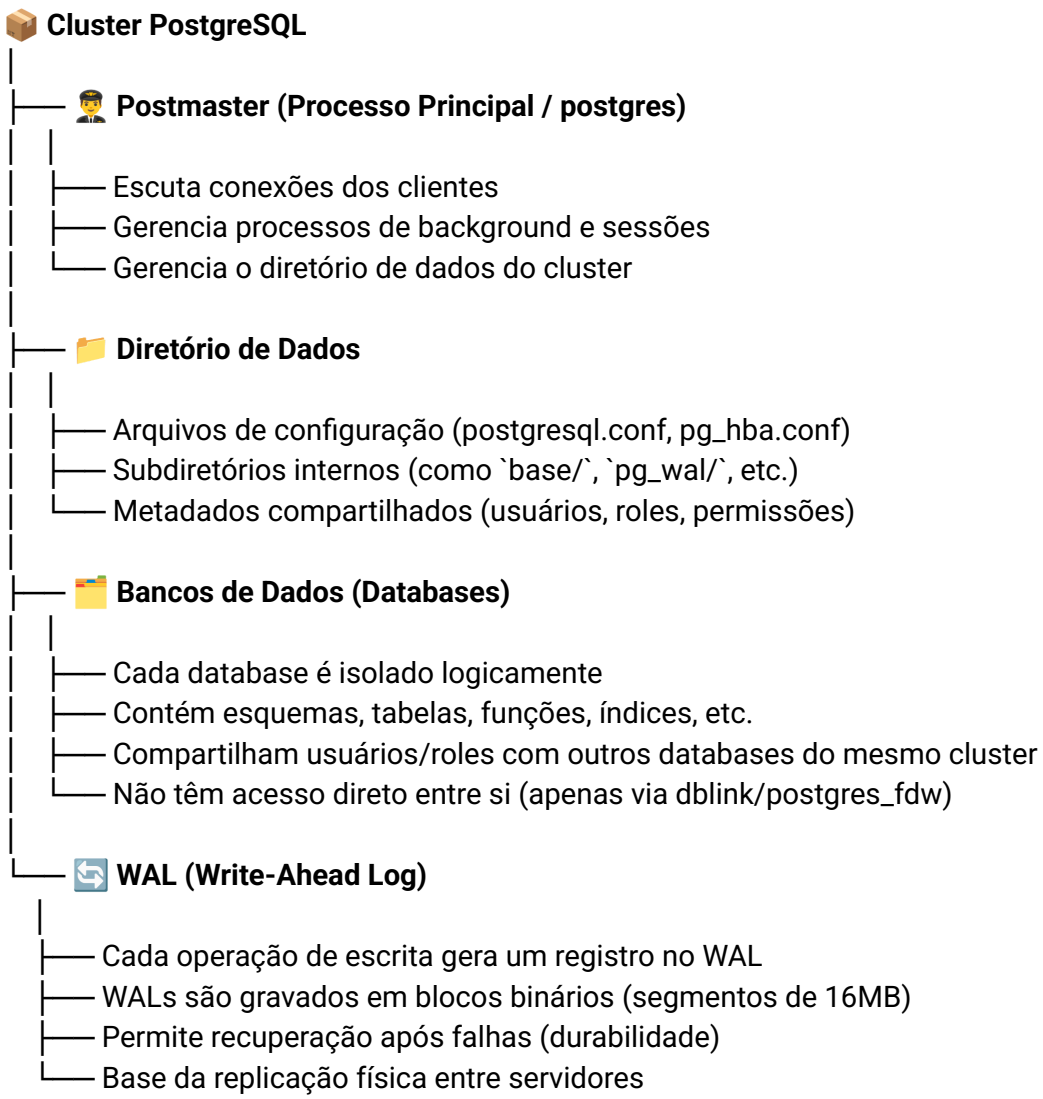
Curso: DBA PostgreSQL com Linux AWS CentOS, Red Hat , Debian

1.Introdução



1.1 CLUSTER

Em PostgreSQL, o termo cluster tem um significado separado e não se refere a um conjunto de servidores trabalhando em conjunto. Um **cluster PostgreSQL** é uma única instância do PostgreSQL que gerencia um conjunto de banco de dados, configurado em um único diretório de dados, com suas próprias configurações, usuários e permissões.

- Um cluster PostgreSQL é uma única instância do servidor PostgreSQL em execução.
- Gerencia múltiplos bancos em um mesmo ambiente isolado.
- Possui um diretório de dados, com arquivos de configuração compartilhados. Portanto, compartilham usuários, roles e configurações.

- São logicamente separados (não é possível fazer joins sem extensões como dblink ou postgres_fdw).

1.2 POSTMASTER

O postmaster é o cérebro da instância PostgreSQL, responsável por inicializar o sistema, gerenciar conexões e processos. Cada cluster PostgreSQL (conjunto de bancos de dados) tem seu próprio postmaster.

1.3 BACKEND PROCESS (servidor do cliente)

Cada conexão com o banco cria um processo separado chamado backend. Ele trata as requisições SQL do cliente. Ou seja, é o processo que o PostgreSQL cria toda vez que alguém se conecta ao banco de dados (usando psql, pgAdmin, sistema PHP, etc)... Ele atende a uma conexão em exclusivo. Se 10 clientes estão conectados ao PostgreSQL, haverá 10 processos backend em execução

1.4 SHARED MEMORY (memória compartilhada):

É usada para a comunicação entre os processos. Nela contém:

- Buffer Cache(armazena blocos de dados usados recentemente).
- Wal Buffers(logs de transações que ainda não foram escritas no banco).
- Lock Tables(controle de concorrência entre transações).

1.5 BUFFER

É um espaço temporário de memória usado para armazenar páginas de dados que estão sendo lidados do disco ou modificadas, antes de serem escritas novamente no disco. Em vez de acessar o disco toda hora, o PostgreSQL guarda esses dados lidos ou modificados na memória ram.

1.6 SHARED BUFFER

Shared Buffer é uma área especial da memória RAM que é compartilhada entre todos os processos do PostgreSQL.

1.7 DATABASE

Database (ou "banco de dados") é um contêiner lógico e isolado que armazena objetos de banco de dados. Cada database é criado dentro de cluster PostgreSQL e é gerenciado pelo mesmo processo postmaster. Mas opera de forma isolada dos outros databases do mesmo cluster.

Os databases compartilham usuários, roles e configurações principais (pois fazem parte do mesmo cluster).
Os objetos (tabelas,esquemas...) são completamente isolados.

Quando se conecta ao PostgreSQL (via psql ou outra aplicação), nos conectamos a um banco de dados específico, não ao cluster inteiro.

1.8 WAL

WAL (Write-Ahead Loggin) registra todas as alterações no banco de dados em arquivos delogs antes que essas mudanças sejam aplicadas. Isso permite que o PostgreSQL recupera o banco de dados mesmo após falhas e incidentes.

- Os arquivos WAL são localizados no diretório pg_wal (antigo pg_xlog). Possuem nomes como: 000000010000000010000003A
- Os comandos que modificam dados (INSERT, UPDATE, DELETE) geram registros WAL, esses registros são armazenados em arquivos WAL.
- Os registros são acumulados sequencialmente em um arquivos WAL até que ele atinja 16MB. Quando está cheio, um novo segmento WAL é criado.
- WALs também são usados na **replicação física** entre servidores (ex: streaming replication).

1.9 CHECKPOINTER

Quando uma transação é feita, as mudanças vão primeiro para o WAL. Isso é rápido e seguro. Essas mudanças ainda não estão no arquivo em disco, somente no WAL e na memória (shared buffer).

O checkpointer verifica de tempos em tempos se não há algo novo no Wal para ser gravado no disco.

Diferença entre PostgreSQL Server e PostgreSQL Client

O **PostgreSQL Server** é o componente principal do sistema de gerenciamento de banco de dados. Ele consiste em um processo daemon (roda continuamente no sistema operacional) que escuta conexões na porta padrão TCP 5432 .

Já o **PostgreSQL Cliente** é qualquer software que se conecta ao servidor para executar operações (psql, pgAdmin, DBeaver)...

Exemplo prático

Exemplo: Sistema WEB

1. O usuário envia um formulário de cadastro para o backend (Exemplo: PHP).

2. O backend se conecta ao PostgreSQL.

O Postmaster está escutando na porta 5432. Ele recebe essa tentativa de conexão e verifica as permissões no pg_hba.conf. Se tudo estiver correto, autentica o usuário.

3. Postmaster cria um backend process

O postmaster cria um backend exclusivo para essa conexão. Esse processo ficará ativo até a conexão acabar, ele roda SQLs, busca e insere dados.

4. O php manda um SQL

INSERT INTO usuarios (nome, email, senha) VALUES ('Carlos', 'carlos@email.com', 'hash');

Esse comando é recebido pelo backend process que consulta a shared memory para ver se os blocos da tabela usuários já estão na memória. Se não tiver, o PostgreSQL carrega do disco para o buffer cache.

5. O backend process escreve no WAL

Antes de inserir os dados na tabela real, ele escreve o comando de forma compactada num arquivo WAL. Nenhuma transação se perde.

6.Dados salvos no disco (ou no buffer)
Depois de registrar no WAL, o dados é escrito no buffer cache e talvez no disco, dependendo da configuração. O postgreSQL pode esperar para escrever no disco real para otimizar o desempenho - por isso o WAL é essencial

7. Conexão encerrada.
O processo backend morre e a memória é liberada.

Fluxo Detalhado referente a Shared Memory:

Modificação na RAM:

A página da tabela users é carregada nos Shared Buffers (se já não estiver lá).

A linha com id = 42 é modificada na memória.

Registro no WAL:

Um registro é adicionado ao WAL buffer (em RAM) descrevendo a mudança.

O WAL buffer é escrito no arquivo WAL em disco (pg_wal/000000010000000010000003A) durante o COMMIT.

Commit:

O cliente recebe confirmação assim que o WAL é persistido.

Persistência no Disco:

Em algum momento futuro, o Checkpointer grava a página modificada do Shared Buffers no arquivo de tabela em disco (base/12345/6789).

Quando fazer um INSERT + COMMIT =

1. Memória e visibilidade

A alteração é feita na memória (buffers em shared memory), visível apenas para a própria transação.

Permite rollback, novas leituras, isolamento entre clientes, etc.

As validações de FK, UNIQUE, CHECK, etc., são feitas antes de gravar na memória.

2. Gravação no WAL e COMMIT

No COMMIT, a alteração é gravada no WAL (Write-Ahead Log), garantindo durabilidade em caso de crash.

O COMMIT é considerado concluído assim que o WAL é escrito com sucesso.

Gravar no WAL é rápido porque o banco apenas faz uma gravação sequencial (append no fim do arquivo), como uma fila.

3. Checkpoint e gravação no disco

Depois, um processo chamado checkpointer varre o WAL periodicamente e grava as alterações de forma definitiva no disco (nos arquivos de dados).

Para consultar os processos do PostgreSQL:

`sudo su -`	para habilitar acesso root
`ps -ef | grep post`	para exibir os processos

2. Instalando o PostgreSQL no AWS Cloud.

2.1 EC2

O EC2 é um serviço da AWS que permite criar servidores virtuais na nuvem.

- Permite escolher o sistema operacional, conectar remotamente, instalar o que quiser, escalar o tamanho (RAM, CPU)...
- É usado para hospedar site, executar um banco de dados ou processar dados em larga escala.

2.2 SSH

SSH é um protocolo de rede que permite acesso remoto seguro a servidores, usando criptografia para proteger a comunicação.

2.3 Como EC2 e SSH se conectam?

- Você cria uma instância EC2 na AWS (Linux, por exemplo)
- A AWS gera uma chave de acesso (arquivo .pem) pra você
- Você usa o SSH + essa chave para se conectar na máquina
- Lá dentro, você instala o PostgreSQL, configura o banco, etc.

2.4 Comandos úteis:

Sistema:

```
sudo -i                          # temporariamente root
sudo dnf update -y               # Atualizar sistema
sudo dnf install [pacote]        # Instalar pacote
sudo reboot                      # Reiniciar
free -h                          # Ver RAM
df -h                            # Ver disco
top                              # Ver processos
```

Arquivos:

```
ls -la                           # Listar arquivos
cd [pasta]                       # Entrar em pasta
mkdir [nome]                     # Criar pasta
rm -rf [arquivo/pasta]           # Apagar (⚠️)
nano [arquivo]                   # Editar arquivo
cat [arquivo]                    # Visualizar o arquivo inteiro
cat [arquivo]  | grep XYZ        # Pesquisa por um trecho
more  [arquivo]                  # Visualizar cada página, permite
rolagem
```

vi ou vim (padrão em servidores):

```
vi [arquivo]                    # Visualizar e editar
cat [arquivo]                   # Visualizar
\work_mem                       # Exemplo, procurar valor no arquivo
i                               # Entrar no modo de edição (INSERT)
Esc                             # Sair do modo de edição
:wq                             # Salvar e sair
:q!                             # Sair sem salvar
:w                              # Salvar sem sair
```

PostgreSQL:

```
sudo systemctl start postgresql-16     # Iniciar serviço
sudo systemctl enable postgresql-16    # Ativar na inicialização
sudo su - postgres                     # Modo usuário postgres
psql                                   # Console do PostgreSQL
sudo -u postgres psql                  # Console do PostgreSQL com usuário
cd /var/lib/pgsql/16/                  # Acessar diretório PostgreSQL
```

Dentro do psql:

```
\l                              # Listar bancos
\du                             # Listar usuários
\x                              # Altera a forma de exibir a consulta
\q                              # Sair
```

2.5 Aceitar conexões com o *postgresql.conf*

Caminho: */var/lib/pgsql/16/data/postgresql.conf*
Ou, pode ser executado no psql: `SHOW config_file;`

O arquivo **postgresql.conf** é o arquivo principal de configuração do servidor do PostgreSQL.

Define os endereços IP que o servidor escuta.

```
listen_addresses = 'localhost'  # Aceita conexões apenas locais
listen_addresses = '*'          # Aceita conexões externas
```

Para aplicar as mudanças:

```
sudo systemctl restart postgresql-16
```

O `listen_addresses` define em quais endereços IP do servidor o PostgreSQL vai escutar as conexões. Ou seja, trata-se dos IPs do SERVIDOR.

Um servidor pode ter vários IPs, pois:
Pode estar conectado a várias redes (internet, rede interna, VPN).
Ou, o administrador pode configurar vários IPs na mesma placa de rede.

Quando colocamos:
`listen_addresses = '*'`
Estamos dizendo ao PostgreSQL: Escute em todos ips que eu (servidor) tiver.

Portanto, ele escutará:
`IP interno (192.168.1.10)`
`IP VPN (10.8.0.1)`
`IP público (201.55.77.88)`

Os ips do cliente configuramos no pg_hba.conf.

Para conexões externas funcionarem, além de listen_addresses, é necessário editar também o arquivo pg_hba.conf.

2.6 Quem e como pode se conectar *pg_hba.conf*

Caminho: */var/lib/pgsql/16/data/pg_hba.conf*
Ou, pode ser executado no psql: `SHOW hba_file;`

O arquivo **pg_hba.conf** é responsável pela autenticação e controle de acesso do PostgreSQL. Ele define quais clientes podem se conectar ao servidor, de onde e com quais métodos de autenticação.

# Tipo	Banco	Usuário	Endereço IP/máscara	Método
host	all	all	192.168.1.0/24	md5

Tipo de conexão: Define o método de conexão.
Exemplos: *local, host, hostssl, hostnossl.*
(SSL é um protocolo de segurança que criptografa os dados, desativada por padrão)

Banco de dados: Nome do banco ou all para todos.
Exemplos: *postgres, meu_aplicativo, all*

Usuário: Nome do usuário ou all para qualquer.
Exemplos: *postgres, usuario_aplicativo, all*

Endereço: IP/mascára (para conexões remotas) ou vazio (para local).

Exemplos: *192.168.1.0/24, 0.0.0.0/0, ::/0 (IPv6).*

Método: Método de autenticação.
Exemplos: *trust, md5, scram-sha-256, cert, reject.*
(O trust é sem autenticação, nunca usado em produção. MD5 está defasado, reject bloqueia explicitamente).

ORDEM: As regras são avaliadas de cima para baixo, a primeira regra correspondente à conexão é a que será aplicada.

```
host  all  all  0.0.0.0/0  reject
host  all  all  192.168.1.0/24  trust
```

Neste caso, todas as conexões serão rejeitadas, pois a primeira regra (0.0.0.0/0) corresponde a qualquer IP.

Portanto, sempre colocar **regras mais específicas antes das genéricas.**

Para aplicar as mudanças de modo menos invasivo (sem reiniciar):
`sudo systemctl reload postgresql-16;`

Para fazer backup do pg_hba.conf:
`sudo cp /caminho/para/pg_hba.conf /caminho/para/backup/pg_hba.conf.bkp`

Questões levantadas:

1. Indentação importa?
No pg_hba.conf a indentação (espaços ou tabs) não importam, é somente para deixar o arquivo legível.

2. O que significa /32 e /24 no IP?

São prefixos de rede.

`192.168.1.100/32:` Somente esse ip específico. (o /32 significa que "todos esses 32 bits do IP estão fixos").

`192.168.2.0/24:` Toda a sub-rede de 256 ips de 192.168.2.0 até 192.168.2.255 (o /24 significa que os primeiros 24 bits estão fixos). Ou seja, a parte 192.168.2 está fixa.

3. No caso do peer não tem address?

Sim, pois quando o type é local o campo de endereço IP é ignorado.

4. Como decidir a ordem das regras?

O PostgreSQL lê o pg_hba.conf de cima para baixo. Portanto, a primeira regra que bater vence. O ideal é permitir quem confia primeiro e no final negar tudo, para garantir que o que não foi coberto será rejeitado.

Exemplo de ordenação básica.

```
# Conexões locais (mais seguras)
local   all         all                         trust

# IP específico (mais específico)
host    all         all         192.168.1.10/32       scram-sha-256

# Rede 10.0.0.0/24 (menos específica)
host    all         all         10.0.0.0/24           md5

# Bloqueia todo o resto
host    all         all         0.0.0.0/0             reject
host    all         all         ::/0                  reject
```

Exemplo mais robusto:

```
# 1. Conexões locais (são as mais seguras)
local   all         postgres                        peer
local   all         all                         peer

# 2. Conexões da máquina local (loopback)
host    all         all         127.0.0.1/32          scram-sha-256
host    all         all         ::1/128               scram-sha-256

# 3. Conexões de administradores de IPs específicos
host    all         admin       192.168.1.10/32       scram-sha-256
host    all         dba_team    192.168.1.0/29        scram-sha-256

# 4. Conexões de aplicações específicas a bancos específicos
host    app_database    app_user    10.0.0.5/32           scram-sha-256
host    reporting_db    report_user 10.0.0.0/24           md5

# 5. Regras mais gerais para redes internas confiáveis
host    all         all         192.168.0.0/16        scram-sha-256

# 6. Regras de negação explícitas (opcional, mas útil)
host    all         all         all                   reject
```

5. Como validar baseado no tipo de ip?

```
# Conexão local via socket (sem rede)
local   all             all                                     trust

# Bloqueia qualquer IPv4
host    all             all             0.0.0.0/0               reject

# Bloqueia qualquer IPv6
host    all             all             ::/0                    reject
```

0.0.0.0/0
Corresponde a qualquer endereço IPv4.

::/0
Corresponde a qualquer endereço IPv6.

Nesse caso haveria diferença entre colocar IPv4 ou IPv6 primeiro? Não faz diferença nesse cenário, pois ambas as regras (0.0.0.0/0 e ::/0) são genéricas e não se sobrepõem.

2.7 Como se relacionam?

- O **postgresql.conf** é a configuração geral do servidor, define como o PostgreSQL se comporta internamente : portas, memória, logs, etc. Ele não controla quem pode acessar, apenas abre espaço para a conexão ser ouvida. Se listen_address estiver como localhost o servidor ignora conexões externas, mesmo que o pg_hba.conf permita.

- O **pg_hba.conf** define quem pode se conectar, a quais bancos, com qual usuário e com qual autenticação. Se não houver uma regra para o IP/usuário a conexão será rejeitada, mesmo que a porta esteja aberta.

2.8 Segurança a nível de instância

1. Roles e usuário são a mesma coisa (diferente do Oracle).

Role: Um conceito genérico que pode representar um usuário (com direito de login) ou um grupo (para agrupar permissões).

Usuário: Um role com a opção LOGIN habilitada.

- *-- Cria um role que pode logar (usuário)*

```
CREATE ROLE app_user WITH LOGIN PASSWORD 'senha_segura';
```

- *Cria um role para agrupar permissões (grupo)*

CREATE ROLE desenvolvedores;

Roles são objetos globais: Quando criadas (via psql, pgAdmin, ou outro cliente), ficam visíveis em toda a instância, e não são restritas ao banco atual.

Em PostgreSQL, temos o role public, que representa todos os usuários, incluindo os que serão criados no futuro.

2. Quem cria um banco de dados é o Owner por padrão.

Quando um banco de dados é criado, o usuário que o cria se torna seu owner (dono) por padrão.

Privilégios do Owner:

- Controle total sobre o banco (criar/alterar/excluir schemas, tabelas, etc.).
- Pode conceder ou revogar privilégios a outros roles.
- Esse cargo pode ser transferido, desde que seja o owner ou um superusuário.
  ALTER TABLE nome_da_tabela OWNER TO novo_dono;

3. Quando o usuário é criado na instância é visível por todos os databases.

- **Roles são globais:** Qualquer role criado na instância do PostgreSQL é visível em todos os databases da instância.
- **Privilégios são locais:** Apesar de serem visíveis globalmente, os privilégios de um role são específicos por database/schema/objeto.

4. Princípio Least Privilege

Revogar tudo e dar a cada usuário/grupo apenas as **permissões mínimas** necessárias para desempenhar suas funções.

Quando é criado um schema, apenas o usuário que criou (o owner) recebe os privilégios sobre ele, o role public não recebe nenhum privilégio ao schema recém criado.

Porém, o schema *public* (que existe por padrão) tem privilégios *públicos (gerais)* automaticamente.

Uma abordagem válida é revogar em níveis diferentes:

```
-- Revoga conexão ao banco
REVOKE CONNECT ON DATABASE nome_do_banco FROM nome_da_role;

-- Revoga acesso ao schema
```

```
REVOKE ALL ON SCHEMA public FROM nome_da_role;

-- Revoga acesso a todas as tabelas
REVOKE ALL ON ALL TABLES IN SCHEMA public FROM nome_da_role;

-- Revoga privilégios em todas as sequences do schema
REVOKE ALL ON ALL SEQUENCES IN SCHEMA public FROM nome_da_role;
```

Em ambientes de produção é comum renomear ou desabilitar o schema public, para evitar uso acidental.

```
-- Bloqueia criação de objetos
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

2.9 Resumo de autenticação e conexão

Em conexões locais feitas via psql sem passar o -U, o postgresql pode assumir automaticamente o nome de usuário do SO para tentar autenticar, por exemplo, via peer. Já em conexões externas via rede, o PostgreSQL não utiliza o usuário do SO, nesse caso é necessário informar explicitamente no momento da conexão.

É importante notar também que o PostgreSQL não utiliza o usuário e senha do próprio pgAdmin (que são apenas para proteger os servidores cadastrados no pgAdmin).

Ou seja, usuário do pgAdmin e do servidores são coisas distintas.

```
psql -h 192.168.1.10 -U desenvolvedores -d minha_base
```

Nesse caso, o usuário **(**role desenvolvedores**)** e senha fornecidos na conexão (a senha será requisitada após a execução do comando).

Esses dados são validados contra as regras do pg_hba.conf e os roles do PostgreSQL. O usuário desenvolvedores também será utilizado como cargo para dar e conceder permissões a ele, ou seja, reforço que cargo e usuário são a mesma coisa.

Se o usuário quiser ter as permissões de outro cargo, será necessário re-logar com ele (se tiver permissão no pg_hba.conf).

3. Instalação PostgreSQL com Docker Containers

3.1 O que é Docker?

Docker é uma plataforma que permite criar, empacotar, distribuir e executar aplicações em "containers". Ele é muito usado para garantir que seu sistema rode igualzinho em qualquer ambiente. É como uma mini-máquina isolada, mas mais leve que uma máquina virtual. O Docker utiliza o mesmo kernel do host, por isso consegue ser mais leve.

3.2 O que é Docker Compose?

O Docker Compose é uma ferramenta que permite definir e rodar vários containers Docker de uma vez só, usando um único arquivo docker-compose.yml

Supondo que queremos subir um ambiente com:

- Um container PostgreSQL
- Outro container PgAdmin
- E queremos que eles conversem entre si na mesma rede

Ao invés de criar cada container manualmente com vários comandos, podemos escrever tudo em um **único arquivo YAML**

3.2.1 Docker Compose CLI (a linha de comando)

Comandos Essenciais:

`docker-compose up`
Função: Iniciar todos os serviços, redes e volumes definidos no arquivo docker-compose.yml

`docker-compose ps`
Função: Listar todos os containers associados ao projeto Docker Compose, mostrando status, portas e nomes.

`docker-compose down`
Função: Parar e remover containers, redes e volumes criados pelo docker-compose up

`docker-compose --help`
Função: Exibe a lista completa de comandos e opções disponíveis no Docker Compose.

3.2.2 Docker Compose File

É o arquivo de configuração chamado **docker-compose.yml**, escrito no formato YAML (uma linguagem de marcação, em formato chave : valor, não usa indentação com tab, somente espaços)

version
Define a versão do Docker Compose
services
Define os containers que compõem a aplicação. **Cada serviço corresponde a um container.**
networks (Opcional)
Permite criar redes personalizadas para comunicação entre containers.
volumes (Opcional)
Gerencia armazenamento persistente para os containers.

6. Conhecendo a estrutura de instalação

6.1 Por que isso é importante?

O PostgreSQL é um banco de dados relacional de objetos open-source, por natureza, gratuito. Porém, há diversas versões e modificações deste SGBD sendo distribuídas não oficialmente, destas, algumas gratuitas, outras não. Portanto, mostra-se essencial quando surge a demanda de administrar um novo banco de dados a necessidade de consultar qual a sua versão, a fim de encontrar as documentações e funcionalidades correspondentes.

Antes da versão 10 do PostgreSQL, tínhamos as versões lançadas da seguinte forma: XYZ (v 9.1.2).

6.2 Como verificar a versão do meu Banco de Dados?

Há inúmeras formas de fazer isso e variam de acordo com a versão. Porém, a que funcionará na maioria dos casos, é através dos comandos a seguir:

```
SELECT version();                # Exibe mais informações, também do SO
SHOW server_version;             # Forma reduzida

psql -V | psql --version          # Via psql
```

6.3 Como verifico o uptime do meu servidor?

```
# Exibe as informações formatadas
SELECT DATE_TRUNC('second',CURRENT_TIMESTAMP -
pg_postmaster_start_time());

#Forma reduzida, indica a data que foi iniciado
SELECT pg_postmaster_start_time();
```

6.4 Onde ficam os arquivos do servidor?

```
# Via sql:
SHOW data_directory;
```

As saídas deverão ser algo semelhante a isso:

Debian ou ubuntu:
`/var/lib/postgresql/MAJOR_RELEASE/main`

Redhat, CentOs, Fedora:
Nesse caso a instalação é padrão do repositório da distro, sem separação por versão.
`/var/lib/pgsql/data`

Podemos ter múltiplas versões do PostgreSQL no mesmo sistema, com diferentes versões e configurações:
`/etc/postgresql/16/dev`
`/etc/postgresql/16/teste`

RESUMO: Diferença entre Ubuntu/Debian vs RedHat/CentOS/Fedora no PostgreSQL
Resumo:

No Ubuntu/Debian, eles separam as coisas:

```
Dados = /var/lib/postgresql/

Configurações = /etc/postgresql/
```

No RedHat/CentOS/Fedora, eles não separam tanto:

Dados e Configurações ficam juntos no data_directory:
`/var/lib/pgsql/VERSAO/data/`

6.5 Onde ficam os arquivos de logs?

Dentro do postgresql.conf podemos configurar como serão salvos os logs. Como, onde, o formato e o nome do arquivo.

Debian ou Ubuntu:
No Debian/Ubuntu, o diretório de logs padrão é compartilhado entre as versões instaladas
/var/log/postgresql

Redhat, CentOs, Fedora:
/var/lib/pgsql/16/data

6.6 Como listo o banco de dados do meu servidor?

Sem se conectar:
`psql -l`

Após se conectar ao psql:
\l
SELECT * FROM pg_database;

6.7 Qual o ID do meu banco de dados?

Cada banco de dados tem um identificador único. Isso é importante pois várias atividades estão ligadas a esse ID, como realizar um backup. A forma de verificação varia com o SO.

Redhat, CentOs, Fedora - Debian ou Ubuntu
Nomes dos diretórios são os OIDs de cada banco:

```
1. SHOW data_directory;          # Chegar neste diretório
2. ls -l                         # Listar diretórios
3. 16384/
   16385/
```

Também é possível verificar via SQL:

```
SELECT oid, datname FROM pg_database;
```

6.8 Quanto de espaço meu banco de dados consome?

É possível fazer calculando o tamanho dos arquivos, mas é muito trabalhoso, pois envolve os subdiretórios.

Para saber do banco de dados atual, passa como parâmetro o current_database:

```
SELECT pg_database_size(current_database());
```

Para saber a soma do tamanho de todos os banco de dados do servidor:

```
SELECT SUM(pg_database_size(datname)) FROM pg_database;
```

6.9 Como alterar a conexão entre os bancos de dados?

```
\c nome_do_banco
```

6.10 Módulos e extensões para PostgreSQL

Em algumas aplicações, há a exigência de ter extensões pré-instaladas para o devido funcionamento. Portanto, é fundamental saber como consultar as extensões.

Para consultar as extensões instaladas:
`SELECT * FROM pg_extension;`

As **extensões**, por serem parte do software PostgreSQL, ficam localizadas num **diretório diferente do servidor**. Pois, por exemplo, isso permite reutilizar os mesmos arquivos de extensão. Também mantém os dados separados do código, que é importante para backups, upgrades e organização do sistema.

Debian ou Ubuntu:
No Debian/Ubuntu, o diretório de logs padrão é compartilhado entre as versões instaladas
/usr/share/postgresql/16/extension/

Redhat, CentOs, Fedora:
/usr/pgsql-16/share/extension/

Para instalar extensões:
```
1. yum install postgresql16-contrib
2. CREATE EXTENSION pgstattuple with schema stats;
```

7. Planejando e criando bancos de dados

O PostgreSQL é um banco de dados muito integrado ao sistema operacional, portanto, é de suma importância alinhar a versão do banco de dados com a versão do SO.

7.1 Escolhendo o melhor layout de storage

Particionamento é a técnica de dividir uma tabela grande em subtabelas menores (partições), com base em um critério como data, ID ou região. A tabela principal continua sendo acessada normalmente pelas aplicações, mas, por baixo dos panos, o PostgreSQL direciona a query para a partição correta — processo conhecido como partition pruning.

- A query lê apenas a partição relevante, o que melhora a performance.
- A manutenção é simplificada, pois é possível, por exemplo, remover uma partição inteira (DROP) sem impactar o restante da tabela.
- Pode ser usado em conjunto com outros recursos como índices, compressão e tablespaces.

Tablespaces são diretórios físicos personalizados onde podemos armazenar objetos do banco de dados (tabelas, índices ou schemas) fora do diretório padrão do PostgreSQL. Eles permitem distribuir o armazenamento entre diferentes discos ou volumes, ajudando na performance, organização e escalabilidade.

- Separação física de dados quentes e frios.

- Balanceamento de carga de I/O (Input/Output) entre diferentes dispositivos.
- Útil para bancos com grande volume de dados ou quando se trabalha com infraestrutura de armazenamento diversificada (ex: SSD + HDD).

Enquanto o **particionamento** atua sobre a divisão lógica de tabelas, o **tablespace** é mais abrangente e faz a distribuição física do storage, atuando sobre tabelas, índices e schemas.

7.2 Novo banco ou novo Schema?

Com um novo banco de dados, as principais vantagens que temos estão relacionadas a segurança, permissões e manutenção.

Usuários e roles são gerenciados por bancos, portanto, evitamos o caso de um usuário ter acesso a dados indevidos. Além disso, podemos fazer backup, restore ou até acidentalmente derrubar um banco sem impactar o outro.

Já com um novo schema, temos economia de recursos e reutilização de conexões:

Tudo está no mesmo banco, então joins e queries entre schemas são simples. Porém, temos menor isolamento entre roles e riscos compartilhados.

Chegamos a conclusão, portanto, que deve-se criar um novo banco quando envolver clientes independentes e aplicações completamente distintas. Criamos um novo schema quando compartilhamos recursos e temos módulos dentro da mesma aplicação (ex: financeiro, rh, estoque).

7.3 Monitoramento

O monitoramento de PostgreSQL é fundamental para manter a performance, escalabilidade e identificar problemas antes que virem crise.

Há ferramentas de monitoramento, entre elas, algumas nativas do PostgreSQL:

pg_stat views (nativas do PostgreSQL)
pg_stat_activity -> conexões e queries em execução.
pg_stat_user -> número de inserts, updates e deletes por tabela.
pg_locks -> locks ativos no momento.
pg_stat_bgwriter -> comportamento do autovacuum e writes em disco.
pg_stat_replication -> status de réplicas

Então assim ficaria melhor?

7.4 LOCK

Locks são travas ou mecanismos de controle de concorrência usados pelo PostgreSQL para garantir que várias transações simultâneas não corrompam os dados.

Se uma transação está atualizando uma linha de uma tabela, o PostgreSQL trava esta linha. Assim, impede que outra transação leia-a ou modifique-a ao mesmo tempo.

7.5 DEADLOCK

Deadlocks ocorrem quando duas ou mais transações (que envolvem locks) impedem entre si que prossigam. Por exemplo:

Transação A bloqueia a linha 1.
Transação B bloqueia a linha 2.

Transação A tenta acessar a linha 2 (bloqueada por B).
Transação B tenta acessar a linha 1 (bloqueada por A).

Resultado: Nenhuma das duas pode continuar, ou seja, um deadlock. O PostgreSQL detecta deadlocks automaticamente e mata uma das transações com o seguinte erro:
ERROR: deadlock detected

7.6 Backup

O Backup **físico** trata-se da cópia exata dos arquivos físicos do PostgreSQL no disco. Ele precisa estar com o PostgreSQL parado, ou, com o WAL ligado e uso de ferramentas específicas. Faz o backup exato dos arquivos, incluindo configurações.

O backup **lógico** exporta os dados em formato SQL ou compactado, como um dump. Ele cria comandos insert, create table, etc, em um arquivo. Gera um script que pode ser executado para "recriar" o banco.

RTO (Recovery Time Objective): Quanto tempo o sistema pode ficar fora do ar.
RPO (Recovery Point Objective): Quanto de dados você pode perder. (Ex: 5 minutos de dados = RPO de 5 minutos)..

7.7 Replicação

Replicação é o processo de manter uma ou mais cópias de um banco de dados sincronizadas com o servidor principal. Utiliza os arquivos de WAL.

1. Replicação **síncrona**

O servidor primário só confirma a transação para o cliente quando o secundário também tiver recebido o dado. Garante zero perda de dados mas impacta a performance.

2. Replicação **assíncrona**.

O primário confirma a transação sem esperar pelo secundário. É mais rápida, porém existe o risco de perda de dados.

7.8 Tablespace

#CRIANDO UMA PARTIÇÃO NO DISCO: CUIDADO O COMANDO ABAIXO PODE DESTRUIR OS DADOS
`fdisk /dev/<CAMINHO-DO-DISCO>`

#LISTANDO AS PARTIÇÕES
`fdisk -l`

#FORMATANDO A PARTIÇÃO COM mkfs
`mkfs -t xfs /dev/xvdg1`

#CRIANDO O PONTO DE MONTAGEM
`mkdir -p /faststorage/tbserp1`

#MONTANDO A PARTIÇÃO
`mount -t xfs /dev/xvdg1 /faststorage/tbserp1`

#AJUSTE DE PERMISSÃO NO PONTO DE MONTAGEM
chown postgres.postgres /faststorage/tbserp1

#VERIFICANDO A PERMISSÃO
`ls -liart  /faststorage/`

#OBS: DEPOIS É PRECISO ADICIONAR O PONTO
#DE MONTAGEM NO `/etc/fstab` para ser permanente

8. Configurando PostgreSQL Server

É vantajoso utilizar a view pg_settings para visualizar as configurações do PostgreSQL, isso pois as configurações podem estar em diferentes arquivos.

8.1 pg_settings

A consulta a seguir conta com um filtro WHERE, que busca somente os parâmetros que não estão com os valores padrão e não foram substituídos por alguma configuração interna. Ou seja, mostra somente aquilo que foi modificado por alguém.

```
SELECT name, source, setting
FROM pg_settings
WHERE source != 'default'
AND source != 'override'
ORDER by 2, 1;
```

O comando SHOW é uma alternativa mais simples de exibir os parâmetros atuais de configuração do PostgreSQL.

8.2 SHOW

```
SHOW ALL;                          # Todos os parâmetros;
SHOW log_directory;                # Valor atual do log_directory;
```

8.3 Alterando valores

É possível alterar os valores a nível de servidor (indo no arquivo de configuração postgresql.conf, por exemplo), ou, a nível de sessão, com o comando SET.

```
SET work_mem = '6MB';
```

Ao alterar a nível de sessão é necessário reiniciar o servidor para que o efeito seja aplicado.

```
/usr/pgsql-16/bin/pg_ctl reload
```

8.4 Alterando parâmetros de configuração a diferentes níveis

#Todos usuários em um banco de dados, por exemplo DB1

```
ALTER DATABASE
SET <configuration_parameter> = valor_parametro;
```

#Um usuário em qualquer banco de dados do meu Database Server

```
ALTER ROLE joao
SET <configuration_parameter> = valor_parametro;
```

#Um usuário específico, mas somente quando conectado em um banco específico
#por exemplo DB1

```
ALTER ROLE joao
IN DATABASE DB1
SET <configuration_parameter> = valor_parametro;
```

9. Privilégios

9.1 Em PostgreSQL, usuário e roles são sinônimos?

Um usuário e role são quase a mesma coisa em PostgreSQL. A diferença é que um usuário é uma role com permissão de login.

ROLE: É um conceito genérico que pode representar usuários, grupos, ou ambos. Pode ter ou não permissão de login. Pode ser usada para agrupar permissões (como um grupo no sistema operacional).

USER: É apenas um atalho/sinônimo para CREATE ROLE com a opção LOGIN. Ou seja, todo USER é uma ROLE, mas nem toda ROLE é um USER.

Usuário = role com LOGIN
Grupo = role sem LOGIN

Há o comando CREATE GROUP, mas ele é apenas um alias para CREATE ROLE sem LOGIN.'

CREATE GROUP equipe; = CREATE ROLE equipe;

No PostgreSQL, usuários são case-sensitive ("user1" != "USER1"). Por padrão, nomes de usuários, tabelas, colunas, etc. são convertidos para minúsculas se não forem colocados entre aspas duplas.

```
CREATE USER = CREATE ROLE + LOGIN PERMISSION
```

Quando um usuário é criado, o PostgreSQL gera um **OID** (Object Identifier), que é o número interno usado para identificar aquele usuário no sistema. Esse OID é usado como owner de objetos como tabelas, esquemas, funções etc.

Criar usuário:

```
createuser --interactive                          #via psql, interativo
CREATE USER dbuser1 WITH password  'Aluno@Postgres'; #via sql
```

Listar usuários:

```
\du
SELECT * FROM pg_user;
```

9.2 Modificar o schema padrão:

Mostra o schema atual:

```
SHOW search_path;
```

Seta o novo schema como atual:

```
SET search_path to erp;
```

9.3 Criar um usuário read-only

Ao criar um novo usuário no PostgreSQL ele recebe por padrão a permissão de criar objetos no schema public. Isso é um problema ao tentar criar um usuário com permissão para somente-leitura.

É necessário revogar os seus privilégios no schema public.
`REVOKE CREATE ON SCHEMA PUBLIC;`

10. Tabelas, dados e visões

10.1 Verificar a estrutura de uma tabela

```
\d                  #lista as tabelas
\d tbfuncionario    #mostra a estrutura da tabela
\d+ tbfuncionario   #mostra a estrutura da tabela com mais detalhes
```

10.1 Povoar tabela com generate_series

```
INSERT INTO tbdistribuidores
SELECT generate_series(1,100,2) col2,
       generate_series(201,250,1) col1;
```

10.2 Alterar tipo de dado de uma coluna com USING

A cláusula USING na instrução ALTER TABLE é utilizada para especificar como os dados existentes em uma coluna devem ser convertidos quando se altera seu tipo. Isso é especialmente útil quando a conversão não é direta ou pode causar perda de dados.

ALTER TABLE tbdistribuidores ALTER COLUMN nome SET DATA TYPE char(20)
USING substr(nome,1,20);

10.3 VIEWS

Uma view pode ser **atualizada** quando baseia-se em apenas uma tabela, inclui a chave primária, e não contém distinct, group by, having, sum, count, union, intersect, joins...

Quando a view é atualizável, qualquer alteração feita nela (INSERT, UPDATE, DELETE) reflete diretamente na tabela base.

VIEW atualizável:

```
CREATE VIEW clientes_ativos AS
SELECT id, nome, email FROM clientes WHERE ativo = true;
```

VIEW NÃO atualizável (pois inclui junção):

```
CREATE VIEW pedidos_com_clientes AS
SELECT p.id, p.data, c.nome
FROM pedidos p JOIN clientes c ON p.cliente_id = c.id;
```

10.4 MATERIALIZED VIEWS

Uma view materializada armazena fisicamente os resultados da consulta. Ela mantém uma cópia estática dos dados no momento da atualização. Portanto, requer uma atualização manual ou programada.

```
CREATE MATERIALIZED VIEW mv_clientes_ativos AS
SELECT * FROM clientes WHERE ativo = true;
```

Para atualizar / dar refresh na view materializada:

```
REFRESH MATERIALIZED VIEW mv_clientes_ativos;
```

10.5 Mover tabelas entre schemas

```
ALTER TABLE tbdistribuidores SET SCHEMA lab2;
```

11. BACKUPS

11.1 COPY

O comando COPY permite importar dados entre o banco de dados e arquivos externos. Ele oferece alta performance para operações de carga e extração de dados. Costuma ser muito mais rápido que comandos INSERT.

1. COPY (SQL)

- Opera no servidor PostgreSQL.
- Acessa arquivos no sistema de arquivos do servidor.
- Requer privilégios de superusuário.

2. \copy (psql)

- Opera no cliente psql.
- Acessa arquivos da máquina do cliente.

- Não requer privilégios especiais.

Exportar dados (de tabela para arquivo):

1. Tabela completa:
```
COPY lab.cliente
TO '/tmp/clienteFull.csv'
WITH (
FORMAT csv, HEADER TRUE, delimiter ','
);
```

2. Com cláusula where:
```
COPY(
      SELECT * FROM lab.cliente WHERE client_id < 501
) TO '/tmp/clienteParcial.csv'
WITH (
FORMAT csv, HEADER TRUE, delimiter ','
);
```

Importar dados (de arquivo para tabela):

```
COPY lab.cliente2
FROM '/caminho/arquivo.csv'
WITH (FORMAT csv, HEADER true, DELIMITER ';');
```

11.1 pg_dump

O pg_dump extrai um banco de dados PostgreSQL em um único arquivo de script ou outro arquivo. Esses arquivos podem ser restaurados com o pg_restore.

Permite fazer backups de forma consistente mesmo se o banco de dados estiver sendo usando no momento, ele não bloqueia acessos, leituras e escritas.

O pg_dump faz o dump somente de um único banco de dados.

Sintaxe básica:

Banco inteiro:
```
pg_dump -d meu_banco -f backup.sql
```

Tabela específica:
```
pg_dump -t minha_tabela -d meu_banco -f backup.sql
```

Opções de formato:

```
pg_dump -F p -d meu_banco -f backup.sql
```

-F p (plano, script SQL, com comandos CREATE e INSERT),
-F c (custom, formato binário, usado com pg_restore),
-F t (tar, compactável, usado com pg_restore)
-F d (diretório, arquivos binários, usado com pg_restore)

Opções de conteúdo:

Há opções curtas e opções longas. São equivalentes. É possível misturar as opções no mesmo comando. Para scripts de longo prazo, é preferível as opções longas por legibilidade.

-a, --data-only: Somente dados (sem DDL)
-s, --schema-only: Somente schema (sem dados)
-n, --schema=NOME_SCHEMA: Backup de schema específico
-t, --table=NOME_TABELA: Backup de tabela específica
-T, --exclude-table=NOME_TABELA: Ignora tabela específica
-N, --exclude-schema=NOME_SCHEMA: Ignora esquema específico

11.2 pg_restore

O pg_restore restaura um banco de dados PostgreSQL através de um arquivo criado pelo pg_dump.

Restaura backups binários (não-SQL), não funciona com arquivos SQL planos (-F p). Ele executará os comandos necessários para restaurar o banco de dados para o estado em que foi salvo.

Portanto, lê os arquivos nos formatos:

- custom (-F c)
- tar (-F t)
- diretório (-F d)

O pg_restore pode ser selecionado e selecionar o que será restaurado, como também a prioridade dos itens.

Sintaxe básica:
pg_restore -F d meu_banco -f tmp/export

Opções:
```
pg_restore [OPÇÕES] [ARQUIVO_DE_BACKUP]
```
Sendo [OPÇÕES] as mesmas citadas no pg_dump

11.3 pg_dumpall

Para fazer o dump de um cluster inteiro utilizamos a função pg_dump_all. Ela basicamente faz um loop que percorre cada banco de dados do cluster e realiza um pg_dump.

Ele não apenas faz backup dos bancos de dados, mas também de usuários (roles), tablespaces e configurações globais. No entanto, configurações globais (como alterações no pg_hba.conf) não são incluídas.

O pg_dumpall gera um único arquivo SQL com o formato de texto plano (plain).

Sintaxe básica:

```
pg_dumpall -U postgres -f /tmp/cluster.sql
```

11.4 Paralelismo no Backup

Por padrão, se não especificado o parâmetro -j (--jobs) o PostgreSQL executa com 1 thread. Ou seja, somente 1 processo.

Paralelismo trata de dividir o processo de dump ou restauração em várias threads (ou processos) simultâneos, o que acelera muito a operação. Essa diferença é vista especialmente em bancos grandes. O paralelismo consome mais CPU e mais disco, então é útil em servidores com bons recursos.

É suportado no pg_dump e pg_restore quando utilizados com o formato custom (-F c) ou directory (-F d).

Não é suportado no pg_dump com o formato plain (-F p) e no pg_dumpall (que nunca é paralelo).

Sintaxe básica:

```
pg_dump -F d -j 4 meu_banco -f /etc/export
```

A diferença está no uso do parâmetro j (jobs), que indica o número de threads. Nesse caso, criará um backup utilizando 4 threads simultâneos.

11.5 Backup Físico

Um backup físico é um backup de baixo nível, que realiza uma cópia / clone do banco. É útil quando queremos recriar completamente um banco de dados ou cluster. O banco não precisa ser parado completamente, permitindo que continue operacional durante o processo. Isso é possível graças ao mecanismo de WAL (Write-Ahead Logging) e ao controle de ponto de verificação (checkpoint).

Com o WAL, ocorre o arquivamento de logs para consistência, assim que o processo termina, o checkpoint trata de sincronizar os dados com o disco, realizando um "replay" de transações pendentes.

Sintaxe básica:

```
pg_basebackup -D /tmp/bck_fisico -l 'Meu primeiro backup físico' -v
-h localhost -p 5432 -U postgres
```

12. Acessando outros bancos com DBLINKS

12.1 O que são DBLINKS?

Database Links são conexões que permitem a um banco de dados acessar dados armazenados em outro banco (até mesmo de outro SGBD, dependendo da extensão utilizada).

Com a extensão DBLINK é possível executar consultas remotas em outro banco, inserir, atualizar ou deletar dados.

Pré-requisitos:

A extensão dblink deve estar instalada no PostgreSQL
É necessário ter credenciais de acesso ao banco remoto
O servidor PostgreSQL deve ter permissão para se conectar ao banco remoto

12.2 Passo-a-passo

1. Dentro do servidor que RECEBERÁ a conexão:
a) Preparar o postgresql.conf para escutar as conexões do servidor que conectará.

listen_addresses = '*' Substituir '*' pelo IP específico do servidor que fará a conexão (mais seguro).

b) Preparar o pg_hba.conf para autenticar a conexão do servidor que conectará.

2. Agora sim, podemos fazer a conexão com o comando dblink_connect().

Mesmo estando na mesma máquina, o PostgreSQL trata os bancos como isolados. Por isso, precisamos do dblink.

Na prática:

1. Verificar se o pacote contrib está instalado:
```
sudo yum list installed | grep contrib
```

2. postgresql.conf: Configurar o listen_adress = *

3. pg_hba.conf: Configurar o host / all / 0.0.0/0 / scram-sha-256

4. Verificar extensões disponíveis no psql:
```
SELECT * FROM pg_available_extensions ORDER BY name;
```
Verificar extensões instaladas:
```
\dew
```

5. Se não estiver instalada:
```
CREATE EXTENSION IF NOT EXISTS dblink;
```

6.
```
CREATE USER userdblink PASSWORD 'Userdb1Pass';
```

7. Criando o server
```
CREATE SERVER server_local FOREIGN DATA WRAPPER dblink_fdw
OPTIONS (host '127.0.0.1', dbname 'db2', port '5432');
```

8. CRIANDO O MAPEAMENTO
```
CREATE USER MAPPING FOR userdblink SERVER server_local
OPTIONS (user 'userdblink', password 'Userdb1Pass');
```

9. Dando o Grant
```
GRANT USAGE ON FOREIGN SERVER server_local to userdblink;
```

10. Testando a Conexão DBLINK server_local
```
SELECT dblink_connect('server_local');
```

11. Vamos criar uma tabela no banco db2
```
saia e conecte no database db2
psql
\c db2
```

12. crie uma tabela
```
create table funcionario (id integer, nome varchar(30));
insert into funcionario values (1, 'JOE');
insert into funcionario values (2, 'JOAO');
```

13. Saia e conecte no db1 novamente com o user que tem o mapeamento
```
psql -U userdblink -d db1
```

14. Testando conexão
```
SELECT dblink_connect('dbname=db2');
```

15. Vamos Adicionar Dados no Banco db2
```
SELECT dblink_exec('insert into funcionario
values(21,''MARIA'');');
```

16. Selecionar os dados remotamente no db2
```
select * from dblink('dbname=db2 user=userdblink
password=Userdb1Pass', 'select * from funcionario;') as t1 (a
int, b varchar(30));
```

14. Criando e Operando TABLESPACES

14.1 Criando passo-a-passo

1. Adicionar o device no servidor. No caso da AWS, adicionaremos um volume e faremos um attach na instância desejada.

2. Listar os devices para identificar o device que foi adicionado:
```
fdisk -l
```

3. Criar uma partição neste device. (Partição se refere a divisão / pedaço do armazenamento, útil para organizar aplicativos, bancos... em partes separadas).
```
fdisk /dev/nome_device
```

4. Pedirá para escolher opções, as usadas na aula foram:
```
n, p, 1, enter, w
```

5. Formatar a partição com o filesystem
```
mkfs.ext4 /dev/nome_particao_device
```

6. Cria uma pasta que será o ponto de montagem (é o diretório onde o dispositivo será acessado. Ou seja, declaramos uma pasta onde, dentro dela, o conteúdo da tablespace aparecerá. O parâmetro -p indica “parents” e criará todos os diretórios faltantes).

```
mkdir -p /data/tbs
```

7. Monta o sistema de arquivos da partição no ponto de montagem. Ou seja, a partição fica acessível através desse diretório.
   ```
   mount /dev/<nome-device> /data/tbs
   ```

8. Muda a permissão de forma recursiva para o usuário postgres
   ```
   chown -R postgres:postgres /data/tbs
   ```

9. Podemos criar diretórios para cada tablespace. Ou seja, 1 device terá 2 tablespaces.
   ```
   mkdir -p /data/tbs/tbs01
   mkdir -p /data/tbs/tbs02
   ```

10. Criando as tablespaces no PostgreSQL:
    ```
    CREATE TABLESPACE tbs01 LOCATION '/data/tbs/tbs01';
    CREATE TABLESPACE tbs02 LOCATION '/data/tbs/tbs02';
    ```

14.2 Dropando TABLESPACE

Para listar as tablespaces:

```
\d pg_tablespace;
SELECT * FROM pg_tablespace;
```

Primeiro, temos que garantir que ela está vazia.

```
DROP TABLESPACE tbs02;
```

14.3 Mover objetos entre TABLESPACE

```
ALTER TABLE LAB.CLIENTE SET TABLESPACE TBS02;
```

15. Saúde do Banco de Dados

15.1 VACUUM

O PostgreSQL utiliza MVCC para garantir isolamento entre transações. Assim, cada modificação em uma linha (tupla) cria uma nova versão dela. Quando a linha é atualizada (UPDATE) ou excluída (DELETE) a versão antiga se torna um "dead tuple", que não é mais visível para transações futuras mas ainda ocupa espaço em disco.

Com o tempo, esses "dead tuples" acumulam-se causando inchaço nas tabelas e índices, que degradam o desempenho.

O VACUUM recupera o espaço, atualiza estatísticas e marca transações antigas como "congeladas".

a. VACUUM padrão
Remove as "dead tuples" e libera espaço para o PostgreSQL. Não bloqueia operações de leitura e escrita.

b. VACUUM ANALYSE
Faz o mesmo com o VACUUM padrão com o adicional que atualiza estatísticas para o planejador de consultas.

c. VACUUM FULL
Reescreve a tabela inteira em um novo arquivo, removendo todo o espaço não utilizado e devolvendo ao sistema operacional. Bloqueia as operações e é mais lento que o VACUUM padrão, porém, reduz drasticamente o tamanho da tabela / arquivo.

15.2 AUTOVACUUM

O AUTOVACUUM é um processo em segundo plano que executa o VACUUM e ANALYSE em segundo plano. Ele monitora tabelas e índices para saber quando agir. É ativado por padrão.

15.3 Wraparound

O PostgreSQL usa ids de transação, quando ele ultrapassa o limite de 4 bilhões ele dá a volta (wraparound) e volta a ser zero. Portanto, transações antigas podem ser interpretadas como transações futuras. Quando o VACUUM encontra uma linha muito antiga, o XID dela passa a ser do tipo FrozenXID, que não entra em conflitos com novos XIDS.

15.4 Configurando a manutenção automática

Parâmetros mais importantes:

É importante setar a porcentagem mas também um número de linhas. Pois, por exemplo, numa tabela com 2 linhas, se deletar 1, já seria 50% de linhas alteradas. Isso já atende a porcentagem, porém, não atenderia ao número absoluto de linhas. O que é vantajoso, pois um VACUUM para 1 linha alterada não é interessante.

Mínimo de linhas para VACUUM:
`autovacuum_vacuum_threshold = 50`

Mínimo de linhas para ANALYZE:
`autovacuum_analyze_threshold = 50`

Mínimo de porcentagem de linhas para VACUUM:
`autovacuum_vacuum_scale_factor = 0.2`

Mínimo de porcentagem de linhas para ANALYZE:
`autovacuum_analyze_scale_factor = 0.1`

Tempo entre verificações de quais tabelas precisam VACUUM/ANALYZE. Define o intervalo de tempo entre cada "rodada" de verificação do autovacuum daemon. Esse timer passa a rodar e resetar após operação
`autovacuum_naptime = 1`

16. Otimização com índices

16.1 Entendendo estágios Geral de um comando SQL

1. Parse (Análise léxica e sintática)
Interpreta o texto SQL digitado pelo usuário.

Verifica a sintaxe e transforma a consulta em uma representação interna chamada árvore de análise (parse tree).

Se houver erro de sintaxe, ele é detectado aqui.

2. Reescrita (Query Rewrite)
A árvore de análise é passada para o rewrite system, que pode modificá-la.

Essa etapa aplica regras do sistema de regras do PostgreSQL (por exemplo, views são expandidas aqui).

Não é focada em otimização ainda — é mais uma transformação lógica da consulta.

3. Otimização (Planner/Optimizer)
O planner analisa a árvore reescrita e gera várias estratégias possíveis para executá-la (diferentes planos).

Com base nas estatísticas do ANALYZE, escolhe o plano de execução com menor custo estimado.

Aqui ocorre a escolha entre index scan, seq scan, nested loop, hash join, etc.

4. Execução

O executor segue o plano de execução escolhido.

Vai até o storage (disco/memória) e realiza a leitura, escrita, atualização ou exclusão dos dados.

Retorna os resultados ao cliente (no caso de SELECTs).

16.2 Otimizador de consultas

CBO - Cost-Base Optimizer
O otimizador toma decisões baseadas em custo estimado, não em regras fixas.

Dentro do CBO, temos o estimador. Ele calcula o número de linhas a serem processadas, o custo estimado de diferentes operações (JOIN, FILTRO, leitura..). Ele obtém esses dados de estatísticas coletadas via ANALYZE e armazenadas no banco.

16.3 Nodes

Tabela heap é a estrutura de armazenamento básica e padrão da tabela.

SEQUENTIAL SCAN
Lê linha por linha da tabela, do início ao fim.
Exemplo: `SELECT * FROM produtos WHERE preco > 10;` (sem índice em preco).

INDEX SCAN
Usa index para encontrar a linha e depois acessa a tabela para buscar os dados desejados.
Exemplo: SELECT nome FROM clientes WHERE id = 5; (índice em id).

INDEX ONLY SCAN
Todas as colunas necessárias são cobertas por índices.
Se tivermos duas colunas (id e nome) num SELECT para que funcione o INDEX ONLY SCAN é necessário um índice composto (multicoluna) envolvendo as colunas utilizadas. Dois ou mais índices separados não são suficientes. Nesse caso, seria necessário acessar a tabela em si (heap).
Exemplo: SELECT id FROM clientes WHERE id = 5; (índice cobre tudo).

NESTED LOOP JOIN
Para cada linha da tabela A, procura correspondência na tabela B

HASH JOIN
Cria uma tabela hash da menor tabela (ou da que possui filtro) e percorre a outra procurando correspondência

MERGE JOIN
Ordena ambas as tabelas pela coluna de join e depois as funde em uma única passagem.

16.4 Obter informações de índices

Substituir “costumer” pelo nome da tabela.

```
SELECT relname, relpages, reltuples,
          i.indisunique, i.indisclustered, i.indisvalid,
          pg_catalog.pg_get_indexdef(i.indexrelid, 0, true)
          FROM pg_class c JOIN pg_index i on c.oid = i.indrelid
          WHERE c.relname = 'customer';
```

16.5 Criar um índice

```
CREATE INDEX  idx_usuarios_id_nome ON tbusuarios(id,nome);
```

16.6 REINDEX

O REINDEX reconstroi o índice existente, é útil quando o índice foi corrompido ou houve muitas atualizações / deletes.

```
REINDEX INDEX nome_do_indice;       -- Reconstrói um índice
específico
REINDEX TABLE nome_da_tabela;       -- Reconstroi todos os índices
de uma tabela
REINDEX DATABASE nome_do_banco;     -- Reconstroi todos os índices
do banco
REINDEX SCHEMA nome_do_esquema;     -- Reconstrói todos os índices
de um esquema
```

16.7 CONCURRENTLY

Criar índices e reindexar bloqueiam operações de escrita (INSERT, UPDATE, DELETE) enquanto rodam. O CONCURRENTLY permite ainda realizar essas operações, mas, quando a indexação terminar, ele fará outro loop procurando os registros que ainda não estavam presentes para indexá-los.

Substituir “costumer” pelo nome da tabela.

16.8 Saber se meu índice está sendo utilizado

```
SELECT indexrelname, idx_scan, idx_tup_read, idx_tup_fetch FROM
pg_stat_user_indexes WHERE relname = 'customer';
```