

Introdução ao PostgreSQL



Introdução ao PostgreSQL

O que é o PostgreSQL?

Sistema de gerenciamento de banco de dados objeto relacional.

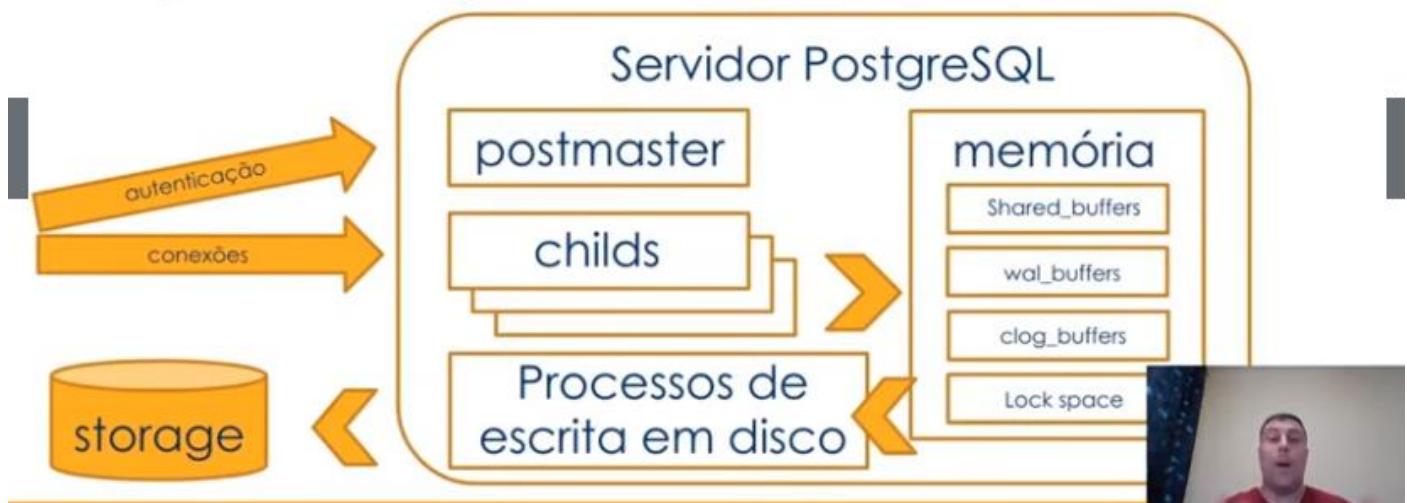
Teve início no Departamento de Ciência da Computação na Universidade da Califórnia em Berkeley em 1986.

SGBD Opensource.



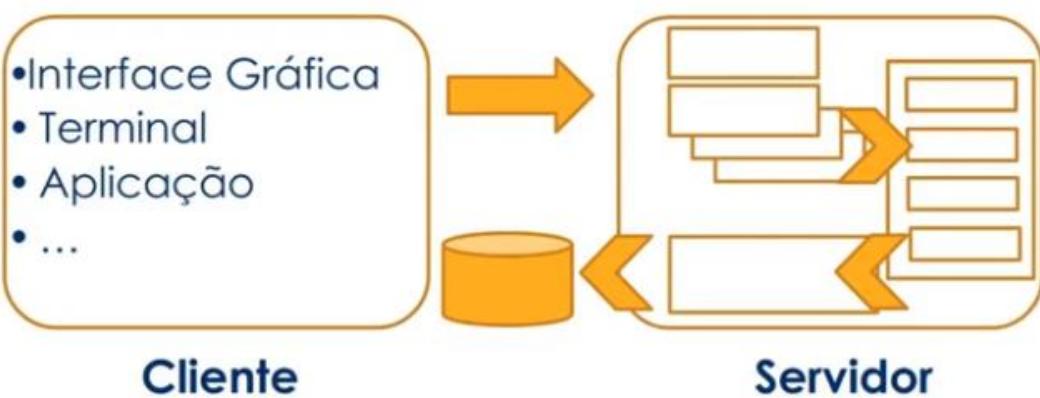
Introdução ao PostgreSQL

Arquitetura multiprocessos



Introdução ao PostgreSQL

Modelo cliente/servidor



Introdução ao PostgreSQL

Principais características

- OpenSource
- Point in time recovery
- Linguagem procedural com suporte a várias linguagens de programação (perl, python, etc)
- Views, functions, procedures, triggers
- Consultas complexas e Common table expressions (CTE)
- Suporte a dados geográficos (PostGIS)
- Controle de concorrência multi-versão



Introdução ao PostgreSQL

Instalação e documentação do PostgreSQL

Site oficial:

<https://www.postgresql.org/>

Download com instruções passo a passo:

<https://www.postgresql.org/download/>

Documentação completa:

<https://www.postgresql.org/docs/manuals/>



Site Oficial:

<https://www.postgresql.org/>

Download:

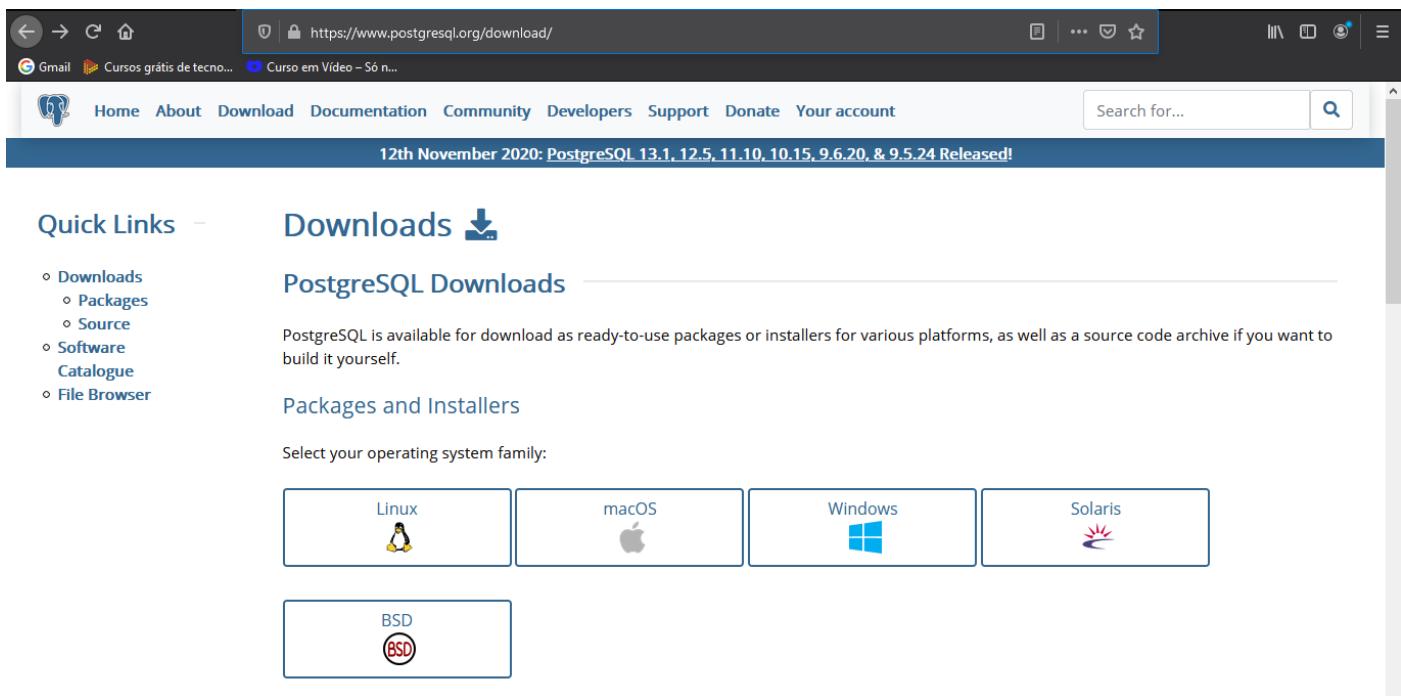
<https://www.postgresql.org/download/>

Documentação:

<https://www.postgresql.org/docs/>

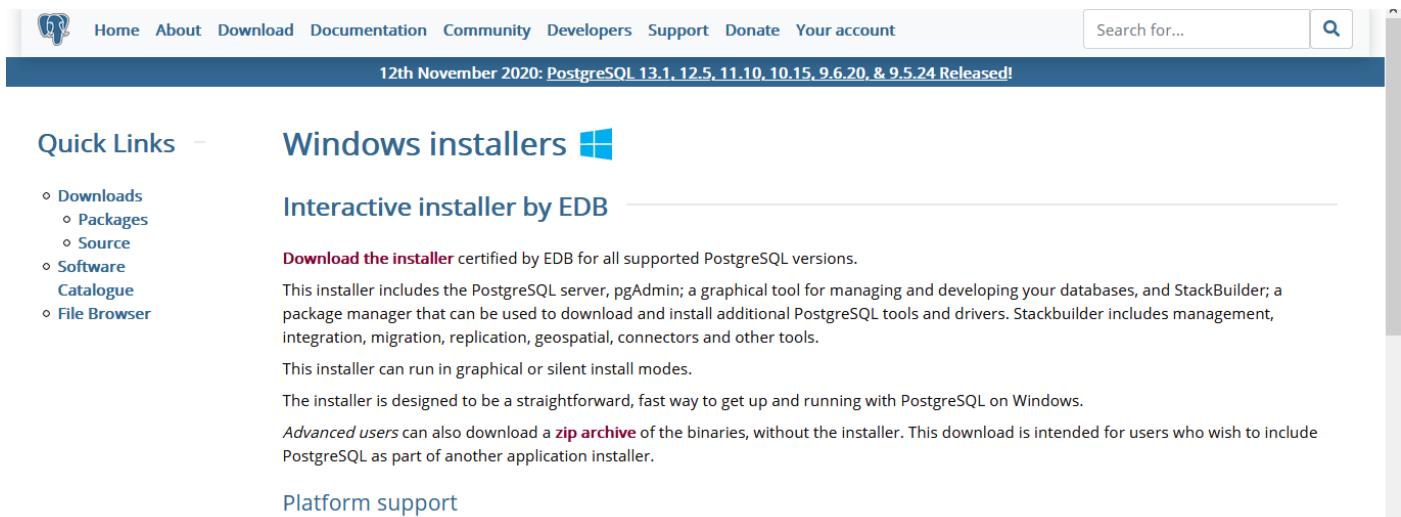
Download e instalação do PostgreSQL

Para fazer o download do PostgreSQL é preciso acessar a página
<https://www.postgresql.org/download/>



The screenshot shows the PostgreSQL download page. At the top, there's a navigation bar with links for Home, About, Download, Documentation, Community, Developers, Support, Donate, and Your account. A search bar is also present. Below the navigation, a banner announces the release of PostgreSQL 13.1, 12.5, 11.10, 10.15, 9.6.20, & 9.5.24. The main content area is titled "Downloads" with a dropdown arrow. Under "PostgreSQL Downloads", it says: "PostgreSQL is available for download as ready-to-use packages or installers for various platforms, as well as a source code archive if you want to build it yourself." It lists "Packages and Installers" and asks to select an operating system family. Five options are shown: Linux (with a penguin icon), macOS (with an apple icon), Windows (with a blue square icon), Solaris (with a sun icon), and BSD (with a red circle icon).

Em seguida escolher o sistema operacional, nesse caso o Windows.



The screenshot shows the PostgreSQL download page again, but this time focusing on the "Windows installers" section. The left sidebar has "Downloads" selected. The main content area is titled "Windows installers" with a Windows logo icon. It features a section for "Interactive installer by EDB". It describes the "Interactive installer" as certified by EDB for all supported PostgreSQL versions. It includes the PostgreSQL server, pgAdmin, StackBuilder, and StackBuilder. It also mentions "Platform support".

Na próxima página vamos navegar através do link “**Download the installer**”



Why EDB?

Products

Services

Support

Resources

Plans

Contact

Sign In

Downloads

PostgreSQL Database Download

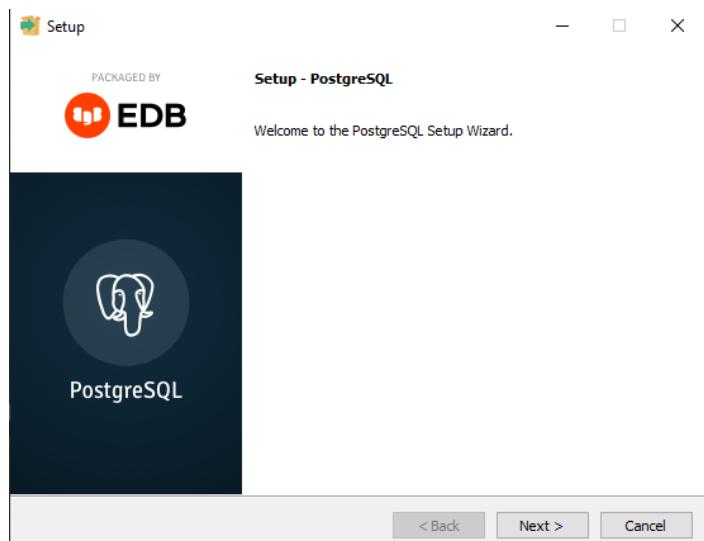
Version	Linux x86-64	Linux x86-32	Mac OS X	Windows x86-64	Windows x86-32
13.1	N/A	N/A	Download	Download	N/A
12.5	N/A	N/A	Download	Download	N/A
11.10	N/A	N/A	Download	Download	N/A
10.14	Download				
9.6.20	Download				

We use cookies on this site to improve performance and enhance your user experience. By browsing this site, you are giving your consent for us to set cookies. For more information, see our [Privacy Policy](#).

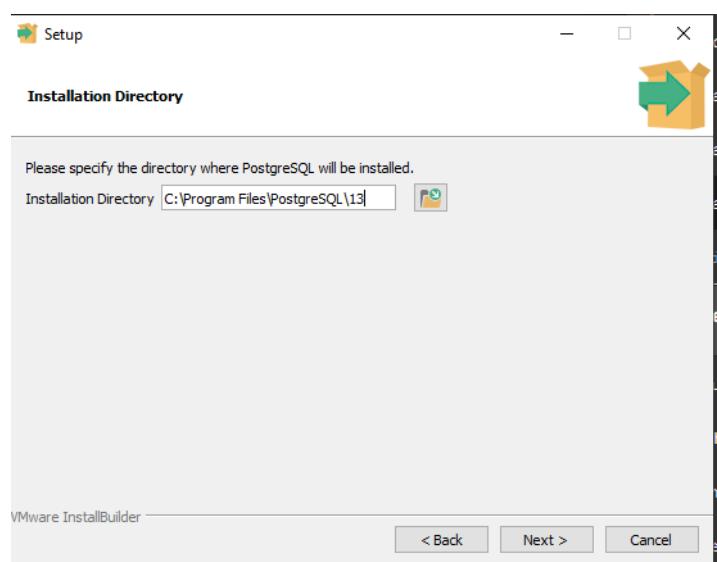
Okay, got it!

Agora nessa página precisamos escolher qual a versão queremos instalar e o download no instalador será feito normalmente. Nesse caso eu escolhi a versão 13.1.

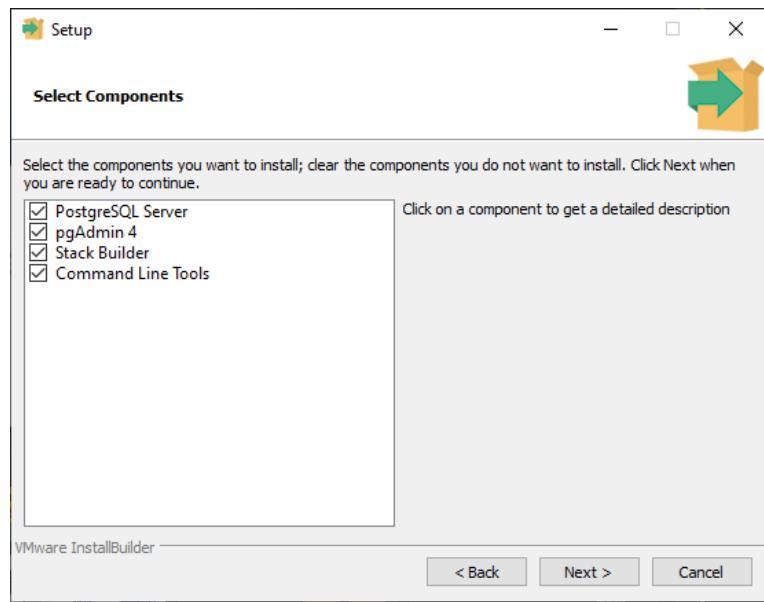
Ao executar o instalador iremos iniciar a instalação do PostgreSQL.



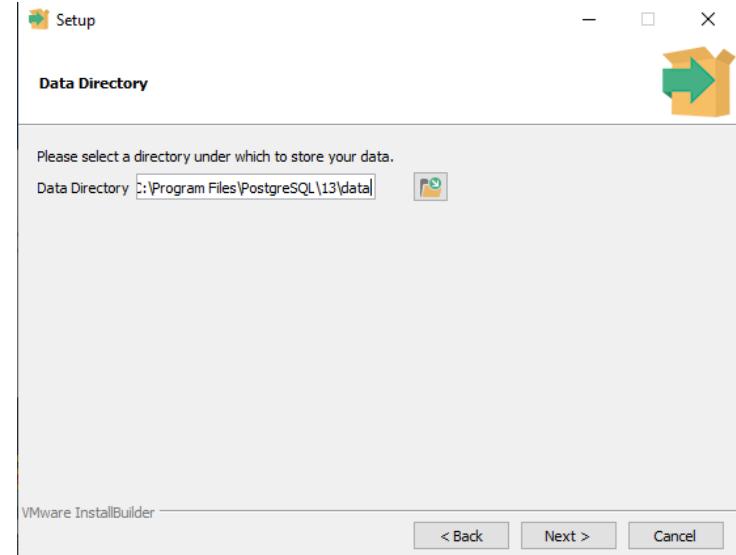
Tela inicial da instalação.



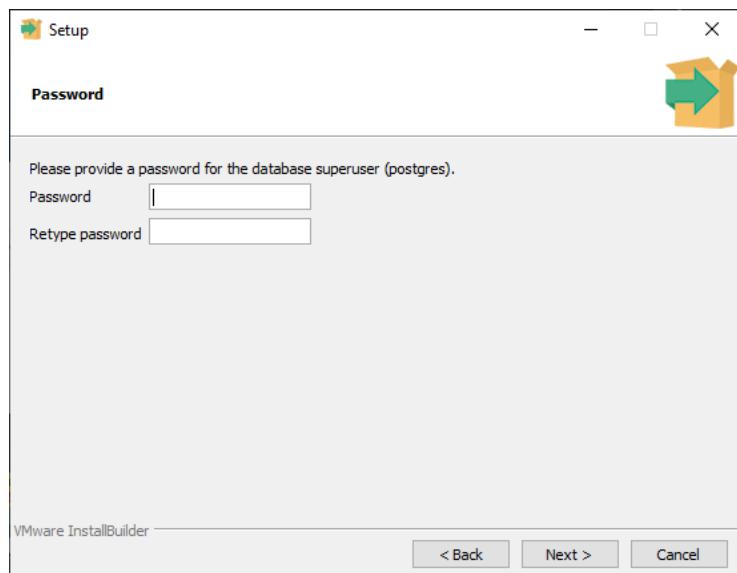
Diretório de instalação



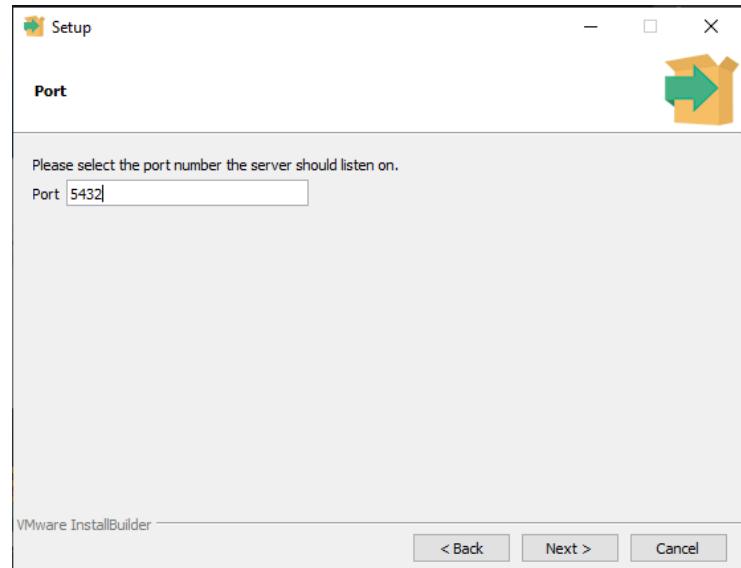
Instalação de componentes essenciais.



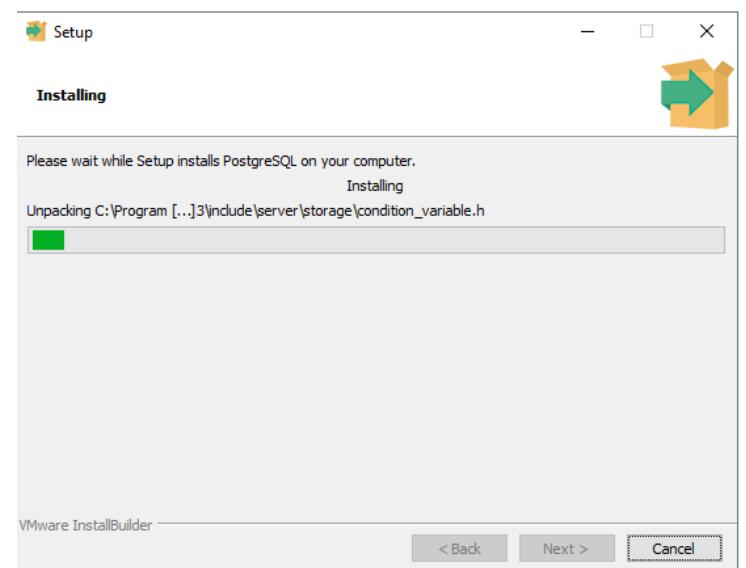
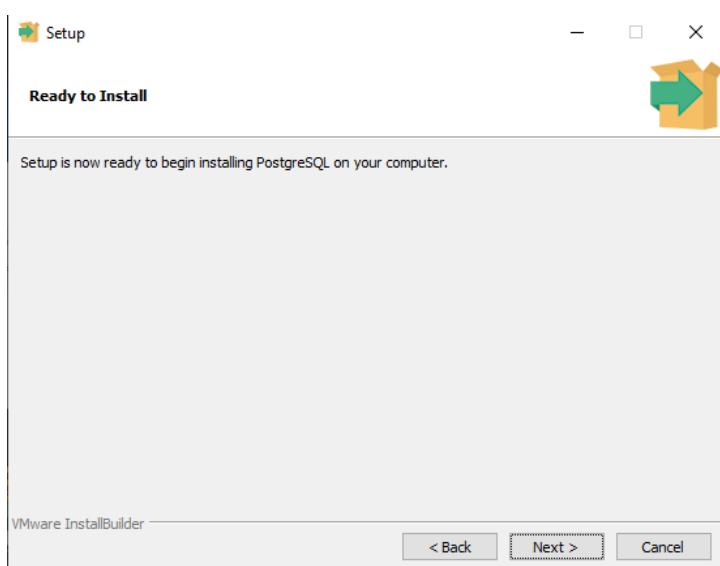
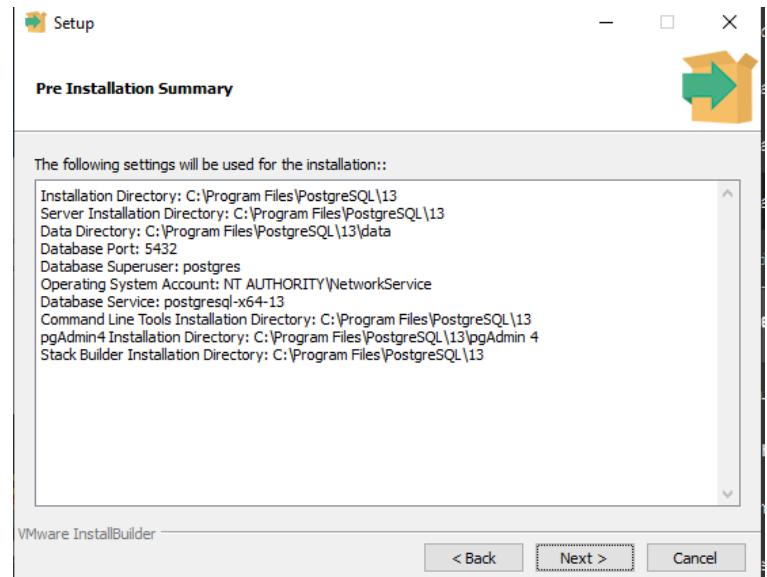
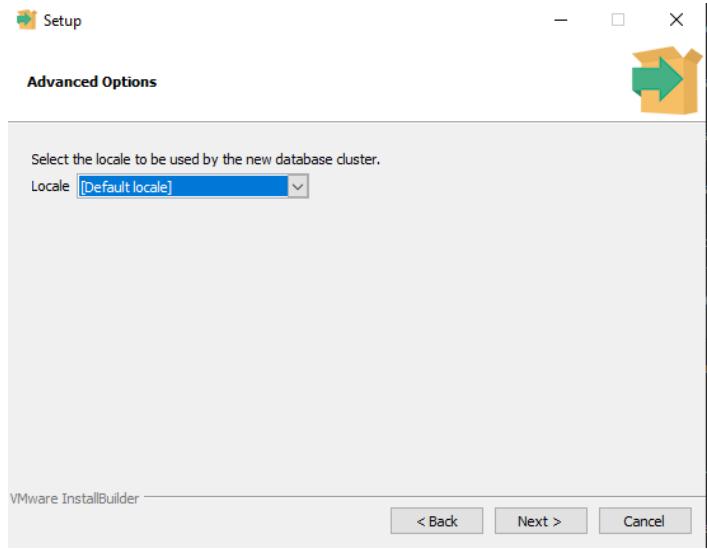
Diretório de dados, como tabelas, views, trigger.

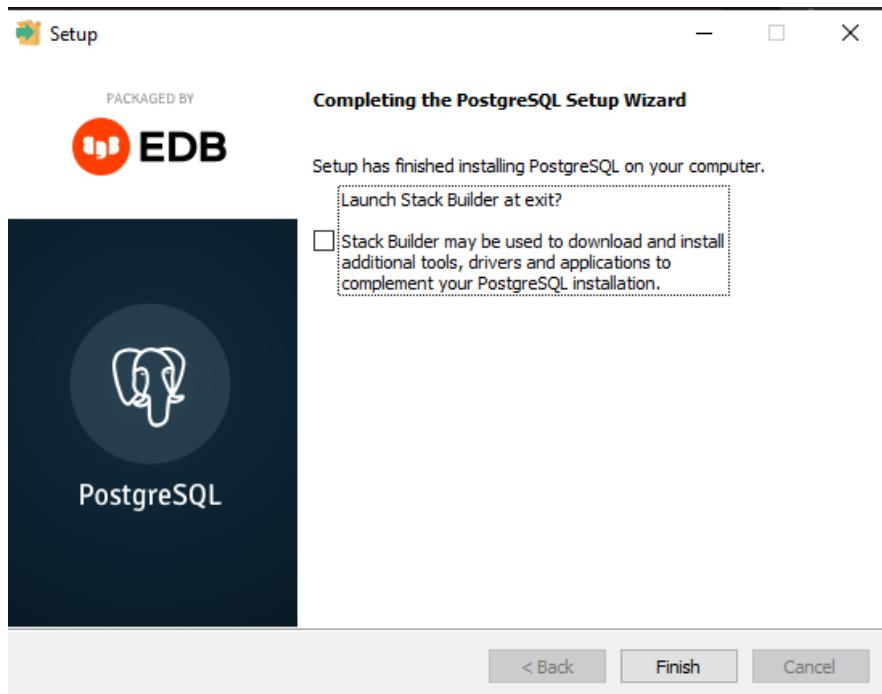


Configuração de senha do usuário postgres. (postgresql)



Configuração de porta (padrão 5432)





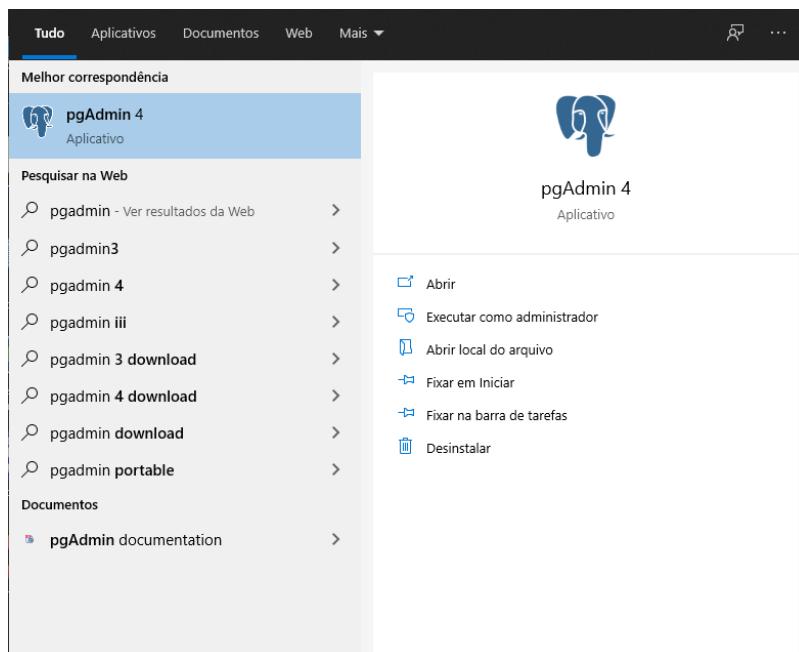
Instalação finalizada.

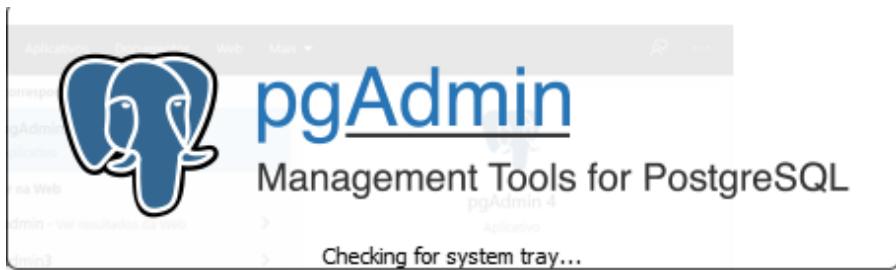
PgAdmin:

No postgres temos a ferramenta PgAdmin, uma ferramenta web com interface gráfica para gerenciar e interagir com o banco de dados.

<https://www.pgadmin.org/>

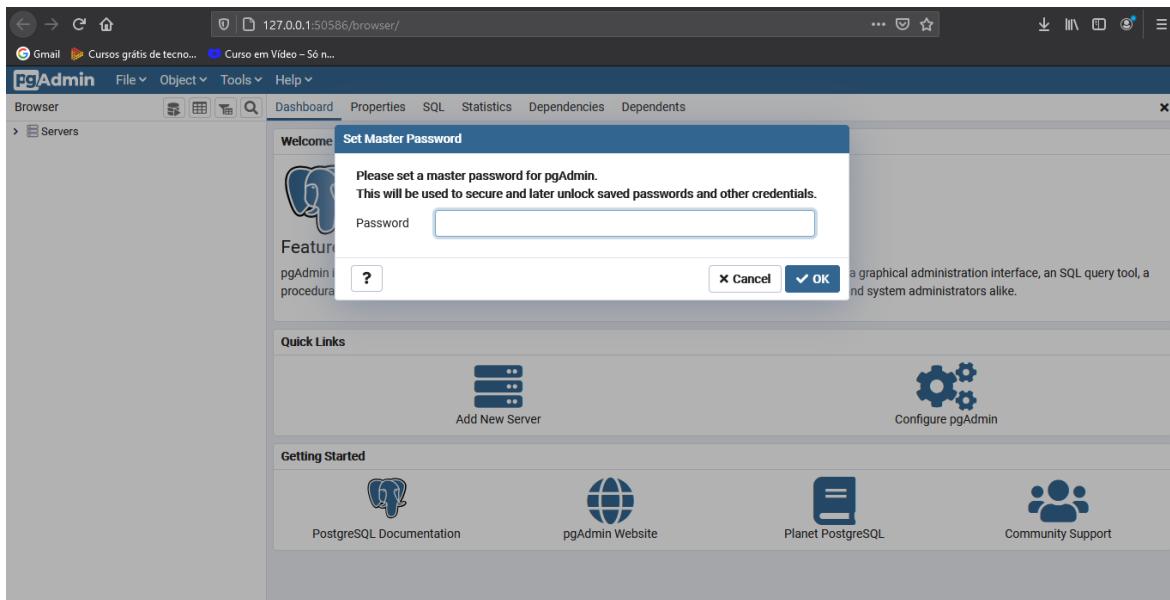
Para abrir a ferramenta que já foi instalada na tela de componentes ao instalar o postgres, é só abrir o menu iniciar do Windows, digitar **pgadmin 4** e executar o programa.



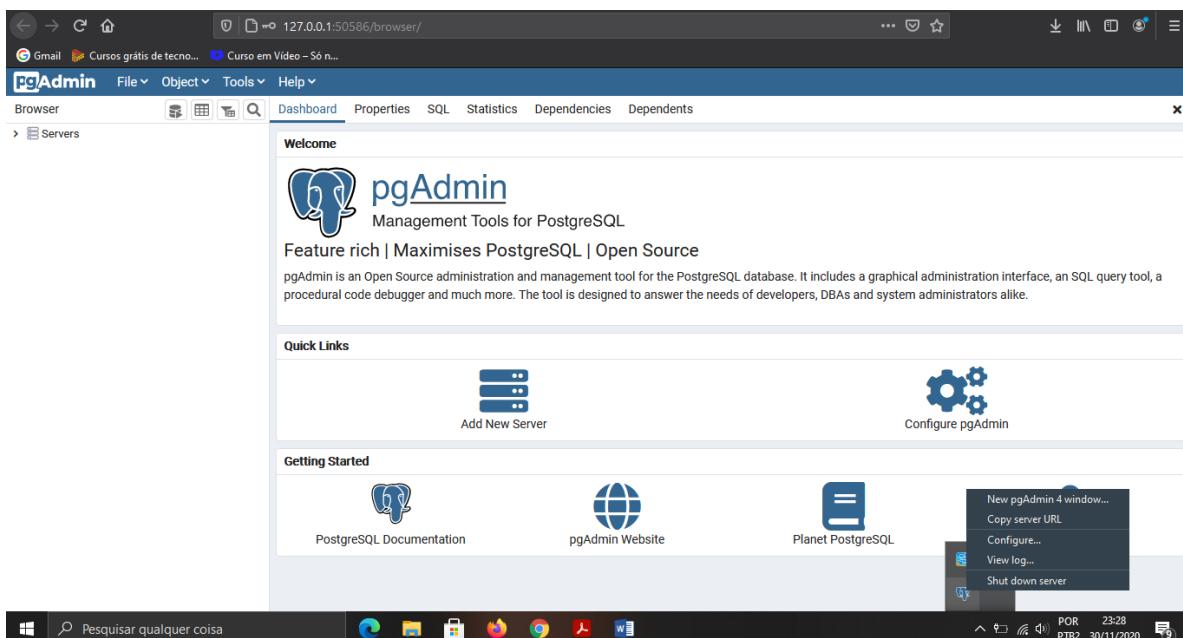


Iniciando o pgadmin4.

O browser iniciará automaticamente já no endereço correto e pedira a senha que foi configurada na instalação.



PgAdmin iniciado.



Página inicial do pgadmin. Ao canto opções do pgadmin na barra de tarefas do Windows.

Arquivos e configurações

1. Arquivo postgresql.conf
2. Arquivo pg_hba.conf
3. Arquivo pg_ident.conf
4. Comandos administrativos



O arquivo postgresql.conf

Definição

Arquivo onde estão definidas e armazenadas todas as configurações do servidor PostgreSQL.

Alguns parâmetros só podem ser alterados com uma reinicialização do banco de dados.

A view pg_settings, acessada por dentro do banco de dados, guarda todas as configurações atuais.

Sobre o arquivo postgresql.conf



O arquivo postgresql.conf

postgresql.conf

Ao acessar a view pg_settings, é possível visualizar todas as configurações atuais:

```
SELECT name, setting  
FROM pg_settings;
```

Ou é possível usar o comando:

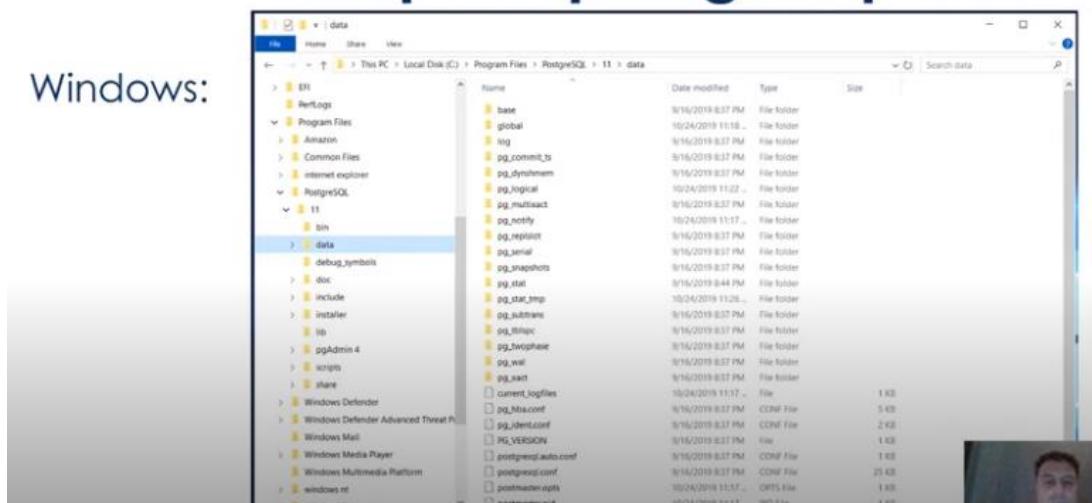
```
SHOW [parâmetro];
```



Acessando a view pg_settings dentro do banco para visualiza as configurações.

O arquivo postgresql.conf

Windows:



Diretório dos arquivos de dados.

O arquivo postgresql.conf

Configurações de conexão

- **LISTEN_ADDRESSES**
Endereço(s) TCP/IP das interfaces que o servidor PostgreSQL vai escutar/liberar conexões.
- **PORT**
A porta TCP que o servidor PostgreSQL vai ouvir. O padrão é 5432.
- **MAX_CONNECTIONS**
Número máximo de conexões simultâneas no servidor PostgreSQL
- **SUPERUSER_RESERVED_CONNECTIONS**
Número de conexões (slots) reservadas para conexões ao banco de dados de super usuários.



Alguns parâmetros importantes.

O arquivo postgresql.conf

Configurações de autenticação

- **AUTHENTICATION_TIMEOUT**
Tempo máximo em segundos para o cliente conseguir uma conexão com o servidor.
- **PASSWORD_ENCRYPTION**
Algoritmo de criptografia das senhas dos novos usuários criados no banco de dados.
- **SSL**
Habilita a conexão criptografada por SSL
(Somente se o PostgreSQL foi compilado com suporte SSL)



O arquivo postgresql.conf

Configurações de memória

- **SHARED_BUFFERS**

Tamanho da memória compartilhada do servidor PostgreSQL para cache/buffer de tabelas, índices e demais relações.

- **WORK_MEM**

Tamanho da memória para operações de agrupamento e ordenação (ORDER BY, DISTINCT, MERGE JOINS).

- **MAINTENANCE_WORK_MEM**

Tamanho da memória para operações como VACUUM, INDEX, ALTER TABLE.



O arquivo pg_hba.conf

Definição

Arquivo responsável pelo controle de autenticação dos usuários no servidor PostgreSQL.

O formato do arquivo pode ser:

```
local    database  user  auth-method [auth-options]
host     database  user  address  auth-method [auth-options]
hostssl  database  user  address  auth-method [auth-options]
hostnoss  database user  address  auth-method [auth-options]
host     database  user  IP-address  IP-mask auth-method [auth-options]
hostssl  database  user  IP-address  IP-mask auth-method [auth-options]
hostnoss  database user  IP-address  IP-mask auth-method [auth-options]
```



O arquivo pg_hba.conf

Métodos de autenticação

- TRUST (conexão sem requisição de senha)
- REJECT (rejeitar conexões)
- MD5 (criptografia md5)
- PASSWORD (senha sem criptografia)
- GSS (generic security service application program interface)
- SSPI (security support provider interface - somente para Windows)
- KRB5 (kerberos V5)
- IDENT (utiliza o usuário do sistema operacional do cliente via ident server)
- PEER (utiliza o usuário do sistema operacional do cliente)
- LDAP (ldap server)
- RADIUS (radius server)
- CERT (autenticação via certificado ssl do cliente)
- PAM (pluggable authentication modules. O usuário precisa estar no b



O arquivo pg_ident.conf

Definição

Arquivo responsável por mapear os usuários do sistema operacional com os usuários do banco de dados. Localizado no diretório de dados PGDATA de sua instalação.

A opção ident deve ser utilizada no arquivo pg_hba.conf

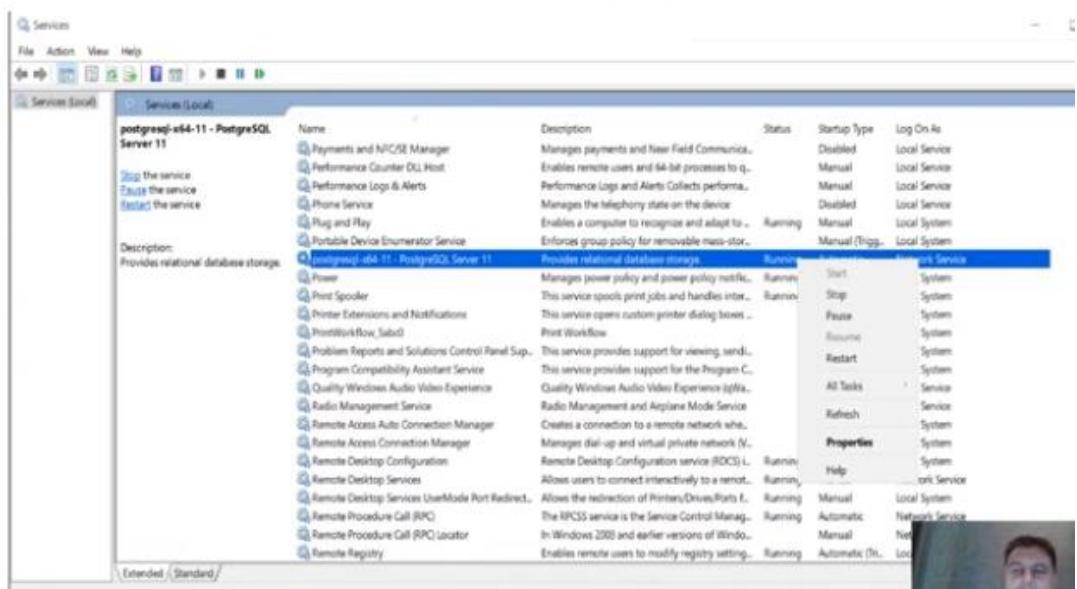
O arquivo pg_ident.conf

Exemplo:

	SYSTEM-USERNAME	PG-USERNAME
1 # MAPNAME		
2 diretoria	daniel	pg_diretoria
3 comercial	tiburcio	pg_comercial
4 comercial	valdeci	pg_comercial

Comandos administrativos

Windows:



Comandos administrativos

Binários do PostgreSQL:

- createdb
- createuser
- dropdb
- dropuser
- initdb
- pg_ctl
- pg_basebackup
- pg_dump / pg_dumpall
- pg_restore
- psql
- reindexdb
- vacuumdb



Arquitetura/Hierarquia

Cluster

Coleção de bancos de dados que compartilham as mesmas configurações (arquivos de configuração) do PostgreSQL e do sistema operacional (porta, listen_addresses, etc).

Arquitetura/Hierarquia

Banco de dados (database)

Conjunto de schemas com seus objetos/relações (tabelas, funções, views, etc)

Arquitetura/Hierarquia

Schema

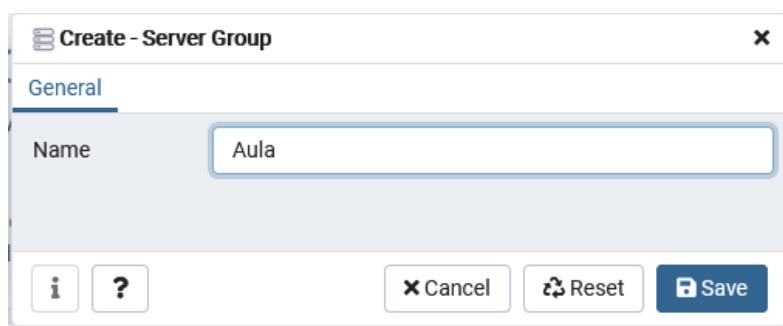
Conjunto de objetos/relações (tabelas, funções, views, etc).

Trabalhando com o pgAdmin

Para começar a trabalhar com o postgresql iremos cirar uma conexão básica.



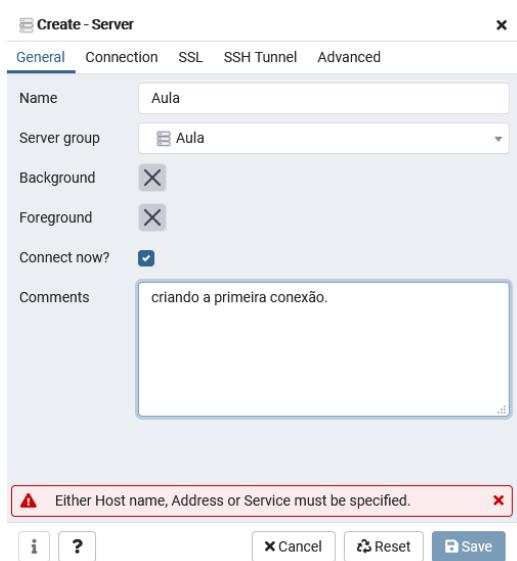
Para criar uma conexão selecionamos Servers, vamos até a opção **object/create/server group**. Dessa forma estamos criando um grupo de servidor.



Será aberta uma caixa para nomear o grupo de servidores que será criado.



Para criar uma nova conexão nesse novo servidor vamos selecionar o grupo de servidores no caso o que acabamos de criar vamos em **object/create/server**.



Será aberta uma janela onde iremos aplicar algumas configurações básicas como:

name: nome da conexão.

Background: escolher cores.

Connect now?: para conectar já ou não.

Comments: utilizar comentário é uma boa pratica.

Agora na aba **connection** vamos configurar

Host: endereço do servidor.

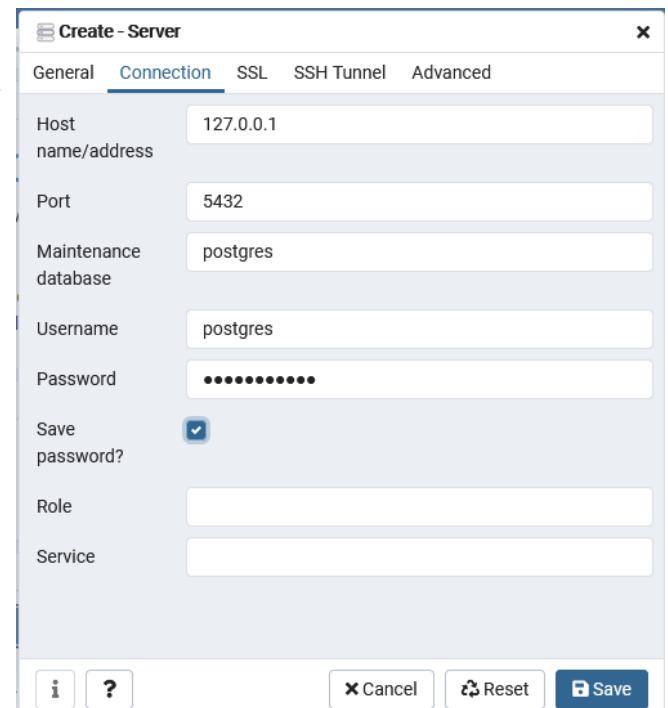
Port: porta de conexão (5432-porta padrão)

Maintenance: banco de dados para manutenção.

Username: Usuário.

Password: Senha configurada.

Save password: salvar senha ou não.

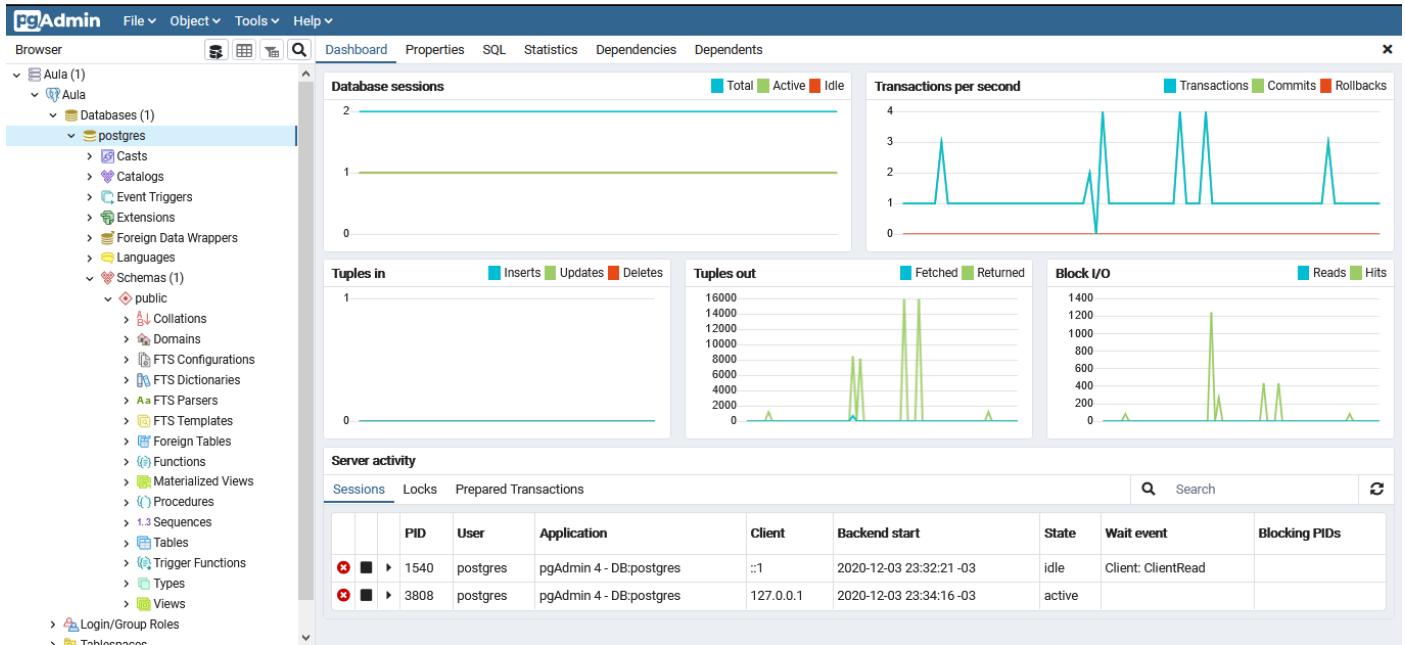


No momento em que a conexão for salva podemos ver os bancos que estão inclusos naquele cluster.

Nesse cluster podemos ter vários bancos de dados, mas no momento só existe um banco chamado postgres e nele diversos objetos como casts, event triggers, schemas.

Em schemas podemos visualizar outros objetos como collections, domains, functions, procedures, tables, triggers, types, views, entre outras. É possível ter vários schemas dentro de um mesmo banco de dados, nesse caso temos um schemas chamado public.

Ao selecionar o banco de dados e acessar a aba **dashboard** podemos visualizar alguns gráficos.

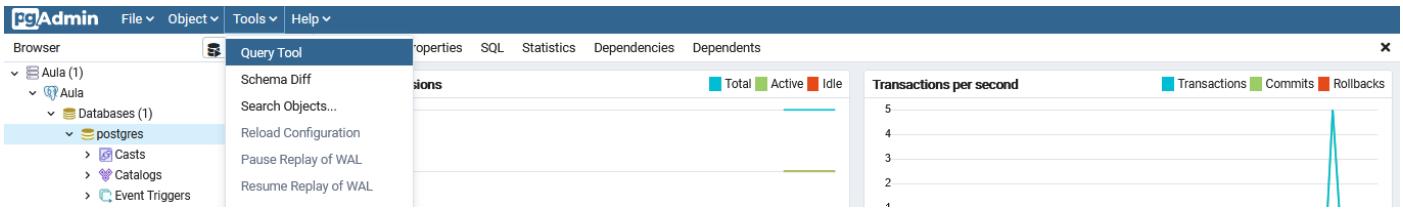


- **Database sessions:** conexão dentro do banco de dados.
- **Transactions per second:** Todas as transações, commits, rollbacks que estão acontecendo naquele segundo.
- **Tuples in – tuples out:** Quantidades de tuplas que estão entrando e saindo do banco. **Tuples in** seria a quantidade de inserts, updates, deletes e **tuples out** quanto está sendo retornado nossos comandos SQL.
- **Block I/O:** Inputs e outputs.

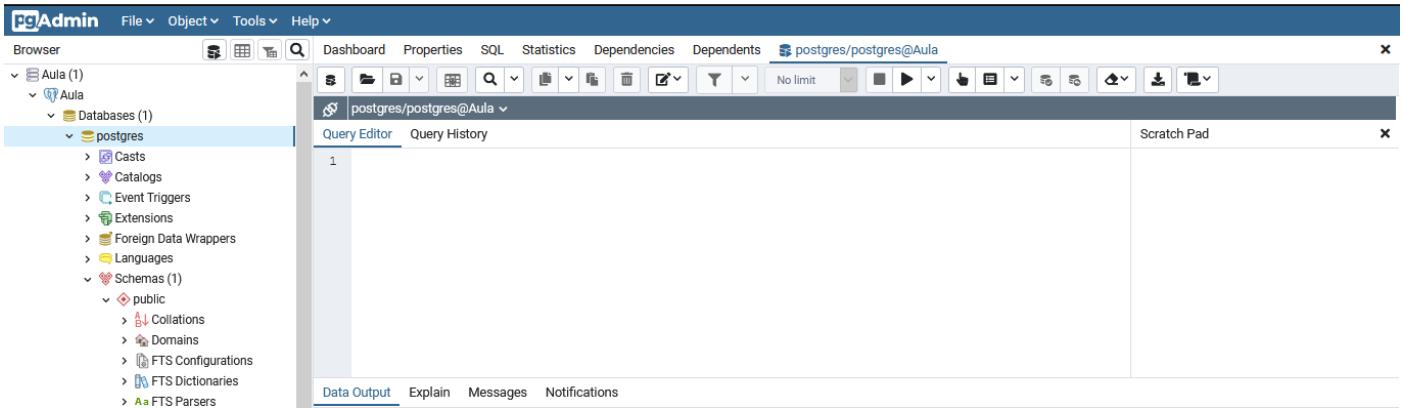
Na área inferior do **dashboard** temos a seção **server activiy** na aba **sessions** vai mostrar todas as seções conectadas e mostrar exatamente o que todas estão fazendo. Já na aba **prepared transactions** estão alguns comandos preparados que estão engatilhados no banco.

Criando um banco de dados.

Basicamente vamos fazer o seguinte.

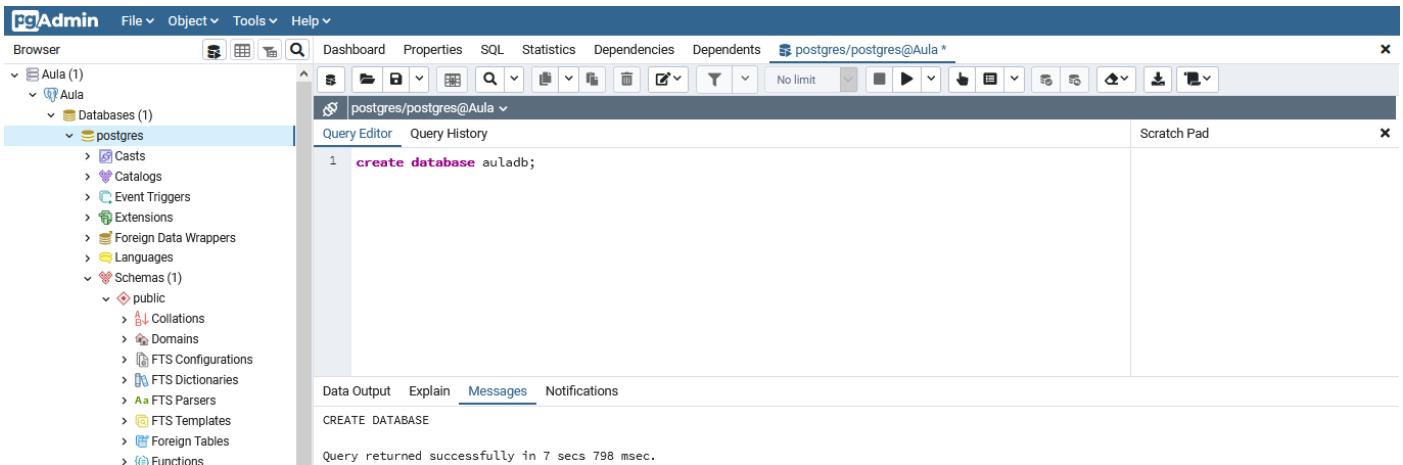


Selecionamos o banco **postgres** vamos em **tools/query tool**.

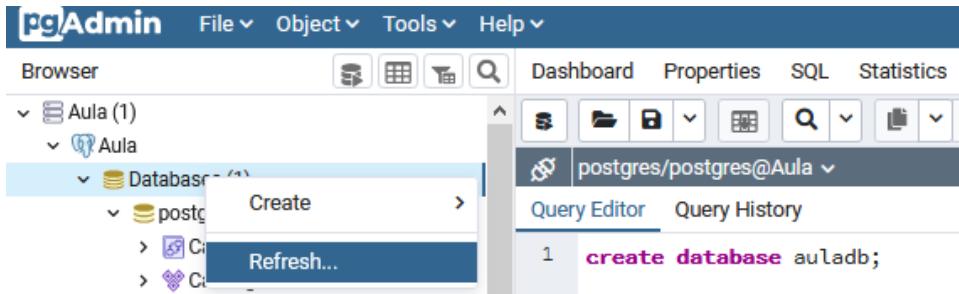


Notemos que abriu uma nova janela onde iremos escrever os nossos comandos.

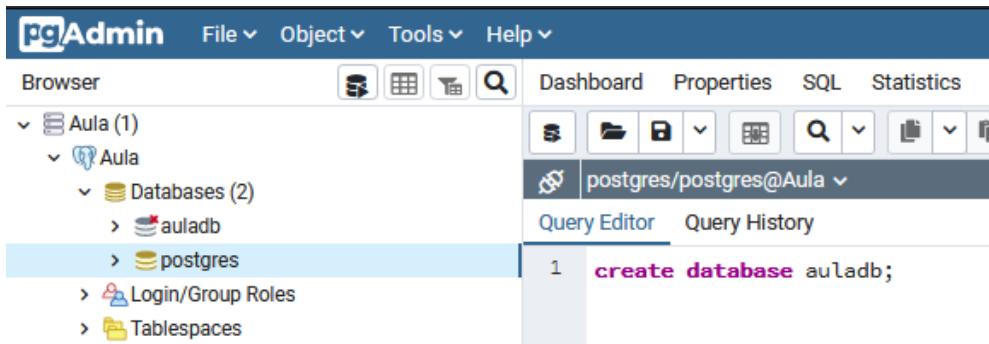
Para criar um novo banco de dados vamos utilizar o comando **create database (nome)**; Para executar o comando é preciso aperta tecla f5.



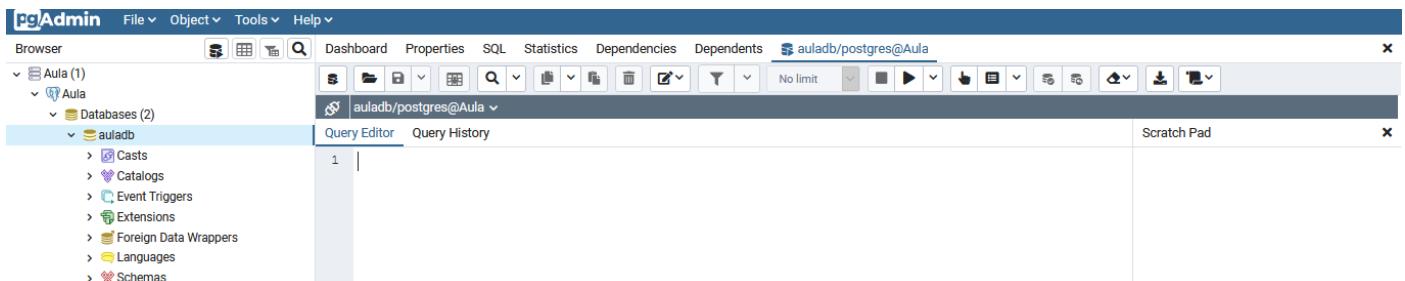
Para atualizar a lista de bancos de dados vamos em **databases** clicamos com o botão direito e em **refresh**.



Agora irá aparecer o novo banco criado.



Notemos que a janela de código **query editor** está conectada ao banco postgres com o usuário postgres dentro do grupo de servidores aula **postgres/postgres @Aula**. Podemos fechar essa janela e para começar a codificar no novo banco de dados o selecionamos e refazemos o processo clicando em **tools/query tool**.



Agora sim estaremos conectados ao novo banco.

Administrando users, roles e groups.



Conceitos users/roles/groups

Definição:

Roles (papéis ou funções), users (usuários) e grupo de usuários são “contas”, perfis de atuação em um banco de dados, que possuem permissões em comum ou específicas.

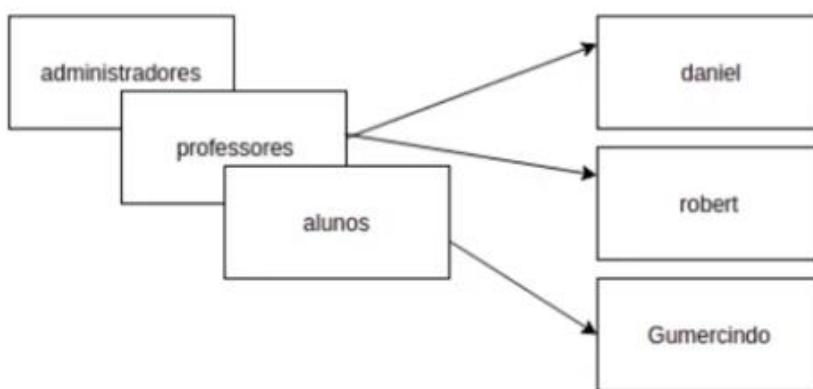
Nas versões anteriores do PostgreSQL 8.1, usuários e roles tinham comportamentos diferentes.

Atualmente, roles e users são alias.

É possível que roles pertençam a outras roles;



Conceitos users/roles/groups



Administrando users/roles/groups

CREATE ROLE name [[WITH] option [...]]

where option can be:

```
SUPERUSER | NOSUPERUSER
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| BYPASSRLS | NOBYPASSRLS
| CONNECTION LIMIT connlimit
| [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
| VALID UNTIL 'timestamp'
| IN ROLE role_name [, ...]
| IN GROUP role_name [, ...]
| ROLE role_name [, ...]
| ADMIN role_name [, ...]
| USER role_name [, ...]
| SYSID uid
```

Comando para se criar uma **role** e opções (configurações) que podem ser atribuídas.

- **create role name [[with] option [...]]**; comando a ser executado no banco de dados para a criação de uma nova role seguido do nome da rola e algumas opções que ela pode ter.
- **superuser / nosuperuser**: uma role que é super usuário tem permissões quase que irrestritas dentro de um banco de dados (cuidado, utilização somente quando for necessário). O padrão é que uma role seja **nosuperuser** (não é super usuário).
- **createdb / nocreatedb**: tem permissão ou não para criar um banco de dados.
- **createrole / nocreaterole**: tem permissão ou não para criar novas roles.
- **inherit / noinherit**: sempre que ela pertencer a uma outra role vai herdar todas as permissões da outra role ou pode pertencer a outra role e não herdar as permissões.
- **login / nologin**: pode ou não fazer login no banco de dados.
- **replication / noreplication**: pode ou não fazer replicações. Exemplo: fazer backup via pg_backup.
- **bypassrls / nobypassrls**: configuração em relação a segurança a nível de roles.
- **connection limit connlimit**: pode definir quantas conexões simultâneas a role pode ter dentro de um banco de dados.
- **[encrypted] password 'password' / password null**: pode ser ou não criptografado, mais utilizado a fazer operações de backup ou restore. Todo password por padrão no banco de dados já é criptografado com md5 mas que pode ser alterado para um padrão como sh256.

- **valid until 'timestamp'**: define até que data essa role terá acesso ao banco de dados.
- **in role role_name [, ...]**: quando criado um novo usuário e define essa opção essa nova role vai pertencer a role definida pela *inrole*.
- **in group role_name [, ...]**: mesmo resultado que o *in role*.
- **role role_name [, ...]**: nessa opção a role informada passa a pertencer ao grupo da nova role que está sendo criada.
- **admin role_name [, ...]**: passa a ter permissões administrativas dentro do grupo de roles especificada.
- **user role_name [, ...]**: !?
- **sysid uid: !?**



Administrando users/roles/groups

```
CREATE ROLE administradores
  CREATEDB
  CREATEROLE
  INHERIT
  NOLOGIN
  REPLICATION
  BYPASSRLS
  CONNECTION LIMIT -1;
```

```
CREATE ROLE professores
  NOCREATEDB
  NOCREATEROLE
  INHERIT
  NOLOGIN
  NOBYPASSRLS
  CONNECTION LIMIT 10;
```

```
CREATE ROLE alunos
  NOCREATEDB
  NOCREATEROLE
  INHERIT
  NOLOGIN
  NOBYPASSRLS
  CONNECTION LIMIT 90;
```

create role administradores: essa role poderá basicamente criar banco de dados, criar novas roles, não pede se logar no banco de dados (como boa prática as novas roles que fará parte desse grupo será criada com permissão para fazer login no banco de dados), pode fazer replicação, pode fazer um bypass, terá número de conexões ilimitadas.

create role professores: não pode criar banco de dados, não pode criar novas roles, vai herdar as permissões das roles a qual pertencer, não pode fazer login, não pode dar bypass, terá conexões limitadas a 10.

create role alunos: não pode criar banco de dados, não pode criar novas roles, vai herdar as permissões das roles a qual pertencer, não pode fazer login, não pode dar bypass, terá conexões limitadas a 90.

Administrando users/roles/groups

Associação entre roles

Quando uma role assume as permissões de outra role.

Necessário a opção **INHERIT**

No momento de criação da role:

- **IN ROLE** (passa a pertencer a role informada)
- **ROLE** (a role informada passa a pertencer a nova role)

Ou após a criação da role:

- GRANT [role a ser concedida] TO [role a assumir as permissões]

Administrando users/roles/groups

Associação entre roles

```
CREATE ROLE professores
  NOCREATEDB
  NOCREATEROLE
  INHERIT
  NOLOGIN
  NOBYPASSRLS
  CONNECTION LIMIT -1;
```

```
CREATE ROLE daniel LOGIN CONNECTION LIMIT 1 PASSWORD '123' IN ROLE professores;
  - A role daniel passa a assumir as permissões da role professores
```

```
CREATE ROLE daniel LOGIN CONNECTION LIMIT 1 PASSWORD '123' ROLE professores;
  - A role professores passar a fazer parte da role daniel assumindo suas permissões.
```

```
CREATE ROLE daniel LOGIN CONNECTION LIMIT 1 PASSWORD '123';
GRANT professores TO daniel;
```

O comando **grant** permite associar as roles mesmo depois de criadas. Nesse caso a role Daniel passará a fazer parte da role professores.

Administrando users/roles/groups

Desassociar membros entre roles

REVOKE [role que será revogada] FROM [role que terá suas permissões revogadas]

```
REVOKE professores FROM daniel;
```

O comando **revoke** desassocia as roles. Nesse caso a role daniel não faz mais parte da role professores.

Administrando users/roles/groups

Alterando uma role

ALTER ROLE role_specification [WITH] option [...]

where option can be:

- | SUPERUSER | NOSUPERUSER
- | CREATEDB | NOCREATEDB
- | CREATEROLE | NOCREATEROLE
- | CREATEUSER | NOCREATEUSER
- | INHERIT | NOINHERIT
- | LOGIN | NOLOGIN
- | REPLICATION | NOREPLICATION
- | BYPASSRLS | NOBYPASSRLS
- | CONNECTION LIMIT connlimit
- | [ENCRYPTED | UNENCRYPTED] PASSWORD 'password'
- | VALID UNTIL 'timestamp'

Alterar uma role é igual a criação de uma. Sendo que ao invés de utilizar **create role** será utilizado o comando **alter role** e passar as configurações.

Administrando users/roles/groups

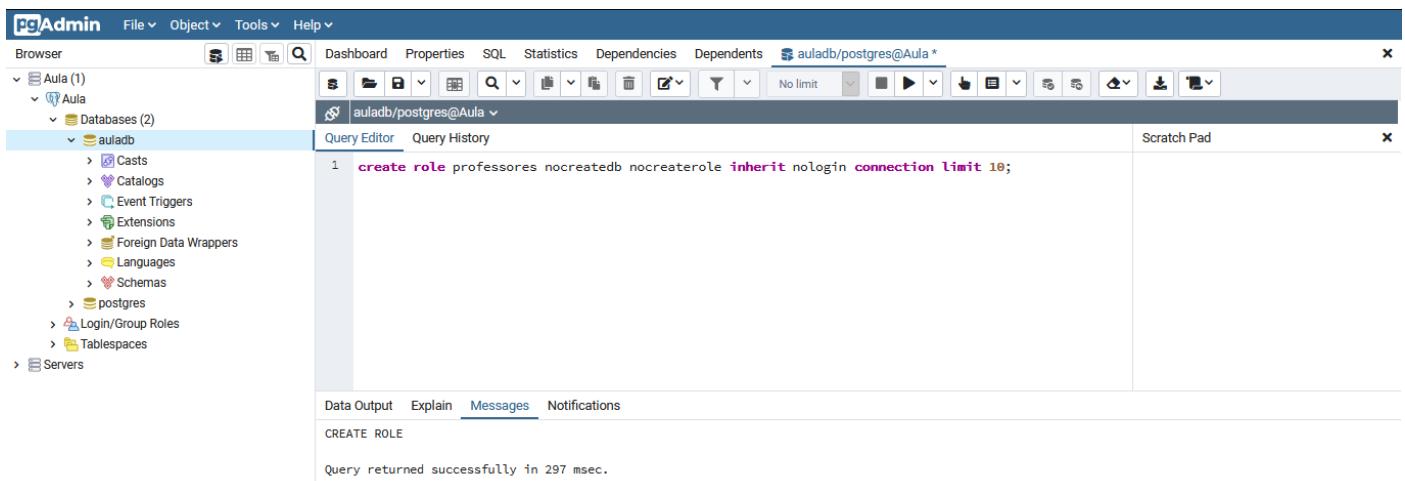
Excluindo uma role

DROP ROLE role_specification;

O comando drop exclui a role especificada.

• Praticando:

Vamos criar algumas roles no nosso banco de dados auladb que criamos e ver na prática como funciona.



The screenshot shows the PgAdmin 4 interface. The left sidebar displays the database structure under 'Aula' and 'auladb'. The 'Query Editor' tab is active, showing the SQL command to create a role:

```
1 create role professores nocreatedb nocreaterole inherit nologin connection limit 10;
```

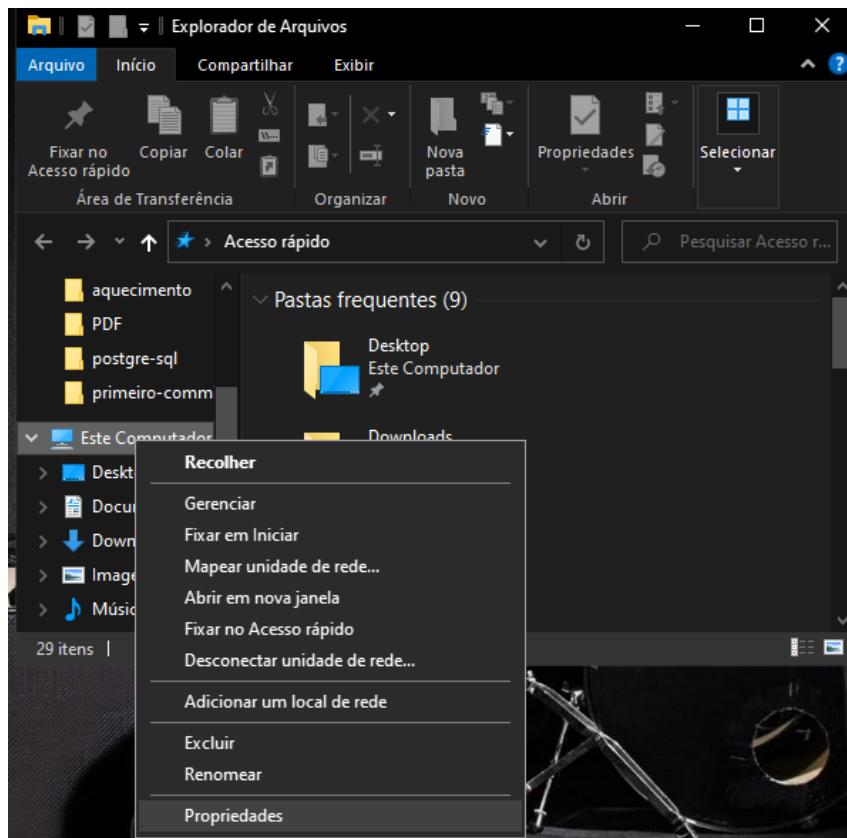
Below the query, the message 'CREATE ROLE' is displayed, followed by 'Query returned successfully in 297 msec.'

Criação da primeira role.

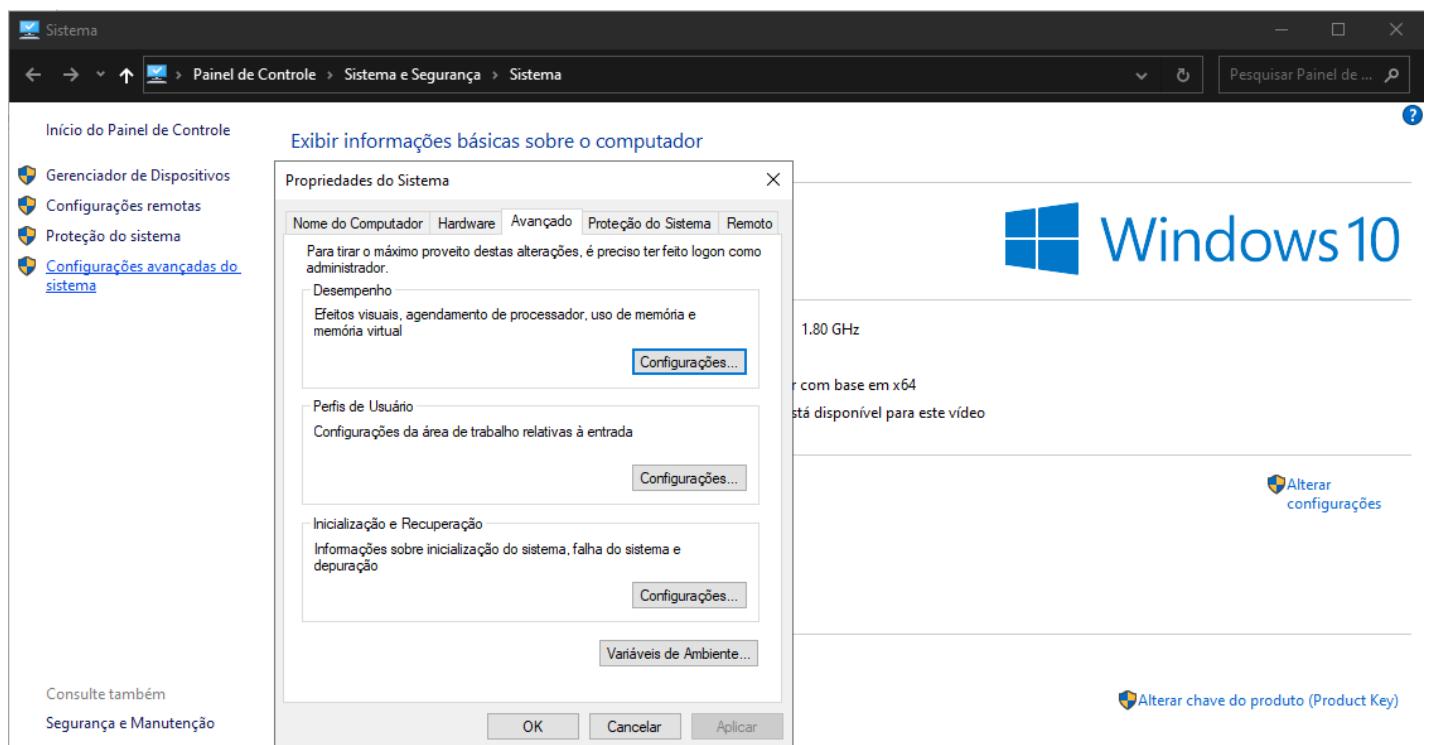
Abrimos a ferramenta **query tool**. Criamos uma role chamada **professores** com o comando: **create role professores nocreatedb nocreaterole inherit nologin connection limit 10;**

Os comandos **nocreatedb / nocreaterole** são o padrão, se eles não forem digitados o resultado será igual, pois já ficará subentendido que essa role não terá essa permissão.

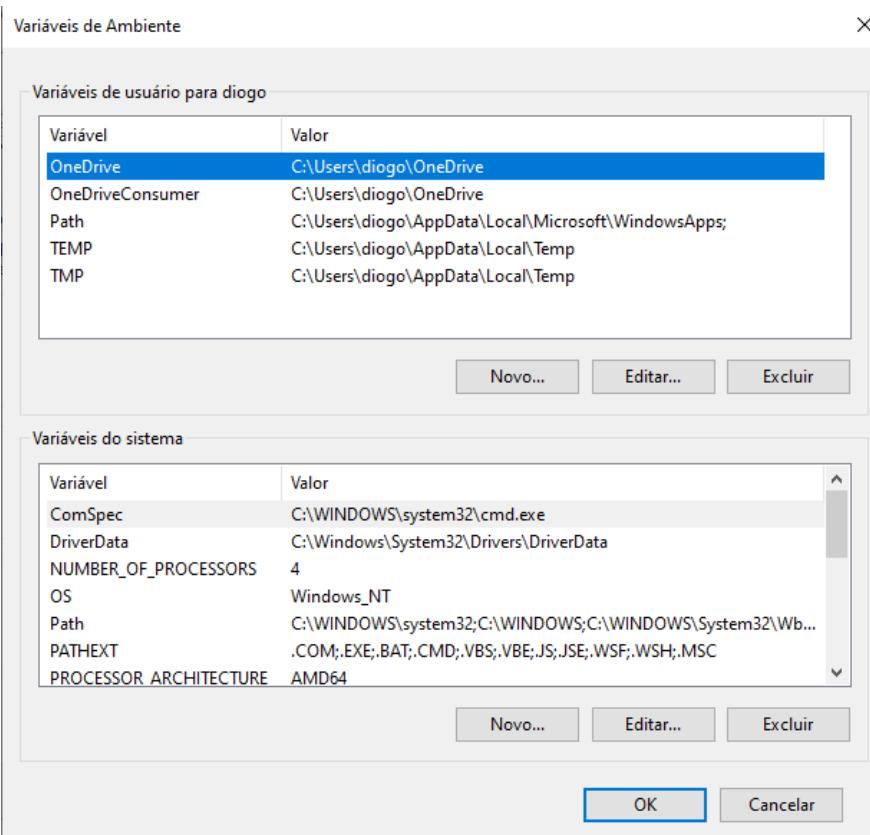
Para demonstrar algumas coisas na prática vamos precisar usar o terminal.
Para acessar esse recurso pelo Windows precisamos fazer algumas configurações.



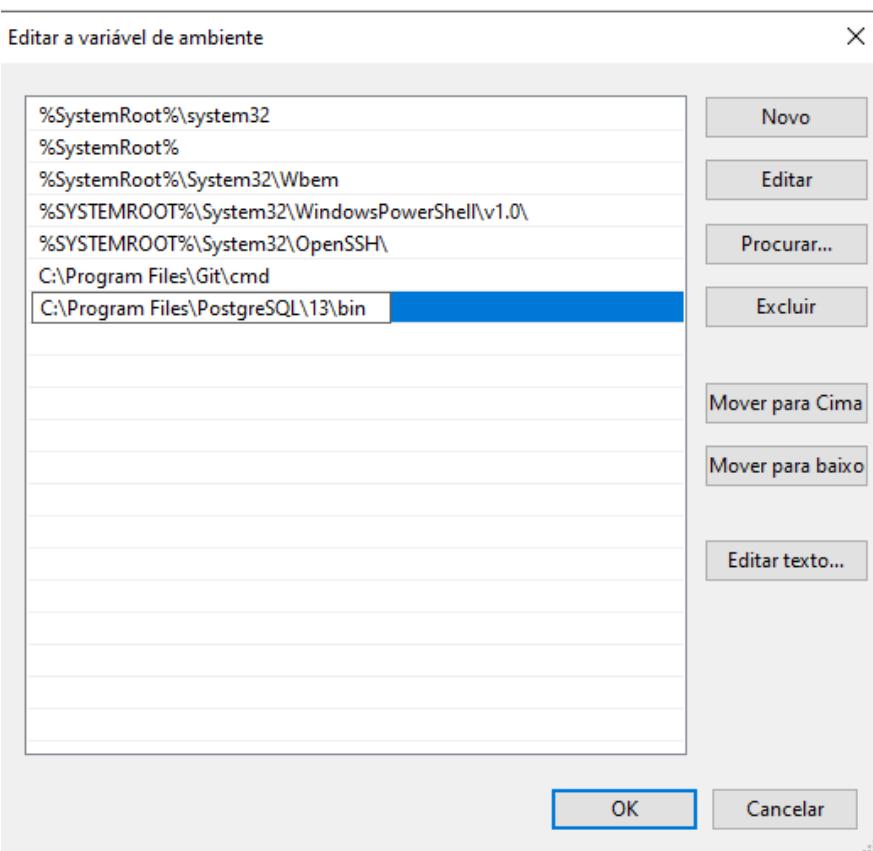
Primeiramente vamos acessar as propriedades do computador.



Nas propriedades do computador clicamos em **configurações avançadas do sistema**. Irá abrir uma janela com as propriedades do sistema. Precisamos acessar a aba **avançado** e depois **variaveis de ambiente**.



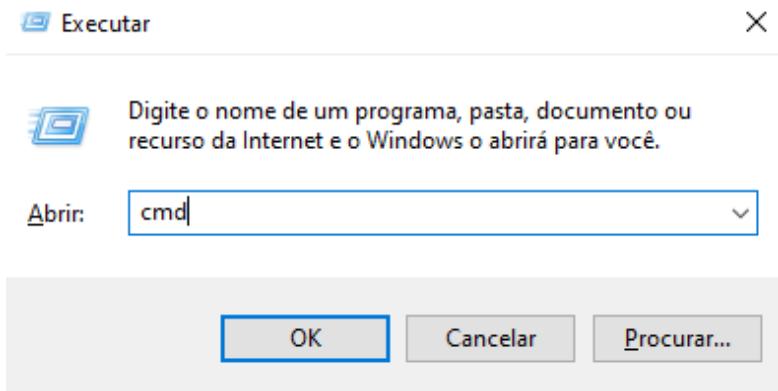
Nessa janela na area de **variáveis do sistema** acessamos a variavel **path**.



Indicamos o caminho da pasta **bin** na pasta de instalação do postgres nesse caso **C:\Program Files\PostgreSQL\13\bin** e confirmamos dando ok.

Essa past contém o arquivo **psql.exe** que é responsável por nos conectar ao postgres via terminal windows.

Por fim, para acessar o postgres via terminal windows basta abrir o cmd.



Tecla **windows + R** digitar cmd e dar enter.

Dentro do cmd digite o comando **psql -U (usuário) -h (local de destino do host)** autenticar com a senha e já estará conectado.

A screenshot of a Windows terminal window titled "C:\WINDOWS\system32\cmd.exe - psql - U postgres - h localhost". The window displays the following text:

```
C:\Users\diogo>psql -U postgres -h localhost
Password for user postgres:
psql (13.1)
WARNING: Console code page (850) differs from Windows code page (1252)
          8-bit characters might not work correctly. See psql reference
          page "Notes for Windows users" for details.
Type "help" for help.

postgres=#
```

Para listar as roles no banco de dados basta utilizar o comando **\du**. Podemos assim ver a role professores que criamos anteriormente.

A screenshot of a Windows terminal window titled "C:\WINDOWS\system32\cmd.exe - psql - U postgres - h localhost". The window displays the following text:

```
C:\Users\diogo>psql -U postgres -h localhost
Microsoft Windows [versão 10.0.19041.630]
(c) 2020 Microsoft Corporation. Todos os direitos reservados.

C:\Users\diogo>psql -U postgres -h localhost
Password for user postgres:
psql (13.1)
WARNING: Console code page (850) differs from Windows code page (1252)
          8-bit characters might not work correctly. See psql reference
          page "Notes for Windows users" for details.
Type "help" for help.

postgres=# \du
              List of roles
   Role name |                         Attributes                         | Member of
   postgres   | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
   professores | Cannot login                                         | {}
```

Voltando para o pgadmin vamos criar a seguinte role **CREATE ROLE daniel LOGIN PASSWORD '123'**; e listar novamente as roles no terminal.

```
C:\WINDOWS\system32\cmd.exe - psql -U postgres -h localhost
Password for user postgres:
psql (13.1)
WARNING: Console code page (850) differs from Windows code page (1252)
          8-bit characters might not work correctly. See psql reference
          page "Notes for Windows users" for details.
Type "help" for help.

postgres=# \du
              List of roles
   Role name   |          Attributes          | Member of
-----+-----+-----+
daniel      |                         |
postgres    | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
professores | Cannot login                           +| {}
                  | 10 connections

postgres=#

```

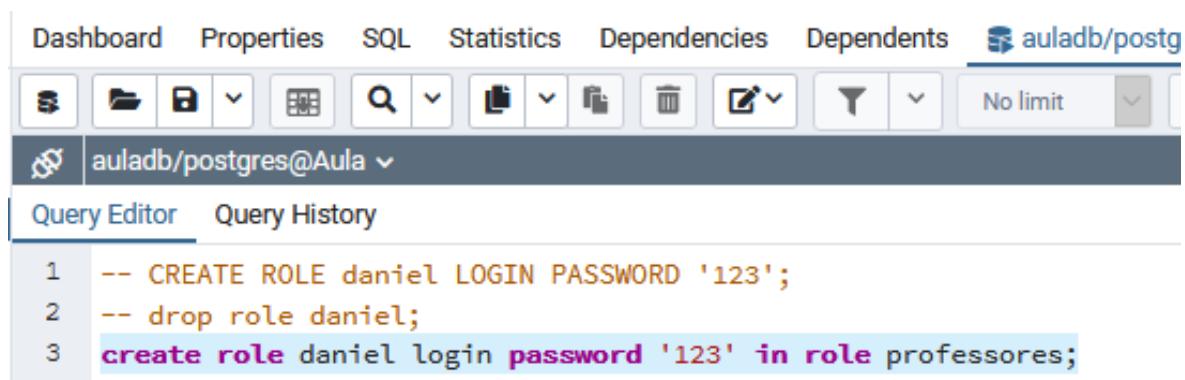
Resultado.

Vamos excluir a role daniel e em seguida criar a mesma relacionada a role professores.

Para excluir uma role no pgadmin é só utilizar o comando **drop role daniel;**

```
C:\WINDOWS\system32\cmd.exe - psql -U postgres -h localhost
postgres=# \du
              List of roles
   Role name   |          Attributes          | Member of
-----+-----+-----+
postgres    | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
professores | Cannot login                           +| {}
                  | 10 connections
```

Para associar o usuario daniel ao grupo professores vamos utilizar o **in role**.



The screenshot shows the pgAdmin 4 interface with the following details:

- Toolbar:** Includes icons for Dashboard, Properties, SQL, Statistics, Dependencies, Dependents, and a connection dropdown set to "auladb/postg".
- Filter Bar:** Shows "No limit" for results.
- User Selection:** Shows "auladb/postgres@Aula".
- Query Editor Tab:** Active tab.
- Query History Tab:** Available tab.
- SQL Area:** Contains the following three lines of SQL:
 - 1 -- CREATE ROLE daniel LOGIN PASSWORD '123';
 - 2 -- drop role daniel;
 - 3 create role daniel login password '123' in role professores;

Agora a role daniel é membro da role professores. Como boa pratica podemos perceber que a role professores não possui permissão para fazer login no banco de dados mas a role daniel possui essa permissão, isso para diferenciar grupo e usuário.

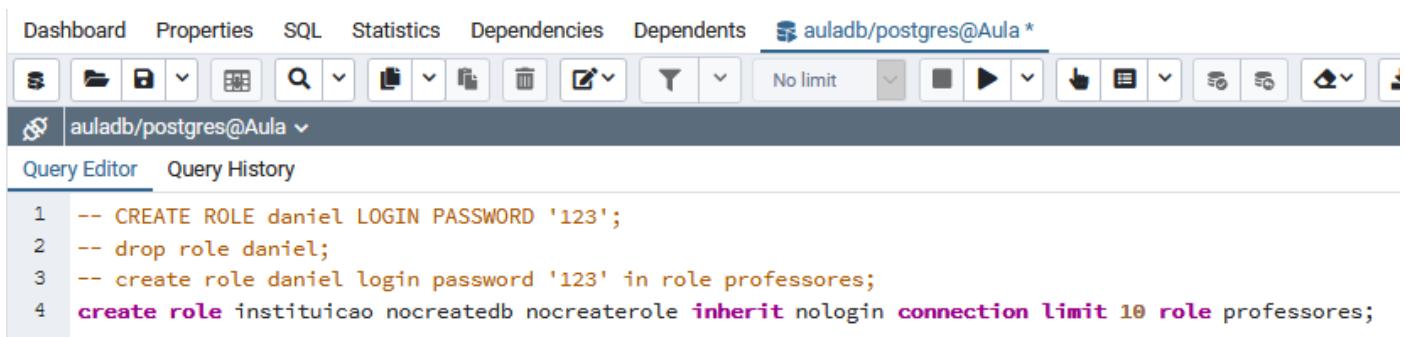
List of roles		
Role name	Attributes	Member of
daniel		{professores}
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS	{}
professores	Cannot login 10 connections	{}

Role Daniel como membro de professores.

Para testar o usuario daniel é só sair do banco utilizando o comando **\q** depois utilizar o comando **psql -U (usuario) (nome_banco)**. Nesse caso **psql -U daniel auladb** autenticar com a senha configurada e pronto.

```
C:\WINDOWS\system32\cmd.exe - psql -U daniel auladb
C:\Users\diogo>psql -U daniel auladb
Password for user daniel:
psql (13.1)
WARNING: Console code page (850) differs from Windows code page (1252)
          8-bit characters might not work correctly. See psql reference
          page "Notes for Windows users" for details.
Type "help" for help.
```

Agora se a role professores tenha que fazer parte de uma nova role? Para isso será preciso utilizar o comando **role** ao invés do **in role**.



The screenshot shows the pgAdmin 4 interface. The top navigation bar includes 'Dashboard', 'Properties', 'SQL', 'Statistics', 'Dependencies', 'Dependents', and a connection tab labeled 'auladb/postgres@Aula *'. Below the navigation is a toolbar with various icons. The main area is titled 'Query Editor' and contains a dropdown menu for connections. The query window displays the following SQL code:

```

1 -- CREATE ROLE daniel LOGIN PASSWORD '123';
2 -- drop role daniel;
3 -- create role daniel login password '123' in role professores;
4 create role instituicao nocreatedb nocreaterole inherit nologin connection limit 10 role professores;
```

Com o comando **create role instituicao nocreatedb nocreaterole inherit nologin connection limit 10 role professores;** foi criada uma nova role chamada instituição da qual a role professores será membro.

Podemos ver que a role daniel é membro da role professores que é membro da role instituição.

List of roles		
Role name	Attributes	Member of
daniel		{professores}
instituicao	Cannot login 10 connections	+ {}
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS	{}
professores	Cannot login 10 connections	+ {instituicao}

Para remover a role professores da role instituição é preciso usar o comando **revoke instituicao from professores;**

List of roles		
Role name	Attributes	Member of
daniel		{professores}
instituicao	Cannot login 10 connections	+ {}
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS	{}
professores	Cannot login 10 connections	+ {}

E para adicionar professores comomembro da role instituição novamente vamos usar o comando **grant instituicao to professores;**

List of roles		
Role name	Attributes	Member of
daniel		{professores}
instituicao	Cannot login 10 connections	+ {}
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS	{}
professores	Cannot login 10 connections	+ {instituicao}

- **Privilégios.**



Administrando acessos (GRANT)

Definição

São os privilégios de acesso aos objetos do banco de dados.

Privilégios:

-- tabela	-- function
-- coluna	-- language
-- sequence	-- large object
-- database	-- schema
-- domain	-- tablespace
-- foreign data wrapper	-- type
-- foreign server	



GRANT são privilégios de acesso que podem ser dados as nossas roles.

Administrando acessos (GRANT)

DATABASE

```
GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [,...] | ALL [ PRIVILEGES ] }
ON DATABASE database_name [,...]
TO role_specification [,...] [ WITH GRANT OPTION ]
```

SCHEMA

```
GRANT { { CREATE | USAGE } [,...] | ALL [ PRIVILEGES ] }
ON SCHEMA schema_name [,...]
TO role_specification [,...] [ WITH GRANT OPTION ]
```

TABLE

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
[,...] | ALL [ PRIVILEGES ] }
ON { [ TABLE ] table_name [,...]
| ALL TABLES IN SCHEMA schema_name [,...] }
TO role_specification [,...] [ WITH GRANT OPTION ]
```

DATABASE

grant [privilégios] on database [nome do banco] to [role especificada]

- **create:** criar banco de dados.
- **connect:** conectar no banco de dados.
- **Temporary:** criar objetos temporários.
- **Temp:** criar objetos temporários.
- **all:** todos os privilégios.

SCHEMA

grant [privilégios] on schema [nome do schema] to [role especificada]

- **create:** criar schema.
- **usage:** usar schema.
- **all:** todos os privilégios.

TABLE

grant [privilégios] on table [nome da tabela] all tables in schema [{todas as tabelas no schema} nome do schema] to [role especificada]

- **select**: fazer consulta na tabela.
- **insert**: inserir dados na tabela.
- **update**: alterar dados da tabela.
- **delete**: deletar dados da tabela.
- **truncate**: deletar todos os dados da tabela.
- **references**: criar relações entre chaves.
- **trigger**: criar gatilhos.
- **all**: todos os privilégios.

Comando **REVOKE**: remove os privilégios dados as roles.



Administrando acessos (GRANT)

DATABASE

```
REVOKE [ GRANT OPTION FOR ]
  { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
  ON DATABASE database_name [, ...]
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]
```

SCHEMA

```
REVOKE [ GRANT OPTION FOR ]
  { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
  ON SCHEMA schema_name [, ...]
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]
```

TABLE

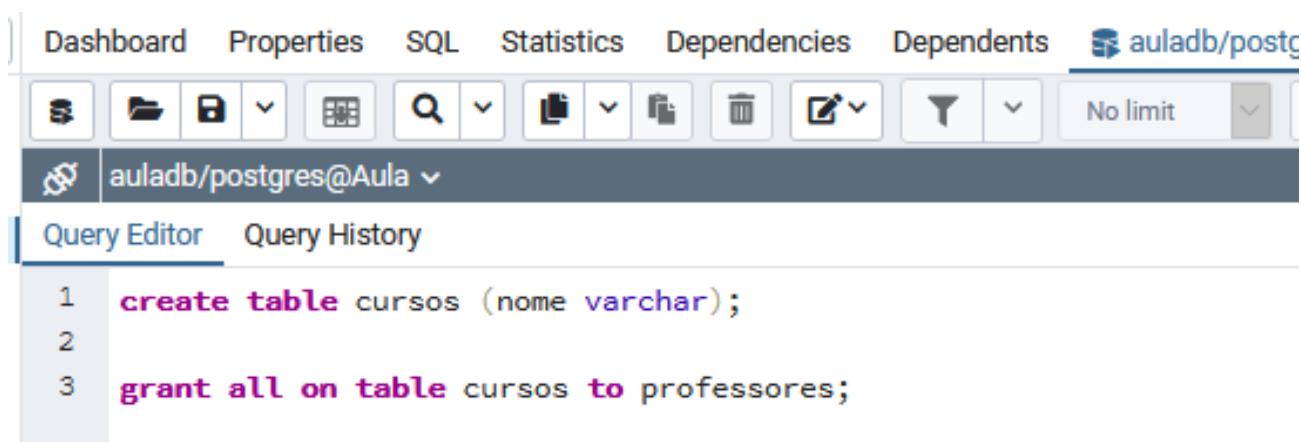
```
REVOKE [ GRANT OPTION FOR ]
  { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
  [, ...] | ALL [ PRIVILEGES ] }
  ON { [ TABLE ] table_name [, ...]
  | ALL TABLES IN SCHEMA schema_name [, ...] }
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]
```

Administrando acessos (GRANT)

REVOGANDO TODAS AS PERMISSÕES (SIMPLIFICADO)

```
REVOKE ALL ON ALL TABLES IN SCHEMA [schema] FROM [role];  
REVOKE ALL ON SCHEMA [schema] FROM [role];  
REVOKE ALL ON DATABASE [database] FROM [role];
```

Para exemplificar vamos criar uma tabela simples e dar permissões para a role professores.



The screenshot shows the pgAdmin 4 interface. At the top, there's a navigation bar with tabs: Dashboard, Properties, SQL, Statistics, Dependencies, Dependents, and a connection dropdown set to 'auladb/postgres@Aula'. Below the navigation bar is a toolbar with various icons for database management. The main area is titled 'Query Editor' and contains the following SQL code:

```
1 create table cursos (nome varchar);  
2  
3 grant all on table cursos to professores;
```

Nesse momento a role professores tem todos as permissões possíveis da tabela cursos, entretanto o usuário daniel que faz parte do grupo professores não, pois ao criar a role daniel não especificamos que ela herdaria todas as permissões de professores.

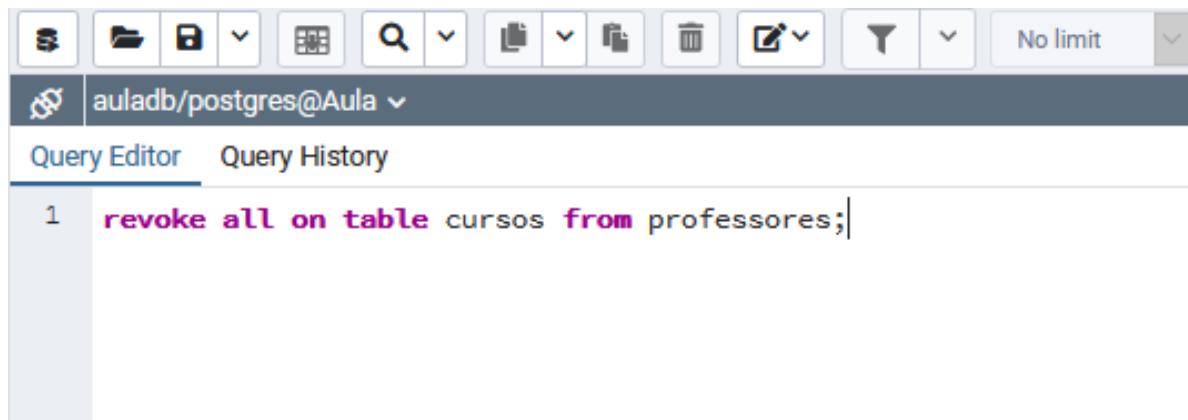
```
auladb=> SELECT nome FROM cursos;  
ERROR: permission denied for table cursos
```

Vamos alterar a role daniel e adicionar a opção **inherit** com o comando **alter role daniel inherit**; agora sim daniel tem as permissões da tabela.

```
auladb=> SELECT nome FROM cursos;  
 nome  
-----  
(0 rows)
```

Para tirar essas permissões de daniel é possível fazer um **revoke professores from daniel** o que tiraria daniel do grupo professores, ou alterar a role daniel com a opção **noinherit**.

Mas para tirar as permissões da tabela cursos da role professores e por consequencia de todas as roles que fazem parte dela vamos utilizar o seguinte comando **revoke all on table cursos from professores;**



```
1 revoke all on table cursos from professores;
```

Objetos e comandos



Database, Schemas e Objetos

Database

É o banco de dados.

Grupo de schemas e seus objetos, como tabelas, types, views, funções, entre outros.

Seus schemas e objetos não podem ser compartilhados entre si.

Cada database é separado um do outro compartilhando apenas usuários/roles e configurações do cluster PostgreSQL.

Schemas

É um grupo de objetos, como tabelas, types, views, funções, entre outros.

É possível relacionar objetos entre diversos schemas.

Por exemplo: schema public e schema curso podem ter tabelas com o mesmo nome (teste por exemplo) relacionando-se entre si.

Objetos

São as tabelas, views, funções, types, sequences, entre outros, pertencentes aos sch



Database, Schemas e Objetos

Database

```
CREATE DATABASE name
```

```
[ [ WITH ] [ OWNER [=] user_name ]  
[ TEMPLATE [=] template ]  
[ ENCODING [=] encoding ]  
[ LC_COLLATE [=] lc_collate ]  
[ LC_CTYPE [=] lc_ctype ]  
[ TABLESPACE [=] tablespace_name ]  
[ ALLOW_CONNECTIONS [=] allowconn ]  
[ CONNECTION LIMIT [=] connlimit ]  
[ IS_TEMPLATE [=] istemplate ] ]
```

```
ALTER DATABASE name RENAME TO new_name
```

```
ALTER DATABASE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

```
ALTER DATABASE name SET TABLESPACE new_tablespace
```

```
DROP DATABASE [nome]
```

- **create database (nome):** cria um banco de dados.
- **alter database (nome) rename to (novo nome):** altera o nome de um bando de dados.
- **drop database (nome):** exclui um banco de dados.

Database, Schemas e Objetos

Schema

```
CREATE SCHEMA schema_name [ AUTHORIZATION role_specification ]
```

```
ALTER SCHEMA name RENAME TO new_name
```

```
ALTER SCHEMA name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

```
DROP SCHEMA [nome]
```

- **create schema (nome):** cria um schema.
- **alter schema (nome) rename to (novo nome):** altera o nome de schema.
- **drop schema (nome):** exclui um schema.

Melhores práticas

```
CREATE SCHEMA IF NOT EXISTS schema_name [ AUTHORIZATION role_specification ]  
DROP SCHEMA IF EXISTS [nome];
```

É uma boa prática utilizar os comandos **if not exists** para criar um banco de dados ou schema e **if exists** para exclui-lo.

- **create database IF NOT EXISTS (nome):** cria um banco de dados se ele não existir.
- **drop database IF EXISTS (nome):** exclui um banco de dados se ele existir.



Tabelas, Colunas e Tipos de dados

Definição

Conjuntos de dados dispostos em colunas e linhas referentes a um objetivo comum.

As colunas são consideradas como “campos da tabela”, como atributos da tabela.

As linhas de uma tabela são chamadas também de tuplas, e é onde estão contidos os valores, os dados.



Exemplo:

TABELA = automovel

COLUNA 1 = tipo (carro, moto, aviao, helicoptero)

COLUNA 2 = ano_fabricacao (2010, 2011, 2020)

COLUNA 3 = capacidade_pessoas (1, 2, 350)

COLUNA 4 = fabricante (Honda, Avianca, Yamaha)

TABELA = produto

COLUNA 1 = codigo serial do produto

COLUNA 2 = tipo (vestuario, eletronico, beleza)

COLUNA 3 = preco

Tabelas, Colunas e Tipos de dados

Primary Key / Chave Primária / PK

No conceito de modelo de dados relacional e obedecendo as regras de normalização, uma PK é um conjunto de um ou mais campos que nunca se repetem em uma tabela e que seus valores garantem a integridade do dado único e a utilização do mesmo como referência para o relacionamento entre demais tabelas.

- não pode haver duas ocorrências de uma mesma entidade com o mesmo conteúdo na PK
- A chave primária não pode ser composta por atributo opcional, ou seja, atributo que aceite nulo.
- Os atributos identificadores devem ser o conjunto mínimo que pode identificar cada instância de um entidade.
- Não devem ser usadas chaves externas.
- Não deve conter informação volátil.



Foreign Key / Chave Estrangeira / FK

Campo, ou conjunto de campos que são referências de chaves primárias de outras tabelas ou da mesma tabela.

Sua principal função é garantir a integridade referencial entre tabelas.

Foreign Key / Chave Estrangeira / FK



No exemplo anterior nós temos 3 tabela, cliente, produto e pedido.

Na tabela pedido temos uma **pk = cpf**

Na tabela produto temos uma **pk = numero_serie**

Já na tabela pedido temos uma **pk = numero** e duas **fk** que fazem referencia aos dados das duas tabelas anteriores **fk = cliente_cpf / produto_numero_serie**

Foreign Key / Chave Estrangeira / FK

Cliente

CPF	NOME	TELEFONE
23433244567	Ermelino Carlos	83977863122
29555467899	Arlinda da Graça	15988900122
27335445017	Gumercindo Orlando	11988000900

Produto

NUMERO_SERIE	NOME	VALOR
10001	Camiseta	R\$ 59,90
10005	Caiça Jeans	R\$ 99,90
10099	Relógio de Ouro	R\$ 1515,90

Pedido

NUMERO	CLIENTE_CPF	PRODUTO_NUMERO_SERIE	VALOR	DESCONTO	
18015	23433244567	10001	R\$ 59,90		
18015	23433244567	10005	R\$ 99,90	R\$ 10,00	
18016	27335445017	10099	R\$ 1515,90		



Tabelas, Colunas e Tipos de dados

Numeric Types

Monetary Types

Character Types

Binary Data Types

Date/Time Types

Boolean Type

Enumerated Types

Geometric Types

Network Address Types

Bit String Types

Text Search Types

UUID Type

XML Type

JSON Types

Arrays

Composite Types

Range Types

Domain Types

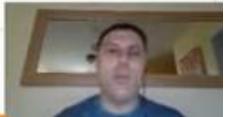
Object Identifier Types

pg_lsn Type

Pseudo-Types

Numéricos

Name	Storage Size	Description	Range
smallint	2 bytes	small-range integer	-32768 to +32767
integer	4 bytes	typical choice for integer	-2147483648 to +2147483647
bigint	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
decimal	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
numeric	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
real	4 bytes	variable-precision, inexact	6 decimal digits precision
double precision	8 bytes	variable-precision, inexact	15 decimal digits precision
smallserial	2 bytes	small autoincrementing integer	1 to 32767
serial	4 bytes	autoincrementing integer	1 to 2147483647
bigserial	8 bytes	large autoincrementing integer	1 to 9223372036854775807



Datas

Name	Storage Size	Description	Low Value	High Value	Resolution
timestamp [(p)] [without time zone]	8 bytes	both date and time (no time zone)	4713 BC	294276 AD	1 microsecond
timestamp [(p)] with time zone	8 bytes	both date and time, with time zone	4713 BC	294276 AD	1 microsecond
date	4 bytes	date (no time of day)	4713 BC	5874897 AD	1 day
time [(p)] [without time zone]	8 bytes	time of day (no date)	00:00:00	24:00:00	1 microsecond
time [(p)] with time zone	12 bytes	time of day (no date), with time zone	00:00:00+1459	24:00:00-1459	1 microsecond
interval [fields] [(p)]	16 bytes	time interval	-178000000 years	178000000 years	1 microsecond

Caracteres

Name	Description
character varying(n), varchar(n)	variable-length with limit
character(n), char(n)	fixed-length, blank padded
text	variable unlimited length

Booleanos

Name	Storage Size	Description
boolean	1 byte	state of true or false

DML e DDL

DML

Data Manipulation Language

Linguagem de manipulação de dados

INSERT, UPDATE, DELETE, **SELECT**

* o select, alguns consideram como DML, outros como DQL, que significa data query language, ou linguagem de consulta de dados

DDL

Data Definition Language

Linguagem de definição de dados

CREATE, ALTER, DROP



DML e DDL

CREATE / ALTER / DROP - TABELAS

```
CREATE TABLE IF NOT EXISTS banco (
    codigo INTEGER PRIMARY KEY,
    nome VARCHAR(50) NOT NULL,
    data_criacao TIMESTAMP NOT NULL DEFAULT NOW()
);
```

```
CREATE TABLE IF NOT EXISTS banco (
    codigo INTEGER,
    nome VARCHAR(50) NOT NULL,
    data_criacao TIMESTAMP NOT NULL DEFAULT NOW(),
    PRIMARY KEY (codigo)
);
```

```
ALTER TABLE banco ADD COLUMN tem_poupanca BOOLEAN;
```

```
DROP TABLE IF EXISTS banco;
```

DML e DDL

INSERT

```
INSERT INTO [nome da tabela] ([campos da tabela,])
VALUES ([valores de acordo com a ordem dos campos acima,]);
```

```
INSERT INTO [nome da tabela] ([campos da tabela,])
SELECT [valores de acordo com a ordem dos campos acima,];
```

DML e DDL

INSERT

```
INSERT INTO banco (codigo, nome, data_criacao)
VALUES (100, 'Banco do Brasil', now());
```

```
INSERT INTO banco (codigo, nome, data_criacao)
SELECT 100, 'Banco do Brasil', now();
```

DML e DDL

UPDATE

```
UPDATE [nome da tabela] SET
[campo1] = [novo valor do campo1],
[campo2] = [novo valor do campo2],
...
[WHERE + condições]
```

ATENÇÃO: muito cuidado com os updates. Sempre utilize-os com condição.

DML e DDL

UPDATE

```
UPDATE banco SET  
codigo = 500  
WHERE codigo = 100;
```

```
UPDATE banco SET  
data_criacao = now()  
WHERE data_criacao IS NULL;
```

DML e DDL

DELETE

```
DELETE FROM [nome da tabela]  
[WHERE + condições]
```

ATENÇÃO: muito cuidado com os deletes. Sempre utilize-os com condição.

DML e DDL

DELETE

```
DELETE FROM banco  
WHERE codigo = 512;
```

```
DELETE FROM banco  
WHERE nome = 'Conta Digital';
```

SELECT

```
SELECT [campos da tabela]  
FROM [nome da tabela]  
[WHERE + condicoes]
```

DICAS DE BOAS PRÁTICAS = Evite sempre que puder o SELECT *

SELECT

```
SELECT codigo, nome  
FROM banco;
```

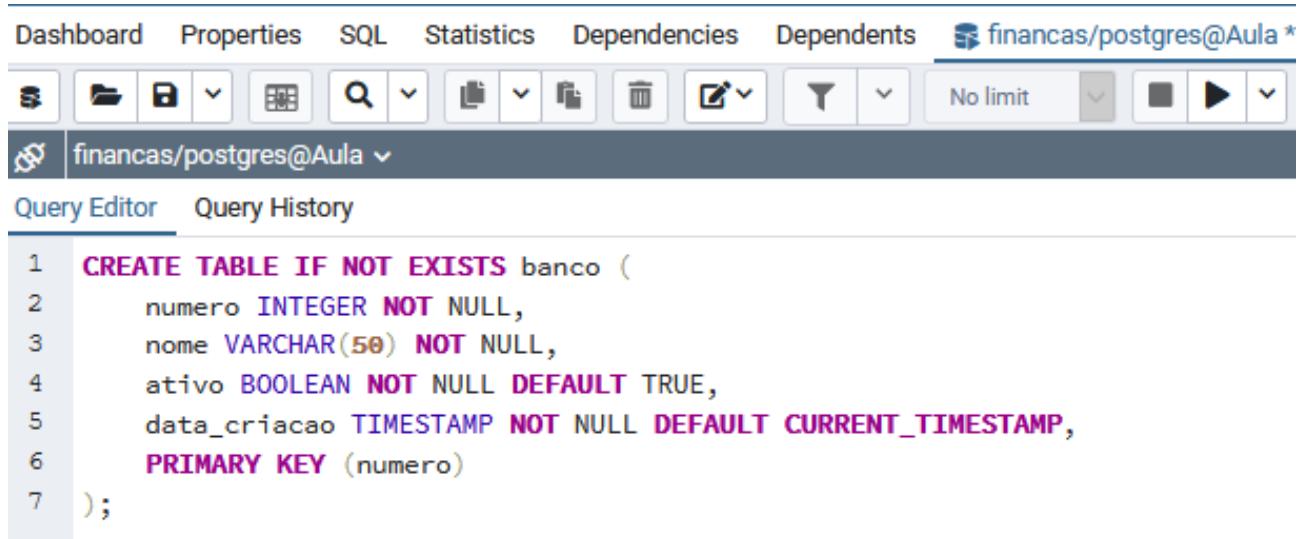
```
SELECT codigo, nome  
FROM banco  
WHERE data_criacao > '2019-10-15 15:00:00';
```

Banco de dados – finanças.

Para iniciar nosso projeto vamos criar um banco de dados de finanças, criar tabelas com os comandos DDL e inserir e manipular dados nessas tabelas com os comandos DML.

Para criar o banco de dados finanças vamos abrir o **query tools** e utilizar o comando **create database finanças;**

Vamos criar algumas tabelas dentro do nosso banco finanças.



The screenshot shows the pgAdmin interface with the following details:

- Toolbar: Dashboard, Properties, SQL, Statistics, Dependencies, Dependents, finanças/postgres@Aula *
- File menu icons: New, Open, Save, Print, Find, Copy, Paste, Delete, Import, Export, Properties, Help.
- Search bar: No limit.
- Table navigation icons: Back, Forward, Home, Refresh, Stop, Run, Stop, Refresh, Stop, Refresh.
- Session bar: finanças/postgres@Aula.
- Bottom tabs: Query Editor (selected), Query History.
- Query Editor content:

```
1 CREATE TABLE IF NOT EXISTS banco (
2     numero INTEGER NOT NULL,
3     nome VARCHAR(50) NOT NULL,
4     ativo BOOLEAN NOT NULL DEFAULT TRUE,
5     data_criacao TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
6     PRIMARY KEY (numero)
7 );
```

Aqui criamos a tabela **banco** com os campos:

numero: número inteiro, não pode ser nulo.

nome: caracteres com 50 posições, não pode ser nulo.

ativo: um boolean para identificar se o banco está ativo = verdadeiro ou não = falso, não pode ser nulo e tem o valor padrão = true(verdadeiro).

data_criacao: campo do tipo data que guarda a data e horário de criação.

primary key: chave primaria da tabela.

```
8
9 CREATE TABLE IF NOT EXISTS agencia (
10     banco_numero INTEGER NOT NULL,
11     numero INTEGER NOT NULL,
12     nome VARCHAR(80),
13     ativo BOOLEAN NOT NULL DEFAULT TRUE,
14     data_criacao TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
15     PRIMARY KEY (banco_numero, numero),
16     FOREIGN KEY (banco_numero) REFERENCES banco (numero)
17 );
18
```

Essa é a criação da tabela **agencia**, iremos daqui em diante falar apenas dos campos necessários.

banco_numero: esse campo fará referência ao campo **numero** da tabela **banco** por tanto precisa ter o mesmo tipo.

primary key: chave primária da tabela composta por 2 campos.

foreign key: chave estrangeira que faz referência a tabela **banco** através da sua chave primária.

```
19
20 CREATE TABLE clientes (
21     numero BIGSERIAL PRIMARY KEY,
22     nome VARCHAR(100) NOT NULL,
23     email VARCHAR(100) NOT NULL,
24     ativo BOOLEAN NOT NULL DEFAULT TRUE,
25     data_criacao TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
26 );
27
```

Tabela **clientes**.

numero: campo do tipo numérico, chave primária dessa tabela.

The screenshot shows the pgAdmin 4 interface with the following details:

- Toolbar:** Includes icons for file operations (New, Open, Save, Print, etc.), search, filter, and various database management functions.
- Connection Bar:** Shows the connection to "financas/postgres@Aula".
- Query Editor:** Contains the SQL code for creating the "conta_corrente" table. The code is as follows:

```
29 CREATE TABLE conta_corrente (
30     banco_numero INTEGER NOT NULL,
31     agencia_numero INTEGER NOT NULL,
32     numero BIGINT NOT NULL,
33     digito SMALLINT NOT NULL,
34     clientes_numero BIGINT NOT NULL,
35     ativo BOOLEAN NOT NULL DEFAULT TRUE,
36     data_criacao TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
37     PRIMARY KEY (banco_numero, agencia_numero, numero, digito, clientes_numero),
38     FOREIGN KEY (banco_numero, agencia_numero) REFERENCES agencia (banco_numero, numero),
39     FOREIGN KEY (clientes_numero) REFERENCES clientes (numero)
40 );
```

Below the Query Editor, there are tabs for Data Output, Explain, Messages, and Notifications.

Criação da tabela **conta_corrente**.

banco_numero: faz referência a tabela banco;

agencia_numero: faz referência a tabela agencia.

clientes_numero: faz referência a tabela clientes.

primary key: chave primaria da tabela composta por seus campos únicos e chaves estrangeiras que referencia outras tabelas.

foreign key: chave estrangeira que faz referência a tabela agencia através da sua chave primaria, chave essa composta com referência a tabela banco.

foreign key: chave estrangeira que sobrou fazendo referência a tabela clientes através de sua chave primaria.

The screenshot shows the pgAdmin 4 interface with the following details:

- Toolbar:** Includes icons for file operations (New, Open, Save, Print, etc.), search, filter, and various database management functions.
- Connection Bar:** Shows the connection to "financas/postgres@Aula".
- Query Editor:** Contains the SQL code for creating the "tipo_transacao" table. The code is as follows:

```
40 );
41
42
43 CREATE TABLE tipo_transacao (
44     id SMALLSERIAL PRIMARY KEY,
45     nome VARCHAR(50) NOT NULL,
46     ativo BOOLEAN NOT NULL DEFAULT TRUE,
47     data_criacao TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
48 );
49
50
```

Criação da tabela **tipo_transacao**.

The screenshot shows a PostgreSQL Query Editor interface. The title bar indicates the connection is to 'financas/postgres@Aula'. The toolbar has various icons for database management. The main area is a code editor with the following SQL script:

```
50  
51 CREATE TABLE cliente_transacoes (  
52     id BIGSERIAL PRIMARY KEY,  
53     banco_numero INTEGER NOT NULL,  
54     agencia_numero INTEGER NOT NULL,  
55     conta_corrente_numero BIGINT NOT NULL,  
56     conta_corrente_digito SMALLINT NOT NULL,  
57     clientes_numero BIGINT NOT NULL,  
58     tipo_transacao SMALLINT NOT NULL,  
59     valor NUMERIC(15,2),  
60     data_criacao TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,  
61     FOREIGN KEY (banco_numero, agencia_numero, conta_corrente_numero, conta_corrente_digito, clientes_numero)  
62         REFERENCES conta_corrente (banco_numero, agencia_numero, numero, digito, clientes_numero)  
63 );
```

Criação da tabela **cliente_transacoes**.

id: chave primaria da tabela.

banco_numero: faz referência a tabela banco;

agencia_numero: faz referência a tabela agencia;

conta_corrente_numero: faz referência a tabela conta_corrente;

conta_corrente_digito: faz referência a conta_corrente;

clientes_numero: faz referência a tabela clientes;

tipo_transacao: faz referência a tabela transacao;

foreign key: criação da chave estrangeira que faz referência a tabela conta_corrente através da sua chave primaria, chave essa composta com referência a outras tabelas.

Fundamentos SQL



Revisão

- **Melhores práticas em DDL**

Importante as tabelas possuírem campos que realmente serão utilizados e que sirvam de atributo direto a um objetivo em comum.

- Criar/Acrecentar colunas que são “atributos básicos” do objeto;
Exemplo: tabela CLIENTE : coluna TELEFONE / coluna AGENCIA_BANCARIA
- Cuidado com regras (constraints)
- Cuidado com o excesso de FKs
- Cuidado com o tamanho indevido de colunas
Exemplo: coluna CEP VARCHAR(255)



DML - CRUD

SELECT

```
SELECT (campos,)  
FROM tabela  
[condições]
```

Exemplo:

```
SELECT numero, nome FROM banco;  
SELECT numero, nome FROM banco WHERE ativo IS TRUE;  
SELECT nome FROM cliente WHERE email LIKE '%gmail.com';
```

```
SELECT numero FROM agencia  
WHERE banco_numero IN (SELECT numero FROM banco WHERE nome ILIKE '%Bradesco');
```



DML: Data manipulation language.

CRUD: Create, Read, Update, Delete.

SELECT - Condição (WHERE / AND / OR)

WHERE (coluna/condição) :

- =
- > / >=
- < / <=
- <> / !=
- LIKE
- ILIKE
- IN

Primeira condição sempre **WHERE**.

Demais condições, **AND** ou **OR**.

SELECT - Idempotência

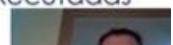
```
SELECT (campos,  
FROM tabela1  
WHERE EXISTS (  
    SELECT (campo,)  
    FROM tabela2  
    WHERE campo1 = valor1  
    [AND/OR campoN = valorN]  
);
```

Não é uma boa prática.

Melhor prática utilizar LEFT JOIN.

IDEMPOTÊNCIA

Propriedade que algumas ações/operações possuem possibilitando-as de serem executadas diversas vezes sem alterar o resultado após a aplicação inicial.



IDEMPOTÊNCIA

- IF EXISTS
- Comandos pertinentes ao DDL e DML

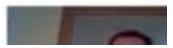
INSERT - IDEMPOTÊNCIA

```
INSERT INTO agencia (banco_numero, numero, nome) VALUES (341,1,'Centro da cidade');
```

```
INSERT INTO agencia (banco_numero, numero, nome)
SELECT 341,1,'Centro da cidade'
WHERE NOT EXISTS (SELECT banco_numero, numero, nome FROM agencia WHERE
banco_numero = 341 AND numero = 1 AND nome = 'Centro da cidade');
```

ON CONFLICT

```
INSERT INTO agencia (banco_numero, numero, nome) VALUES (341,1,'Centro da cidade') ON
CONFLICT (banco_numero, numero) DO NOTHING;
```



Uma boa prática para se trabalhar idempotência é o **on conflict**. Nesse código podemos dizer a inserir dados na tabela que se houver algum conflito na chave primária não faça nada.

Definição

“Esvazia” a tabela.

```
TRUNCATE [ TABLE ] [ ONLY ] name [ * ] [, ... ]
[ RESTART IDENTITY | CONTINUE IDENTITY ] [ CASCADE | RESTRICT ]
```

restart identity: Reinicia a contagem do campo sequencial da tabela.

continue identity (padrão): Continua a contagem do campo sequencial da tabela do último registro.

cascade: Ao limpar a tabela apaga também as referências de chaves estrangeiras de outras tabelas.

restrict (padrão): não permite limpar a tabela quando existem chaves estrangeiras.

Vamos realizar algumas inserções no banco **finanças** que criamos.

```
1 INSERT INTO banco (numero, nome) VALUES (010203, 'BANCO TESTE INSERT');
2
3 SELECT numero, nome, ativo, data_criacao FROM banco WHERE numero = 010203;
```

	numero [PK] integer	nome character varying (50)	ativo boolean	data_criacao timestamp without time zone	
1	10203	BANCO TESTE INSERT	true	2020-12-13 14:08:14.549406	

Aqui fizemos a inserção na tabela **banco** e em seguida um **select** na tabela trazendo as informações do banco que acabamos de inserir com clausula **where**.

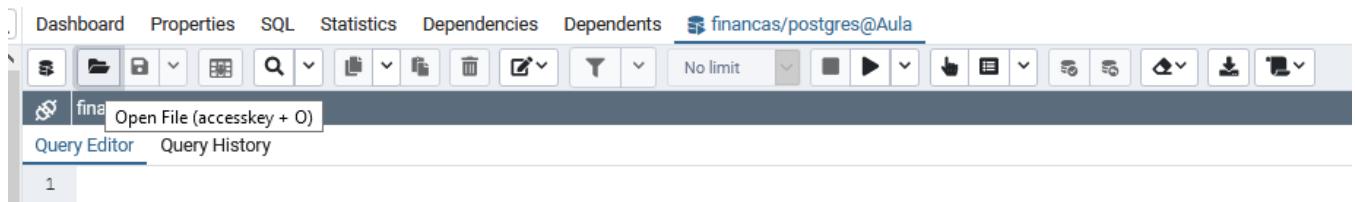
```
1 INSERT INTO banco (numero, nome) VALUES (010203, 'BANCO TESTE INSERT');
2
3 SELECT numero, nome, ativo, data_criacao FROM banco WHERE numero = 010203;
4
5 UPDATE banco SET ativo = false WHERE numero = 010203;
```

	numero [PK] integer	nome character varying (50)	ativo boolean	data_criacao timestamp without time zone	
1	10203	BANCO TESTE INSERT	false	2020-12-13 14:08:14.549406	

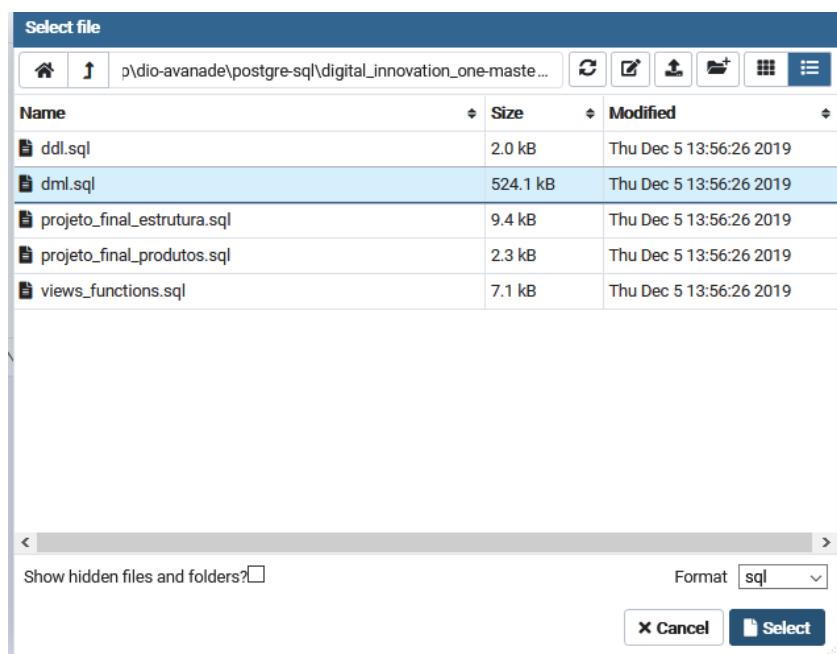
Aqui fizemos uma alteração (**update**) nos dados da tabela especificada pela clausula **where**, e em seguida um **select** novamente mostra a modificação feita na coluna **ativo**.

Vamos utilizar um arquivo para inserir todos os dados que iremos precisar para praticar. Os arquivos estão em https://github.com/drobcosta/digital_innovation_one vamos fazer o download e importar para o nosso pg admin.

Para importar o arquivo vamos no ícone da pastinha (open file).



Ao abrir a caixinha selecionamos o arquivo **dml.sql** no destino e clicamos em select.



```
1 INSERT INTO banco (numero, nome) VALUES (654, 'Banco A.J.Renner S.A.'::VARCHAR(50));
2 INSERT INTO banco (numero, nome) VALUES (246, 'Banco ABC Brasil S.A.'::VARCHAR(50));
3 INSERT INTO banco (numero, nome) VALUES (25, 'Banco Alfa S.A.'::VARCHAR(50));
4 INSERT INTO banco (numero, nome) VALUES (641, 'Banco Alvorada S.A.'::VARCHAR(50));
5 INSERT INTO banco (numero, nome) VALUES (213, 'Banco Arbi S.A.'::VARCHAR(50));
6 INSERT INTO banco (numero, nome) VALUES (19, 'Banco Azteca do Brasil S.A.'::VARCHAR(50));
7 INSERT INTO banco (numero, nome) VALUES (29, 'Banco Banerj S.A.'::VARCHAR(50));
8 INSERT INTO banco (numero, nome) VALUES (0, 'Banco Bankpar S.A.'::VARCHAR(50));
9 INSERT INTO banco (numero, nome) VALUES (740, 'Banco Barclays S.A.'::VARCHAR(50));
10 INSERT INTO banco (numero, nome) VALUES (107, 'Banco BBM S.A.'::VARCHAR(50));
11 INSERT INTO banco (numero, nome) VALUES (31, 'Banco Beg S.A.'::VARCHAR(50));
12 INSERT INTO banco (numero, nome) VALUES (739, 'Banco BGN S.A.'::VARCHAR(50));
13 INSERT INTO banco (numero, nome) VALUES (96, 'Banco BM&F de Serviços de Liquidação e Custódia S.A.'::VARCHAR(50));
14 INSERT INTO banco (numero, nome) VALUES (318, 'Banco BMG S.A.'::VARCHAR(50));
```

The pgAdmin interface is shown again, this time with the 'Query Editor' tab selected. The main pane displays the 14 'INSERT INTO' statements for the 'banco' table. At the bottom of the interface are tabs for 'Data Output', 'Explain', 'Messages', and 'Notifications'.

Agora é só executar todo o código para inserir os dados na tabela.

Caso ainda não tenha criado as tabelas basta importar e executar o arquivo **ddl.sql**

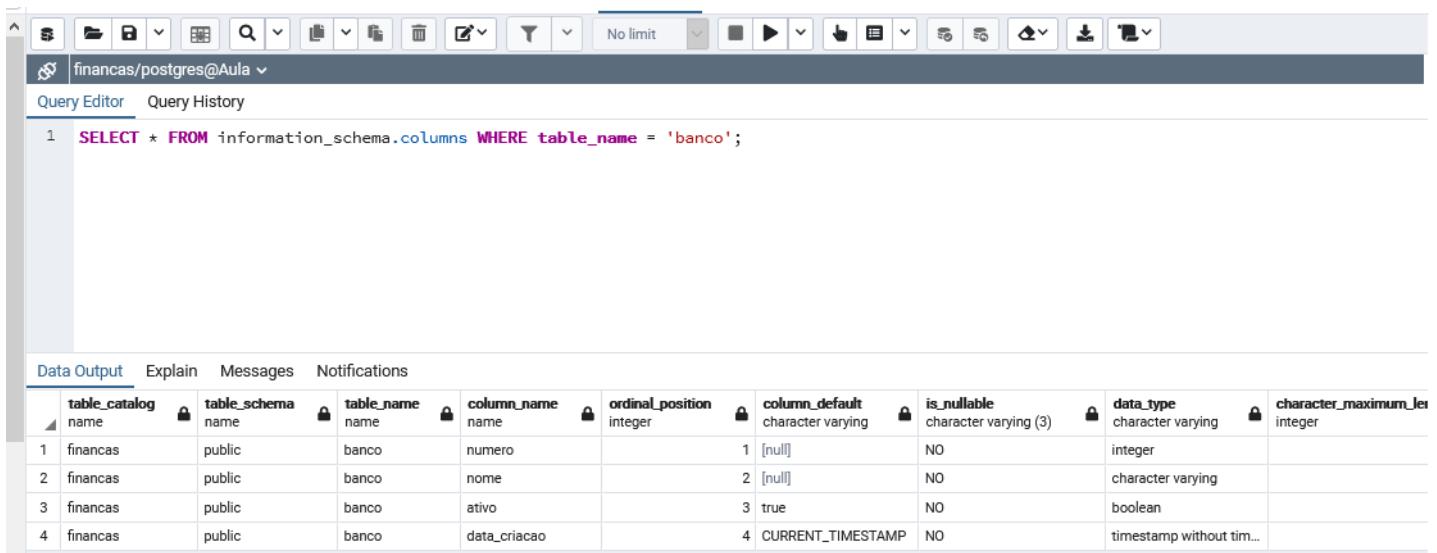
Após isso vamos fazer alguns **SELECTs** nas tabelas e ver o resultado diretamente no pgadmin.

The screenshot shows the pgAdmin Query Editor interface. The title bar says "financas/postgres@Aula". The "Query Editor" tab is selected. Below it, there is a list of nine SELECT statements numbered 1 to 9. Statement 1 is highlighted. The "Data Output" tab is selected, showing a table with three columns: "numero" (PK integer), "nome" (character varying(50)), and "data_criacao" (timestamp without time zone). The table contains four rows of data.

	numero [PK] integer	nome character varying (50)	data_criacao timestamp without time zone
1	654	Banco A.J.Renner S.A.	2020-12-10 23:11:45.414778
2	246	Banco ABC Brasil S.A.	2020-12-10 23:11:45.414778
3	25	Banco Alfa S.A.	2020-12-10 23:11:45.414778
4	641	Banco Alvorada S.A.	2020-12-10 23:11:45.414778

1. **SELECT numero, nome, ativo FROM banco;**
2. **SELECT numero, nome, data_criacao FROM banco WHERE ativo = true;**
3. **SELECT numero, nome, data_criacao FROM banco WHERE ativo = false;**
4. **SELECT banco_numero, numero, nome FROM agencia;**
5. **SELECT numero, nome, email FROM cliente;**
6. **SELECT id, nome FROM tipo_transacao;**
7. **SELECT banco_numero, agencia_numero, numero, cliente_numero
FROM conta_corrente;**
8. **SELECT banco_numero, agencia_numero, cliente_numero FROM
cliente_transacoes;**
9. **SELECT * FROM cliente_transacoes;**

Para ter acesso as informações das tabelas como, colunas, tipos de dados, schema, entre outras é possível utilizar a view do postgres **information_schema** como nos exemplos a seguir.

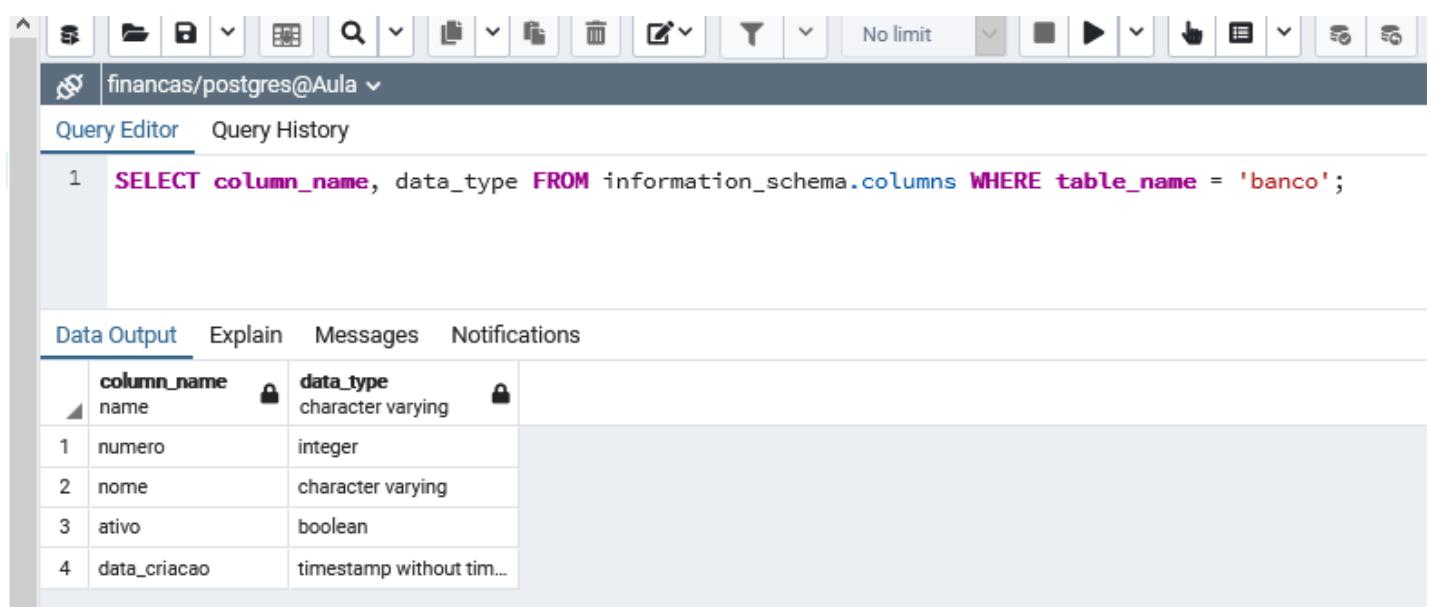


The screenshot shows the pgAdmin interface with the following details:

- Toolbar:** Standard pgAdmin toolbar with various icons for file operations, search, and database management.
- Status Bar:** Shows the connection information: `financas/postgres@Aula`.
- Query Editor:** The active tab, showing the query:

```
1  SELECT * FROM information_schema.columns WHERE table_name = 'banco';
```
- Data Output:** The results of the query are displayed in a table:

	table_catalog name	table_schema name	table_name name	column_name name	ordinal_position integer	column_default character varying	is_nullable character varying (3)	data_type character varying	character_maximum_lei integer
1	financas	public	banco	numero		1 [null]	NO	integer	
2	financas	public	banco	nome		2 [null]	NO	character varying	
3	financas	public	banco	ativo		3 true	NO	boolean	
4	financas	public	banco	data_criacao		4 CURRENT_TIMESTAMP	NO	timestamp without tim...	



The screenshot shows the pgAdmin interface with the following details:

- Toolbar:** Standard pgAdmin toolbar with various icons for file operations, search, and database management.
- Status Bar:** Shows the connection information: `financas/postgres@Aula`.
- Query Editor:** The active tab, showing the query:

```
1  SELECT column_name, data_type FROM information_schema.columns WHERE table_name = 'banco';
```
- Data Output:** The results of the query are displayed in a table:

	column_name name	data_type character varying
1	numero	integer
2	nome	character varying
3	ativo	boolean
4	data_criacao	timestamp without tim...

Funções Agregadas

- **AVG**
- **COUNT (opção: HAVING)**
- **MAX**
- **MIN**
- **SUM**

<https://www.postgresql.org/docs/11/functions-aggregate.html>

AVG: Retorna a média dos valores requisitados.

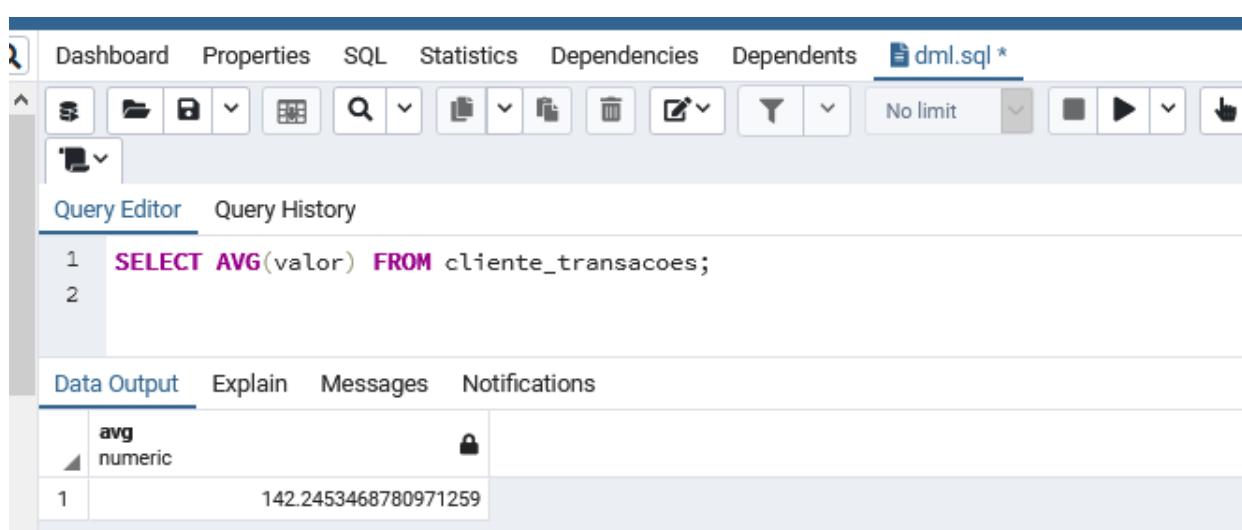
COUNT(having): Retorna a quantidade de registros selecionados

MAX: Retorna o maior número dos valores requisitados.

MIN: Retorna o menor número dos valores requisitados.

SUM: Retorna a soma dos valores requisitados.

AVG:



The screenshot shows a PostgreSQL query editor interface. The top navigation bar includes 'Dashboard', 'Properties', 'SQL', 'Statistics', 'Dependencies', 'Dependents', and a file tab labeled 'dml.sql *'. Below the navigation is a toolbar with various icons for database management. The main area is divided into 'Query Editor' and 'Query History' tabs, with 'Query Editor' selected. A SQL query is entered: 'SELECT AVG(valor) FROM cliente_transacoes;'. The results section, titled 'Data Output', displays a single row with the column name 'avg' and its value '142.2453468780971259'. Other tabs like 'Explain', 'Messages', and 'Notifications' are also visible.

avg	
numeric	
1	142.2453468780971259

Retorna a média da coluna valores na tabela cliente_transacoes.

COUNT (HAVING):

```
6 -- select count - having
7
8 SELECT COUNT(numero) FROM cliente;
```

Data Output Explain Messages Notifications

	count bigint
1	500

Retorna a quantidade de registros na coluna numero da tabela cliente.

```
22 SELECT COUNT(numero), email FROM cliente WHERE email ILIKE '%gmail.com'
23 GROUP BY email;
24
```

Data Output Explain Messages Notifications

	count bigint	email character varying (250)
1	1	piedade_cordeiro@gmail.com
2	1	simeao_simões@gmail.com
3	1	emilio_morais@gmail.com
4	1	filena_blanco@gmail.com
5	1	ubirata_marmou@gmail.com

Retorna a quantidade de email onde o email seja gmail.com como especifica o **ILIKE**.

Como as funções agregadas são funções

de agrupamento é preciso agrupar a coluna email com **GROUP BY** para que as colunas resitadas sejam executadas em conjunto como se fosse apenas uma.

```
24
25 SELECT COUNT(id), tipo_transacao_id FROM cliente_transacoes
26 GROUP BY tipo_transacao_id;
27
```

Data Output Explain Messages Notifications

	count bigint	tipo_transacao_id smallint
1	640	1
2	278	3
3	960	2
4	140	4

Retorna a quantidade de transações por tipo de transação.

```
27  
28 SELECT COUNT(id), tipo_transacao_id FROM cliente_transacoes  
29 GROUP BY tipo_transacao_id  
30 HAVING COUNT(id) > 150;  
31
```

Data Output Explain Messages Notifications

	count bigint	tipo_transacao_id smallint	
1	640		1
2	278		3
3	960		2

Nesse caso com a cláusula **HAVING** só irá retornar os registros que tem essa contagem maior que 150.

MAX:

```
24  
25 -- MAX  
26  
27 SELECT MAX(numero) FROM cliente;
```

Data Output Explain Messages Notifications

	max bigint
1	500

Retorna o maior registro na coluna numero da tabela cliente.

```
28  
29 SELECT MAX(valor) FROM cliente_transacoes;  
30
```

Data Output Explain Messages Notifications

	max numeric
1	859.85

Retorna o maior registro na coluna valor da tabela cliente_transacoes ou seja o maior valor de transações registrado.

```
30
31 SELECT MAX(valor), tipo_transacao_id FROM cliente_transacoes
32 GROUP BY tipo_transacao_id;
```

Data Output Explain Messages Notifications

	max numeric	tipo_transacao_id smallint
1	270.19	1
2	100.50	3
3	859.85	2
4	76.32	4

Retorna o maior valor de cada tipo de transação registrados.

MIN:

```
30 -- MIN
31
32 SELECT MIN(numero) FROM cliente;
33
34
```

Data Output Explain Messages Notifications

	min bigint
1	1

Retorna o menor registro na coluna numero da tabela cliente.

```
34
35 SELECT MIN(valor) FROM cliente_transacoes;
```

Data Output Explain Messages Notifications

	min numeric
1	0.02

Retorna o menor registro na coluna valor da tabela cliente_transacoes ou seja o menor valor de transações registrado.

```
40 SELECT MIN(valor), tipo_transacao_id FROM cliente_transacoes  
41 GROUP BY tipo_transacao_id;  
42
```

Data Output Explain Messages Notifications

	min numeric	tipo_transacao_id smallint	
1	0.05	1	
2	0.87	3	
3	0.29	2	
4	0.02	4	

Retorna o menor valor de cada tipo de transação registrados.

SUM:

```
51 -- SUM  
52  
53 SELECT SUM(valor) FROM cliente_transacoes;  
54
```

Data Output Explain Messages Notifications

	sum numeric
1	287051.11

Retorna a soma de todas as transações.

```
55 SELECT SUM(valor), tipo_transacao_id FROM cliente_transacoes  
56 GROUP BY tipo_transacao_id;  
57
```

Data Output Explain Messages Notifications

	sum numeric	tipo_transacao_id smallint	
1	49484.52	1	
2	14192.55	3	
3	220486.93	2	
4	2887.11	4	

Retorna a soma das transações por tipo de transação.

Para ordenar valores é só utilizar o comando **order by tipo_transacao_id;**

O **order by** por padrão ordena de forma ascendente, para ordenar de forma descendente é só utilizar o comando **desc** ficando:

order by tipo_transacao_id desc;

Relacionamentos entre tabelas

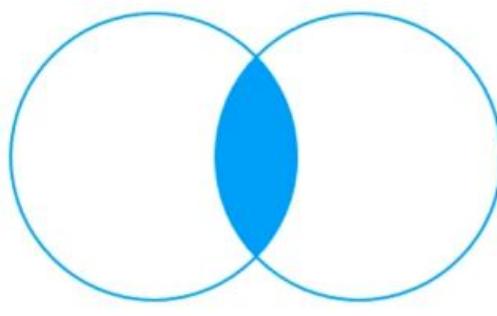


JOINs

Definição

- JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN
- CROSS JOIN

JOIN (INNER)



INNER JOIN

Retorna campos em comum em duas tabelas diferentes.

JOIN (INNER)

```
SELECT tabela_1.campos, tabela_2.campos  
FROM tabela_1  
JOIN tabela_2  
ON tabela_2.campo = tabela_1.campo
```

JOIN (INNER)

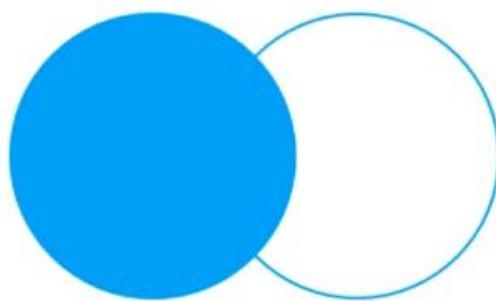
ID	NOME
1	Genivaldo
2	Emengarda
3	Hercules

ID	TBL1_ID	VALOR
1	1	R\$ 10,00
2	1	R\$ 15,00
3	3	R\$ 50,00

TBL_1	TBL_2
Genivaldo	R\$ 10,00
Genivaldo	R\$ 15,00
Hercules	R\$ 50,00



LEFT JOIN (OUTER)



LEFT OUTER JOIN

No relacionamento entre duas tabelas A e B o left join retorna toda tabela A por completo e a tabela B se houver um relacionamento irá retornar os dados, se não retornará nulo.

LEFT JOIN (OUTER)

```
SELECT tabela_1.campos, tabela_2.campos
FROM tabela_1
LEFT JOIN tabela_2
ON tabela_2.campo = tabela_1.campo
```

LEFT JOIN (OUTER)

ID	NOME
1	Genivaldo
2	Emengarda
3	Hercules

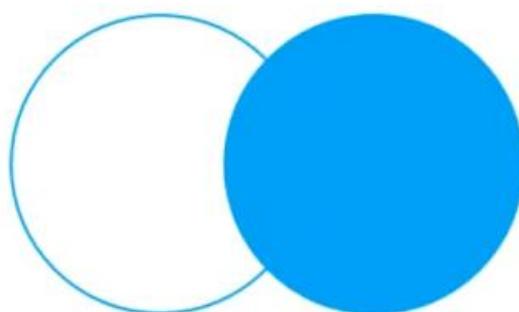
ID	TBL1_ID	VALOR
1	1	R\$ 10,00
2	1	R\$ 15,00
3	3	R\$ 50,00

TBL_1	TBL_2
Genivaldo	R\$ 10,00
Genivaldo	R\$ 15,00
Emengarda	
Hercules	R\$ 50,00



JOINs

RIGHT JOIN (OUTER)



RIGHT OUTER JOIN

Mesmo raciocínio do LEFT JOIN porém dará mais prioridade a tabela B.

RIGHT JOIN (OUTER)

```
SELECT tabela_1.campos, tabela_2.campos
FROM tabela_1
RIGHT JOIN tabela_2
ON tabela_2.campo = tabela_1.campo
```

RIGHT JOIN (OUTER)

ID	PRODUTO
1	Camiseta Polo
2	Calça Jeans
3	Saia longa

ID	TAMANHO
1	P
2	M
3	G

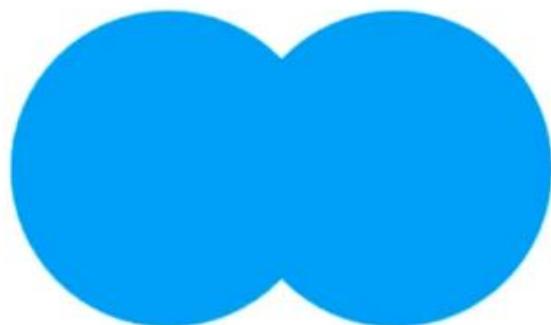
TBL_1	TBL_2
1	2
1	3
2	2

TBL_1	TBL_2
Camiseta Polo	M
Camiseta Polo	G
Calça Jeans	M
	P



JOINS

FULL JOIN



FULL OUTER JOIN

Retorna todas as relações possíveis.

FULL JOIN

```
SELECT tabela_1.campos, tabela_2.campos  
FROM tabela_1  
FULL JOIN tabela_2  
ON tabela_2.campo = tabela_1.campo
```

FULL JOIN

ID	PRODUTO
1	Camiseta Polo
2	Calça Jeans
3	Saia longa

ID	TAMANHO
1	P
2	M
3	G

TBL_1	TBL_2
Camiseta Polo	M
Camiseta Polo	G
Calça Jeans	M
Saia Longa	
	P



CROSS JOIN

Todos os dados de uma tabela serão cruzados com todos os dados da tabela referenciada no CROSS JOIN criando uma matriz.

```
SELECT tabela_1.campos, tabela_2.campos  
FROM tabela_1  
CROSS JOIN tabela_2
```

CROSS JOIN

Todos os dados de uma tabela serão cruzados com todos os dados da tabela referenciada no CROSS JOIN criando uma matriz.

ID	NOME
1	José Colméia
2	Airton Senna

ID	VALOR
1	R\$ 10,00
2	R\$ 15,00
3	R\$ 25,00



TBL 1 = ID	TBL 1 = NOME	TBL 2 = ID	TBL 2 = VALOR
1	José Colméia	1	R\$ 10,00
1	José Colméia	2	R\$ 15,00
1	José Colméia	3	R\$ 25,00
2	Airton Senna	1	R\$ 10,00
2	Airton Senna	2	R\$ 15,00
2	Airton Senna	3	R\$ 25,00



Prática

INNER JOIN

Vamos ao nosso banco finanças e relacionar algumas tabelas.

```
8 -- INNER JOIN
9
10 SELECT banco.numero, banco.nome, agencia.numero, agencia.nome
11 FROM banco
12 JOIN agencia ON agencia.banco_numero = banco.numero;
13
14
```

Data Output Explain Messages Notifications

	numero integer	nome character varying (50)	numero integer	nome character varying (80)
1	1	Banco do Brasil S.A.	1	Agência número 1 do banco ...
2	1	Banco do Brasil S.A.	2	Agência número 2 do banco ...
3	1	Banco do Brasil S.A.	3	Agência número 3 do banco ...
4	1	Banco do Brasil S.A.	4	Agência número 4 do banco ...
5	1	Banco do Brasil S.A.	5	Agência número 5 do banco ...
6	1	Banco do Brasil S.A.	6	Agência número 6 do banco ...
7	1	Banco do Brasil S.A.	7	Agência número 7 do banco ...

O código acima faz um relacionamento entre as tabelas banco e agencia, especificamente nas colunas numero e nome de ambos onde o campo banco_numero (chave estrangeira na tabela agencia referente a tabela banco) em agencia seja igual ao campo numero na tabela banco. Retornando todos os banco que e agencias relacionados.

```
14 SELECT banco.numero FROM banco
15 JOIN agencia ON agencia.banco_numero = banco.numero
16 GROUP BY banco.numero;
17
18
```

Data Output Explain Messages Notifications

	numero [PK] integer
1	623
2	477
3	237
4	399
5	104
6	655

O código acima trás o comando **group by**. Semelhante o resultado anterior retorna todos os bancos que possuem relacionamento com a tabela agencia mas agrupado pelo numero do banco evitando trazer vários registros.

```

17
18 SELECT COUNT(DISTINCT banco.numero) FROM banco
19 JOIN agencia ON agencia.banco_numero = banco.numero;
20

```

Data Output Explain Messages Notifications

	count bigint	lock
1	9	

Já esse código retorna a quantidade de bancos que tem relacionamento com a tabela agencia com o comando **count** que já conhecemos e o comando **distinct** que tem a função de retornar apenas os bancos que são distintos.

LEFT JOIN

```

22 --LEFT JOIN
23
24 SELECT banco.numero, banco.nome, agencia.numero, agencia.nome
25 FROM banco LEFT JOIN agencia
26 ON agencia.banco_numero = banco.numero;

```

Data Output Explain Messages Notifications

	numero integer	lock	nome character varying (50)	lock	numero integer	lock	nome character varying (80)	lock
294	477		Citibank N.A.		14		Agência número 14 do banc...	
295	477		Citibank N.A.		15		Agência número 15 do banc...	
296	477		Citibank N.A.		16		Agência número 16 do banc...	
297	630		Banco Intercap S.A.		[null]		[null]	
298	756		Banco Cooperativo do Brasil...		[null]		[null]	
299	214		Banco Dibens S.A.		[null]		[null]	
300	25		Banco Alfa S.A.		[null]		[null]	

Como vimos no conceito do **left join**, o código aqui irá retornar todos os bancos que tem relacionamentos com a tabela agencia e também os bancos que não possuem registros relacionados a tabela agencia.

```

28 SELECT agencia.numero, agencia.nome, banco.numero, banco.nome
29 FROM agencia LEFT JOIN banco
30 ON banco.numero = agencia.banco_numero;
31

```

Data Output Explain Messages Notifications

	numero integer	lock	nome character varying (80)	lock	numero integer	lock	nome character varying (50)	lock
1	1		Agência número 1 do banco ...		1		Banco do Brasil S.A.	
2	2		Agência número 2 do banco ...		1		Banco do Brasil S.A.	
3	3		Agência número 3 do banco ...		1		Banco do Brasil S.A.	
4	4		Agência número 4 do banco ...		1		Banco do Brasil S.A.	
5	5		Agência número 5 do banco ...		1		Banco do Brasil S.A.	
6	6		Agência número 6 do banco ...		1		Banco do Brasil S.A.	
7	7		Agência número 7 do banco ...		1		Banco do Brasil S.A.	

Aqui é retornado todas as agencia que tem registros relacionados a bancos e também as agencias que não tem relações se for o caso.

RIGHT JOIN

```
33 -- RIGHT JOIN
34
35 SELECT agencia.numero, agencia.nome, banco.numero, banco.nome
36 FROM agencia RIGHT JOIN banco
37 ON banco.numero = agencia.banco_numero;
```

Data Output Explain Messages Notifications

	numero integer	nome character varying (80)	numero integer	nome character varying (50)
294	14	Agência número 14 do banc...	477	Citibank N.A.
295	15	Agência número 15 do banc...	477	Citibank N.A.
296	16	Agência número 16 do banc...	477	Citibank N.A.
297	[null]	[null]	630	Banco Intercap S.A.
298	[null]	[null]	756	Banco Cooperativo do Brasil...
299	[null]	[null]	214	Banco Dibens S.A.
300	[null]	[null]	25	Banco Alfa S.A.

Nesse caso como aprendemos no conceito de **right join**, será retornado todas as agencias que tem relacionamentos com bancos e todos os bancos relacionados as agencias e também os bancos que não tem registros de relação com agencias.

FULL JOIN

```
42 SELECT banco.numero, banco.nome, agencia.numero, agencia.nome
43 FROM banco FULL JOIN agencia
44 ON banco.numero = agencia.banco_numero;
45
```

Data Output Explain Messages Notifications

	numero integer	nome character varying (50)	numero integer	nome character varying (80)
294	477	Citibank N.A.	14	Agência número 14 do banc...
295	477	Citibank N.A.	15	Agência número 15 do banc...
296	477	Citibank N.A.	16	Agência número 16 do banc...
297	630	Banco Intercap S.A.	[null]	[null]
298	756	Banco Cooperativo do Brasil...	[null]	[null]
299	214	Banco Dibens S.A.	[null]	[null]
300	25	Banco Alfa S.A.	[null]	[null]

No **full join** como aprendemos no conceito, o código acima irá retornar todos os relacionamentos de bancos com agencias, todos os bancos sem relacionamentos e todas as agencias sem relacionamentos com bancos.

Trabalhando JOIN com várias tabelas.

```
47 -- JOIN COM VARIAS TABELAS
48
49 SELECT banco.nome, agencia.nome,
50     conta_corrente.numero, conta_corrente.digito,
51     cliente.nome
52 FROM banco
53     JOIN agencia ON agencia.banco_numero = banco.numero
54     JOIN conta_corrente ON conta_corrente.banco_numero = banco.numero
55 --ou JOIN conta_corrente ON conta_corrente.banco_numero = agencia.banco_numero
56         AND conta_corrente.agencia_numero = agencia.numero
57     JOIN cliente ON cliente.numero = conta_corrente.cliente_numero;
```

Data Output Explain Messages Notifications					
	nome character varying (50)	nome character varying (80)	numero bigint	digito smallint	nome character varying (120)
98	Banco do Brasil S.A.	Agência número 48 do banc...	88442466	4	Amália Vilaça
99	Banco do Brasil S.A.	Agência número 49 do banc...	364941217	4	Leopoldina Varanda
100	Banco do Brasil S.A.	Agência número 50 do banc...	267909423	3	Ramiro Viégas
101	Banco Itaú S.A.	Agência número 1 do banco ...	430754320	4	Ítalo Caires
102	Banco Itaú S.A.	Agência número 1 do banco ...	251144344	3	Zuleica Valentim
103	Banco Itaú S.A.	Agência número 2 do banco ...	201484438	0	Caim Quintão
104	Banco Itaú S.A.	Agência número 2 do banco ...	305145937	8	Doroteia Souza
105	Banco Itaú S.A.	Agência número 2 do banco ...			

Aqui temos um relacionamento mais complexo com várias tabelas.

O código retornar os registros como nome do banco e agencia, numero e digito da conta corrente e nome do cliente, onde existir relação entre as tabelas banco, agencia, conta corrente e cliente.

O relacionamento fica da seguinte forma: o numero do banco na tabela banco que esteja relacionado ao numero do banco na tabela agencia, numero do banco na tabela banco ou na tabela agencia relacionados ao numero do banco na tabela conta_corrente e numero da agencia na tabela agencia relacionado com o numero da agencia na tabela conta_corrente, por fim o numero do cliente na tabela cliente relacionado com o numero do cliente na tabela conta_corrente. Todas esses registros em tabelas diferentes são relacionados através das chaves estrangeiras.

Query Editor Query History

```
83 SELECT banco.nome, agencia.nome, conta_corrente.numero, conta_corrente.digito,
84     cliente.nome, tipo_transacao.nome, cliente_transacoes.valor
85 FROM banco
86     JOIN agencia ON agencia.banco_numero = banco.numero
87     JOIN conta_corrente ON conta_corrente.banco_numero = banco.numero
88         AND conta_corrente.agencia_numero = agencia.numero
89     JOIN cliente ON cliente.numero = conta_corrente.cliente_numero
90     JOIN cliente_transacoes ON cliente_transacoes.cliente_numero = cliente.numero
91     JOIN tipo_transacao ON tipo_transacao.id = cliente_transacoes.tipo_transacao_id
92 ORDER BY cliente.nome, cliente_transacoes.valor;
```

Data Output Explain Messages Notifications

	nome character varying (50)	nome character varying (80)	numero bigint	digito smallint	nome character varying (120)	nome character varying (50)	valor numeric (15,2)
1	Banco do Brasil S.A.	Agência número 23 do banc...	431262421	3	Aarão Valentín	Débito	11.94
2	Banco do Brasil S.A.	Agência número 23 do banc...	431262421	3	Aarão Valentín	Crédito	51.54
3	Banco do Brasil S.A.	Agência número 23 do banc...	431262421	3	Aarão Valentín	Crédito	52.89
4	Banco do Brasil S.A.	Agência número 23 do banc...	431262421	3	Aarão Valentín	Transferência	57.85
5	Banco Santander (Brasil) S.A.	Agência número 29 do banc...	105294373	2	Abigail Belém	Crédito	36.39
6	Banco Santander (Brasil) S.A.	Agência número 29 do banc...	105294373	2	Abigail Belém	Crédito	36.62
7	Banco Santander (Brasil) S.A.	Agência número 29 do banc...	105294373	2	Abigail Belém	Transferência	96.34
8	Banco Santander (Brasil) S.A.	Agência número 29 do banc...	105294373	2	Abigail Belém	Débito	135.45
9	HSBC Bank Brasil S A - Ban...	Anência número 7 do banco...	194158634	6	Ahbraão Faro	Transferência	46.42

Aqui segue o mesmo raciocínio do código anterior, apenas adicionando os relacionamentos dos registros com a tabela de clientes, transações e tipos de transações. Ao final ordenamos os registros pelo nome do cliente e pelo valor da transação com o **order by**.

CTE – Common Table Expressions

Definição

Forma auxiliar de organizar “statements”, ou seja, blocos de códigos, para consultas muito grandes, gerando tabelas temporárias e criando relacionamentos entre elas.

Dentro dos statements podem ter SELECTs, INSERTs, UPDATEs ou DELETEs.

WITH STATEMENTS

```
WITH [nome1] AS (
    SELECT (campos,)
    FROM tabela_A
    [WHERE]
), [nome2] AS (
    SELECT (campos,)
    FROM tabela_B
    [WHERE]
)
SELECT [nome1].[campos,], [nome2].[campos,]
FROM [nome1]
JOIN [nome2] .....
```

Criando STATEMENTS

```
4 WITH tbl_tmp_banco AS (
5     SELECT numero, nome
6     FROM banco
7 )
8 SELECT numero, nome FROM tbl_tmp_banco;
```

Data Output			Explain	Messages	Notifications
	numero [PK] integer	nome character varying (50)			
1	654	Banco A.J.Renner S.A.			
2	246	Banco ABC Brasil S.A.			
3	25	Banco Alfa S.A.			
4	641	Banco Alvorada S.A.			
5	213	Banco Arbi S.A.			
6	19	Banco Azteca do Brasil S.A.			

No código acima criamos uma tabela temporária (statement) simples chamada **tbl_tmp_banco** e logo após fizemos um select na tabela temporária.

```

11 WITH params AS (
12     SELECT 213 AS banco_numero
13 ), tbl_tmp_banco AS (
14     SELECT numero, nome FROM banco
15     JOIN params ON params.banco_numero = banco.numero
16 )
17 SELECT numero, nome FROM tbl_tmp_banco;

```

Data Output Explain Messages Notifications

	numero [PK] integer	nome character varying (50)
1	213	Banco Arbi S.A.

Nesse código foi criada um statement chamado **params** e um statement chamado **tbl_tmp_banco** juntamente com um **join** para a tabela **params**.

```

20 -- SUB SELECT
21 SELECT banco.numero, banco.nome FROM banco
22     JOIN (
23         SELECT 213 AS banco_numero
24     ) params ON params.banco_numero = banco.numero;

```

Data Output Explain Messages Notifications

	numero [PK] integer	nome character varying (50)
1	213	Banco Arbi S.A.

Existe algo semelhante ao statement que criamos anteriormente chamado de **sub select**. Apesar do resultado ser o mesmo o statement trás um código mais organizado, fácil de manter, com separações bem definidas entre tabelas criadas temporariamente.

Statement com várias tabelas

```
44 WITH clientes_e_transacoes AS (
45     SELECT cliente.nome AS cliente_nome, --muda nome do campo
46         tipo_transacao.nome AS tipo_transacao_nome, --muda nome do campo
47         cliente_transacoes.valor AS tipo_transacao_valor --muda nome do campo
48     --relação correta
49     FROM cliente
50     JOIN cliente_transacoes ON cliente_transacoes.cliente_numero = cliente.numero
51     JOIN tipo_transacao ON tipo_transacao.id = cliente_transacoes.tipo_transacao_id
52 )
53 SELECT cliente_nome, tipo_transacao_nome, tipo_transacao_valor
54 FROM clientes_e_transacoes ORDER BY cliente_nome, tipo_transacao_valor;
```

Data Output Explain Messages Notifications

	cliente_nome character varying (120)	tipo_transacao_nome character varying (50)	tipo_transacao_valor numeric (15,2)	
1	Aarão Valentín	Débito	11.94	
2	Aarão Valentín	Crédito	51.54	
3	Aarão Valentín	Crédito	52.89	
4	Aarão Valentín	Transferência	57.85	
5	Abigail Belém	Crédito	36.39	
6	Abigail Belém	Crédito	36.62	
7	Abigail Belém	Transferência	96.34	

Aqui criamos uma tabela temporária **clientes_e_transacoes** a partir dos relacionamentos com **join** e o **select** acessa essa tabela criada externamente.

VIEWS



Views

Definição

View são visões.

São "camadas" para as tabelas.

São "alias" para uma ou mais queries.

Aceitam comandos de **SELECT, INSERT, UPDATE e DELETE**.

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name [ ( column_name [, ...] ) ]
[ WITH ( view_option_name [= view_option_value] [, ...] ) ]
AS query
[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

- **or replace**: idepotencia da view, cria ou substitui a view.
- **temp**: view temporaria, só irá existir dentro da sua seção.
- **recursive**: select dentro da view que irá chamar a própria view que irá executar até se esgotar determinada opção.
- **whit [cascade | local] check option**: São validações pro comando insert, update, delete da sua view.

IDEMPOTÊNCIA

```
CREATE OR REPLACE VIEW vw_bancos AS (
    SELECT numero, nome, ativo
    FROM banco
);
```

```
SELECT numero, nome, ativo
FROM vw_bancos;
```

```
CREATE OR REPLACE VIEW vw_bancos (banco_numero, banco_nome, banco_ativo) AS (
    SELECT numero, nome, ativo
    FROM banco
);
```

```
SELECT banco_numero, banco_nome, banco_ativo
FROM vw_bancos;
```



INSERT, UPDATE e DELETE

```
CREATE OR REPLACE VIEW vw_bancos AS (
    SELECT numero, nome, ativo
    FROM banco
);
```

```
SELECT numero, nome, ativo
FROM vw_bancos;
```

- Funcionam apenas para VIEWS com apenas 1 tabela

```
INSERT INTO vw_bancos (numero, nome, ativo) VALUES (100, 'Banco CEM', TRUE);
```

```
UPDATE vw_bancos SET nome = 'Banco 100' WHERE numero = 100;
```

```
DELETE FROM vw_bancos WHERE numero = 100;
```



Comandos de insert, update e delete só funcionam pra view que são referência pra uma tabela.

A view irá servir como uma janela pra manipular os dados diretamente na tabela.

TEMPORARY

```
CREATE OR REPLACE TEMPORARY VIEW vw_bancos AS (
    SELECT numero, nome, ativo
    FROM banco
);
```

```
SELECT numero, nome, ativo
FROM vw_bancos;
```

VIEW presente apenas na sessão do usuário.

Se você se desconectar e conectar novamente, a VIEW não estará disponível.



RECURSIVE

```
CREATE OR REPLACE RECURSIVE VIEW (nome_da_view)(campos_da_view) AS (
    SELECT base
    UNION ALL
    SELECT campos
    FROM tabela_base
    JOIN (nome_da_view)
);
```

- Obrigatório a existência dos campos da VIEW
- UNION ALL

WITH OPTIONS

```
CREATE OR REPLACE VIEW vw_bancos AS (
    SELECT numero, nome, ativo
    FROM banco
);
```

```
INSERT INTO vw_bancos (numero, nome, ativo) VALUES (100, 'Banco CEM', FALSE)
-- OK
```

```
CREATE OR REPLACE VIEW vw_bancos AS (
    SELECT numero, nome, ativo
    FROM banco
    WHERE ativo IS TRUE
) WITH LOCAL CHECK OPTION;
```

```
INSERT INTO vw_bancos (numero, nome, ativo) VALUES (100, 'Banco CEM', FALSE)
-- ERRO
```

WITH OPTIONS

```
CREATE OR REPLACE VIEW vw_bancos_ativos AS (
    SELECT numero, nome, ativo
    FROM banco
    WHERE ativo IS TRUE
) WITH LOCAL CHECK OPTION;

CREATE OR REPLACE VIEW vw_bancos_maiores_que_100 AS (
    SELECT numero, nome, ativo
    FROM vw_banco
    WHERE numero > 100
) WITH LOCAL CHECK OPTION;

INSERT INTO vw_bancos_maiores_que_100 (numero, nome, ativo) VALUES (99, 'Banco DIO', FALSE)
-- ERRO
INSERT INTO vw_bancos_maiores_que_100 (numero, nome, ativo) VALUES (200, 'Banco DIO', FALSE)
-- ERRO
```

WITH OPTIONS

```
CREATE OR REPLACE VIEW vw_bancos_ativos AS (
    SELECT numero, nome, ativo
    FROM banco
    WHERE ativo IS TRUE
);

CREATE OR REPLACE VIEW vw_bancos_maiores_que_100 AS (
    SELECT numero, nome, ativo
    FROM vw_banco
    WHERE numero > 100
) WITH LOCAL CHECK OPTION;

INSERT INTO vw_bancos_maiores_que_100 (numero, nome, ativo) VALUES (99, 'Banco DIO', FALSE)
-- ERRO
INSERT INTO vw_bancos_maiores_que_100 (numero, nome, ativo) VALUES (200, 'Banco DIO', FALSE)
-- OK
```

WITH OPTIONS

```
CREATE OR REPLACE VIEW vw_bancos_ativos AS (
    SELECT numero, nome, ativo
    FROM banco
    WHERE ativo IS TRUE
);

CREATE OR REPLACE VIEW vw_bancos_maiores_que_100 AS (
    SELECT numero, nome, ativo
    FROM vw_banco
    WHERE numero > 100
) WITH CASCADING CHECK OPTION;

INSERT INTO vw_bancos_maiores_que_100 (numero, nome, ativo) VALUES (99, 'Banco DIO', FALSE)
-- ERRO
INSERT INTO vw_bancos_maiores_que_100 (numero, nome, ativo) VALUES (200, 'Banco DIO', FALSE)
-- ERRO
```

Criando VIEWS

Query Editor Query History

```
1 SELECT numero, nome, ativo FROM banco;
2
3 --VIEW SIMPLES
4
5 CREATE OR REPLACE VIEW vw_bancos AS(
6     SELECT numero, nome, ativo
7     FROM banco
8 );
```

Data Output Explain Messages Notifications

CREATE VIEW

Query returned successfully in 667 msec.

Comando cria uma **view** chamada **vw_bancos** com os campos da tabela **banco** **numero**, **nome** e **ativo**.

```
3 --VIEW SIMPLES
4
5 CREATE OR REPLACE VIEW vw_bancos AS(
6     SELECT numero, nome, ativo
7     FROM banco
8 );
9 SELECT numero, nome, ativo FROM vw_bancos;
```

Data Output Explain Messages Notifications

	numero integer	nome character varying (50)	ativo boolean
1	654	Banco A.J.Renner S.A.	true
2	246	Banco ABC Brasil S.A.	true
3	25	Banco Alfa S.A.	true
4	641	Banco Alvorada S.A.	true
5	213	Banco Arbi S.A.	true

A partir disso nós podemos fazer um **select** na **view** criada.

```
11 CREATE OR REPLACE VIEW vw_bancos_2 (banco_numero, banco_nome, banco_ativo) AS (
12     SELECT numero, nome, ativo FROM banco
13 );
14 SELECT banco_numero, banco_nome, banco_ativo FROM vw_bancos_2;
```

Data Output Explain Messages Notifications

	banco_numero integer	banco_nome character varying (50)	banco_ativo boolean
1	654	Banco A.J.Renner S.A.	true
2	246	Banco ABC Brasil S.A.	true
3	25	Banco Alfa S.A.	true
4	641	Banco Alvorada S.A.	true
5	213	Banco Arbi S.A.	true
6	19	Banco Azteca do Brasil S.A.	true

Criação de mais uma **view** simples.

```

16 --INSERT
17
18 INSERT INTO vw_bancos_2 (banco_numero, banco_nome, banco_ativo)
19 VALUES (51, 'Banco Boa Ideia', TRUE);
20
21 SELECT banco_numero, banco_nome, banco_ativo FROM vw_bancos_2 WHERE banco_numero = 51;
22 SELECT numero, banco, ativo FROM banco WHERE numero = 51;

```

Data Output Explain Messages Notifications

	numero [PK] integer	banco banco	ativo boolean	
1		51 ('Banc...')	true	

Fizemos um insert e agora o banco existe nas duas tabelas.

```

24 --UPDATE
25 UPDATE vw_bancos_2 SET banco_ativo = FALSE WHERE banco_numero = 51;

```

Data Output Explain Messages Notifications

	numero [PK] integer	banco banco	ativo boolean	
1		51 ('Banco Boa Ideia', f'2020-12-24 12:32:26.515858')	false	

Agora um update.

```

27 --DELETE
28 DELETE FROM vw_bancos_2 WHERE banco_numero = 51;
29

```

Data Output Explain Messages Notifications

	numero [PK] integer	banco banco	ativo boolean	

Agora um delete.

VIEW Temporária.

```
31 --VIEW TEMPORÁRIA
32
33 CREATE OR REPLACE TEMPORARY VIEW vw_agencia AS (
34     SELECT nome FROM agencia
35 );
36 SELECT nome FROM vw_agencia;
```

Data Output Explain Messages Notifications

	nome character varying (80) 
1	Agência número 1 do banco ...
2	Agência número 2 do banco ...
3	Agência número 3 do banco ...
4	Agência número 4 do banco ...
5	Anência número 5 do banco ...

Nesse código criamos uma view temporária, o que significa que por exemplo, ao abrir outra seção no query tools e tentar fazer um select nessa view não será possível pois ela existe apenas nessa seção.

WHIT OPTIONS

```
40 --WHIT OPTIONS
41
42 CREATE OR REPLACE VIEW vw_bancos_ativos AS (
43     SELECT numero, nome, ativo FROM banco
44     WHERE ativo IS TRUE
45 ) WITH LOCAL CHECK OPTION;
46
47 INSERT INTO vw_bancos_ativos (numero, nome, ativo)
48 VALUES (51, 'Banco Boa Ideia', FALSE);
49
```

Data Output Explain Messages Notifications

```
ERROR: new row violates check option for view "vw_bancos_ativos"
DETAIL: Failing row contains (51, Banco Boa Ideia, f, 2020-12-24 13:02:46.907673).
SQL state: 44000
```

Nesse código criamos uma view e dissemos que o campo ativo tem que ser TRUE, para validar isso utilizamos a opção **with local check option**. Então ao tentar fazer um insert com o campo ativo = FALSE será retornado um erro como podemos ver.

```

40 --WITH OPTIONS
41
42 CREATE OR REPLACE VIEW vw_bancos_ativos AS (
43     SELECT numero, nome, ativo FROM banco
44     WHERE ativo IS TRUE
45 ) WITH LOCAL CHECK OPTION;
46
47 INSERT INTO vw_bancos_ativos (numero, nome, ativo)
48 VALUES (51, 'Banco Boa Ideia', FALSE);
49
50 CREATE OR REPLACE VIEW vw_bancos_com_a AS (
51     SELECT numero, nome, ativo
52     FROM vw_bancos_ativos
53     WHERE nome ILIKE 'a%'
54 ) WITH LOCAL CHECK OPTION;
55
56 INSERT INTO vw_bancos_com_a (numero, nome, ativo) VALUES (337,'Beta Omega',FALSE);
57 SELECT * FROM vw_bancos_com_a;
58

```

Data Output Explain Messages Notifications

```

ERROR: new row violates check option for view "vw_bancos_ativos"
DETAIL: Failing row contains (337, Alfa Omega, f, 2020-12-24 14:54:21.395335).
SQL state: 44000

```

Aqui criamos uma nova view com referência a view anterior e com a condição que o nome deve iniciar com a letra 'a' como especifica o ILIKE.

Ao tentar fazer um insert percebemos que ainda não é possível pois além do nome do banco iniciar com a letra 'b' temos o campo ativo = FALSE, o que contraria o with option da primeira view.

Para esse insert dar certo teríamos que iniciar o nome do banco com a letra 'a' e o campo ativo teria que ser = TRUE.

Ou se quiser q o campo ativo seja realmente = FALSE teria que remover a with option da view vw_bancos_ativos.

```

40 --WITH OPTIONS
41
42 CREATE OR REPLACE VIEW vw_bancos_ativos AS (
43     SELECT numero, nome, ativo FROM banco
44     WHERE ativo IS TRUE
45 );
46
47 INSERT INTO vw_bancos_ativos (numero, nome, ativo)
48 VALUES (51, 'Banco Boa Ideia', FALSE);
49
50 CREATE OR REPLACE VIEW vw_bancos_com_a AS (
51     SELECT numero, nome, ativo
52     FROM vw_bancos_ativos
53     WHERE nome ILIKE 'a%'
54 ) WITH LOCAL CHECK OPTION;
55
56 INSERT INTO vw_bancos_com_a (numero, nome, ativo) VALUES (338,'Alfa Omega',FALSE);
57 SELECT * FROM vw_bancos_com_a;
58

```

Data Output Explain Messages Notifications

INSERT 0 1

Query returned successfully in 110 msec.

```

40 --WITH OPTIONS
41
42 CREATE OR REPLACE VIEW vw_bancos_ativos AS (
43     SELECT numero, nome, ativo FROM banco
44     WHERE ativo IS TRUE
45 );
46
47 INSERT INTO vw_bancos_ativos (numero, nome, ativo)
48 VALUES (51, 'Banco Boa Ideia', FALSE);
49
50 CREATE OR REPLACE VIEW vw_bancos_com_a AS (
51     SELECT numero, nome, ativo
52     FROM vw_bancos_ativos
53     WHERE nome ILIKE 'a%'
54 ) WITH CASCDED CHECK OPTION;
55
56 INSERT INTO vw_bancos_com_a (numero, nome, ativo) VALUES (340,'Alfa Omega',FALSE);
57 SELECT * FROM vw_bancos_com_a;
58

```

Data Output Explain Messages Notifications

```

ERROR: new row violates check option for view "vw_bancos_ativos"
DETAIL: Failing row contains (340, Alfa Omega, f, 2020-12-24 15:08:41.950253).
SQL state: 44000

```

Outra situação que podemos observar é o with option com a opção CASCDED. Essa opção faz com que a clausula da view vw_bancos_ativos seja verificada mesmo que ela não tenha uma with option.

Então ao tentar fazer esse insert é retornado erro.

RECURSIVE

Para exemplificar essa recursividade precisamos criar uma nova tabela.

Após criar a tabela vamos inserir dados e fazer um select da tabela.

```
61
62 CREATE TABLE IF NOT EXISTS funcionarios(
63     id SERIAL,
64     nome VARCHAR(50),
65     gerente INTEGER,
66     PRIMARY KEY (id),
67     FOREIGN KEY (gerente) REFERENCES funcionarios(id)
68 );
69
70 INSERT INTO funcionarios (nome, gerente) VALUES ('Ancelmo', null);
71 INSERT INTO funcionarios (nome, gerente) VALUES ('Beatriz', 1);
72 INSERT INTO funcionarios (nome, gerente) VALUES ('Magno', 1);
73 INSERT INTO funcionarios (nome, gerente) VALUES ('Cremilda', 2);
74 INSERT INTO funcionarios (nome, gerente) VALUES ('Wagner', 4);
75
76 SELECT id, nome, gerente FROM funcionarios;
--
```

Data Output Explain Messages Notifications

	id [PK] integer	nome character varying (50)	gerente integer
1	1	Ancelmo	[null]
2	2	Beatriz	1
3	3	Magno	1
4	4	Cremilda	2
5	5	Wagner	4

```
78 CREATE OR REPLACE RECURSIVE VIEW vw_func (id, gerente, funcionario) AS (
79     SELECT id, gerente, nome FROM funcionarios
80     WHERE gerente IS NULL
81
82     UNION ALL
83
84     SELECT funcionarios.id, funcionarios.gerente, funcionarios.nome
85     FROM funcionarios
86     JOIN vw_func ON vw_func.id = funcionarios.gerente
87 );
88
89 SELECT id, gerente, funcionario FROM vw_func;
90
```

Data Output Explain Messages Notifications

	id integer	gerente integer	funcionario character varying (50)
1	1	[null]	Ancelmo
2	2	1	Beatriz
3	3	1	Magno
4	4	2	Cremilda
5	5	4	Wagner

Aqui já criamos a view recursiva que chama a ela mesma através de um join criando um loop até esgotar os recursos produzindo a saída onde podemos observar a que gerente pertence tal funcionário.

Transações



Transações

Definição

Conceito fundamental de todos os sistemas de bancos de dados.

Conceito de múltiplas etapas/códigos reunidos em apenas 1 transação, onde o resultado precisa ser **tudo ou nada**.

Exemplos:

```
BEGIN;  
  
    UPDATE conta SET valor = valor - 100.00  
    WHERE nome = 'Alice';  
  
    UPDATE conta SET valor = valor + 100.00  
    WHERE nome = 'Bob';  
  
COMMIT;
```

Uma transação se inicia com um **begin**. Se a execução dos códigos que há dentro desse **begin** forem bem sucedidas então será executado um **commit** e essa transação estará em produção no banco de dados, mas se alguma coisa der errado acontecerá um **rollback** e todas as alterações serão desfeitas, ou seja, ou todas as alterações são bem sucedidas ou nada será alterado.

Exemplos:

```
BEGIN;  
  
    UPDATE conta SET valor = valor - 100.00  
    WHERE nome = 'Alice';  
  
    SAVEPOINT my_savepoint;  
  
    UPDATE conta SET valor = valor + 100.00  
    WHERE nome = 'Bob';  
    -- oops ... não é para o Bob, é para o Wally!!!  
  
ROLLBACK TO my_savepoint;  
  
    UPDATE conta SET valor = valor + 100.00  
    WHERE nome = 'Wally';  
COMMIT;
```

Esse exemplo mostra como funciona um **savepoint**, ao iniciar uma transação será salva um ponto de consistência para onde deve ser retornado caso algo dê errado voltando para aquele determinado ponto com um **rollback**, e refazendo a transação da forma correta chegando a um **commit**.

Rollback

```
7 BEGIN;
8 UPDATE banco SET ativo = true WHERE numero = 0;
9 SELECT numero, nome, ativo FROM banco ORDER BY numero;
10
```

Data Output Explain Messages Notifications

	numero [PK] integer	nome character varying (50)	ativo boolean	
1	0	Banco Bankpar S.A.	true	

Antes de executarmos esse código, tínhamos o campo ativo desse banco = false.

Ao executar o código primeiramente iniciamos uma transação com o **begin**, e atualizamos o campo ativo para true como mostra o select. Mas por ser uma transação nós podemos retornar esse dado ao que era antes do update com um **rollback**.

```
7 BEGIN;
8 UPDATE banco SET ativo = true WHERE numero = 0;
9 SELECT numero, nome, ativo FROM banco ORDER BY numero;
10
11 ROLLBACK;
12 SELECT numero, nome, ativo FROM banco ORDER BY numero;
```

Data Output Explain Messages Notifications

	numero [PK] integer	nome character varying (50)	ativo boolean	
1	0	Banco Bankpar S.A.	false	

O **rollback** desfaz todas as alterações que foram feitas, retornando os dados ao que eram antes da transação.

Commit.

```
15 --COMMIT
16
17 BEGIN;
18 UPDATE banco SET ativo = true WHERE numero = 0;
19 SELECT numero, nome, ativo FROM banco ORDER BY numero;
```

Data Output Explain Messages Notifications

	numero [PK] integer	nome character varying (50)	ativo boolean	
1	0	Banco Bankpar S.A.	true	

Por outro lado, ao executar um commit ao final da transação, essas mudanças se tornarão permanentes no banco de dados.

```
15 --COMMIT
16
17 BEGIN;
18 UPDATE banco SET ativo = true WHERE numero = 0;
19 SELECT numero, nome, ativo FROM banco ORDER BY numero;
20
21 COMMIT;
22 SELECT numero, nome, ativo FROM banco ORDER BY numero;
23
24
```

Data Output Explain Messages Notifications

	numero [PK] integer	nome character varying (50)	ativo boolean
1	0	Banco Bankpar S.A.	true

SAVEPOINT

Dada a seguinte tabela.

```
26 --SAVEPOINT
27 SELECT id, gerente, nome FROM funcionarios;
```

Data Output Explain Messages Notifications

	id [PK] integer	gerente integer	nome character varying (50)
1	1	[null]	Ancelmo
2	2	1	Beatriz
3	3	1	Magno
4	4	2	Cremilda
5	5	4	Wagner

Vamos iniciar uma transação onde vamos trocar o campo gerente da Beatriz para null.

```
26 --SAVEPOINT
27 SELECT id, gerente, nome FROM funcionarios;
28 BEGIN;
29 UPDATE funcionarios SET gerente = null WHERE id = 2;
30 SAVEPOINT sp_func;
31 SELECT id, gerente, nome FROM funcionarios ORDER BY id;
32
```

Data Output Explain Messages Notifications

	id [PK] integer	gerente integer	nome character varying (50)
1	1	[null]	Ancelmo
2	2	[null]	Beatriz
3	3	1	Magno
4	4	2	Cremilda
5	5	4	Wagner

Após fazer as alterações citadas podemos ver que criamos um **savepoint** desse update para restaurar a esse ponto caso algo dê errado.

Agora vamos pensar que executamos um comando errado onde pode ter comprometido os dados no banco como veremos a seguir.

```
26 --SAVEPOINT
27 SELECT id, gerente, nome FROM funcionarios;
28 BEGIN;
29 UPDATE funcionarios SET gerente = null where id = 2;
30 SAVEPOINT sp_func;
31 SELECT id, gerente, nome FROM funcionarios ORDER BY id;
32
33 UPDATE funcionarios SET gerente = null;
34 SELECT id, gerente, nome FROM funcionarios ORDER BY id;
35
```

Data Output Explain Messages Notifications

	id [PK] integer	gerente integer	nome character varying (50)
1	1	[null]	Ancelmo
2	2	[null]	Beatriz
3	3	[null]	Magno
4	4	[null]	Cremilda
5	5	[null]	Wagner

Aqui um update sem uma cláusula where comprometeu todos os dados na tabela;

```
26 --SAVEPOINT
27 SELECT id, gerente, nome FROM funcionarios;
28 BEGIN;
29 UPDATE funcionarios SET gerente = null where id = 2;
30 SAVEPOINT sp_func;
31 SELECT id, gerente, nome FROM funcionarios ORDER BY id;
32
33 UPDATE funcionarios SET gerente = null;
34 SELECT id, gerente, nome FROM funcionarios ORDER BY id;
35
36 ROLLBACK TO sp_func;
37 SELECT id, gerente, nome FROM funcionarios ORDER BY id;
38
```

Data Output Explain Messages Notifications

	id [PK] integer	gerente integer	nome character varying (50)
1	1	[null]	Ancelmo
2	2	[null]	Beatriz
3	3	1	Magno
4	4	2	Cremilda
5	5	4	Wagner

Mas como criamos um **savepoint** pudemos retornar nosso banco de dados ao um ponto de consistencia na tabela evitando tornar permanente esse erro gravíssimo no nosso banco de dados.

```

26 --SAVEPOINT
27 SELECT id, gerente, nome FROM funcionarios;
28 BEGIN;
29 UPDATE funcionarios SET gerente = null WHERE id = 2;
30 SAVEPOINT sp_func;
31 SELECT id, gerente, nome FROM funcionarios ORDER BY id;
32
33 UPDATE funcionarios SET gerente = null;
34 SELECT id, gerente, nome FROM funcionarios ORDER BY id;
35
36 ROLLBACK TO sp_func;
37 SELECT id, gerente, nome FROM funcionarios ORDER BY id;
38
39 UPDATE funcionarios SET gerente = 2 WHERE id = 5;
40 COMMIT;
41 SELECT id, gerente, nome FROM funcionarios ORDER BY id;
42

```

Data Output Explain Messages Notifications

	id [PK] integer	gerente integer	nome character varying (50)
1	1	[null]	Ancelmo
2	2	[null]	Beatriz
3	3	1	Magno
4	4	2	Cremilda
5	5	2	Wagner

E por fim nós terminamos nossas possíveis alterações com um commit tornando assim permanente e consistente nossas atualizações e encerrando assim nossa transação.

Funções



Funções

Definição

Conjunto de códigos que são executados **dentro de uma transação** com a finalidade de facilitar a programação e obter o reaproveitamento/reutilização de códigos.

Existem 4 tipos de funções:

- query language functions (funções escritas em SQL)
- procedural language functions (funções escritas em, por exemplo, PL/pgSQL ou PL/py)
- internal functions
- C-language functions

Porém, o foco aqui é falar sobre **USER DEFINED FUNCTIONS**.
Funções que podem ser criadas pelo usuário.

Linguagens

- SQL
- **PL/PGSQL**
- PL/PY
- PL/PHP
- PL/RUBY
- PL/Java
- PL/Lua
-

<https://www.postgresql.org/docs/11/external-pl.html>

Podemos escrever funções em diversas linguagens de programação dentro do postgres.

Vamos abordar nesse curso como trabalhar com **PL/PGSQL** linguagem exclusiva do PostgreSQL.

Definição

```
CREATE [ OR REPLACE ] FUNCTION
  name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
  [ RETURNS rettype
  | RETURNS TABLE ( column_name column_type [, ...] ) ]
{ LANGUAGE lang_name
  | TRANSFORM { FOR TYPE type_name } [, ...]
  | WINDOW
  | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
  | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
  | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
  | PARALLEL { UNSAFE | RESTRICTED | SAFE }
  | COST execution_cost
  | ROWS result_rows
  | SET configuration_parameter { TO value | = value | FROM CURRENT }
  | AS 'definition'
  | AS 'obj_file', 'link_symbol'
} ...
```

IDEMPOTÊNCIA

CREATE **OR REPLACE** FUNCTION [nome da função]

- Mesmo nome
- Mesmo tipo de retorno
- Mesmo número de parâmetros/argumentos

RETURNS

- Tipo de retorno (data type)
 - INTEGER
 - CHAR / VARCHAR
 - BOOLEAN
 - ROW
 - TABLE
 - JSON

SEGURANÇA

- **SECURITY**
 - INVOKER
 - DEFINER
- **INVOKER:** (PADRÃO) Permite que a função seja executada com as permissões do usuário que está executando a função. Se o usuário que está executando a função não tem permissão de fazer insert na tabela e dentro da função existe um insert provavelmente esse usuário irá tomar um erro.
- **DEFINER:** Faz com que o usuário que está executando a função a execute com as permissões do usuário que criou a função. Se o usuário não tem permissão de determinado comando dentro da função mas o usuário que criou sim, então ele conseguirá executar a função com as permissões de quem a criou.

COMPORTAMENTO

- **IMMUTABLE**

Não pode alterar o banco de dados.

Funções que garantem o mesmo resultado para os mesmos argumentos/parâmetros da função. Evitar a utilização de selects, pois tabelas podem sofrer alterações.

- **STABLE**

Não pode alterar o banco de dados.

Funções que garantem o mesmo resultado para os mesmos argumentos/parâmetros da função. Trabalha melhor com tipos de current_timestamp e outros tipos variáveis. Podem conter selects.

- **VOLATILE**

Comportamento padrão. Aceita todos os cenários.



SEGURANÇA E BOAS PRÁTICAS

- **CALLED ON NULL INPUT**

Padrão.

Se qualquer um dos parâmetros/argumentos for NULL, a função será executada.

- **RETURNS NULL ON NULL INPUT**

Se qualquer um dos parâmetros/argumentos for NULL, a função retornará NULL

RECURSOS

- **COST**

Custo/row em unidades de CPU

- **ROWS**

Número estimado de linhas que será analisada pelo planner

SQL FUNCTIONS

Não é possível utilizar **TRANSAÇÕES**

```
CREATE OR REPLACE FUNCTION fc_somar(INTEGER,INTEGER)
RETURNS INTEGER
LANGUAGE SQL
AS $$          SELECT $1 + $2;
$$;

CREATE OR REPLACE FUNCTION fc_somar(num1 INTEGER,num2 INTEGER)
RETURNS INTEGER
LANGUAGE SQL
AS $$          SELECT num1 + num2;
$$;
```

SQL FUNCTIONS

```
CREATE OR REPLACE FUNCTION fc_bancos_add(p_numero INTEGER,p_nome VARCHAR,p_ativo BOOLEAN)
RETURNS TABLE (numero INTEGER,nome VARCHAR)
RETURNS NULL ON NULL INPUT
LANGUAGE SQL
AS $$          INSERT INTO banco (numero, nome, ativo)
VALUES (p_numero, p_nome, p_ativo);

          SELECT numero, nome
          FROM banco
         WHERE numero = p_numero;
$$;
```

Funções

PLPGSQL

```
CREATE OR REPLACE FUNCTION banco_add(p_numero INTEGER, p_nome VARCHAR, p_ativo BOOLEAN)
RETURNS BOOLEAN
LANGUAGE PLPGSQL
AS $$
DECLARE variavel_id INTEGER;
BEGIN
    SELECT INTO variavel_id numero FROM banco WHERE nome = p_nome;

    IF variavel_id IS NULL THEN
        INSERT INTO banco (numero, nome, ativo) VALUES (p_numero, p_nome, p_ativo);
    ELSE
        RETURN FALSE;
    END IF;
    *
    SELECT INTO variavel_id numero FROM banco WHERE nome = p_nome;

    IF variavel_id IS NULL THEN
        RETURN FALSE;
    ELSE
        RETURN TRUE;
    END IF;
END; $$;

SELECT banco_add(13, 'Banco azarado', true);
```

```

1 --FUNCTIONS
2
3 CREATE OR REPLACE FUNCTION func_somar(INTEGER,INTEGER)
4 RETURNS INTEGER
5 SECURITY DEFINER
6 --RETURNS NULL ON NULL INPUT
7 CALLED ON NULL INPUT
8 LANGUAGE SQL
9 AS $$*
10    SELECT($1) + ($2);
11 $$;
12
13 SELECT func_somar(2,3);

```

Data Output Explain Messages Notifications

	func_somar	integer	lock	
1		5		

Aqui neste código criamos uma função simples para somar 2 valores inteiros.

Para chamar e executar a função utilizamos o select passando como parâmetro 2 valores que foram somados retornando um valor inteiro igual à soma desses dois valores.

```

1 --FUNCTIONS
2
3 CREATE OR REPLACE FUNCTION func_somar(INTEGER,INTEGER)
4 RETURNS INTEGER
5 SECURITY DEFINER
6 --RETURNS NULL ON NULL INPUT
7 CALLED ON NULL INPUT
8 LANGUAGE SQL
9 AS $$*
10    SELECT($1) + ($2);
11 $$;
12
13 SELECT func_somar(null,3);

```

Data Output Explain Messages Notifications

	func_somar	integer	lock	
1		[null]		

Agora se algum desses valores passados for vazio a função retornará vazio ou trará como resultado = vazio [null].

Para fazer a tratativa desse tipo de situação vamos usar o comando **coalesce**.

```

16 --FUNCTION SQL 2
17 CREATE OR REPLACE FUNCTION func_somar_2(INTEGER,INTEGER)
18 RETURNS INTEGER
19 SECURITY DEFINER
20 --RETURNS NULL ON NULL INPUT
21 CALLED ON NULL INPUT
22 LANGUAGE SQL
23 AS $$ 
24     SELECT COALESCE($1,0) + COALESCE($2,0);
25 $$;
26
27 SELECT func_somar_2(null,10);
28

```

Data Output Explain Messages Notifications

	func_somar_2	integer	lock
1		10	

Com essa função se houver algum número vazio, a função não retornará [null] mas sim o valor que encontrar que não seja [null], isso porque colocamos uma tratativa com o **coalesce** na nossa função fazendo com que se houver um parâmetro [null] ele utilizará o 0[zero] na soma.

```

30 --FUNCTION PLPGSQL
31
32 CREATE OR REPLACE FUNCTION bancos_add(p_numero INTEGER, p_nome VARCHAR, p_ativo BOOLEAN)
33 RETURNS INTEGER
34 SECURITY INVOKER
35 LANGUAGE PLPGSQL
36 CALLED ON NULL INPUT
37 AS $$ 
38 DECLARE variavel_id INTEGER;
39 BEGIN
40     SELECT INTO variavel_id numero
41     FROM banco
42     WHERE numero = p_numero;
43
44     RETURN variavel_id;
45 END; $$;
46
47 SELECT bancos_add(333, '', null);
48

```

Data Output Explain Messages Notifications

	bancos_add	integer	lock
1		333	

Esta é uma função **PLPGSQL**, essa função faz um select na tabela banco de acordo com os parâmetros que forem passados e retorna um numero da variável declarada = ao número do banco se ele já existir na tabela ou [null] se aquele numero não existir.

Nosso select chama a função passando como parâmetro o numero 333 e o retorno é o próprio numero indicando que esse numero já existe na tabela banco se não existir provavelmente o retorno seria [null].

```
49 SELECT numero, nome, ativo FROM banco WHERE numero = 333;  
50
```

Data Output Explain Messages Notifications

	numero [PK] integer	nome character varying (50)	ativo boolean
1	333	Alfa Omega	true

Como podemos ver ao fazer um select normal na tabela banco podemos confirmar que realmente tal banco já existe na tabela. Portanto ao tentar fazer um insert com esse código seria retornado erro por duplicidade de chave primária e a transação não seria concluída.

```

54
55 CREATE OR REPLACE FUNCTION bancos_add_2(p_numero INTEGER, p_nome VARCHAR, p_ativo BOOLEAN)
56 RETURNS INTEGER
57 SECURITY INVOKER
58 LANGUAGE PLPGSQL
59 CALLED ON NULL INPUT
60 AS $$ 
61 DECLARE variavel_id INTEGER;
62 BEGIN
63 IF p_numero IS NULL OR p_nome IS NULL OR p_ativo IS NULL THEN
64     RETURN 0;
65 END IF;
66
67 SELECT INTO variavel_id numero
68 FROM banco
69 WHERE numero = p_numero;
70
71 IF variavel_id IS NULL THEN
72     INSERT INTO banco(numero, nome, ativo)
73     VALUES (p_numero, p_nome, p_ativo);
74 END IF;
75
76 SELECT INTO variavel_id numero
77 FROM banco
78 WHERE numero = p_numero;
79
80 RETURN variavel_id;
81 END; $$;
82
83 SELECT bancos_add_2(331615, 'Banco Novo XYZ', true);
84

```

Data Output Explain Messages Notifications

	bancos_add_2	
	integer	
1	331615	

Aqui temos uma função mais complexa, mas que complementa a função anterior adicionando algumas tratativas.

O primeiro IF faz uma tratativa de erro onde o retorno da função não será mais [null] ao não encontrar registro de banco mas sim 0[zero].

O segundo IF faz um insert dos parâmetros passados se a verificação da função não encontrar registro para aquele numero. Ou seja se a variável for [null].

O ultimo select da função faz novamente a verificação do numero passado como parâmetro e retorna o numero do registro se ele for inserido com êxito.

```

85 SELECT numero, nome, ativo FROM banco WHERE numero = 331615;
86
87

```

Data Output Explain Messages Notifications

	numero	nome	ativo
	[PK] integer	character varying (50)	boolean
1	331615	Banco Novo XYZ	true

Por fim um select na tabela banco confirma que os dados foram realmente inseridos.