 | niklaus0021@gmail.com
  8 | mmdhsn@gmail.com
(2 rows)
```

now we set rule for it for example if

Username starts with a(email starts with a) do something or save it somewhere else

RULE

- Now , NEW variable equals to our new record in table: id = 8 email = mmdhsn@gmail.com
- Now for we set rule for a_users → means that we want to filter users who starts emails with a and set rule for save it into a users if user`s email starts with a

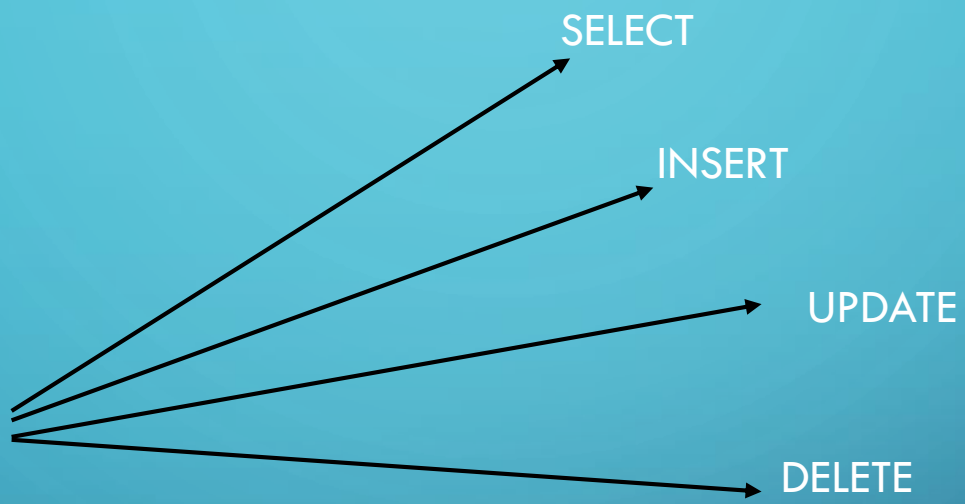
RULE

- First we create a table which called a_users
- Second we have to set our role at **user table** . The rule will copy details on a_users
- **Syntax:**

CREATE OR REPLACE RULE name AS ON event TO table_name WHERE condition DO {ALSO/INSTADE} {NOTHING/ COMMAND}

EVENT

- We have 4 event



IN THIS TASK we want to set rule on INSERT event. So syntax have to be like this
CRAETE OR REPLACE RULE name AS ON INSERT

WHERE

- By using WHERE syntax we can filter or make condition for a command
- For example we can filter the search by using where

DO ALSO/ INSTEAD

- DO means do! When we use also which means do the command in users table and a_users table means do the both commands
- But instead means ignore users table command and do the command that I write

INSTEAD → NOTHING

- It`s simple and easy it means do nothing !
- 1- instead which means ignore the main command (the command that wrote on users)
- 2- and now do nothing ! Easy!

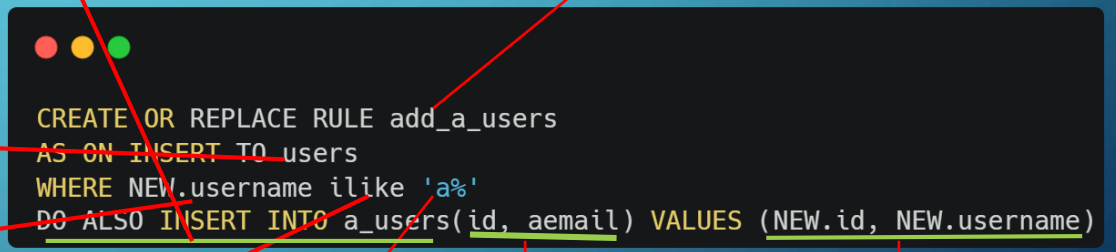
INSTEAD → COMMAND

- Which means ignore the main command and do what I command you right now

Do also command… the command here is INSERT TO

CODE!

- The syntax and code is hard a little bit but I explain everything here



Rule name

The main table

Remember new ? Everything adds on table and it`s new it goes into new variable

an sql command for finding character

Starts with a

Column that we created on a_users (new table)

Find values from new.id and new.username column from users table(target table)

EXPLANATION

```
CREATE OR REPLACE RULE add_a_users
AS ON INSERT TO users
WHERE NEW.username ilike 'a%'
DO ALSO INSERT INTO a_users(id, aemail) VALUES (NEW.id, NEW.username)
```

1- where NEW.username → which means where is filtering for searching (as base) so we filter the search in new usernames and the usernames starts with a
2- NEW.username → NEW.username is the new email that client write in database when a new record saves it goes to NEW variable and username is one of our column in users table so NEW.username means the new record on users table at username column.
3- ilike → using for searching character into string
4- 'a%' → means start with a and after a isn`t important (the % is for showing what character is important for us) and if we write this code like this '%a' that means ends with a
5-DO ALSO INSERT INTO a_user(id, aemail) → it`s a simple command that says insert into our_table(our_column)
6- VALUES(NEW.id, NEW.username) → it means where I can find values to insert into a_users(id, aemail)? We say: from (NEW.id, NEW.username) which means the newest record (NEW variable) from our column in users (main table)

CHECK RULES

- Checking rules is possible by following command: \d name_of_table

```
postgres=# \d users
                    Table "public.users"
  Column  |          Type          | Collation | Nullable | Default
----------+------------------------+-----------+----------+---------
 id       | integer                |           | not null |
 username | character varying(100) |           |          |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
Rules:
    add_a_users AS
    ON INSERT TO users
   WHERE new.username::text ~~* 'a%'::text DO  INSERT INTO a_users (id, aemail)
  VALUES (new.id, new.username)
```

We can see rules at the end of this command`s answer

CHECK RULES SECOND WAY

- By sql command and reading pg_rules

```
postgres=# SELECT * FROM pg_rules;
-[ RECORD 1 ]-----------------------------------------------------------------------------------------
schemaname | pg_catalog
tablename  | pg_settings
rulename   | pg_settings_u
definition | CREATE RULE pg_settings_u AS
    +
           |     ON UPDATE TO pg_catalog.pg_settings
     +
           |    WHERE (new.name = old.name) DO  SELECT set_config(old.name, new.setting, false) AS
set_config;
-[ RECORD 2 ]-----------------------------------------------------------------------------------------
schemaname | pg_catalog
tablename  | pg_settings
rulename   | pg_settings_n
definition | CREATE RULE pg_settings_n AS
    +
           |     ON UPDATE TO pg_catalog.pg_settings DO INSTEAD NOTHING;
-[ RECORD 3 ]-----------------------------------------------------------------------------------------
schemaname | public
tablename  | users
rulename   | add_a_users
definition | CREATE RULE add_a_users AS
    +
           |     ON INSERT TO public.users
     +
           |    WHERE ((new.username)::text ~~* 'a%'::text) DO  INSERT INTO a_users (id, aemail)
     +
           |   VALUES (new.id, new.username);
```

RESULT AND TESTING

```
postgres=# INSERT INTO users(id, username) VALUES(4, 'aghapedgam@gmail.com');
INSERT 0 1
postgres=# SELECT * FROM users
postgres-# ;
-[ RECORD 1 ]-------------------
id       | 7
username | niklaus0021@gmail.com
-[ RECORD 2 ]-------------------
id       | 8
username | mmdhsn@gmail.com
-[ RECORD 3 ]-------------------
id       | 4
username | aghapedgam@gmail.com


postgres=# SELECT * FROM a_users;
-[ RECORD 1 ]----------------
id     | 4
aemail | aghapedgam@gmail.com
```

WHAT IF WE USE INSTEAD?

Pay attention to insert answer

```
postgres=# INSERT INTO users(id, username) VALUES(5, 'anna@gmail.com');
INSERT 0 0
```

INSERT 0 0 → insert nothing in user if you remember when we add user the answer was INSERT 0 1

```
postgres=# CREATE OR REPLACE RULE add_a_users
postgres-# AS ON INSERT TO users
postgres-# WHERE NEW.username ilike 'a%'
postgres-# DO INSTEAD INSERT INTO a_users(id, aemail) VALUES (NEW.id, NEW.username);
CREATE RULE
```

```
postgres=# SELECT * FROM users;
-[ RECORD 1 ]-------------------
id       | 7
username | niklaus0021@gmail.com
-[ RECORD 2 ]-------------------
id       | 8
username | mmdhsn@gmail.com
-[ RECORD 3 ]-------------------
id       | 4
username | aghapedgam@gmail.com

postgres=# SELECT * FROM a_users;
-[ RECORD 1 ]----------------
id     | 4
aemail | aghapedgam@gmail.com
-[ RECORD 2 ]----------------
id     | 5
aemail | anna@gmail.com
```

In users there no anna@gmail.com but in a_users we can see the id and user were added because of using INSTEAD command

WHAT IF WE USE INSTEAD NOTHING

Means that if user email starts with a nothing happened. (no need to see the result test it.)

syntax

```
CREATE OR REPLACE RULE add_a_users
AS ON INSERT TO users
WHERE NEW.username ilike 'a%'
DO INSTEAD NOTHING
```

WHAT IF WE USE ALSO NOTHING

It has irrational problem ! It`s like writing no rule !

HOW TO REMOVE RULE

- By following command : DROP RULE rule_name ON table_name

```
postgres=# DROP RULE add_a_users ON users;
DROP RULE
```

TRIGGER

- Triggers are same as rules but more advance and you can do something more than rules but in the other hand first you have to give it a function and trigger active the function and function will do the operation that we want.
- For trigger we have to give it function first
- We don`t write codes in trigger we give it a function

IF THEN

- IF Boolean expression THEN

  statements

END IF;

And I think IF THEN is so clear but I will give you more example later

STRING FUNCTION AND OPERATORS

- This is full list of string functions https://www.postgresql.org/docs/16/functions-string.html
- You can read the following link and learn all of string function like upper, lower and etc.
- But now we need substring

SUBSTRING

- Substring is like for in python but no to much it`s for splitting strings.
- Syntax: (the syntax is lower)
- substring(starting text(from start integer)(for count integer) → text
- Example:
- Substring('Thomas' from 2 for 3) → hom
- From 2 means start from second character and for 3 means resume it till resumed characters reach 3

TRIGGER FUNCTION

- We want to create trigger like the rule that we made
- 1- we need to create a function by trigger method!
- The diffrents are:
- 1- LANGUAGE at bottom of the code
- 2- RETURNS trigger

```
CREATE OR REPLACE FUNCTION add_a_user()
RETURNS trigger
AS $$
BEGIN
    IF lower(substring(NEW.username FROM 1 FOR 1)) = 'a' THEN
    INSERT INTO a_users(id, aemail) VALUES (NEW.id, NEW.username);
    END IF;
    RETURN NEW;
END;
$$
LANGUAGE plpgsql;
```

EXPLANATION

lower make the input lower (same as python) using substring for split username for finding first character

This is our statement and it`s sql language

End of if statement

Return the variable to insert it into a_users

```
CREATE OR REPLACE FUNCTION add_a_user()
RETURNS trigger
AS $$
BEGIN
    IF lower(substring(NEW.username FROM 1 FOR 1)) = 'a' THEN
    INSERT INTO a_users(id, aemail) VALUES (NEW.id, NEW.username);
    END IF;
    RETURN NEW;
END;
$$
LANGUAGE plpgsql;
```

CREATE A TRIGGER

- Syntax:
- CREATE [OR REPLACE] TRIGGER trigger_name [BEFORE/AFTER/INSTEAD OF] [event] ON table_name FOR EACH [ROW/STATEMENT] EXECUTE [FUNCTION/PROCEDURE] function_name
- BEFORE/AFTER /INSTEAD : BEFORE means before the main table operation execute, AFTER means after main table operation execute and INSTEAD means ignore the main table operation execute

FOR EACH ROW/ STATEMENT

- ROW means that we want to execute our trigger on each ROW
- And statement means we want to execute our trigger when we use statement

EXECUTE FUNCTION/PROCEDURE

- As we said plpgsql is a procedure postgresql language so we have to choose procedure

CREATE A TRIGGER

```
CREATE TRIGGER add_a_user BEFORE INSERT ON users FOR EACH ROW
EXECUTE PROCEDURE add_a_user();
```

By following the syntax we create a trigger

HOW TO SEE TRIGGERS

- By using \d name_of_table

```
postgres=# \d users
                        Table "public.users"
  Column  |          Type          | Collation | Nullable | Default
----------+------------------------+-----------+----------+---------
 id       | integer                |           | not null |
 username | character varying(100) |           |          |
Indexes:
    "users_pkey" PRIMARY KEY, btree (id)
Triggers:
    add_a_user BEFORE INSERT ON users FOR EACH ROW EXECUTE FUNCTION add_a_user()
```

RESULT

First we create a user by Leyla@gmail.com and as we can see There is no such an id in a_users

But Aamir@gmail.com starts with a and our trigger works correctly

1

```
postgres=# INSERT INTO users(id, username) VALUES(5, 'leyla@gmail.com');
INSERT 0 1
postgres=# SELECT * FROM users;
-[ RECORD 1 ]--------------------
id       | 7
username | niklaus0021@gmail.com
-[ RECORD 2 ]--------------------
id       | 8
username | mmdhsn@gmail.com
-[ RECORD 3 ]--------------------
id       | 4
username | aghapedgam@gmail.com
-[ RECORD 4 ]--------------------
id       | 5
username | leyla@gmail.com


postgres=# SELECT * FROM a_users;
-[ RECORD 1 ]----------------
id     | 4
aemail | aghapedgam@gmail.com
-[ RECORD 2 ]----------------
id     | 5
aemail | anna@gmail.com
```

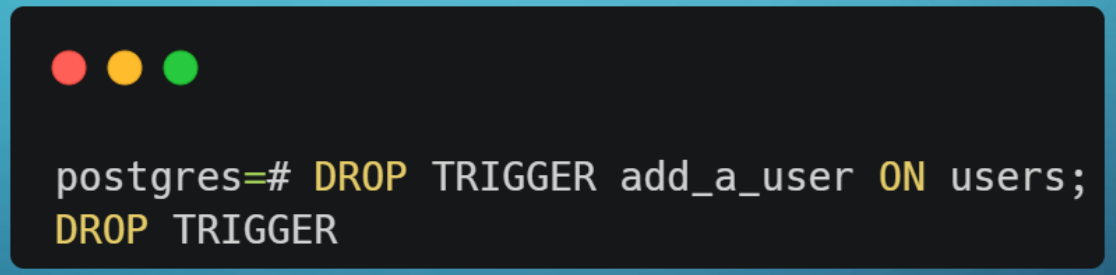
2

```
postgres=# INSERT INTO users(id, username) VALUES(6, 'Aamir@gmail.com');
INSERT 0 1
postgres=# SELECT * FROM users;
-[ RECORD 1 ]--------------------
id       | 7
username | niklaus0021@gmail.com
-[ RECORD 2 ]--------------------
id       | 8
username | mmdhsn@gmail.com
-[ RECORD 3 ]--------------------
id       | 4
username | aghapedgam@gmail.com
-[ RECORD 4 ]--------------------
id       | 5
username | leyla@gmail.com
-[ RECORD 5 ]--------------------
id       | 6
username | Aamir@gmail.com


postgres=# SELECT * FROM a_users;
-[ RECORD 1 ]----------------
id     | 4
aemail | aghapedgam@gmail.com
-[ RECORD 2 ]----------------
id     | 5
aemail | anna@gmail.com
-[ RECORD 3 ]----------------
id     | 6
aemail | Aamir@gmail.com
```

HOW TO DELETE TRIGGER

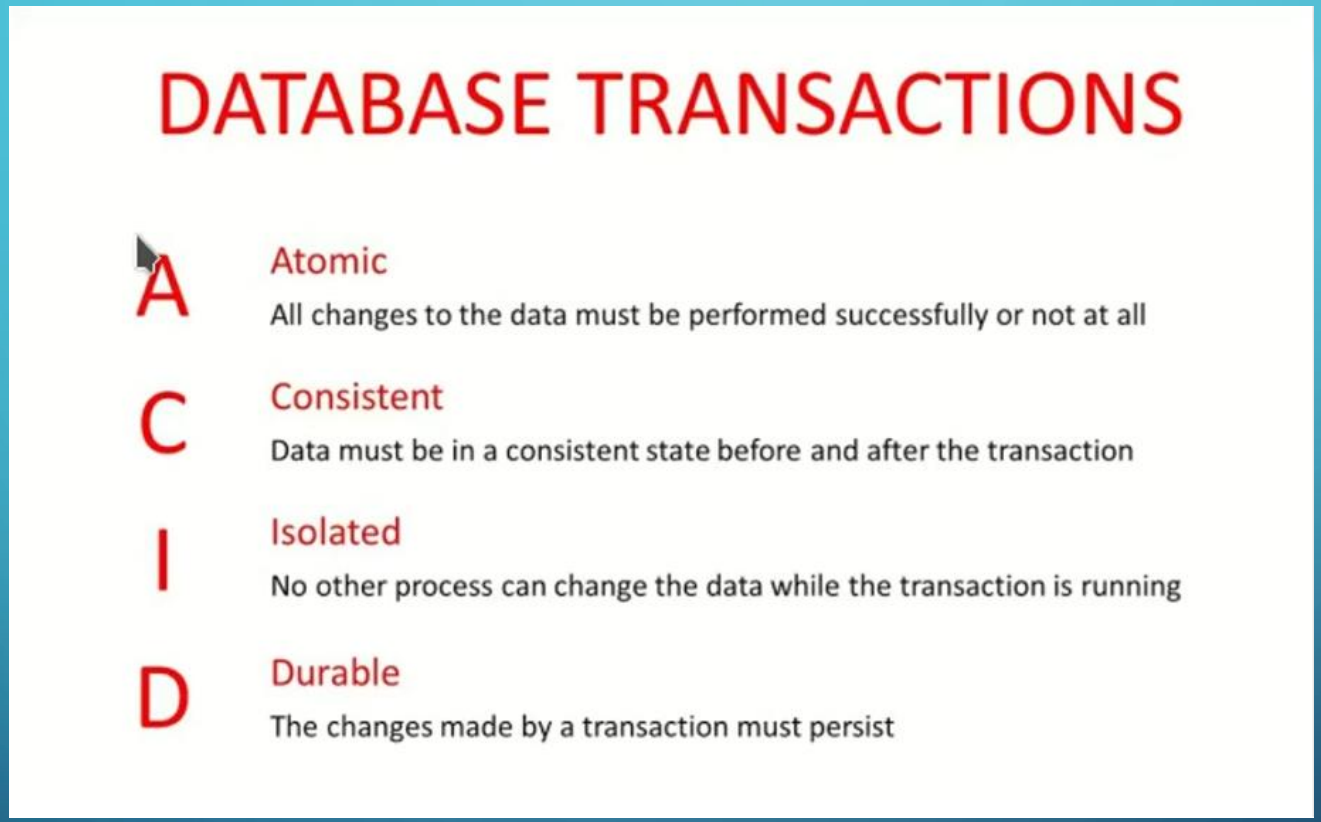
- By using following command
- DROP TRIGGER trigger_name ON table_name;



TRANSACTION

- A database transaction is single unit of work that consist of one or more operation
- When we merge 4 command in 1 transaction means that these commands and operations are related

ACID



4 rules of database transaction which called ACID

A: all command must be successful or failure it`s has to be 4/4 or 0/4

C: Transaction should not ruin or interrupt our data

i: it`s like single task function in python

D: means that should fully changes the data or don`t work on data → 0/4 or 4/4

NOTE

- All commands in postgresql are using transaction like DROP DATABASE or CREATE TABLE or etc.
- But we can manage our transaction too for manage the transaction using following **syntax:**
- **BEGIN;**

COMMAND

CONDITION

COMMIT;

WHY WE USE TRANSACTION

- Imagine that wanna withdraw money from our bank account and transfer it into another account this is 2 actions that has to be fully successful if 1 of this 2 action failed the whole thing gonna run it wrong so that`s why we use transaction to manage our transaction
- If a transaction won`t run completely postgresql gonna return it to the first update and won`t change anything

REMINDER

```
CREATE TABLE name_of_table (COLUMN) VALUES (DATATYPE)
```

XMIN

- XMIN is the last transaction number for changing details which means the last time that postgresql changes information is XMIN and it has specific number
- **SYNTAX:**

SELECT XMIN, * FROM name_of_table

```
postgres=# INSERT INTO members(id, email, age) VALUES(1, 'niklaus@gmail.com', 24);
INSERT 0 1
postgres=# SELECT * FROM members;
 id |       email       | age
----+-------------------+-----
  1 | niklaus@gmail.com |  24
(1 row)


postgres=# SELECT XMIN, * FROM members;
 xmin | id |       email       | age
------+----+-------------------+-----
  844 |  1 | niklaus@gmail.com |  24
(1 row)
```

XMIN

- XMIN numbers increase automatically one by one which means if we do another transaction the number will be increase

```
postgres=# INSERT INTO members(id, email, age) VALUES(2, 'niklaus99@gmail.com', 26);
INSERT 0 1
postgres=# SELECT XMIN, * FROM members
postgres-# ;
 xmin | id |        email        | age
------+----+---------------------+-----
  844 |  1 | niklaus@gmail.com   |  24
  845 |  2 | niklaus99@gmail.com |  26
(2 rows)
```

TRANSACTION

- Now for manage transaction pay attention

We start transaction by BEGIN;
Then INSERT a user into member
Now rationally if we SELECT table we should see the user information on table BUT…

```
postgres=# BEGIN;
BEGIN
postgres=*# INSERT INTO members(id, email, age) VALUES(3, 'niklaus99100@gmail.com', 27);
INSERT 0 1
```

We can`t do it till finishing transaction and…

```
postgres=!# SELECT * FROM members ;
ERROR:  current transaction is aborted, commands ignored until end of transaction block
```

If we open another session postgresql won`t show us the information because the transaction haven`t finished yet

```
postgres=# SELECT * FROM members
postgres-# ;
 id |        email        | age
----+---------------------+-----
  1 | niklaus@gmail.com   |  24
  2 | niklaus99@gmail.com |  26
(2 rows)
```

NOTE

- Open another session means that opens another sql shell and using it, but because the transaction not COMMIT; yet in another session we can`t read the information that we would transaction

TRANSACTION

Remember the ACID . And compare it to this code this transaction following the acid and in the end we COMMIT; it and… done !
Note : now if we open a new session and want to read columns now we can read and we will see the third id

```
postgres=# BEGIN;
BEGIN
postgres=*# INSERT INTO members(id, email, age) VALUES(3, 'niklaus99100@gmail.com', 27);
INSERT 0 1
postgres=*# COMMIT;
COMMIT
postgres=# SELECT * FROM members ;
 id |         email          | age
----+------------------------+-----
  1 | niklaus@gmail.com      |  24
  2 | niklaus99@gmail.com    |  26
  3 | niklaus99100@gmail.com |  27
(3 rows)
```

ROLLBACK

- If we wanna drop transaction and don`t want to COMMIT we use ROLLBACK

```
postgres=# BEGIN;
BEGIN
postgres=*# INSERT INTO members(id, email, age) VALUES(4, 'mail@gmail.com', 84);
INSERT 0 1
postgres=*# ROLLBACK;
ROLLBACK
postgres=# SELECT * FROM members;
 id |         email          | age
----+------------------------+-----
  1 | niklaus@gmail.com      |  24
  2 | niklaus99@gmail.com    |  26
  3 | niklaus99100@gmail.com |  27
(3 rows)
```

COUNT

- For counting informations into table we using count by using count we can see how much record do we have in our table
- Note : count is a built in function

```
postgres=# SELECT count(*) FROM members;
 count
-------
     3
(1 row)
```

AUTO ROLLBACK

- If one of our commands has error postgresql give us a rollback automatically

There is no counteeeeeee function in postgresql and postgresql gives us a rollback

```
postgres=# BEGIN;
BEGIN
postgres=*# SELECT count(*) FROM members;
 count
-------
     3
(1 row)


postgres=*# SELECT counteeeeee(*) FROM members;
ERROR:  function counteeeeee() does not exist
LINE 1: SELECT counteeeeee(*) FROM members;
               ^
HINT:  No function matches the given name and argument types. You might need to add explicit type casts.
postgres=!# COMMIT;
ROLLBACK
```

SAVEPOINT

- We cant have nested transaction by savepoint we can split our transaction and manage that
- **Syntax:**

SAVEPOINT savepoint_name

Savepoints are likes save point games! Imagine that we created 40 commands transaction and the 34th command is wrong we need to have save point at 33th one ! In the other hand we have to rewrite the whole transaction

CRAETE SAVEPOINT

- Savepoints are in the transaction body after each command at transaction we can write a save point
- For closing save point we have 2 option
  - 1- rollback
  - 2- release savepoint

```
BEGIN;
INSERT INTO members(id, email, age) VALUES (4, 'mmd@gmail.com', 35)
SAVEPOINT first_command;
INSERT INTO members(id, email, age) VALUES (5, 'mmd2@gmail.com', 36)
```

savepoint

Second command

ROLLBACK IN SAVEPOINT

- If we use rollback save point the the commands in between savepoints won`t run and getting ignore
- **Syntax:**

```
BEGIN;
INSERT INTO members(id, email, age) VALUES (4, 'mmd@gmail.com', 35)
SAVEPOINT first_command;
INSERT INTO members(id, email, age) VALUES (5, 'mmd2@gmail.com', 36)
ROLLBACK TO SAVEPOINT name_of_savepoint
INSERT INTO members(id, email, age) VALUES (6, 'mmd232@gmail.com', 45)
COMMIT;
```

Getting ignore because it`s between savepoint and rollback

savepoint

rollback

After rollback it`s back to where the savepoint is and ignore the commands between rollback and savepoint

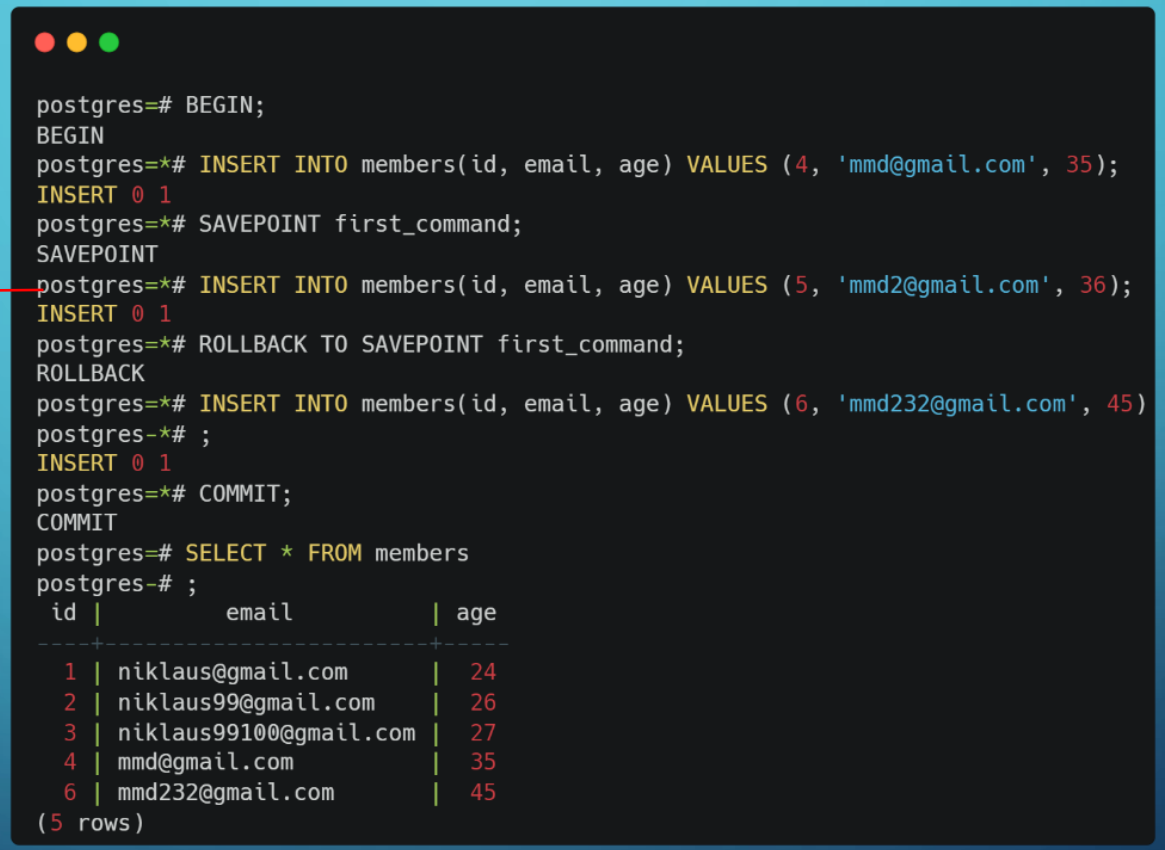
RELEASE SAVEPOINT

```
BEGIN;
INSERT INTO members(id, email, age) VALUES (4, 'mmd@gmail.com', 35);
SAVEPOINT first_command;
INSERT INTO members(id, email, age) VALUES (5, 'mmd2@gmail.com', 36);
RELEASE SAVEPOINT name_of_savepoint;
COMMIT;
```

If using RELEASE SAVEPOINT it`s just release savepoint and run all commands

ROLLBACK TO SAVEPOINT RESULT

ignored



```
postgres=# BEGIN;
BEGIN
postgres=*# INSERT INTO members(id, email, age) VALUES (4, 'mmd@gmail.com', 35);
INSERT 0 1
postgres=*# SAVEPOINT first_command;
SAVEPOINT
postgres=*# INSERT INTO members(id, email, age) VALUES (5, 'mmd2@gmail.com', 36);
INSERT 0 1
postgres=*# ROLLBACK TO SAVEPOINT first_command;
ROLLBACK
postgres=*# INSERT INTO members(id, email, age) VALUES (6, 'mmd232@gmail.com', 45)
postgres-*# ;
INSERT 0 1
postgres=*# COMMIT;
COMMIT
postgres=# SELECT * FROM members
postgres-# ;
 id |         email          | age
----+------------------------+-----
  1 | niklaus@gmail.com      |  24
  2 | niklaus99@gmail.com    |  26
  3 | niklaus99100@gmail.com |  27
  4 | mmd@gmail.com          |  35
  6 | mmd232@gmail.com       |  45
(5 rows)
```

RELEASE SAVEPOINT RESULT

```
postgres=# BEGIN;
BEGIN
postgres=*#
postgres=*# INSERT INTO members(id, email, age) VALUES (4, 'mmd@gmail.com', 35);
INSERT 0 1
postgres=*# SAVEPOINT first_command;
SAVEPOINT
postgres=*# INSERT INTO members(id, email, age) VALUES (5, 'mmd2@gmail.com', 36);
INSERT 0 1
postgres=*# RELEASE SAVEPOINT first_command;
RELEASE
postgres=*# INSERT INTO members(id, email, age) VALUES (6, 'mmd232@gmail.com', 45);
INSERT 0 1
postgres=*#
postgres=*# COMMIT;
COMMIT
postgres=# SELECT * FROM members;
 id |         email          | age
----+------------------------+-----
  1 | niklaus@gmail.com      |  24
  2 | niklaus99@gmail.com    |  26
  3 | niklaus99100@gmail.com |  27
  4 | mmd@gmail.com          |  35
  4 | mmd@gmail.com          |  35
  4 | mmd@gmail.com          |  35
  5 | mmd2@gmail.com         |  36
  6 | mmd232@gmail.com       |  45
(8 rows)
```

DELETE

- By delete command we can delete records from our tables by following syntax:
  - **DELETE FROM name_of_table WHERE column,(or xmin and max) = value_of_record;**
- this is our table

As you can see we have some repeatet records in 2 way i`ll show you how to delete the records

```
postgres=# SELECT * FROM members;
 id |         email          | age
----+------------------------+-----
  1 | niklaus@gmail.com      |  24
  2 | niklaus99@gmail.com    |  26
  3 | niklaus99100@gmail.com |  27
  4 | mmd@gmail.com          |  35
  4 | mmd@gmail.com          |  35
  4 | mmd@gmail.com          |  35
  5 | mmd2@gmail.com         |  36
  6 | mmd232@gmail.com       |  45
(8 rows)
```

DELETE BY COLUMN

But here we have 3 ids of 4 if we delete by id we gonna lost our 3 records so we need to delete records by xmin and max

```
postgres=# DELETE FROM members WHERE id = 6;
DELETE 1
postgres=# SELECT * FROM members;
 id |         email          | age
----+------------------------+-----
  1 | niklaus@gmail.com      |  24
  2 | niklaus99@gmail.com    |  26
  3 | niklaus99100@gmail.com |  27
  4 | mmd@gmail.com          |  35
  4 | mmd@gmail.com          |  35
  4 | mmd@gmail.com          |  35
  5 | mmd2@gmail.com         |  36
(7 rows)
```

DELETE BY XMIN

As u can by using xmin we delete a record.

```
postgres=# SELECT XMIN, * FROM members;
 xmin | id |         email          | age
------+----+------------------------+-----
  844 |  1 | niklaus@gmail.com      |  24
  845 |  2 | niklaus99@gmail.com    |  26
  847 |  3 | niklaus99100@gmail.com |  27
  849 |  4 | mmd@gmail.com          |  35
  852 |  4 | mmd@gmail.com          |  35
  856 |  4 | mmd@gmail.com          |  35
  857 |  5 | mmd2@gmail.com         |  36
(7 rows)


postgres=# DELETE FROM members WHERE XMIN = 852;
DELETE 1
postgres=# SELECT * FROM members;
 id |         email          | age
----+------------------------+-----
  1 | niklaus@gmail.com      |  24
  2 | niklaus99@gmail.com    |  26
  3 | niklaus99100@gmail.com |  27
  4 | mmd@gmail.com          |  35
  4 | mmd@gmail.com          |  35
  5 | mmd2@gmail.com         |  36
(6 rows)
```

DELETE LAST RECORD BY MAX

- If we want to delete last record using following command:
- Syntax:
  - DELETE FROM name_of_table WHERE column = (SELECT MAX(column) FROM name_of_table
- Example:
  - DELETE FROM name_of_table WHERE id = (SELECT MAX(id) FROM name_of_table);
    - Notice that paranthceses are part of the syntax

DELETE LAST RECORD BY MAX

```
postgres=# DELETE FROM members WHERE id = (SELECT MAX(id) FROM members);
DELETE 1
postgres=# SELECT * FROM members;
 id |         email          | age
----+------------------------+-----
  1 | niklaus@gmail.com      |  24
  2 | niklaus99@gmail.com    |  26
  3 | niklaus99100@gmail.com |  27
  4 | mmd@gmail.com          |  35
  4 | mmd@gmail.com          |  35
(5 rows)
```

INDEX

- Indexes are construction which make our database and search more faster
- Index is like list of a book instead of searching a word in book you can search in list in some of conditions we shouldn`t use indexes cause may make the search speed even slower (i`ll say in which conditions.)

SEARCH FILTER

- By using where we can filter our searches

But here we have 5 ids of 4 now imagine that

We have million records and it`s actually

Impossible to find the right record when ids

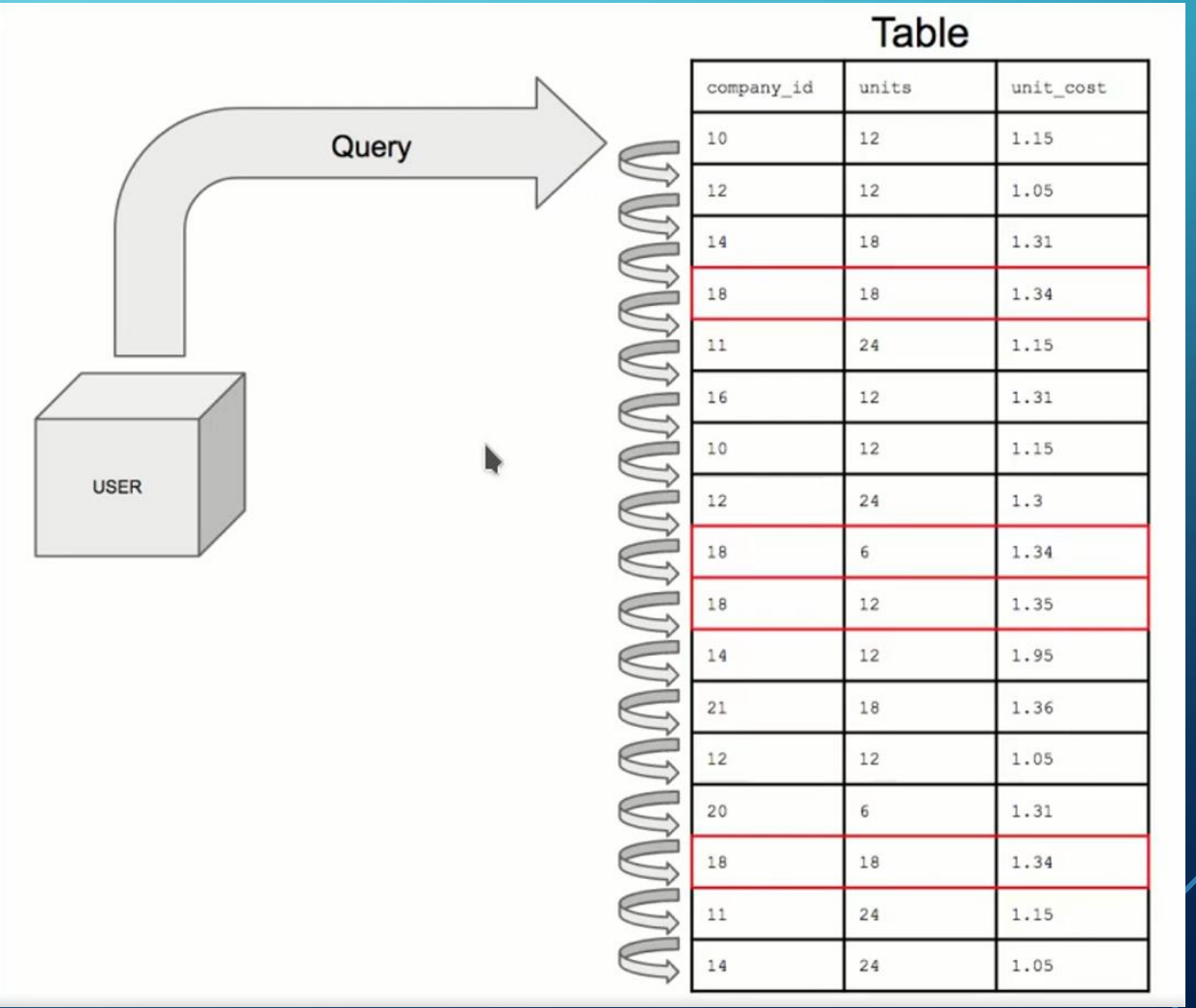
Are the same so we using indexes

```
postgres=# SELECT * FROM members ;
 id |          email           | age
----+--------------------------+-----
  1 | niklaus@gmail.com        |  24
  2 | niklaus99@gmail.com      |  26
  3 | niklaus99100@gmail.com   |  27
  4 | mmd@gmail.com            |  35
  4 | mmd@gmail.com            |  35
  4 | hsn@gmail.com            |  36
  4 | hsn213@gmail.com         |  46
  4 | hsn21453@gmail.com       |  26
(8 rows)


postgres=# SELECT * FROM members WHERE id = 4
postgres-# ;
 id |       email         | age
----+---------------------+-----
  4 | mmd@gmail.com       |  35
  4 | mmd@gmail.com       |  35
  4 | hsn@gmail.com       |  36
  4 | hsn213@gmail.com    |  46
  4 | hsn21453@gmail.com  |  26
(5 rows)
```

WHY WE USE INDEX

Its example like the last slide and if we have for example 100,000 ids of 18 it`s a high loading process even for postgresql. Because postgresql has to serach record by record



WHAT IS INDEX

- When we command postgresql to index the records (information) postgresql sort it by chosen column
- (like sort in python)

COMPANY_ID	UNIT	UNIT_COST
10	12	1.15
10	12	1.15
11	24	1.15
11	24	1.15
12	12	1.05
12	24	1.3
12	12	1.05
14	18	1.31
14	12	1.95
14	24	1.05
16	12	1.31
18	18	1.34
18	6	1.34
18	12	1.35
18	18	1.34
20	6	1.31

HOW INDEX WORKS

- **Index don`t change the main table** cause it`s take a load from postgres so how index works? By copy a table and sort it. So in the first step index make a copy from our table and then sort it by chosen condition. (for example sort it by id)
- In last slide we had 4 ids of 18 so how postgres understand the whole details?

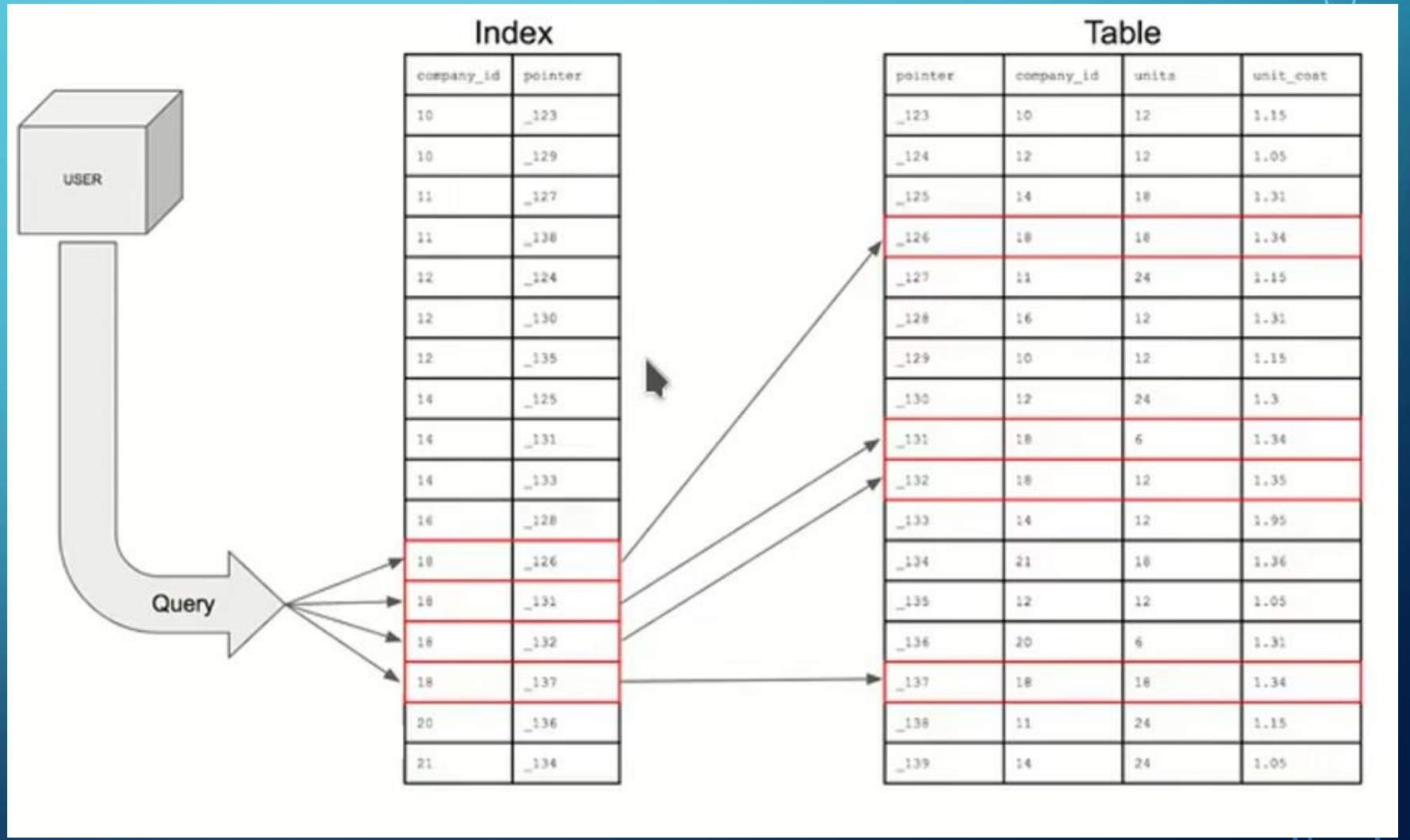
HOW INDEX WORKS

By getting help from pointers , pointers are memory id which point to the information that saved on memory for example: we want 16 company`s id so index using pointer find 16 and details then shows information to us. Or for 18 find the pointer set the information and shows it to us. That why the information comes sorted which means ids and company name are match and information are correct

COMPANY_ID	POINTER
10	_123
10	_129
11	_127
11	_138
12	_124
12	_130
12	_135
14	_125
14	_131
14	_133
16	_128
18	_126
18	_131
18	_132
18	_137
20	_136

HOW INDEX WORKS

User type a query , postgres comes into index and find pointers then show us the records



WHEN WE SHOULDN`T USE INDEX

- 1- when our table is small and information are not to much (we have little information and few records) cause make the search speed slower.
- 2- if we have table and we want to edit it rapidly in this situation we shouldn`t use index
- 3- if we have many NULL(s) in our tables we shouldn`t use indexes cause make the search more slower
  - Note : null is none in python

INDEX TYPES

- Index types are:
  - B-Tree
  - Hash
  - GIST
  - SP-Gist
  - GIN
  - BRIN
- B-Tree has the most usage and the examples in last slides was B-Tree

HOW TO USE INDEX

- First we chose the column or columns that we want to index it, the limitation columns in postgres is 32 columns
- **SYNTAX:**
  - CREATE INDEX name_of_index ON name_of_table(name_of_column);

CREATE INDEX AND SEE IT

```
postgres=# CREATE INDEX name_indexing ON names(name);
CREATE INDEX
postgres=# \d names
                    Table "public.names"
 Column |          Type          | Collation | Nullable | Default
--------+------------------------+-----------+----------+---------
 id     | integer                |           |          |
 name   | character varying(255) |           |          |
Indexes:
    "name_indexing" btree (name)
```

All things are clear in picture

SEARCH BY INDEX

- Searching by index is automatically when we define an index it`s automatically first search in index then using pointer and at last shows us the table

  This is our table and we search for specific name. first postgres using index finding pointers and then shows us information
  But if we search from index postgres search directly from tables

```
postgres=# SELECT * FROM names;
 id |   name
----+----------
  1 | mohammad
  2 | pedram
  3 | shakib
  4 | khashi
  5 | amir
  6 | ashkan
  7 | amir
(7 rows)
postgres=# SELECT * FROM names where name = 'amir';
 id | name
----+------
  5 | amir
  7 | amir
(2 rows)
```

DELETE INDEX

- By using following command
  - DROP INDEX name_of_index;

```
postgres=# DROP INDEX name_indexing;
DROP INDEX
postgres=# \d names
                    Table "public.names"
 Column |          Type          | Collation | Nullable | Default
--------+------------------------+-----------+----------+---------
 id     | integer                |           |          |
 name   | character varying(255) |           |          |
```

LOG

- Log is report of database it has variety information so if you want to know logs completely use following link
  - https://www.postgresql.org/docs/current/runtime-config-logging.html
- There is 2 ways to reading log files
  - 1-after logs we can read it line by lines
  - 2- using automatic tools
- We learn first way in this powerpoint

SAVING LOG FILES

- By default there is no place to save log files so we have to give them a place to save. For saving postgresql logs you need to config it in postgresql.conf
- At :
- Localdisc/porgramfile(or etc.)/postgresql/16/data/postgresql.conf
- Then open cmd and write following command:
  - Type postgresql.conf

SAVING LOG FILES

- Then you can see a part of postgres.conf that write : REPORTING AND LOGGING
- If you see # before the commands which means it`s not active and you have activate it

example

```
#------------------------------------------------------------------------------
# REPORTING AND LOGGING
#------------------------------------------------------------------------------

# - Where to Log -

log_destination = 'stderr'              # Valid values are combinations of
                                        # stderr, csvlog, jsonlog, syslog, and
                                        # eventlog, depending on platform.
                                        # csvlog and jsonlog require
                                        # logging_collector to be on.

# This is used when logging to stderr:
logging_collector = on          # Enable capturing of stderr, jsonlog,
                                        # and csvlog into log files. Required
                                        # to be on for csvlogs and jsonlogs.
                                        # (change requires restart)

# These are only used if logging_collector is on:
#log_directory = 'log'                  # directory where log files are written,
                                        # can be absolute or relative to PGDATA
#log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'        # log file name pattern,
                                        # can include strftime() escapes
```

EDIT POSTGRESQL.CONF

- For edit postgres.conf open cmd wirte your notepad(editor) and then type postgresql.conf
  - Notepad postgresql.conf → this is the command

EDITING POSTGRESQL.CONF

Where does log save?

4 types of showing logs

```
#------------------------------------------------------------------------------
# REPORTING AND LOGGING
#------------------------------------------------------------------------------

# - Where to Log -

log_destination = 'stderr'      # Valid values are combinations of
                                # stderr, csvlog, jsonlog, syslog, and
                                # eventlog, depending on platform.
                                # csvlog and jsonlog require
                                # logging_collector to be on.
```

Stderr → is our terminal, when we set stderr means shows the logs in terminal
Csvlog → means that save log files by csv format
Syslog → in great systems we have some server which their command is saving log they are syslog
Eventlog → is for windows system

ACTIVATE LOG DESTINATION

- For activate log destination just remove (#) before log destination

LOGIN COLLECTOR

- Login collector collect to logs and save it into the file

```
# This is used when logging to stderr:
logging_collector = on      # Enable capturing of stderr, jsonlog,
                  # and csvlog into log files. Required
                  # to be on for csvlogs and jsonlogs.
                  # (change requires restart)
```

For activate it remove (#) before logging collector

LOG DIRECTORY

- Log directory is saving logs in the path that we chose
- If we say absolute means save it in desktop if we say log means that save it on PGDATA
- Logfile name → it`s clear but pay attention to naming it

```
# These are only used if logging_collector is on:
#log_directory = 'log'                  # directory where log files are written,
                                        # can be absolute or relative to PGDATA
#log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'    # log file name pattern,
```

Y = year, m = month, d = day, h = hour, m = minute, s = second

LOGFILE MODE

- Logfile mode means how is the access to log file it`s 0600 or 0640 by default which 0600 means → the only server owner can read and write into log files
- And 0640 means → the owner can read and edit and the group can read only

LOG_TRUNCATE_ON_ROTATION

- Log… rotation uses for split logs when we get logged it could be a huge file by rotation we set that if log file get bigger than 10 mb close the log file and open another one

Means everyday
create a new logfile

What is the
size of logfile

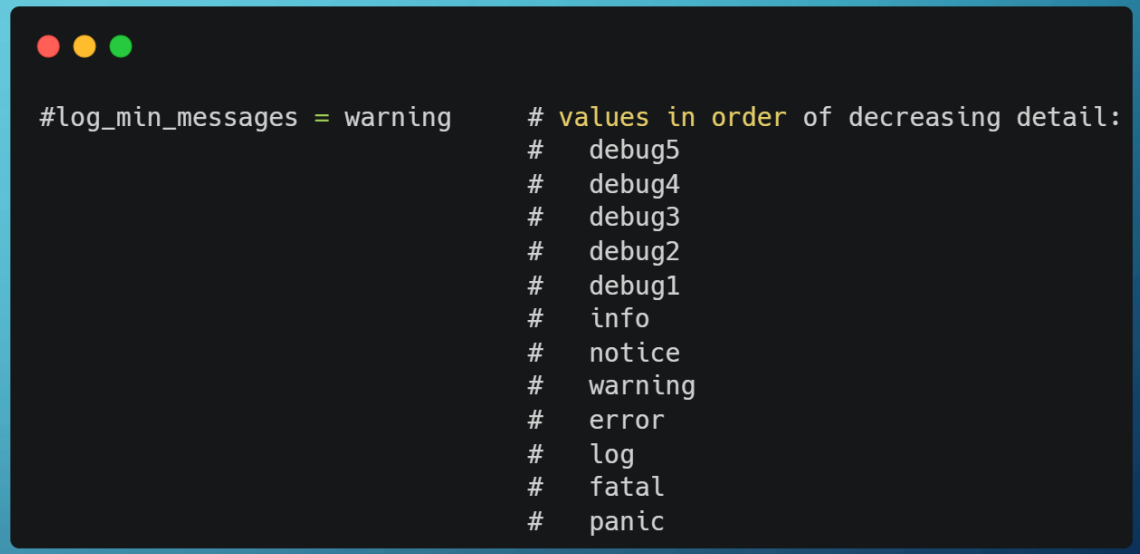
These was where to log setting

```
#log_rotation_age = 1d          # Automatic rotation of logfiles will
                                # happen after that time.  0 disables.
#log_rotation_size = 10MB       # Automatic rotation of logfiles will
                                # happen after that much log output.
                                # 0 disables.
#log_truncate_on_rotation = off     # If on, an existing log file with the
                                # same name as the new log file will be
                                # truncated rather than appended to.
                                # But such truncation only occurs on
                                # time-driven rotation, not on restarts
                                # or size-driven rotation.  Default is
                                # off, meaning append to existing files
                                # in all cases.
```

WHEN TO LOG

Means set the level of log

The top list logs are not dangerous(debug 5 is the safest) but bottom is dangerous (panic is the most dangerous) it sets on warning means logs the warning and the bottom of the logs.



Activate it by removing (#)

log_timezone = 'Asia/Tehran‘
It`s clear !

WHAT TO LOG

- Log_statement → means that which operation should get logged
  - There is for option for statement:
    - None → clear !
    - Ddl → data definition language → means that when we change an schema of table or etc. (changing context droping data base or changing table or etc.)
    - Mod → INSERT , UPDATE the lower important of operations and commands
    - All → clear !

LOG STATEMENT

Remove hash and set it on all

```
log_statement = 'all'                   # none, ddl, mod, all
```

AFTER CHANGES

- After changing we have to restart postgresql server by following command
  - 1- open cmd on administrator
  - 2- navigate it to postgresql → c:\ programfile\postgresql\16\bin
  - 3-pg_ctl.exe restart -D "C:\Program Files\PostgreSQL\16\data" use this command
  - 4-pg_ctl.exe start –D "C:\Program Files\PostgreSQL\16\data" use this command

RESTART THE SERVER

```
C:\Program Files\PostgreSQL\16\bin>pg_ctl.exe restart -D "C:\Program Files\PostgreSQL\16\data"


pg_ctl: old server process (PID: 4192) seems to be gone
starting server anyway
waiting for server to start....2023-11-13 21:41:28.443 +0330 [5760] LOG:  redirecting log output to
logging collector process
2023-11-13 21:41:28.443 +0330 [5760] HINT:  Future log output will appear in directory "log".
 stopped waiting
pg_ctl: could not start server
Examine the log output.

C:\Program Files\PostgreSQL\16\bin>pg_ctl.exe start -D "C:\Program Files\PostgreSQL\16\data"


waiting for server to start....2023-11-13 21:42:14.192 +0330 [9896] LOG:  redirecting log output to
logging collector process
2023-11-13 21:42:14.192 +0330 [9896] HINT:  Future log output will appear in directory "log".
 done
server started
```

READING LOGS

- For reading logs go to this address
- C:\Program Files\PostgreSQL\16\data\log
- And read logs !
- You can read it in cmd by using this command
- Type C:\Program Files\PostgreSQL\16\data\log\log_file_name

Note : creating database and etc… is statement

BUG NOTE

- When I was creating this powerpoint I got the bug and I would like to share it with you, accidently my server shut down and I couldn`t connect it I got this error:
  - connection to server at "localhost" (127.0.0.1), port 5432 failed: Connection refused (0x0000274D/10061) Is the server running on that host and accepting TCP/IP connections?
  - The solution is go to services.mcs find posgresql and start/restart it

BACKUP & RESTORE

- There is 2 ways to taking backup from data base
  - Logical backup
  - Physical backup
- In logical backup postgresql create a client and copy all tables and server and getting backup from that but if information quantities is high other clients gonna be slower if we have a lot of quantities we can use physical backup specially In extra large project

PG_DUMP AND PG_RESTORE

- It`s run on cmd not postgresql and for getting backup use following command:
  - Pg_dump.exe –U username database_name
  - But first you have to go and run cmd in this address : C:\Program Files\PostgreSQL\16\bin
  - Use following command and read notes in next slide :
  - pg_dump.exe -U postgres hlweb > E:backup\posg.sql

IMPORTANT NOTES

- 1- u can`t save backups in postgresql folder at C drive
- 2- ensure that create backup folder in another drive that pg_dump has primission there
- These 2 note takes 30 minutes of my time ☺

RESTORE

- When our file is sql we can use sql shell (postgresql) to restore it but in windows … we can`t ! So use the following orders :
  - 1- open cmd at bin address C:\Program Files\PostgreSQL\16\bin
  - 2- psql.exe -U postgres -d hlweb -f E:\backup\posg.sql
  - 3- in postgresql use \i and address with forward slashe E:/back/posg.sql
  - 4- now need to reconnect to database again (important one)
    - If you don`t do that you can`t see the change and got error for multiple primary keys

FD,FC,FT

- Fc → save backup as binary
- Fd → save backup in directory
- Ft → save backup in tar file (compressed)

AGGREGATION

- Aggregation functions are the functions which take some inputs and gives you 1 output
- For knowing all aggregation functions read the following link
  - https://www.postgresql.org/docs/16/functions-aggregate.html
- Before reading table by select we can use aggregation function and do some operation on records

AVG

- AVG gives us and average of the column that we define
- We have table like this
- Now we want to know the AVG price of our productions
- Use following command:
  - SELECT AVG(column) FROM name_of_table → SELECT AVG(price) FROM production

```
shop=# SELECT * FROM production
shop-# ;
 id | pro_name | price | group_id
----+----------+-------+----------
  1 | htc one  |   400 |
  2 | hp       |   500 |
  3 | lenovo   |  1500 |
  4 | samsung  | 21500 |
  5 | asus     | 21500 |
  6 | acer     |   700 |
(6 rows)
```

```
shop=# SELECT AVG(price) FROM production;
        avg
-------------------
 7683.333333333333
(1 row)
```

MIN/MAX

- Min is one of aggregation function that show us the minimum of choosing column
- Command : SELECT MIN(column) FROM name_of_table
- Max is opposite of minimum it`s clear !
- Command : SELECT MAX(column) FROM name_of_table

```
shop=# SELECT MIN(price) FROM production;
 min
-----
 400
(1 row)
```

```
shop=# SELECT MAX(price) FROM production;
  max
-------
 21500
(1 row)
```

AGGREGATION

- As u see we had about 7 prices the aggregation function takes all and return a single answer for us
- **SUM:**
  - Gives us a sum of column.
  - Syntax:
    - SELECT SUM(column) FROM name_of_table

```
shop=# SELECT SUM(price) FROM production;
  sum
-------
 46100
(1 row)
```

COUNT

- It`s like len in python gives us the count of records of column
- Syntax is clear right now
- Also we can count all columns by (*)

```
shop=# SELECT COUNT(price) FROM production;
 count
-------
     6
(1 row)
```

```
shop=# SELECT COUNT(*) FROM production;
 count
-------
     6
(1 row)
```

CONDITION IN AGGREGATIONS

- We can define a condition in aggregation functions by following syntax:
  - SELECT COUNT(*) FROM name_of_table WHERE condition

```
shop=# SELECT COUNT(*) FROM production WHERE price < 400;
 count
-------
     0
(1 row)

shop=# SELECT COUNT(*) FROM production WHERE price < 2000;
 count
-------
     4
(1 row)
```

ARRAY_AGG

- By using array_agg we can merge all items in one line

```
shop=# SELECT ARRAY_AGG(pro_name) FROM production;
                array_agg
-----------------------------------------
 {"htc one",hp,lenovo,samsung,asus,acer}
(1 row)
```

JSON_AGG

- Create a json of items in table

```
shop=# SELECT JSON_AGG(pro_name) FROM production;
                          json_agg
-------------------------------------------------------------
 ["htc one", "hp", "lenovo", "samsung", "asus", "acer"]
(1 row)
```

CORR

- CORR or correlation is one of statics aggregation functions and it`s for doing some math !
- For understanding better you have to know

what is correlation

```
shop=# SELECT CORR(id, price) FROM production;
        corr
---------------------
 0.42175529286244656
(1 row)
```

REGR_COUNT

- Give us a details about how many records(items) are not null

We have 6 not null records

```
shop=# SELECT REGR_COUNT(id, price) FROM production;
 regr_count
------------
          6
(1 row)
```

WINDOW FUNCTION

- Window functions gets some inputs and give us multiple answer (opposite of aggregation functions)
- The are is some specific window functions but you use aggregation functions as window functions too
- For knowing all window functions use following link:
  - https:\\www.postgresql.org\docs\current\functions-window.html

AGGREGATION FUNCTION AS WINDOW FUNCTION

1- We have a table like this

```
 group_id | group_name
----------+------------
        1 | LAPTOP
        2 | PHONE
        2 | MOUSE
(3 rows)


shop=# SELECT * FROM products;
 id |   name   | price | group_id
----+----------+-------+----------
  1 | xiaomi   |  2000 |        2
  2 | samsung  |  3000 |        2
  3 | htc      |  3040 |        2
  4 | sony z3  |  3000 |        2
  5 | razor    |  4000 |        3
  6 | asus     |  5000 |        1
  7 | acer     |  5600 |        1
  8 | lenovo   |  6800 |        1
(8 rows)
```

Now we want to use AVG as window function

SYNTAX:

SELECT column_name, column_name … , AVG(column_name) OVER() FROM name_of_table

Now as u can see by OVER() we create window function and that how window function works

```
shop=# SELECT name, group_id, AVG(price) OVER( ) FROM products;
  name    | group_id | avg
----------+----------+------
 xiaomi   |        2 | 4055
 samsung  |        2 | 4055
 htc      |        2 | 4055
 sony z3  |        2 | 4055
 razor    |        3 | 4055
 asus     |        1 | 4055
 acer     |        1 | 4055
 lenovo   |        1 | 4055
(8 rows)
```

OVER()

- OVER() is a sign of window function in OVER() we can do some operation too ! Like partitioning or etc.
  - In partitioning look at last slide example we want to give us avg by group id so we use partitioning. SYNTAX:
    - SELECT name_of_column , name_of_column … , AVG(name_of_column) OVER(PARTITION BY name_of_column) FROM name_of_table

```
shop=# SELECT name, group_id, AVG(price) OVER(PARTITION BY group_id) FROM products;
  name    | group_id | avg
----------+----------+------
 lenovo   |        1 | 5800
 asus     |        1 | 5800
 acer     |        1 | 5800
 sony z3  |        2 | 2760
 xiaomi   |        2 | 2760
 samsung  |        2 | 2760
 htc      |        2 | 2760
 razor    |        3 | 4000
(8 rows)
```

WINDOW FUNCTIONS

As reminder → window function sign is OVER()

- Now going for pure window functions :
- row_number() → row_number work is counting the rows
  - SYTNAX:
    - SELECT name_of_column … ROW_NUMBER() **OVER()** FROM name_of_table

```
shop=# SELECT name, group_id, id, ROW_NUMBER( ) OVER( ) FROM products;
  name   | group_id | id | row_number
---------+----------+----+------------
 xiaomi  |        2 |  1 |          1
 samsung |        2 |  2 |          2
 htc     |        2 |  3 |          3
 sony z3 |        2 |  4 |          4
 razor   |        3 |  5 |          5
 asus    |        1 |  6 |          6
 acer    |        1 |  7 |          7
 lenovo  |        1 |  8 |          8
(8 rows)
```

ROW_NUMBER() AND PARTITION

Which means in each group
how many product do we have

```
shop=# SELECT name, group_id, id, ROW_NUMBER( ) OVER(PARTITION BY group_id) FROM products;
  name    | group_id | id | row_number
----------+----------+----+------------
 lenovo   |        1 |  8 |          1
 asus     |        1 |  6 |          2
 acer     |        1 |  7 |          3
 sony z3  |        2 |  4 |          1
 xiaomi   |        2 |  1 |          2
 samsung  |        2 |  2 |          3
 htc      |        2 |  3 |          4
 razor    |        3 |  5 |          1
(8 rows)
```

RANK()

- Give the same records 1 number and it has gap ! (I couldn`t explain it better !)
  - It`s need a special syntax or in the other hand see the all t
Special syntax is → OVER(ORDER BY name_of_column)

```
shop=# SELECT id, name, price, group_id, RANK() OVER(ORDER BY price) FROM products;
 id |   name    | price | group_id | rank
----+-----------+-------+----------+------
  1 | xiaomi    |  2000 |        2 |    1
  2 | samsung   |  3000 |        2 |    2
  4 | sony z3   |  3000 |        2 |    2
  3 | htc       |  3040 |        2 |    4
  5 | razor     |  4000 |        3 |    5
  6 | asus      |  5000 |        1 |    6
  7 | acer      |  5600 |        1 |    7
  8 | lenovo    |  6800 |        1 |    8
```

```
shop=# SELECT id, name, price, group_id, RANK() OVER() FROM products;
 id |   name    | price | group_id | rank
----+-----------+-------+----------+------
  1 | xiaomi    |  2000 |        2 |    1
  2 | samsung   |  3000 |        2 |    1
  3 | htc       |  3040 |        2 |    1
  4 | sony z3   |  3000 |        2 |    1
  5 | razor     |  4000 |        3 |    1
  6 | asus      |  5000 |        1 |    1
  7 | acer      |  5600 |        1 |    1
  8 | lenovo    |  6800 |        1 |    1
(8 rows)
```

DENSE_RANK

- Compare this two codes please

```
shop=# SELECT id, name, price, group_id, DENSE_RANK() OVER(ORDER BY price) FROM products;
 id |   name   | price | group_id | dense_rank
----+----------+-------+----------+------------
  1 | xiaomi   |  2000 |        2 |          1
  2 | samsung  |  3000 |        2 |          2
  4 | sony z3  |  3000 |        2 |          2
  3 | htc      |  3040 |        2 |          3
  5 | razor    |  4000 |        3 |          4
  6 | asus     |  5000 |        1 |          5
  7 | acer     |  5600 |        1 |          6
  8 | lenovo   |  6800 |        1 |          7
(8 rows)
```

1, 2 2 , 3 , 4 5, 6, 7

```
shop=# SELECT id, name, price, group_id, RANK() OVER(ORDER BY price) FROM products;
 id |   name   | price | group_id | rank
----+----------+-------+----------+------
  1 | xiaomi   |  2000 |        2 |    1
  2 | samsung  |  3000 |        2 |    2
  4 | sony z3  |  3000 |        2 |    2
  3 | htc      |  3040 |        2 |    4
  5 | razor    |  4000 |        3 |    5
  6 | asus     |  5000 |        1 |    6
  7 | acer     |  5600 |        1 |    7
  8 | lenovo   |  6800 |        1 |    8
```

1,2 2 , 4 , 5 ,6 ,7 ,8

1 , 2 2 , 4 it`s gap ! Cause we had two (2) rank ignore number 3 and goes for 4
But DENSE_RANK doesn't have a gap

LAG

- Giving us details about last record of row (example will explain you better)
- Lag arguments : LAG (name_of_column , offset number)
  - Offset number means → how much previous column we want to see

It partition by group id and that`s why the start of each group id lag`s is blank

```
shop=# SELECT id, name, price, group_id, LAG(price, 1) OVER(PARTITION BY group_id ORDER BY price) FROM
products;
 id |  name   | price | group_id | lag
----+---------+-------+----------+------
  6 | asus    |  5000 |        1 |
  7 | acer    |  5600 |        1 | 5000
  8 | lenovo  |  6800 |        1 | 5600
  1 | xiaomi  |  2000 |        2 |
  4 | sony z3 |  3000 |        2 | 2000
  2 | samsung |  3000 |        2 | 3000
  3 | htc     |  3040 |        2 | 3000
  5 | razor   |  4000 |        3 |
(8 rows)
```

LEAD

- LEAD() is opposite function of LAG()

```
shop=# SELECT id, name, price, group_id, LEAD(price, 1) OVER(PARTITION BY group_id ORDER BY price) FROM
products;
 id |  name   | price | group_id | lead
----+---------+-------+----------+------
  6 | asus    |  5000 |        1 | 5600
  7 | acer    |  5600 |        1 | 6800
  8 | lenovo  |  6800 |        1 |
  1 | xiaomi  |  2000 |        2 | 3000
  4 | sony z3 |  3000 |        2 | 3000
  2 | samsung |  3000 |        2 | 3040
  3 | htc     |  3040 |        2 |
  5 | razor   |  4000 |        3 |
(8 rows)
```

VIEW

- VIEW is the name that we can give it to our query imagine that we have a complicated query and we have to call it many times instead of rewriting it many times we just call view
- SYNTAX:
  - CREATE OR REPLACE [TEMP|TEMPORARY] VIEW name_of_view AS name_of_query
    - Note : TEMPORARY is an option you can write it or you can`t it`s optional but if you write TEMP when the session is over the VIEW will be removed (DROPED)

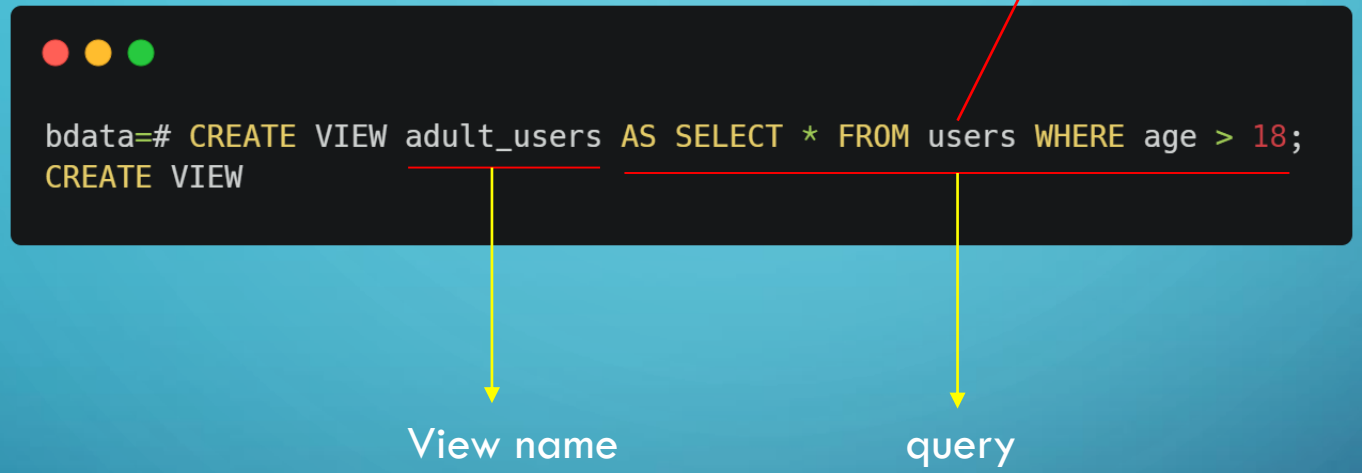
VIEW

- We have table like this and wanna search users WHERE age > 18

SELECT * FROM users WHERE age > 18 → it`s query that we want to use it many times so instead of writing it again and again we make VIEW of it

```
bdata=# SELECT * FROM users;
 id |   name   | age
----+----------+-----
  1 | mohammad |  22
  2 | ali      |  25
  3 | soha     |  24
  4 | amir     |  27
  5 | nazanin  |  21
  6 | nini     |   3
  7 | nojavon  |  16
(7 rows)
```

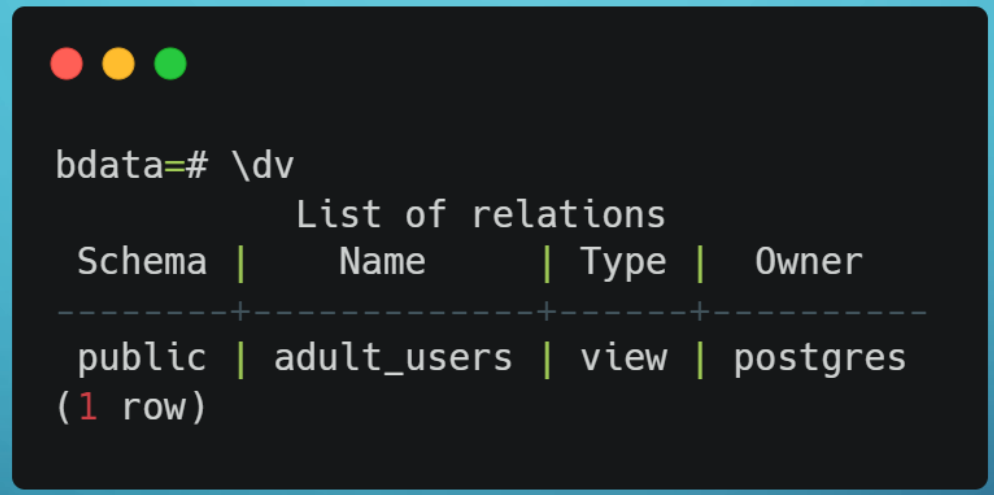
CREATE VIEW



After creating view we don`t need to write query every time just need to call view

HOW TO SEE OUR VIEWS

- By \dv



```
bdata=# \dv
           List of relations
 Schema |    Name      | Type |  Owner
--------+--------------+------+----------
 public | adult_users | view | postgres
(1 row)
```

HOW TO CALL VIEW

- For calling view use the following command:
  - SELECT * FROM name_of_view

```
bdata=# SELECT * FROM adult_users;
 id |   name   | age
----+----------+-----
  1 | mohammad |  22
  2 | ali      |  25
  3 | soha     |  24
  4 | amir     |  27
  5 | nazanin  |  21
(5 rows)
```

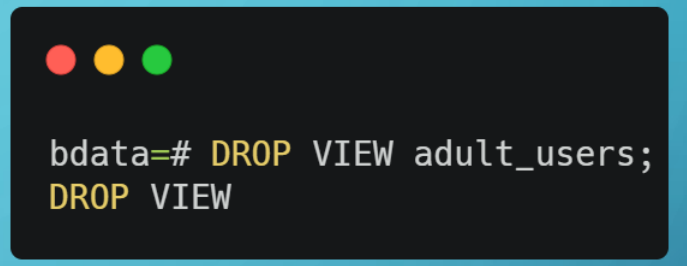
ADD CONDITION IN VIEW

- Also you can add condition in view by using WHERE

```
bdata=# SELECT * FROM adult_users WHERE name = 'mohammad' AND age > 21;
 id |   name   | age
----+----------+-----
  1 | mohammad |  22
(1 row)
```

DELETE VIEW

- By using DROP VIEW name_of_view



MATERIALIZED VIEW

- MATERIALIZED VIEW is using for huge database and very complicated queries MATERIALIZED VIEW is same as view but difference is that this VIEW save information on itself.

Means I saved 5 records

You can`t see MATERIALIZED VIEW in \dv cause it saved information on itself you can use \d and find your MATERIALIZED VIEW

For delete MATERIALIZED VIEW use DROP MATERIALIZED VIEW name_of_materialized_view

```
bdata=# DROP MATERIALIZED VIEW adult_users ;
DROP MATERIALIZED VIEW
```

```
bdata=# CREATE MATERIALIZED VIEW adult_users AS SELECT * FROM users WHERE age > 18;
SELECT 5
bdata=# \d
                List of relations
 Schema |    Name     |       Type        |  Owner
--------+-------------+-------------------+----------
 public | adult_users | materialized view | postgres
 public | users       | table             | postgres
(2 rows)


bdata=# SELECT * FROM adult_users;
 id |   name   | age
----+----------+-----
  1 | mohammad |  22
  2 | ali      |  25
  3 | soha     |  24
  4 | amir     |  27
  5 | nazanin  |  21
(5 rows)
```

REFRESH MATERIALIZED VIEW

- As we said MATERIALIZED VIEW saves information on itself so if we update our table we need to update our MATERIALIZED VIEW too by following command:
  - REFRESH MATERIALIZED VIEW name_of_materialized_view

```
bdata=# REFRESH MATERIALIZED VIEW adult_users;
REFRESH MATERIALIZED VIEW
```

THE END

THANKS FOR YOU ATTENTION