Guia Básico para Administração de Bancos de Dados:

PostgreSQL e MySQL (MariaDB)

Eduardo Maroñas Monks

Guia Básico para Administração de Bancos de Dados: PostgreSQL e MySQL (MariaDB)

Eduardo Maroñas Monks

ISBN: 978-65-01-59716-4

2025

Sumário

Sobre o Autor

Possui graduação em Bacharel em Análise de Sistemas pela Universidade Católica de Pelotas (1998), Mestrado em Computação pela Universidade Federal do Rio Grande do Sul (2006) e Doutorado pela Universidade Federal de Pelotas (2023). Começou a atuar na área de redes em 1995 como estagiário na UCPEL, onde trabalhou por 14 anos. No período de 1999 a 2004 e 2006 a 2007, atuou como docente em cursos de Ciência da Computação e Tecnologia em Análise e Desenvolvimento de Sistemas na UCPEL. Em 2007, passou a exercer a função de docente da UniSenac Pelotas, em que esteve na coordenação da criação do curso superior em Redes de Computadores, curso que atingiu a nota máxima no ENADE de 2017. Em 2014 tornou-se Analista de TI na UFPEL, atuando na administração de redes. Possui certificados Cisco CCAI, Cisco CCNA e LPIC-1.

Para saber mais: Canal do Youtube criado em 2009 com mais de 600 vídeos sobre a área de redes, disponível em **http://www.youtube.com/emmonks**

Aviso Importante

Sobre esta Obra

Este livro pode ser usado de forma livre, desde que os créditos e as referências sejam publicados. As informações contidas no documento são de minha autoria, fruto de experiências de mais de 25 anos atuando na área. Nos casos em que são apontadas ferramentas ou sites, poderá haver inconsistências no acesso futuro e poderão causar indisponibilidades.

Se este livro tiver ajudado de alguma forma, por favor, contribua com o valor que achar justo como forma de agradecimento, um cafezinho está ótimo. Obrigado!

Chave do PIX: **7f7fa84a-6ce9-45da-a42f-a77d18eb232c**

Este livro também pode ser obtido em formato físico na loja da editora UICLAP, disponível em https://loja.uiclap.com/

Introdução

Este livro é um guia prático para administrar bancos de dados PostgreSQL e MySQL/MariaDB, dois dos sistemas de código aberto mais utilizados no mercado. Para um iniciante na área de bancos de dados ou para um profissional que precisa gerenciar esses sistemas como parte de suas responsabilidades em infraestrutura, este material poderá ser de grande ajuda nas rotinas de administração da infraestrutura de TI.

Os bancos de dados relacionais operam principalmente com a linguagem SQL (*Structured Query Language*) para manipular dados e estruturas. No entanto, a administração de um SGBD (Sistema Gerenciador de Banco de Dados) vai além: inclui configuração, criação de usuários, backups, monitoramento, otimização e replicação. Embora modelagem e consultas SQL não sejam o foco principal de um administrador de infraestrutura, entender o básico ajuda a solucionar problemas e dimensionar recursos com mais eficiência.

Cada SGBD tem suas particularidades. Enquanto o MySQL/MariaDB usa comandos como **SHOW TABLES**, o PostgreSQL emprega meta-comandos como **\d** no seu console. Este livro aborda justamente essas diferenças, apresentando os comandos e configurações essenciais para administrar PostgreSQL e MySQL com confiança.

Seja para implantar um novo servidor, garantir a segurança dos dados ou melhorar o desempenho, este guia oferece as ferramentas necessárias para otimizar as rotinas dos administradores de bancos de dados.

Boa leitura!

1) Fundamentos

Um SGBD (Sistema Gerenciador de Banco de Dados) é, em termos simples, um software que permite armazenar, organizar e recuperar grandes volumes de dados de forma eficiente e segura. Uma analogia interessante para um SGBD seria tal como um servidor de arquivos com funcionalidades avançadas para o gerenciamento de arquivos, com a diferença que ele gerencia dados estruturados em vez de arquivos comuns.

Do ponto de vista da infraestrutura de redes, o SGBD é um serviço que é executado em um sistema operacional, que fica à espera de conexões de clientes por meio de protocolos de rede. Assim como um servidor web ou de e-mail, os servidores de banco de dados utilizam portas de comunicação específicas (por exemplo, a porta 5432 para PostgreSQL e 3306 para MySQL) para receber requisições de clientes de aplicações locais ou em rede e entregar as informações solicitadas. Gerenciar um SGBD é garantir que esse serviço esteja sempre disponível, com desempenho condizente com as aplicações disponíveis e seguro na rede.

Dentro de um SGBD, os dados são organizados de forma hierárquica e relacional.

- **Banco de Dados (*Database*):** É o contêiner principal que agrupa todos os dados e objetos relacionados a uma aplicação ou sistema específico. Podem haver vários bancos de dados em um mesmo SGBD.
- **Esquema (*Schema*):** No PostgreSQL, um esquema é um *namespace* que organiza objetos como tabelas, funções e índices dentro de um banco de dados. É como uma pasta para organizar objetos. No MySQL, o conceito de esquema é frequentemente sinônimo de banco de dados.
- **Tabela (*Table*):** É onde os dados são realmente armazenados, organizados em linhas e colunas, como uma planilha. Por exemplo, uma tabela "clientes" pode conter informações sobre os usuários de um determinado sistema.
- **Coluna (*Column*):** Cada coluna em uma tabela representa um atributo específico dos dados. Por exemplo, na tabela "clientes", haveria colunas como "Nome", "Email", "DataNascimento". Cada coluna tem um tipo de dado (texto, número, data, entre outros).

- **Linha (*Row*) / Registro (*Record*):** Uma linha representa um único item ou registro dentro de uma tabela. No exemplo da tabela "clientes", cada linha seria um registro de um cliente.
- **Chave Primária (*Primary Key*):** É uma coluna (ou conjunto de colunas) que identifica de forma única cada linha em uma tabela. Ela garante que não haja duas linhas idênticas e é fundamental para a integridade dos dados. É uma forma de identificar unicamente um registro em uma tabela no banco de dados.
- **Chave Estrangeira (*Foreign Key*):** É uma coluna em uma tabela que faz referência à chave primária de outra tabela. Ela estabelece um relacionamento entre as tabelas, garantindo a integridade referencial. Por exemplo, em uma tabela "pedidos", a chave estrangeira "id_cliente" ligaria cada pedido ao cliente correspondente na tabela "clientes".

A normalização é um conjunto de regras para organizar as colunas e tabelas de um banco de dados, visando **r**eduzir a redundância de dados e melhorar a integridade das informações. Para o profissional de infraestrutura de TI, não é preciso ser um especialista em normalização, mas é útil saber que bancos de dados bem normalizados tendem a ser mais fáceis de gerenciar, menos propensos a erros e, muitas vezes, mais eficientes em termos de armazenamento. Dados redundantes ou mal estruturados podem levar a problemas de desempenho e consumo excessivo de armazenamento em disco, que são preocupações diretas da infraestrutura, principalmente, no desempenho das consultas SQL.

Quando se trata de escolher entre PostgreSQL e MySQL (ou MariaDB, um fork popular do MySQL), ambos são excelentes opções de SGBDs relacionais de código aberto. A escolha ideal muitas vezes depende dos requisitos específicos do ambiente e das preferências operacionais da equipe de desenvolvimento. Para a equipe de infraestrutura de TI são importantes os recursos computacionais necessários para que o banco de dados possa ser executado com desempenho esperado.

O PostgreSQL (https://www.postgresql.org/) é reconhecido por sua robustez, extensibilidade e estrita conformidade com os padrões SQL. Ele é considerado um banco de dados de nível empresarial, muitas vezes comparado a soluções proprietárias como Oracle. O PostgreSQL é conhecido por seu sistema de transações avançado e recuperação de falhas, o que é crítico para garantir a integridade dos dados em cenários de alta exigência. Oferece suporte nativo a tipos de dados complexos (como JSONB para documentos, arrays), expressões regulares avançadas e extensões (PostGIS para dados geográficos, por exemplo). Isso pode ser relevante

se as aplicações usam esses recursos e precisam de um banco de dados que os suporte de forma nativa e eficiente.

Geralmente, o PostgreSQL suporta muito bem com grandes volumes de dados e operações complexas em um único servidor, exigindo mais recursos (CPU, RAM, I/O de disco) do hardware ou dos recursos de máquina virtual onde está sendo executado. Possui um formato de licença baseado em BSD que permite grande flexibilidade de uso. Os logs são extremamente detalhados e configuráveis, o que facilita muito a depuração e auditoria, um ponto chave para a equipe de infraestrutura. Alguns cenários de uso do PostgreSQL:

- Sistemas que demandam alta integridade transacional (ex: sistemas financeiros, e-commerce crítico)
- Aplicações com dados geográficos (com PostGIS)
- Ambientes que buscam compatibilidade com SQL padrão e flexibilidade para tipos de dados não convencionais
- Cenários que necessitam de *logs* detalhados para monitoramento e auditoria de segurança
- Quando a equipe de infraestrutura valoriza estabilidade e um ecossistema mais tradicional de banco de dados

O MySQL (https://www.mysql.com/) (e versão a partir de um fork do projeto original MariaDB (https://mariadb.org/)) é o SGBD de código aberto mais popular do mundo, amplamente utilizado em aplicações web, especialmente no famoso stack LAMP (Linux, Apache, MySQL, PHP). É conhecido por sua facilidade de uso, velocidade e escalabilidade horizontal. O MySQL, geralmente, é mais simples de configurar e começar a usar, do que o PostgreSQL, o que pode ser uma vantagem em ambientes que exigem agilidade na implantação. É otimizado para lidar com um grande volume de operações de leitura, sendo ideal para aplicações que servem muitas requisições de dados. Possui diversas opções de replicação nativas (*Master-Slave*, *Master-Master*, *Group Replication*) que facilitam a distribuição da carga de trabalho entre múltiplos servidores, um recurso valioso para infraestrutura de TI.

Devido à sua popularidade, há uma vasta comunidade, muitos recursos online e diversas ferramentas de gerenciamento e monitoramento disponíveis. Permite escolher diferentes *storage engines* (como InnoDB para transações e MyISAM para leitura rápida), permitindo otimizar o banco para diferentes cargas de trabalho. Alguns cenários de uso do MySQL:

- Aplicações Web e Sites de Alto Tráfego: Especialmente onde o volume de leituras é muito maior que o de escritas (ex: blogs, e-commerce com muitos visitantes)
- Projetos que exigem implantação rápida e curva de aprendizado mais suave
- Ambientes que se beneficiam de escalabilidade horizontal e de replicação avançada
- Sistemas que já utilizam o ecossistema LAMP.
- Quando a equipe de infraestrutura prefere um banco de dados leve e rápido, com um grande suporte da comunidade

Não há uma competição para saber se há uma melhor entre MySQL e PostgreSQL, o ideal é aquele que melhor se adequa às necessidades de um projeto, junto a equipe de desenvolvimento e da equipe de infraestrutura de TI. Entretanto, se a integridade dos dados, a conformidade rigorosa com SQL, a extensibilidade e a robustez são as prioridades máximas, mesmo que isso signifique um pouco mais de complexidade na configuração e uso de recursos, o PostgreSQL pode ser a escolha ideal. Se a facilidade de uso, a velocidade para cargas de trabalho intensivas em leitura, a vasta comunidade e a capacidade de escalar horizontalmente são mais importantes, o MySQL (ou MariaDB) pode ser o mais indicado.

A escolha por um banco de dados favorito, em muitas vezes, não dependerá da escolha do administrador. Em muitos cenários os sistemas de bancos de dados já estão sendo utilizados pelas aplicações e deverão ser administrados independentemente da preferência do profissional será o responsável por dar suporte.

2) PostgreSQL

O banco de dados PostgreSQL é um projeto em código-fonte aberto bastante popular. O começo foi com o Projeto POSTGRES iniciado nos anos 80 na Universidade de Berkeley. Nos anos 90, com o nome de Postgres95 foi disponibilizado como software livre. Os desenvolvedores tinham como foco o padrão SQL (*Strucuture Query Language*). Posteriormente, o nome foi simplificado para PostgreSQL. O PostgreSQL é um banco robusto, confiável e possui uma comunidade bastante ativa e está disponível para Linux, Windows, MacOS, FreeBSD entre outros.

A versão do banco de dados PostgreSQL utilizada para os exemplos de comandos foi a **13.18** sendo executada em sistema operacional **Linux** na distribuição **CentOS Stream release 9**.

Para instalar o servidor e cliente do PostgreSQL no CentOS Stream release 9

```
yum install postgresql-contrib postgresql-server
```

Os comandos e procedimentos deste guia dependem da versão do PostgreSQL, MySQL e MariaDB utilizada. Além disso, as localizações de arquivos e nomes de pacotes podem variar entre as distribuições Linux, tais como CentOS, Ubuntu e Debian. Por isso, recomenda-se sempre consultar a documentação oficial da versão de banco de dados e distribuição, pois pequenos ajustes podem ser necessários para adaptar os exemplos.

As configurações do PostgreSQL ficam em dois arquivos principais:

- **postgresql.conf** : É o arquivo de configuração principal, contendo parâmetros que afetam o comportamento geral do servidor
- **pg_hba.conf** : Onde estão definidas as regras de autenticação para conexões de clientes ao servidor

Os arquivos de configuração e os dados onde ficam os registros, por padrão em servidores Linux, estão localizados em "**/var/lib/pgsql/data**" (este caminho está definido no arquivo de configuração postgresql.conf). O cliente nativo é o aplicativo **psql**, o usuário padrão é o postgres, que tem o papel de usuário similar ao "**root**" (SUPERUSER). A porta de comunicação padrão é a **TCP 5432**.

*Os comandos precedidos de **"#"** devem ser executados no shell do sistema operacional. Os outros comandos são executados dentro da console do **psql***

Criar banco de dados

Cria o banco "empresa" com as configurações padrão

```
CREATE DATABASE empresa;
```

Cria o banco "empresa", sendo o dono o usuário "aluno", com a codificação "UTF8" e o locale "en_US" (idioma inglês americano)

```
CREATE DATABASE "empresa"
  WITH OWNER "aluno"
  ENCODING 'UTF8'
  LC_COLLATE = 'en_US.UTF-8'
  LC_CTYPE = 'en_US.UTF-8';
```

Clonar o banco empresa para "empresaX" e o usuário "postgres" será o proprietário do novo banco criado.

```
CREATE DATABASE empresaX WITH TEMPLATE empresa OWNER postgres;
```

O banco empresa deve estar sem atividade ou em somente leitura no momento da cópia para manter a integridade da cópia.

Para encerrar as atividades no banco “empresa”:

```
SELECT pg_terminate_backend(pg_stat_activity.pid)
FROM pg_stat_activity
WHERE pg_stat_activity.datname = 'empresa'
AND pid != pg_backend_pid();
```

Remover banco de dados

Remove o banco de dados "empresaX"

```
drop database empresaX
```

Forçar a remoção do banco “empresa”, caso tenha atividades em andamento

```
drop database empresaX FORCE
```

Criar tabela

Cria a tabela “fornecedores”

```
CREATE TABLE fornecedores (
        cod_forn        integer,
        nome  varchar(80)
);
```

Cria a tabela “adesivoB” a partir do conteúdo da tabela adesivo

```
INSERT INTO adesivoB (conteudo, dia, fim)
```

```
SELECT conteudo, dia, fim
FROM adesivo;
```

Remover tabela

Remove as tabelas “adesivoB” e “adesivoC”

```
DROP TABLE adesivoB, adesivoC;
```

Limpa o conteúdo da tabela “adesivoB”

```
TRUNCATE adesivoB;
```

Limpa o conteúdo da tabela “adesivoB”, zerando os contadores de sequência

```
TRUNCATE adesivoB RESTART IDENTITY;
```

Criar usuários

Criar usuário (role)

```
CREATE USER novo_usuario WITH PASSWORD 'senha_segura';
CREATE ROLE novo_usuario LOGIN PASSWORD ‘senha_segura’;
```

Cria os usuários “aluno” e “aluno5”

```
CREATE USER aluno WITH PASSWORD ‘@T3tesT32o24@’;
CREATE ROLE aluno5 LOGIN PASSWORD '@P4ssW0rd$';
```

Cria o “aluno3” com privilégios de superusuário e acesso remoto

```
CREATE ROLE aluno3 SUPERUSER LOGIN PASSWORD '@P4ssW0rd$';
```

O CREATE USER é equivalente ao CREATE ROLE exceto que o CREATE USER inclui o LOGIN (permissão de acesso remoto) por default e o CREATE ROLE não.

Operações com usuários

Para trocar a senha do usuário “aluno3”

ALTER ROLE aluno3 PASSWORD '@P4ssW0rd$45';

Para remover os privilégios de superusuário do “aluno5”

ALTER ROLE aluno5 NOSUPERUSER;

Para retirar a permissão de conexão remota do “aluno3”

ALTER ROLE aluno3 NOLOGIN;

Para alterar o nome do usuário “aluno2” para “aluno4” (o password será resetado)

ALTER ROLE aluno2 RENAME TO aluno4;

Para configurar o tempo de validade (1º/05/2026) do usuário “aluno1”

ALTER ROLE aluno1 valid until '2026-05-01';

Remover um usuário, no exemplo o usuário “aluno3”

DROP ROLE aluno3;

Caso o usuário seja dono de algum objeto não será possível removê-lo antes de trocar o dono ou remover os objetos

Troca a propriedade dos objetos do “aluno3” para “postgres”

reassign owned by aluno3 to postgres;

Remove todos os objetos de propriedade de “aluno3”

```
drop owned by aluno3;
```

Listar o usuário atual

```
SELECT current_role;
```

Faz um “sudo”, assumindo a identidade de outro usuário

```
SET ROLE aluno;
```

Para retornar ao usuário original

```
RESET ROLE;
```

Listar os usuários existentes

```
\du

select * from pg_catalog.pg_user;
```

Permissões de usuários

Concede todos os privilégios para o usuário "aluno6" no banco de dados "empresa"

```
GRANT ALL PRIVILEGES ON DATABASE empresa TO aluno6;
```

Conecta no banco "empresa" e concede todos os privilégios nas tabelas do banco para o usuário "aluno6"

```
\c empresa

GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO "aluno6";
```

Para verificar os privilégios usar o comando “\z”

Conecta no banco "empresa" e concede o privilégio de SELECT nas tabelas do banco para o usuário "aluno6"

```
\c empresa

GRANT SELECT ON ALL TABLES IN SCHEMA public TO "aluno6";
```

Concede todos os privilégios na tabela

```
GRANT ALL PRIVILEGES ON adesivo TO "aluno6";
```

Altera os privilégios padrão para ALL no esquema public para o usuário “aluno”

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT ALL ON TABLES TO aluno;
```

Remove o privilégio padrão de INSERT no esquema public para o usuário “aluno”

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public REVOKE INSERT ON TABLES FROM aluno;
```

Para listar os privilégios do usuário “aluno”

```
SELECT * FROM information_schema.role_table_grants where grantee = 'aluno';
```

A mudança dos privilégios padrão ALTER DEFAULT PRIVILEGES só afetam as tabelas futuras!

Trocar senha

Para trocar a senha do usuário “aluno3”

ALTER ROLE aluno3 PASSWORD '@P4ssW0rd$45';

Este comando altera a senha do usuário “aluno6” para “@P4ssW0rd$45”. Se a sessão estiver ativa, deve encerrar a conexão para forçar a nova senha

ALTER USER aluno6 WITH PASSWORD '@P4ssW0rd$45';

Para alterar a senha do usuário “postgres” (superusuário padrão)

ALTER USER postgres WITH PASSWORD '@P4ssW0rd$45';

Em caso de ser perdida a senha do usuário “postgres”

Alterar a forma de autenticação do localhost (127.0.0.1) para trust

Fazer um reload no serviço do Postgresql

 # systemctl reload postgresql.service

Acessar a console psql com o usuário do Linux postgres

 # su – postgres

 # psql

Realizar a troca da senha do usuário “postgres”

ALTER USER postgres PASSWORD '@P4ssW0rd$45';

ALTER ROLE e ALTER USER são equivalentes

Tamanho do banco

Para listar todos os bancos de dados com o tamanho ocupado em disco

\l+

Para listar o banco “empresa” com o tamanho ocupado em disco

```
\l+ empresa
```

Para listar todos os bancos de dados com o tamanho ocupado em disco usando SQL

```
SELECT pg_database.datname AS "Bancos", pg_size_pretty(pg_database_size(pg_database.datname)) AS “Tamanho” FROM pg_database;
```

Para listar o tamanho em disco ocupado pelo banco de dados “empresa” usando SQL

```
SELECT pg_database.datname AS "Bancos", pg_size_pretty(pg_database_size(pg_database.datname)) AS "Tamanho“ FROM pg_database where pg_database.datname = 'empresa';
```

Tamanho de tabela

Para listar todas as tabelas de um banco de dados com o tamanho ocupado em disco

```
\d+
```

Para listar o tamanho em disco da tabela “adesivo4” do banco de dados conectado

```
SELECT table_name AS "Tabela",pg_size_pretty(pg_total_relation_size(table_name::regclass)) AS "Tamanho"FROM information_schema.tables WHERE table_schema = 'public' and table_name='adesivo4‘ ORDER BY pg_total_relation_size(table_name::regclass) DESC;
```

Modo somente leitura (manutenção)

Para bloquear todas as transações no banco (somente leitura)

```
ALTER SYSTEM SET default_transaction_read_only = on;

SELECT pg_reload_conf();
```

Para reativar as transações no banco (leitura/escrita)

```
ALTER SYSTEM SET default_transaction_read_only = off;

SELECT pg_reload_conf();
```

Rotinas de checagem

Este comando limpa e atualiza as estatísticas de uma tabela ou banco de dados, o que poder ser útil para manter o desempenho

```
VACUUM ANALYZE;
```

O comando "vacuum" realiza a desfragmentação de tabelas que podem reduzir bastante o espaço em disco e melhorar o tempo de buscas

```
VACUUM FULL;
```

Executa um "vacuum" na tabela "adesivo"

```
VACUUM adesivo;
```

Estes procedimentos podem causar indisponibilidade de acesso ao banco durante a execução (poderá levar horas, depende da complexidade do banco)

Modo Monousuário

Parâmetros para ativar o modo monousuário (--single).

 # postgres --single -O -P -D $PGDATA $DB_NAME

-D local dos arquivos do banco

-O permite alterações nas tabelas do sistema

-P - ignora índices do sistema e atualiza os índices se as tabelas forem modificadas, importante para índices corrompidos

Exemplo de uso do modo monousuário para o banco "video1". Após acessar em modo monousuário, foi executado o comando para reindexar os índices e para sair do ambiente a combinação de teclas CTRL+D.

 # postgres --single -P -D /var/lib/pgsql/data/ video1

REINDEX SYSTEM video1;

CTRL+D para sair

O modo monousuário é indicado para casos onde há corrompimento de índices ou tabelas.

Backup

Para realizar um backup (dump) do banco de dados "empresa"

pg_dump -U postgres -W -Fp -v -f backup_empresa.dump empresa

-W – solicita a senha do usuário

-Fp – formato em texto plano (padrão)

-v – modo verbose

-f – arquivo com o dump do banco

Script com rotinas completas de backup para Postgresql e MySQL disponível em https://github.com/emmonks/bkp_bancos

Restore

Para realizar a restauração do backup (arquvo de dump) do banco de dados “empresa” para um novo banco de nome “empresa2”.

Como usuário “postgres”, criar o banco de dados “empresa2”

 # createdb empresa2

Executar o com comando “pg_restore”

 # pg_restore -d empresa2 backup_empresa.dump

Script com rotinas completas de backup para Postgresql e MySQL disponível em https://github.com/emmonks/bkp_bancos

Logs

Para ativar os logs, ajustar as configurações no arquivo “postgresql.conf”

log_destination = 'stderr'

logging_collector = on

log_directory = '/var/log/postgresql'

log_filename = 'postgresql-%Y-%m-%d.log'

O serviço do PostgreSQL deve ser reiniciado para aplicar as novas configurações **“systemctl restart postgresql.service”**

Na console “psql” para verificar o formato e se os logs estão ativos

```
SHOW log_destination;

SHOW logging_collector;
```

Para ativar os logs das queries em formato CSV, ajustar as configurações no “postgresql.conf”

```
log_destination = 'csvlog'

log_directory = '/var/log/pg_log/'

log_rotation_size = 10MB

log_truncate_on_rotation = on

log_filename = 'postgresql-%Y-%m-%d.log'

log_statement = 'mod'
```

O consumo de espaço em disco poderá ser alto com um volume grande de transações

Tuning

O tuning do PostgreSQL envolve ajustar parâmetros de configuração para otimizar o desempenho no arquivo “**postgresql.conf**”. Os ajustes são relacionados aos recursos de CPU, memória e disco do servidor.

Exemplos de parâmetros para tuning:

```
max_connections = 250
```

```
shared_buffers = 512

work_mem = 100MB

maintenance_work_mem = 512MB

wal_buffers = 16MB

default_statistics_target = 100

max_wal_size = 1GB

min_wal_size = 80MB
```

Na console usando o comando "**psql**" é possível realizar alterações de parâmetros em tempo de execução. Entretanto, ao reiniciar o banco de dados estas configurações serão perdidas. Para manter os parâmetros deve ser usado o arquivo "**postgresql.conf**".

Este comando define o tamanho de memória de trabalho para a sessão atual

```
SET work_mem = '64MB';
```

Para listar todos os parâmetros

```
SHOW all;
```

Para listar o parâmetro work_mem

```
SHOW work_mem;
```

Os valores dos parâmetros podem ser ajustados a partir do monitoramento do servidor. Outra alternativa é o uso de simuladores para estimar os valores tal como o Pgtune https://pgtune.leopard.in.ua/

Monitoramento

Mostra as atividades em todos os bancos de dados e processos do sistema

```
SELECT * FROM pg_stat_activity;
```

Mostra somente as atividades do banco de dados de nome “empresa”

```
SELECT * FROM pg_stat_activity where datname=‘empresa';
```

Mostra estatísticas de todos os banco de dados

```
SELECT * FROM pg_stat_database;
```

Mostra estatísticas somente do banco de dados “empresa”

```
SELECT * FROM pg_stat_database where datname = ‘empresa';
```

Lista as conexões de usuários no banco de nome “empresa”

```
SELECT pid, usename, client_addr FROM pg_stat_activity WHERE datname =‘empresa';
```

Encerra todos as conexões ativas no banco de nome “empresa”

```
SELECT pg_terminate_backend (pid) FROM pg_stat_activity WHERE datname = ‘empresa';
```

Mostra as queries em execução ordenado pelo tempo de início

```
SELECT pid, age(clock_timestamp(), query_start), usename, query FROM pg_stat_activity WHERE query != '<IDLE>' AND query NOT ILIKE '%pg_stat_activity%‘ ORDER BY query_start desc;
```

Encerra o processo com o PID 1817

Lista os processos

```
SELECT * FROM pg_stat_activity;
```

Encerra de forma elegante

```
SELECT pg_cancel_backend(1817);
```

Força o encerramento

```
SELECT pg_terminate_backend(1817);
```

Números de Sequência

No PostgreSQL, os campos incrementais são geralmente implementados usando sequências (*sequences*). Quando é criada uma coluna com o tipo **SERIAL**, **BIGSERIAL** ou **GENERATED ALWAYS AS IDENTITY**, o PostgreSQL cria automaticamente uma sequência associada a essa coluna.

Exemplo de criação de um campo de sequência do tipo "SERIAL", será criado automaticamente uma sequência de nome "pedidos_id_seq"

```
CREATE TABLE pedidos (
        id SERIAL PRIMARY KEY,
        descricao VARCHAR(255)
);
```

Insere dois registros na tabela "pedidos" que receberão as identificações "1" e "2" por serem as primeiras duas inserções na tabela.

```
INSERT INTO pedidos (descricao) VALUES ('Pedido Alpha'), ('Pedido Beta');
```

O campo "id" será incrementado a cada inserção na tabela "pedidos". Por exemplo, após algumas inserções e também remoções, o maior "id" atual é "500". Para mostrar o "id" atual, no schema "public" (padrão).

```
SELECT pg_get_serial_sequence('pedidos', 'id');
```

Ajusta o número de sequência para um valor específico, neste caso para "501"

```
SELECT setval('public.pedidos_id_seq', 501);
```

Ajusta o número de sequência para o maior valor de “id” existente + 1

```
SELECT setval('public.pedidos_id_seq', (SELECT MAX(id) FROM pedidos) + 1);
```

Índices

No contexto de um banco de dados, um índice é uma estrutura de dados especial que armazena uma cópia dos dados de uma ou mais colunas de uma tabela em uma ordem específica. Essa ordem permite que o SGBD localize linhas de forma muito mais rápida do que teria que fazer ao escanear a tabela inteira, linha por linha, que é conhecido como "**Table Scan**" ou "**Sequential Scan**".

A principal função dos índices é acelerar as operações de leitura (**SELECT**). Ao otimizar a forma como o banco de dados encontra os dados, eles reduzem a quantidade de I/O (Input/Output) de disco e o uso de CPU, tornando as consultas mais rápidas e o servidor mais responsivo. No entanto, índices não são uma solução sem custos. Eles afetam o desempenho de outras maneiras:

- Espaço em Disco: Índices consomem espaço em disco, pois são cópias organizadas de parte dos dados.
- Performance de Escrita: Toda vez que dados são inseridos (INSERT), atualizados (UPDATE) ou deletados (DELETE) em uma tabela indexada, o banco de dados precisa não apenas modificar os dados da tabela, mas também atualizar o(s) índice(s) associado(s). Isso adiciona uma sobrecarga e pode tornar as operações de escrita mais lentas.

O desafio para a infraestrutura é balancear esses fatores: ter índices onde eles são mais benéficos para leituras frequentes, e evitar índices desnecessários que apenas consomem espaço e degradam as escritas. A ausência de índices em colunas frequentemente usadas em cláusulas **WHERE** (condições de filtro), **JOINs** (conexões entre tabelas) ou **ORDER BY** (ordenação) é uma das causas mais comuns de problemas de desempenho em bancos de dados. Quando isso ocorre, o banco de dados é forçado a realizar um "**Table Scan**", algo que não é desejável.

No caso de um “**Table Scan**” ocorrem as seguintes ações no banco de dados:

- O SGBD lê todas as linhas da tabela, do início ao fim

- Para cada linha, o banco verifica se atende à condição da consulta
- Se a tabela tem milhões de linhas e apenas algumas dezenas satisfazem a condição, o SGBD desperdiça muitos recursos lendo dados irrelevantes
- Isso sobrecarrega o subsistema de I/O de disco, a CPU do servidor e pode fazer com que outras consultas também fiquem lentas, pois os recursos estão sendo monopolizados

Para o profissional de infraestrutura de TI, uma query que causa um “**Table Scan”** em uma tabela com grande número de registros aparece nos relatórios de monitoramento como uma "query lenta" (**Slow Query**) ou um processo com alto consumo de I/O, indicando que o servidor está trabalhando excessivamente para processar essa consulta.

Alguns cenários práticos onde a falta de um índice causa sérios problemas de desempenho no PostgreSQL.

Cenário 1 - Busca por Nome de Usuário

Estrutura de uma tabela **“usuarios”** com milhões de registros:

```
CREATE TABLE usuarios (
  id INT PRIMARY KEY AUTO_INCREMENT,
nome VARCHAR(255) NOT NULL,
 email VARCHAR(255) UNIQUE NOT NULL,
data_cadastro DATE,
ativo BOOLEAN
);
```

Consulta a tabela “usuarios” em busca do valor “João Silva” na coluna “nome”

```
SELECT * FROM usuarios WHERE nome = 'João Silva';
```

Problema sem Índice: Se não houver um índice na coluna “**nome”**, o banco de dados terá que escanear a tabela “**usuarios”** inteira toda vez que alguém procurar

por um nome. Para milhões de usuários, isso é extremamente lento e consome muitos recursos.

Adiciona um índice de nome “idx_nome” na coluna “nome” da tabela “usuarios”

```
CREATE INDEX idx_nome ON usuarios (nome);
```

Com esse índice, a busca por nome se torna uma operação de índice eficiente, lendo apenas as poucas linhas relevantes, reduzindo drasticamente o tempo de execução e o uso de recursos.

Cenário 2 - Filtragem por Período de Tempo em Logs

Estrutura de uma tabela **“log_eventos”** que armazena bilhões de entradas de log:

```
CREATE TABLE log_eventos (
 id BIGINT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
  timestamp TIMESTAMP NOT NULL,
 tipo_evento VARCHAR(50),
mensagem TEXT
);
```

Consulta a tabela “log_eventos” para buscar logs registrados no dia “01/01/2024”

```
SELECT * FROM log_eventos WHERE timestamp BETWEEN '2024-01-01 00:00:00' AND '2024-01-01 23:59:59';
```

Problema sem Índice: Sem um índice na coluna “**timestamp”**, o SGBD terá que ler a tabela “**log_eventos”** inteira, que possui bilhões de registros, para encontrar os eventos do dia especificado. Essa operação pode levar minutos ou até horas e saturar o I/O de disco.

Adiciona um índice de nome “idx_timestamp” na coluna “timestamp” da tabela “log_eventos”

```
CREATE INDEX idx_timestamp ON log_eventos (timestamp);
```

Um índice em **“timestamp”** permitirá que o banco de dados faça buscas diretamente para a parte dos dados relevantes para o período, otimizando drasticamente a consulta.

Cenário 3 - Junção de Tabelas (JOIN)

Estrutura de duas tabelas, “**pedidos”** e “**clientes”**, e é necessário listar todos os pedidos de um cliente específico

Tabela “pedidos” (com milhões de registros)

```
CREATE TABLE pedidos (
 id INT PRIMARY KEY AUTO_INCREMENT,
 cliente_id INT NOT NULL, -- Chave estrangeira para clientes
 data_pedido DATE,
  valor DECIMAL(10,2)
);
```

Tabela “clientes” (com milhares de registros)

```
CREATE TABLE clientes (
 id INT PRIMARY KEY AUTO_INCREMENT,
nome VARCHAR(255)
);
```

A consulta para obter “pedidos” de um cliente específico

```
SELECT p.id, p.data_pedido, p.valor, c.nome
```

```
FROM pedidos p

JOIN clientes c ON p.cliente_id = c.id

WHERE c.nome = 'Empresa XYZ';
```

Problema sem Índice: Embora “**id”** seja uma chave primária em “**clientes”,** que é indexada automaticamente, se não houver um índice na coluna “**cliente_id”** da tabela “**pedidos**”, que é a chave estrangeira usada no **JOIN**, o SGBD pode ter que escanear a tabela “**pedidos**” inteira ou usar um algoritmo de **join** menos eficiente para encontrar os pedidos relacionados ao cliente. Em tabelas muito grandes, isso causa lentidão massiva.

Adiciona um índice de nome “idx_cliente_id” na chave estrangeira “cliente_id” da tabela “pedidos”

```
CREATE INDEX idx_cliente_id ON pedidos (cliente_id);
```

Indexar a coluna “**cliente_id”** na tabela “**pedidos”** otimiza a operação de **JOIN**, permitindo que o banco de dados encontre rapidamente os pedidos de um “**cliente_id”** específico, reduzindo significativamente o tempo de execução da query e o impacto nos recursos do servidor.

*Os índices são ferramentas poderosas para melhorar o desempenho de consultas de leitura. Para o profissional de infraestrutura de TI, saber identificar quando uma query está sofrendo por falta de um índice (muitas vezes detectado por **EXPLAIN** ou por ferramentas de monitoramento que apontam "**Table Scans**") é tão importante quanto saber como criá-los. Implementar índices estrategicamente pode transformar um servidor de banco de dados lento e sobrecarregado em um ambiente responsivo e eficiente, liberando recursos preciosos para outras operações.*

Função EXPLAIN

A ferramenta **EXPLAIN** mostra o plano de execução que o banco de dados pretende seguir para executar uma query. Isso não é sobre otimizar a query em si (tarefa mais do desenvolvedor), mas sim sobre entender por que uma query está consumindo muitos recursos do servidor (CPU, memória, I/O de disco) e identificar se ela está ocorrendo uma carga excessiva e talvez desnecessária.

Ao analisar o **EXPLAIN**, devem ser percebidos indicadores de ineficiência, tais como:

- **Scans completos de tabela:** O banco de dados está lendo a tabela inteira em vez de usar um índice?
- **Joins ineficientes:** Como as tabelas estão sendo unidas?
- **Uso de arquivos temporários em disco:** O banco está usando o disco para processar a query, o que é muito mais lento que a memória?

No PostgreSQL, recomenda-se usar "**EXPLAIN ANALYZE**" para ver o plano de execução real e os tempos gastos.

Exemplo 1: Query Eficiente (usando índice)

Uma tabela "**produtos**" com um índice na coluna "**codigo_barras**".

Mostra os campos "nome_produto" e "preco" da tabela "produtos" onde o "codigo_barras" seja igual a "1234567890" e apresenta o resultado da função EXPLAIN ANALYZE

```
EXPLAIN ANALYZE SELECT nome_produto, preco FROM produtos WHERE codigo_barras = '1234567890';
```

Saída Esperada (Simplificada):

```
                                QUERY PLAN
---------------------------------------------------------------------------------------
```

Index Scan using idx_codigo_barras on produtos (cost=0.42..8.44 rows=1 width=28) (actual time=0.045..0.046 rows=1 loops=1)

Index Cond: (codigo_barras = '1234567890'::text)

Planning Time: 0.081 ms

Execution Time: 0.067 ms

Análise (Simplificada):

***Index Scan**: Indica que o PostgreSQL utilizou um índice (`idx_codigo_barras`)*

***actual time** (Tempo real): Mostra o tempo real gasto para cada etapa e para a query completa (0.067 ms). Tempos baixos são um bom sinal.*

***rows=1**: Apenas 1 linha foi retornada, confirmando a eficiência.*

Essa query é muito rápida e eficiente, com baixo impacto nos recursos.

Exemplo 2: Query Ineficiente (Table Scan)

Uma tabela “**log_eventos**” gigante sem um índice na coluna “**tipo_evento**”.

Seleciona todas as colunas da tabela “log_eventos” que sejam iguais a “erro” no campo “tipo_evento”

*EXPLAIN ANALYZE SELECT * FROM log_eventos WHERE tipo_evento = 'erro';*

Saída Esperada (Simplificada):

QUERY PLAN

Seq Scan on log_eventos (cost=0.00..150000.00 rows=500000 width=100) (actual time=0.010..5000.000 rows=500000 loops=1)

Filter: ((tipo_evento)::text = 'erro'::text)

Rows Removed by Filter: 9500000

Planning Time: 0.090 ms

Execution Time: 5000.123 ms

Análise (Simplificada):

***Seq Scan**: Significa "**Sequential Scan**", ou seja, o PostgreSQL está lendo a tabela inteira sequencialmente. O efeito disto é consumo excessivo de recursos do servidor e aumento no tempo de resposta para o usuário.*

***rows=500000**: Estima que 500.000 linhas serão retornadas, mas a parte mais crítica é que o banco precisou escanear um número muito maior (`**Rows Removed by Filter: 9500000**`) para encontrar essas 500.000.*

***actual time=5000.123 ms**: A query levou 5 segundos (ou mais, dependendo do tamanho real da tabela) para ser executada. Isso é um tempo considerável e pode impactar a performance do servidor.*

*Uma "**Seq Scan**" em uma tabela grande, como "**log_eventos**", gera grande volume de requisições de I/O de disco e uso intenso de CPU, resultando em lentidão para outras operações no banco de dados e sobrecarga para o servidor. Para o profissional de infraestrutura de TI, essa é a indicação clara de que há uma query ineficiente consumindo recursos excessivos.*

Replicação

No PostgreSQL, a replicação nativa é baseada no conceito de **WAL** (**Write-Ahead Log)**. Todas as alterações no banco de dados são registradas no WAL antes de serem efetivadas nos arquivos de dados. Os servidores de réplica (chamados de **standby** ou **replica**) executam um fluxo contínuo dos registros WAL do primário e os aplicam localmente.

A forma mais comum e moderna de replicação é o "**Streaming Replication"**, que pode ser configurado de forma síncrona ou assíncrona. Geralmente, usa-se a assíncrona para evitar latência no primário.

Exemplo de Configuração Básica com recursos de "Streaming Replication" Assíncrona

No Servidor Primário (Principal):

Editar o arquivo **postgresql.conf**

*listen_addresses = **'*'** # Permite conexões de qualquer IP (ou IPs específicos)*

*wal_level = **replica** # Nível de log necessário para replicação*

*max_wal_senders = **10** # Número máximo de conexões de réplicas*

*archive_mode = **on** # Habilita o modo de arquivamento (importante para recuperação)*

archive_command = 'cp %p /var/lib/pgsql/data/archive/%f' # Comando para arquivar WALs

*Ajustar o caminho do diretório de arquivo, caso não exista o diretório "**/var/lib/pgsql/data/archive**", deverá ser criado com o dono e o grupo para o usuário **postgres** . Comando para mudar o dono e o grupo para "postgres": **chown postgres:postgres***

Editar o arquivo "**pg_hba.conf"** para permitir que a réplica faça a conexão. No exemplo, o IP do banco de dados réplica é "**192.168.254.93**" e o usuário "**replica_user**".

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD

host    replication     replica_user    192.168.254.93/32       md5
```

Cria um usuário para replicação no PostgreSQL no servidor primário

```
CREATE USER replica_user WITH REPLICATION ENCRYPTED PASSWORD 'senha@123';
```

Para aplicar as alterações deve ser realizado um reinício no serviço do PostgreSQL com o comando **systemctl restart postgresql.service**

O serviço do PostgreSQL não exclui automaticamente os arquivos WAL do diretório de arquivamento (archive), só realiza as cópias dos arquivos. Estes arquivos podem consumir um volume considerável de espaço em disco rapidamente. Desta forma, é necessária uma rotina de limpeza dos arquivos mais antigos, que já foram consolidados no processo de backup. Por exemplo,se é realizado um backup diário, na rotina de pós backup deverão ser removidos os arquivos WAL com mais de um dia. Um script simples para remoção dos arquivos WAL mais antigos poderia ser:

```
#!/bin/bash

# Diretório onde os arquivos WALs são arquivados
```

```
WAL_ARCHIVE_DIR="/var/lib/pgsql/data/archive"

# Número de dias para manter os arquivos WAL

DAYS_TO_KEEP=1

find "$WAL_ARCHIVE_DIR" -type f -mtime +"$DAYS_TO_KEEP" -delete

echo "$(date): Limpeza dos arquivos WAL completa. Arquivos mais antigos que $DAYS_TO_KEEP dia(s) foram removidos de $WAL_ARCHIVE_DIR." >> /var/log/wal_limpeza.log
```

No Servidor de Réplica (Standby):

- **Parar o PostgreSQL na réplica:** systemctl stop postgresql.service
- **Para sincronizar com o servidor primário, apagar dados antigos (se houver):**

```
rm -rf /var/lib/pgsql/data/*
```

Este procedimento é necessário porque o PostgreSQL implementa a replicação a nível de bloco. Para garantir a integridade e consistência dos dados, a réplica deve ser um espelho exato do primário a partir de um ponto no tempo.

- **Copiar os dados base do primário:** Usar o comando "**pg_basebackup**" para criar uma cópia inicial dos dados. No exemplo, o IP do servidor primário é "**192.168.254.34**", o usuário remoto "**replica_user**" e o caminho é o "**PG_DATA"** que aponta para "**/var/lib/pgsql/data**", os parâmetros "**-P**" mostra o progresso, "**-v**" ativa o modo verbose e "**-R**" Cria automaticamente o arquivo **standby.signal** e adiciona configurações de conexão ao **postgresql.conf** da réplica.

```
pg_basebackup -h 192.168.254.34 -U replica_user -D /var/lib/pgsql/data -P -v -R
```

- **Verificar e ajustar o postgresql.conf na réplica:** O “pg_basebackup -R” já deve ter configurado o necessário. Certificar de que a diretiva “**hot_standby = on**” esteja definida e é uma boa prática colocar uma identificação no parâmetro “**primary_conninfo**” para saber qual é a réplica, por exemplo “**primary_conninfo=’replica1’**”. Importante verificar se os arquivos e pastas em “/var/lib/pgsql/data” estão com o dono “**postgres**” e o grupo “**postgres**”.
- **Iniciar o PostgreSQL na réplica:** systemctl start postgresql.service

Monitoramento da Replicação

No servidor primário, para ver as conexões de replicação ativas

```
SELECT usename, client_addr, state, sync_state FROM pg_stat_replication;
```

Exemplo de saída do comando:

```
 username    | client_addr   |  state   | sync_state
-------------+---------------+----------+------------
 replica_user | 192.168.254.93 | streaming | async
```

No servidor de réplica, para ver o atraso (lag) da replicação

```
SELECT pg_last_wal_receive_lsn(), pg_last_wal_replay_lsn(),
pg_is_in_recovery();
```

Exemplo de saída do comando:

```
 pg_last_wal_receive_lsn | pg_last_wal_replay_lsn | pg_is_in_recovery
-------------------------+------------------------+-------------------
 0/53000148              | 0/53000148             | t(1 row)
```

*O valor da diretiva “**pg_last_wal_replay_lsn**” deve ser o mais próximo possível do valor da diretiva “**pg_last_wal_receive_lsn**”. A diretiva “**pg_is_in_recovery()**” deve retornar “**t (true)**” na réplica.*

psql - Atalhos

\s - histórico de comandos

\q - para sair

\t - ativar o temporizador

\d - mostrar as tabelas, views

\d+ - mostrar as tabelas, views com o tamanho em disco

\dS+ - mostrar as tabelas, views, incluindo do sistema, com o tamanho em disco

\watch [SEG] repete a execução de uma query a cada X segundos

\c - conecta em um banco

\l+ - lista banco de dados, com o tamanho em disco

\d+ tabela - mostra informações sobre uma tabela

\du - lista usuário e permissões

\i nome_do_arquivo - executa comandos contidos no arquivo

Ferramentas complementares

pg_top – ferramenta similar ao top para monitoramento de processos no PostgreSQL

Instalação: # yum install pg_top

Uso básico

Como usuário postgres, executar

 # pg_top

Usar a letra "h" para acessar a ajuda da ferramenta

Exemplo:

Letra "Q" e o número do processo mostra a query sendo executada

initdb: Inicializa um novo cluster de banco de dados PostgreSQL.

initdb - inicializa o ambiente de dados, similar a uma formatação.

Criação do diretório de dados em "/var/banco_dados"

 # initdb -D /var/banco_dados

-D: Especifica o diretório para os arquivos de dados.

createdb - Cria um novo banco de dados no prompt do sistema operacional.

Cria o banco de dados "empresa"

 # createdb empresa

dropdb - Remove um banco de dados existente no prompt do sistema operacional.

Remove o banco de dados "empresa"

 # dropdb empresa

psql - Interface de linha de comando interativa para o PostgreSQL.

Conecta no serviço do PostgreSQL no endereço de host "192.168.10.5", na porta TCP "7000", com o usuário "aluno" e no banco de dados "empresa"

 # psql -h 192.168.10.5 -p 7000 -U aluno -d empresa

Conecta no serviço do PostgreSQL no endereço de host “192.168.10.5”, na porta TCP “7000”, com o usuário “aluno” e solicita a senha no banco de dados “empresa”

```
# psql -h 192.168.10.5 -p 7000 -U aluno -W -d empresa
```

Lista os bancos de dados do servidor local

```
# psql -l
```

Lista os bancos de dados do servidor remoto de IP “192.168.10.5” e porta padrão TCP “5432”, acessando como usuário “aluno” e sem solicitar senha

```
# psql -l -h 192.168.10.5 -w -U aluno
```

pg_ctl - Controla o servidor de banco de dados PostgreSQL (iniciar, parar, reiniciar).

Realiza o gerenciamento do serviço onde o diretório de dados está em “/var/banco_dados”

```
# pg_ctl -D /var/banco_dados start

# pg_ctl -D /var/banco_dados stop

# pg_ctl -D /var/banco_dados restart

# pg_ctl -D /var/banco_dados reload
```

pgbench - Ferramenta nativa do PostgreSQL que provê funcionalidades de benchmark com possibilidade de configurações de consultas, concorrência e iterações para simular uma carga de trabalho no servidor.

Cria o banco “testador” para os testes

```
# createdb testador
```

Cria a tabela para ser usada no teste no banco "testador", usando o arquivo “cria_tabela.sql” com os comandos

```
# psql -U postgres -d testador -f cria_tabela.sql
```

Exemplo de conteúdo do arquivo “cria_tabela.sql”

```
DROP TABLE IF EXISTS a;

CREATE TABLE a (b INT);

INSERT INTO a VALUES (23);
```

Executa a bateria de testes com as consultas no arquivo "teste_script.sql" no banco "testador", com os parâmetros:

- **-c 80:** Especifica o número de clientes (conexões) simultâneos que executarão as transações
- **-j 8:** Define o número de threads que o pgbench usará para coordenar os clientes
- **-t 20:** Especifica o número de transações (iterações) que cada cliente executará

```
#pgbench -U postgres -c 80 -j 8 -t 20 --file=teste_script.sql testador
```

Exemplo de conteúdo do arquivo "teste_script.sql"

```
SELECT * FROM a;

INSERT INTO a VALUES (230);
```

É possível utilizar as tabelas padrão da ferramenta pgbench para realizar os testes de forma genérica. Neste caso, está sendo criado um ambiente de testes com 1.000.000 de linhas (parâmetro "-s"). O banco "testador" deverá ser criado previamente para receber as tabelas do pgbench

```
Para inicializar o ambiente com 1.000.000 de linhas

# pgbench -i -s 1 -U postgres testador

Para executar os testes padrão da ferramenta

# pgbench testador
```

3) MySQL (MariaDB)

O banco de dados MySQL é um projeto em código-fonte aberto bastante popular, com a primeira versão do MySQL lançada em 1995. O nome MySQL é uma homenagem à filha de um dos criadores, Monty Widenius chamada My. Em 2008, a Sun Microsystems adquiriu a MySQL AB, empresa por trás do MySQL, solidificando ainda mais sua posição no mercado. Em 2010, o projeto MySQL mudou de proprietário com a aquisição da Sun pela empresa Oracle. Com a aquisição por empresas, o MySQL passou a ser disponibilizado em versões comercial e comunitária. O projeto MariaDB é um fork do projeto MySQL com o objetivo de ser livre para qualquer tipo de uso e compatível com o MySQL. O MariaDB é a alternativa padrão nas distribuições Linux atuais. A porta padrão de comunicação do MySQL é a TCP 3306.

A versão do banco de dados MariaDB utilizada para os exemplos de comandos foi a **10.5.27** sendo executada em sistema operacional **Linux** na distribuição **CentOS Stream release 9**.

Para instalar o servidor e cliente do MySQL no CentOS Stream release 9

```
yum install mysql-server mysql
```

Para instalar o servidor e cliente do MariaDB no CentOS Stream release 9

```
yum install mariadb-server mariadb-server-utils
```

Os comandos e procedimentos deste guia dependem da versão do PostgreSQL, MySQL e MariaDB utilizada. Além disso, as localizações de arquivos e nomes de pacotes podem variar entre as distribuições Linux, tais como CentOS, Ubuntu e Debian. Por isso, recomenda-se sempre consultar a documentação oficial da versão de banco de dados e distribuição, pois pequenos ajustes podem ser necessários para adaptar os exemplos.

As configurações do MySQL ficam em um arquivo principal denominado “**my.cnf**”. Este arquivo é o principal para a configuração do banco de dados, contendo parâmetros que afetam o comportamento geral do servidor. Normalmente, é usada a diretiva “**includedir**” para apontar para mais arquivos de configuração. Os arquivos de configuração ficam em “**/etc**” e o diretório dos arquivos com os dados chamado datadir está localizado em “**/var/lib/mysql**”. Entretanto, os caminhos podem variar de acordo com a distribuição ou com a configuração definida pelo administrador do sistema. Por exemplo, os caminhos do arquivo “**my.cnf**” nas distribuições CentOS e Debian:

CentOS - /etc/my.cnf

Debian - /etc/mysql/my.cnf

O cliente nativo é o aplicativo mysql que funciona como uma console (shell) para interação com o usuário e possibilita o gerenciamento e a operação dos bancos de dados.

*Os comandos precedidos de **“#”** devem ser executados no shell do sistema operacional. Os outros comandos são executados dentro da console do **mysql***

Criar banco de dados

Cria o banco "empresa" com as configurações padrão

CREATE DATABASE empresa;

Cria o banco "empresa", sendo o dono o usuário “aluno”, com a codificação “UTF8” e o locale “en_US” (Inglês americano)

CREATE DATABASE empresa DEFAULT CHARACTER SET = 'utf8' DEFAULT COLLATE 'utf8_general_ci';

Clonar o banco “empresa” para “empresa_bk”

```
# mysqladmin create empresa_bk|mysqldump --routines --triggers empresa | mysql empresa_bk
```

O banco empresa deve estar sem atividade ou em somente leitura no momento da cópia para manter a integridade da cópia.

Remover banco de dados

Remove o banco de dados "empresaX"

```
drop database empresaX;

mysqladmin drop empresaX;
```

Remove o banco “empresaX”, caso exista um banco com este nome

```
drop database IF EXISTS empresaX;
```

Criar tabela

Cria a tabela “fornecedores”

```
CREATE TABLE fornecedores (
        cod_forn        integer,
        nome  varchar(80)
);
```

Cria a tabela “adesivoB” a partir do conteúdo da tabela adesivo

```
CREATE TABLE adesivoB SELECT * FROM adesivo;
```

Remover tabela

Remove as tabelas “adesivoB” e “adesivoC”

```
DROP TABLE adesivoB, adesivoC;
```

Limpa o conteúdo da tabela “adesivoB”

```
TRUNCATE adesivoB;
```

Limpa o conteúdo da tabela “adesivoB”, zerando os contadores de sequência

```
TRUNCATE adesivoB;

ALTER TABLE adesivoB AUTO_INCREMENT = 1
```

Criar usuários

Cria o usuário “aluno”

```
CREATE USER 'aluno' IDENTIFIED BY '@P4ssW0rd$';
```

Cria o usuário “aluno10” com restrições de acesso somente para localhost

```
CREATE USER 'aluno10‘@’127.0.0.1’ IDENTIFIED BY '@P4ssW0rd$';
```

Cria o usuário “aluno11” com permissão de acesso somente da rede 192.168.10.0/24

```
CREATE USER 'aluno11‘@’192.168.10.0/24’ IDENTIFIED BY '@P4ssW0rd$';
```

Cria o usuário “aluno12” com permissão de acesso de qualquer endereço

```
CREATE USER 'aluno11‘@’%’ IDENTIFIED BY '@P4ssW0rd$';
```

Nas versões mais recentes o usuário sem definição de origem de acesso assume "%" por padrão

Operações com usuários

Para trocar a senha do usuário "aluno3"

```
ALTER USER 'aluno13' IDENTIFIED BY '@P4ssW0rd$';
```

Para trocar a senha do usuário "aluno13" para "@P4ssW0rd$" e habilitar as conexões de qualquer origem

```
ALTER USER 'aluno13'@'%' IDENTIFIED BY '@P4ssW0rd$';
```

Remove um usuário "aluno3"

```
DROP USER 'aluno13';
```

Lista o usuário atual

```
SELECT CURRENT_USER();
```

Faz um "sudo", assumindo a identidade de outro usuário

```
system mysql -u aluno13 -p
```

Lista os usuários existentes

```
SELECT user FROM mysql.user;
```

Permissões de usuários

Concede todos os privilégios para o usuário "aluno" no banco de dados "video1" vindo de qualquer origem

```
grant all privileges on video1.* to aluno@'%' ;
```

Concede ao usuário “aluno13” o privilégio de SELECT na tabela “usuarios” no banco “video1” com acesso de localhost

```
grant select on video1.usuarios to 'aluno13'@'localhost';
```

Concede permissões de SELECT, INSERT e UPDATE no banco “video1” para o usuário “aluno13” com acesso de localhost e renova as permissões

```
GRANT SELECT, INSERT, UPDATE ON video1.* TO ‘aluno13'@'localhost';

FLUSH PRIVILEGES;
```

Concede todos os privilégios para o usuário “aluno14” na tabela “usuarios” do banco “video1” com origem da rede “192.168.254.0/24”

```
grant all privileges on video1.usuarios to 'aluno14'@'192.168.254.0/24';
```

Revoga todos os privilégios do usuário aluno no banco "video1" originado de localhost

```
REVOKE all privileges ON video1.* FROM 'aluno'@'localhost';
```

Revoga o privilégio de DELETE do usuário “aluno13” no banco "video1" vindo que qualquer origem

```
REVOKE DELETE ON video1.* FROM 'aluno13'@'%';
```

Mostra os usuários e as origens permitidas

```
SELECT user,host FROM mysql.user;
```

Mostra as permissões do usuário “aluno” com qualquer origem

```
SHOW GRANTS FOR 'aluno'@'%';
```

Trocar senha

Para trocar a senha do usuário “aluno3”

```
ALTER USER 'aluno3' IDENTIFIED BY '@P4ssW0rd$';
```

Para trocar a senha do usuário “aluno13” para “@P4ssW0rd$” e habilitar as conexões de qualquer origem

ALTER USER 'aluno13'@'%' IDENTIFIED BY '@P4ssW0rd$';

Um usuário poderá ter senhas e permissões diferentes de acordo com a origem

Em caso de ser perdida a senha do usuário root

Parar o serviço do MySQL

 # systemctl stop mariadb.service

Executar o serviço em modo seguro (ignorar os grants) e colocar o processo em segundo plano

 # mysqld_safe --skip-grant-tables &

Entrar na console (mysql -u root)

Executar os comandos

use mysql;

FLUSH PRIVILEGES;

ALTER USER 'root'@'localhost' IDENTIFIED BY '@P4ssW0rd$';

Parar o serviço em modo seguro: ***mysqladmin shutdown***

Tamanho do banco

Para listar todos os bancos de dados com o tamanho ocupado em disco

```
SELECT table_schema AS 'Nome do Banco', ROUND(SUM(data_length + index_length) / 1024 / 1024, 2) AS 'Tamanho (MB)' FROM information_schema.tables GROUP BY table_schema;
```

Tamanho de tabela

Para listar o tamanho em disco das tabelas do banco de dados “empresa”

```
SELECT table_name AS 'Tabela', ROUND((data_length + index_length) / 1024 / 1024, 2) AS 'Tamanho (MB)‘  FROM information_schema.tables WHERE table_schema = 'empresa‘ ORDER BY (data_length + index_length) DESC;
```

Modo somente leitura (manutenção)

Bloqueia as tabelas para operações de leitura e escrita, permitindo operações de manutenção (somente leitura)

```
FLUSH TABLES WITH READ LOCK;
```

Para reativar as transações no banco (leitura/escrita)

```
UNLOCK TABLES;
```

Rotinas de checagem

Para verificar a integridade da tabela “funcionarios”

```
check table funcionarios;
```

O comando analyze verifica se as estatísticas para o otimizar de queries do MySQL estão atualizadas na tabela

```
ANALYZE table funcionarios;
```

Executa uma checagem geral e otimização em todos os bancos de dados

```
#  mysqlcheck --auto-repair -o --all-databases
```

Estes procedimentos podem causar indisponibilidade de acesso ao banco durante a execução (poderá levar horas, depende da complexidade do banco)

Modo Monousuário

Uma das formas de ativar algo parecido com um modo monousuário é ignorando as permissões e o acesso por rede

```
# mysqld_safe --skip-grant-tables –skip-networking
```

O modo Monousuário é indicado para casos onde há corrompimento de índices ou tabelas.

Backup

Para realizar um backup (dump) do banco de dados “empresa” no arquivo “backup_empresa_01032025_0915.sql”

```
# mysqldump -u root -p empresa > backup_empresa_01032025_0915.sql
```

Para realizar o backup de todos os bancos de dados como usuário “root” e solicitando senha

```
# mysqldump -u root -p --all-databases > backup_todos_01032025_0915.sql
```

Para realizar o backup da tabela “funcionarios” do banco “empresa”

```
# mysqldump –u root -p empresa funcionarios > backup_tabela_funcionarios.sql
```

Script com rotinas completas de backup para Postgresql e MySQL disponível em https://github.com/emmonks/bkp_bancos

Restore

Para realizar a restauração do backup (dump) do banco de dados “empresa” para um novo servidor com IP “192.168.10.3”

```
# mysql --host=192.168.10.3 --user=root --port=3306 -p empresa < empresa.sql
```

Para restaurar a tabela “funcionarios” do banco “empresa”

```
# mysql --user=root -p empresa < backup_tabela_funcionarios.sql
```

Para restaurar todos os banco de dados

```
# mysql -u root -p < backup_todos_01032025_0915.sql
```

Script com rotinas completas de backup para Postgresql e MySQL disponível em https://github.com/emmonks/bkp_bancos

Logs

Para ativar os logs, ajustar as configurações no “my.cnf (50-server.cnf)”

```
[mysqld]

log-error=/var/log/mysql/error.log

general-log-file=/var/log/mysql/mysql.log

slow-query-log-file=/var/log/mysql/slow.log
```

O serviço do MySQL deve ser reiniciado para aplicar as novas configurações

Na console mysql para verificar o formato e se os logs estão ativos

```
SHOW VARIABLES LIKE 'log_error';

SHOW VARIABLES LIKE 'log%';
```

Para desabilitar os logs

```
SET GLOBAL general_log = 'OFF';

SET GLOBAL slow_query_log = 'OFF';
```

Para reativar os logs

```
SET GLOBAL general_log = 'ON';

SET GLOBAL slow_query_log = 'ON';
```

Para definir o tempo considerado para uma "slow query" 0 a 10 segundos

```
SET GLOBAL long_query_time = 8;
```

Utilitário mysqldumpslow apresenta um resumo dos logs de queries lentas

```
# mysqldumpslow
```

*O diretório dos logs deve possuir como dono o usuário "**mysql**". O consumo de espaço em disco poderá ser alto com um volume grande de transações*

Tuning

O tuning do MySQL envolve ajustar parâmetros de configuração para otimizar o desempenho no arquivo "**my.cnf**". Os ajustes são relacionados aos recursos de CPU, memória e disco do servidor.

Exemplos de parâmetros para tuning:

```
[mysqld]

innodb_buffer_pool_size=1G

query_cache_size=64M

max_connections=200
```

Na console usando o comando mysql é possível realizar alterações de parâmetros em tempo de execução. Ao reiniciar o banco de dados estas configurações serão perdidas. Para manter os parâmetros deve ser usado o arquivo "my.cnf".

Este comando define o número máximo de conexões para 500

```
SET GLOBAL max_connections = 500;
```

Para listar todos os parâmetros

```
SHOW variables;
```

Para listar parâmetros relacionados a conexões

```
show variables like '%connections%';
```

Os valores podem ser estimados com o uso de simuladores tal como o MySQLTuner - https://github.com/major/MySQLTuner-perl

Monitoramento

Mostra as atividades em todos os bancos de dados e processos do sistema

```
SHOW PROCESSLIST;
```

Mostra somente as atividades do banco de dados de nome "empresa"

*SELECT * FROM INFORMATION_SCHEMA.PROCESSLIST where DB='empresa';*

Mostra estatísticas de todos os banco de dados

SHOW STATUS;

Para listar o número de queries executadas

show status where variable_name='Queries';

Lista as conexões de usuários no banco de nome “empresa”

SELECT pid, usename, client_addr FROM pg_stat_activity WHERE datname =‘empresa';

Mostra as queries em execução no banco "empresa" ordenado pelo tempo em execução

SELECT id,time FROM INFORMATION_SCHEMA.PROCESSLIST WHERE DB =‘empresa' order by time desc;

Encerra todas as conexões ativas no banco “empresa”

select concat('KILL ',id,';') from information_schema.processlist where db='empresa' into outfile '/tmp/t_empresa.txt';

source /tmp/t_empresa.txt;

Encerra o processo com o “PID 1812”

Lista as threads ordenado por tempo em execução

*SELECT * FROM INFORMATION_SCHEMA.PROCESSLIST order by time desc;*

Encerra a conexão e as threads

kill connection 1812;

Encerra apenas a thread da query

kill query 1812;

Números de Sequência

No MySQL (e MariaDB), o valor incremental é controlado pela propriedade "**AUTO_INCREMENT**" de uma coluna. Para definir ou ajustar o próximo valor de "**AUTO_INCREMENT**", deve ser usada a instrução "**ALTER TABLE**".

Exemplo de criação de uma sequência "AUTO_INCREMENT", será criado automaticamente uma sequência no campo "**id**"

```
CREATE TABLE produtos (
  id INT PRIMARY KEY AUTO_INCREMENT,
nome_produto VARCHAR(255)
);
```

Insere dois registros na tabela "produtos" que receberão as identificações "1" e "2" por serem as primeiras duas inserções na tabela.

```
INSERT INTO produtos (nome_produto) VALUES ('Produto A'), ('Produto B');
```

O campo "id" será incrementado a cada inserção na tabela "produtos". Após algumas inserções e talvez remoções, o maior "id" é 500. Para mostrar o "id" atual

```
SHOW CREATE TABLE produtos;
```

Ajustar o número de sequência para um valor específico, neste caso para "501"

```
ALTER TABLE produtos AUTO_INCREMENT = 501;
```

Ajustar o número de sequência para o maior valor de "id" existente + 1

```
SELECT MAX(id) FROM produtos;
ALTER TABLE produtos AUTO_INCREMENT = 501;
```

Índices

No contexto de um banco de dados, um índice é uma estrutura de dados especial que armazena uma cópia dos dados de uma ou mais colunas de uma tabela em uma ordem específica. Essa ordem permite que o SGBD localize linhas de forma muito mais rápida do que teria que fazer ao escanear a tabela inteira, linha por linha, que é conhecido como "**Table Scan**" ou "**Sequential Scan**".

A principal função dos índices é acelerar as operações de leitura (**SELECT**). Ao otimizar a forma como o banco de dados encontra os dados, eles reduzem a quantidade de I/O (Input/Output) de disco e o uso de CPU, tornando as consultas mais rápidas e o servidor mais responsivo. No entanto, índices não são uma solução sem custos. Eles afetam o desempenho de outras maneiras:

- Espaço em Disco: Índices consomem espaço em disco, pois são cópias organizadas de parte dos dados.
- Performance de Escrita: Toda vez que dados são inseridos (INSERT), atualizados (UPDATE) ou deletados (DELETE) em uma tabela indexada, o banco de dados precisa não apenas modificar os dados da tabela, mas também atualizar o(s) índice(s) associado(s). Isso adiciona uma sobrecarga e pode tornar as operações de escrita mais lentas.

O desafio para a infraestrutura é balancear esses fatores: ter índices onde eles são mais benéficos para leituras frequentes, e evitar índices desnecessários que apenas consomem espaço e degradam as escritas. A ausência de índices em colunas frequentemente usadas em cláusulas **WHERE** (condições de filtro), **JOINs** (conexões entre tabelas) ou **ORDER BY** (ordenação) é uma das causas mais comuns de problemas de desempenho em bancos de dados. Quando isso ocorre, o banco de dados é forçado a realizar um "**Table Scan**", algo que não é desejável.

No caso de um “**Table Scan**” ocorrem as seguintes ações no banco de dados:

- O SGBD lê todas as linhas da tabela, do início ao fim
- Para cada linha, o banco verifica se atende à condição da consulta

- Se a tabela tem milhões de linhas e apenas algumas dezenas satisfazem a condição, o SGBD desperdiça muitos recursos lendo dados irrelevantes
- Isso sobrecarrega o subsistema de I/O de disco, a CPU do servidor e pode fazer com que outras consultas também fiquem lentas, pois os recursos estão sendo monopolizados

Para o profissional de infraestrutura de TI, uma query que causa um "**Table Scan"** em uma tabela com grande número de registros aparece nos relatórios de monitoramento como uma "query lenta" (**Slow Query**) ou um processo com alto consumo de I/O, indicando que o servidor está trabalhando excessivamente para processar essa consulta.

Alguns cenários práticos onde a falta de um índice causa sérios problemas de desempenho no MySQL.

Cenário 1 - Busca por Nome de Usuário

Estrutura de uma tabela **"usuarios"** com milhões de registros

```
CREATE TABLE usuarios (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    data_cadastro DATE,
    ativo BOOLEAN
);
```

Consulta a tabela "usuarios" em busca do valor "João Silva" na coluna "nome"

```
SELECT * FROM usuarios WHERE nome = 'João Silva';
```

Problema sem Índice: Se não houver um índice na coluna "**nome"**, o banco de dados terá que escanear a tabela "**usuarios"** inteira toda vez que alguém procurar

por um nome. Para milhões de usuários, isso é extremamente lento e consome muitos recursos.

Cria um índice de nome “idx_nome” na coluna “nome” da tabela “usuarios”

```
CREATE INDEX idx_nome ON usuarios (nome);
```

Com esse índice, a busca por nome se torna uma operação de índice eficiente, lendo apenas as poucas linhas relevantes, reduzindo drasticamente o tempo de execução e o uso de recursos.

Cenário 2 - Filtragem por Período de Tempo em Logs

Estrutura de uma tabela **“log_eventos”** que armazena bilhões de entradas de log:

```
CREATE TABLE log_eventos (
    id BIGINT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    timestamp TIMESTAMP NOT NULL,
    tipo_evento VARCHAR(50),
    mensagem TEXT
);
```

Consulta a tabela “log_eventos” para buscar logs registrados no dia “01/01/2024”

```
SELECT * FROM log_eventos WHERE timestamp BETWEEN '2024-01-01 00:00:00' AND '2024-01-01 23:59:59';
```

Problema sem Índice: Sem um índice na coluna “**timestamp”**, o SGBD terá que ler a tabela “**log_eventos”** inteira (bilhões de registros!) para encontrar os eventos do dia especificado. Essa operação pode levar minutos ou até horas e saturar o I/O de disco.

Cria um índice de nome “idx_timestamp” na coluna “timestamp” da tabela “log_eventos”

```
CREATE INDEX idx_timestamp ON log_eventos (timestamp);
```

Um índice em **timestamp** permitirá que o banco de dados faça buscas diretamente para a parte dos dados relevantes para o período, otimizando drasticamente a consulta.

Cenário 3 - Junção de Tabelas (JOIN)

Duas tabelas, "**pedidos"** e "**clientes"**, e é necessário listar todos os pedidos de um cliente específico

Tabela pedidos (com milhões de registros)

```
CREATE TABLE pedidos (
    id INT PRIMARY KEY AUTO_INCREMENT,
    cliente_id INT NOT NULL, -- Chave estrangeira para clientes
    data_pedido DATE,
    valor DECIMAL(10,2)
);
```

Tabela clientes (com milhares de registros)

```
CREATE TABLE clientes (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(255)
);
```

Consulta para obter pedidos de um cliente específico de nome "Empresa XYZ", relacionando os campos "id" das tabelas "pedidos" e "clientes"

```
SELECT p.id, p.data_pedido, p.valor, c.nome
FROM pedidos p
JOIN clientes c ON p.cliente_id = c.id
```

```
WHERE c.nome = 'Empresa XYZ';
```

Problema sem Índice: Embora “**id”** seja uma chave primária em “**clientes”,** que é indexada automaticamente, se não houver um índice na coluna “**cliente_id”** da tabela pedidos, que é a chave estrangeira usada no **JOIN**, o SGBD pode ter que escanear a tabela “**pedidos**” inteira ou usar um algoritmo de **join** menos eficiente para encontrar os pedidos relacionados ao cliente. Em tabelas muito grandes, isso causa lentidão massiva.

Cria um índice de nome “idx_cliente” na chave estrangeira “cliente_id” na tabela “pedidos”

```
CREATE INDEX idx_cliente_id ON pedidos (cliente_id);
```

Indexar a coluna **cliente_id** na tabela **pedidos** otimiza a operação de JOIN, permitindo que o banco de dados encontre rapidamente os pedidos de um **cliente_id** específico, reduzindo significativamente o tempo de execução da query e o impacto nos recursos do servidor.

*Os índices são ferramentas poderosas para melhorar o desempenho de consultas de leitura. Para o profissional de infraestrutura de TI, saber identificar quando uma query está sofrendo por falta de um índice (muitas vezes detectado por **EXPLAIN** ou por ferramentas de monitoramento que apontam "**Table Scans**") é tão importante quanto saber como criá-los. Implementar índices estrategicamente pode transformar um servidor de banco de dados lento e sobrecarregado em um ambiente responsivo e eficiente, liberando recursos preciosos para outras operações.*

Função EXPLAIN

A ferramenta **EXPLAIN** mostra o plano de execução que o banco de dados pretende seguir para executar uma query. Isso não é sobre otimizar a query em si

(tarefa mais do desenvolvedor), mas sim sobre entender por que uma query está consumindo muitos recursos do servidor (CPU, memória, I/O de disco) e identificar se ela está ocorrendo uma carga excessiva e talvez desnecessária.

Ao analisar o **EXPLAIN**, devem ser percebidos indicadores de ineficiência, tais como:

- **Scans completos de tabela:** O banco de dados está lendo a tabela inteira em vez de usar um índice?
- **Joins ineficientes:** Como as tabelas estão sendo unidas?
- **Uso de arquivos temporários em disco:** O banco está usando o disco para processar a query, o que é muito mais lento que a memória?

No banco de dados MySQL, para ativar a função **EXPLAIN** basta usar como prefixo a palavra-chave "**EXPLAIN**" nas consultas com o comando **SELECT**.

Exemplo 1: Query Eficiente (usando índice)

Uma tabela "**usuarios**" com milhões de registros e um índice na coluna "**email**".

Mostra os campos "nome" e "email" da tabela usuários onde o "email" seja igual a "fulano@empresa.com.br" e apresenta o resultado da função EXPLAIN

```
EXPLAIN SELECT nome, email FROM usuarios WHERE email = 'fulano@empresa.com.br';
```

Saída Esperada (Simplificada):

```
id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra

1 | SIMPLE | usuarios | NULL | ref | idx_email | idx_email | 768 | const | 1 | 100.00 | Using where
```

Análise (Simplificada):

***type: ref**: Indica que o banco usou um índice "**idx_email**" para encontrar as linhas. Isto indica que a busca foi direta e eficiente.*

***rows: 1**: Estima que apenas 1 linha será lida para satisfazer a condição, o que é excelente.*

Essa query é muito rápida e eficiente, com baixo impacto nos recursos.

Exemplo 2: Query Ineficiente (sem usar índice)

Neste exemplo não existe um índice na coluna "**data_cadastro**" e a tabela "**pedidos**" possui centenas de milhares de registros.

Mostra todos os registros da tabela "pedidos" onde o campo "data_cadastro" for mais recente que "01/01/2023" e apresenta o resultado da função EXPLAIN

```
EXPLAIN SELECT * FROM pedidos WHERE data_cadastro < '2023-01-01';
```

Saída Esperada (Simplificada):

id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra

*1 | SIMPLE | **pedidos** | NULL | ALL | NULL | NULL | NULL | NULL | 1000000 | 10.00 | Using where*

Análise (Simplificada):

***type: ALL**: Indica que o MySQL está fazendo um "Table Scan" completo, lendo todas as centenas de milhares de linhas da tabela "pedidos" para encontrar as correspondências.*

***rows: 1000000**: Confirma que um milhão de linhas foram lidas. Isso consome muitos recursos de I/O de disco e CPU, especialmente em tabelas grandes.*

***possible_keys: NULL** e **key: NULL**: Nenhum índice foi considerado ou usado.*

Uma query como essa, se executada frequentemente ou em tabelas muito grandes, pode esgotar os recursos do servidor, levando a lentidão generalizada e até mesmo travamentos. A solução para o desenvolvedor seria criar um índice em "**data_cadastro**", mas para um administrador de infraestrutura de TI, o "**EXPLAIN**" identifica o problema de consumo de recursos.

Replicação

A replicação é uma técnica fundamental na administração de bancos de dados para garantir alta disponibilidade, tolerância a falhas e escalabilidade de leitura. Em essência, a replicação envolve copiar os dados de um servidor de banco de dados (o servidor primário, *master* ou principal) para um ou mais servidores (os servidores de réplica ou *slave*). Isso cria cópias idênticas do banco de dados que podem ser usadas para:

- **Recuperação de Desastres:** Se o servidor primário falhar, uma réplica pode ser promovida para assumir as operações.
- **Balanceamento de Carga de Leitura**: Aplicações podem direcionar as consultas de leitura para os servidores de réplica, aliviando a carga sobre o servidor primário e melhorando o desempenho geral.
- **Relatórios e Análise**: Tarefas que exigem muitas leituras podem ser executadas nas réplicas sem impactar o desempenho do servidor primário de produção.

No MySQL e MariaDB, a replicação nativa é baseada em **binlogs** (logs binários). O servidor primário registra todas as alterações de dados (INSERTs, UPDATEs, DELETEs, DDLs) em seus **binlogs**. Os servidores de réplica leem esses **binlogs** e aplicam as mesmas alterações, garantindo que estejam sincronizados. A

configuração mais comum é a replicação assíncrona **Master-Slave**, onde o servidor primário não espera a confirmação da réplica para continuar processando transações.

Exemplo de Configuração Básica (Master-Slave Assíncrona)

No Servidor Primário (Master):

Editar o arquivo de configuração do MySQL/MariaDB geralmente localizado em "**/etc/mysql/my.cnf**" ou "**/etc/my.cnf**" e e após a edição, reiniciar o serviço com o comando "**systemctl restart mysql"**.

```
[mysqld]

 # ID único para este servidor (deve ser um inteiro)

server-id = 1

 # Habilita o binlog e define o prefixo dos arquivos

log_bin = mysql-bin

# Permite conexões de outras máquinas (ou um IP específico)

bind-address = 0.0.0.0
```

Cria um usuário "replica_user" com a senha "senha@123" para replicação e concede as permissões necessárias

```
CREATE USER 'replica_user'@'%' IDENTIFIED BY 'senha@123';

GRANT REPLICATION SLAVE ON *.* TO 'replica_user'@'%';

FLUSH PRIVILEGES;
```

Para sincronizar com um banco já em uso, obtenha a posição atual do binlog para iniciar a réplica. Em caso de bancos "Master" e "Slave" sendo iniciados, este procedimento não é necessário.

```
SHOW MASTER STATUS;
```

*Anotar "**File**" e "**Position**" da saída. Ex: **File: 'mysql-bin.000001', Position: 1234***

No Servidor de Réplica (Slave):

Editar o arquivo de configuração do MySQL/MariaDB ("**my.cnf"**) e após a edição, reiniciar o serviço MySQL "**systemctl restart mysql".**

```
[mysqld]

# ID único para este servidor (deve ser diferente do master)

server-id = 2

# Opcional: torna o slave somente leitura para evitar escritas acidentais

read_only = 1

# Permite conexões de outras máquinas (ou um IP específico)

bind-address = 0.0.0.0
```

Configurar a réplica para se conectar ao **master** e iniciar a replicação (usando as informações obtidas (**File** e **Position**) na execução de **SHOW MASTER STATUS** no **Master**. No exemplo, o endereço IP do servidor Master é "192.168.254.33". Estes comandos devem ser enviados na console do utilitário **mysql** no servidor **slave**.

```
CHANGE MASTER TO

 MASTER_HOST='192.168.254.33',

MASTER_USER='replica_user',

MASTER_PASSWORD='senha@123',
```

*MASTER_LOG_FILE='**mysql-bin.000001**';*

*MASTER_LOG_POS=**1234**;*

START SLAVE;

Para verificar o status da replicação no servidor de réplica

SHOW SLAVE STATUS\G;

*Observar na saída da execução do comando "**SHOW SLAVE STATUS\G;**" os seguintes parâmetros para saber o status da réplica (Slave):*

- *Slave_IO_Running: **Yes***
- *Slave_SQL_Running: **Yes***
- *Last_IO_Error e Last_SQL_Error (**devem estar vazios**)*
- *Seconds_Behind_Master: 0 (**idealmente zero ou muito próximo, indica quão atrasada a réplica está**)*

Algumas considerações sobre o uso de replicação em bancos de dados:

- **Latência de Rede:** Os recursos de rede (largura de banda, latência, perdas, jitter) entre o primário e as réplicas impactam diretamente o parâmetro "**Seconds_Behind_Master"**
- **Monitoramento Constante:** É essencial monitorar ativamente o status da replicação. Falhas na rede, espaços de armazenamento cheios ou erros de configuração podem quebrar a replicação, deixando as réplicas desatualizadas
- **Recursos da Réplica:** As réplicas precisam ter recursos (CPU, RAM, I/O de disco) suficientes para acompanhar as operações do primário
- **Segurança:** O usuário de replicação deve ter apenas as permissões mínimas necessárias para essa tarefa

- **Failover e Promoção:** Embora a replicação crie cópias, a promoção de uma réplica a primário em caso de falha (*failover*) geralmente requer ferramentas adicionais ou *scripts* manuais para garantir que a aplicação se reconecte ao novo primário

mysql - Atalhos

connect (\r) *- Conecta em um servidor, parâmetros são nome do banco e endereço do host*

\r empresa localhost

edit (\e) *- Edita o último comando com o editor definido na variável do sistema $EDITOR (editor padrão é o vi).*

Para alterar para o nano:

EXPORT EDITOR="nano"

ego (\G) *- Apresenta os resultados verticalmente*

*select * from funcionarios limit 5 \G*

exit (\q) *- Mesma função do comando quit*

help (\h) *- Apresenta as opções de ajuda para comandos e parâmetros*

help show grants

pager (\P) *- Define o paginador para a saída dos comandos*

\P less utilitário "less" como paginador

nopager (\n) *- Desabilita a saída pelo pager e retorna para a saída padrão (stdout)*

notee (\t) *- Desabilita o redirecionamento dos comandos e saídas para um arquivo*

prompt (\R) *- Modifica o prompt padrão. Ao executar sem parâmetros retorna para o prompt padrão.*

prompt (\u@\h) [\d]>_ --> (root@localhost) [empresa]>

***quit (\q)** - Encerra a sessão*

***source (\.)** - Executa um arquivo com comandos sql*

source /tmp/comandos.sql

***status (\s)** - Apresenta informações sobre o status do servidor*

***system (\!)** - Executa um comando no shell do sistema operacional*

system grep "show" ~/.mysql_history

***tee (\T)** - Direciona a saída dos comandos para um arquivo*

\T /tmp/comandos.log

***use (\u)** - Acessa um banco de dados*

use empresa;

***SHOW databases;** - lista banco de dados*

***SHOW tables;** - lista as tabelas do banco de dados corrente*

***DESC nome_da_tabela;** - mostra a estrutura de uma tabela*

DESC funcionarios;

*Mais opções de **prompt** - https://dev.mysql.com/doc/refman/8.4/en/mysql-commands.html*

Ferramentas complementares

mytop – ferramenta similar ao top para monitoramento de processos no MySQL

Instalação: yum install mytop

Uso básico

Para analisar os processos do banco empresa em localhost

 # mytop -h 127.0.0.1 -d empresa

Atalhos

d – troca de banco de dados

k – encerra uma thread

h – lista threads de um host específico

mysqlshow - lista banco de dados

 # mysqlshow

Lista as tabelas do banco "empresa"

 # mysqlshow empresa

mysqladmin - Utilitário para tarefas administrativas no shell do sistema operacional.

Cria o banco de dados "empresa"

 # mysqladmin create empresa

Remove o banco de dados "empresa"

 # mysqladmin drop empresa

Lista os processos em execução

 # mysqladmin processlist

Termina o processo com o ID “1812”

 # mysqladmin kill 1812

Troca a senha do usuário root (MySQL)

 # mysqladmin password

Mostra a versão do banco e outras informações

 # mysqladmin version

Mostra informações resumidas sobre o estado do banco

 # mysqladmin status

Para o serviço do banco de dados

 # mysqladmin shutdown

Faz um reload na tabelas de permissões (grants)

 # mysqladmin reload

mysql - Interface de linha de comando interativa para o MySQL.

Conecta no serviço do MySQL no endereço de host “192.168.10.5”, na porta TCP “7000”, com o usuário “aluno” e no banco de dados “empresa” (será solicitada a senha)

 # mysql -h 192.168.10.5 --port=7000 -u aluno -D empresa

Conecta no serviço do MySQL no endereço de host “192.168.10.5”, na porta TCP 7000, com o usuário “aluno” e envia a senha no banco de dados “empresa”

 # mysql -h 192.168.254.34 -p'senac2010' -u aluno10 -D empresa

Lista os bancos de dados do servidor local, parâmetro “-e” executa um comando SQL

 # mysqlshow mysql -h 127.0.0.1 -u root -e "SHOW databases;"

Lista os bancos de dados do servidor remoto de IP “192.168.10.5” e porta padrão TCP “7000”, acessando como usuário “aluno10” e será pedida a senha

 # mysql -h 192.168.10.5 --port=7000 -u aluno10 -p’@P4ssW0rd$’ -s -e “SHOW databases;”

 # mysqlshow -h192.168.10.5 -P7000 -u aluno10 -p’@P4ssW0rd$’

mysqlslap - Ferramenta de diagnóstico que acompanha o MySQL que possibilita testar a carga de servidores de banco de dados. Pode emular um grande número de conexões de cliente que chegam ao servidor de banco de dados simultaneamente.

Realiza a consulta “*SELECT count(*) FROM funcionarios;*” com 88 conexões simultâneas, por 100 vezes, no banco de dados “empresa”, com o usuário “root” (será pedido a senha)

 # mysqlslap --user=root --password --host=localhost --concurrency=88 --iterations=100 --create-schema=empresa --query="SELECT count(*) FROM funcionarios;"

Cria uma tabela temporária "a" e insere um valor nela, simula 80 usuários simultâneos executando a consulta "SELECT * FROM a", repete todo esse processo de teste 20 vezes para coletar métricas de desempenho.

*# mysqlslap --user=root --password --delimiter=";" --create="CREATE TABLE a (b int); INSERT INTO a VALUES (23)" --query="SELECT * FROM a" --concurrency=80 --iterations=20*

4) Exemplos de queries SQL

Estes são alguns exemplos de consultas SQL simples para entendimento da sintaxe do tratamento de dados armazenados em sistemas de bancos de dados.

Conta o número de registros da tabela “funcionarios”

```
select count(*) from funcionarios;
```

Seleciona os registros da tabela “funcionarios” que o campo ”nome” contenham os caracteres “ed”

```
select * from funcionarios where nome LIKE '%ed%';
```

Calcula a média de salário de todos os funcionários, função AVG (média) da coluna “salario” na tabela “funcionarios”

```
select avg(salario) from funcionarios;
```

Seleciona os dez maiores salários, ordena a coluna “salario” da tabela “funcionarios” de forma decrescente e mostrar os primeiros 10 registros “LIMIT 10”

```
SELECT * FROM funcionario ORDER BY salario DESC  LIMIT 10;
```

Soma o total de salários, utiliza a função SUM (somatório) na coluna salários da tabela “funcionarios”

```
SELECT sum(salario) FROM funcionarios;
```

Seleciona todos os dados dos funcionários os quais possuem “ed” no nome e salário maior do que 99999, apresenta todas as colunas da tabela “funcionarios” onde o campo “nome” contenha os caracteres “ed” e que os campo “salario” sejam menores que “99999”

```
SELECT * FROM funcionarios WHERE nome LIKE '%ed' AND salario < 99999;
```

Atualiza em 10% os salários menores do que 99999, altera a coluna “salario” da tabela “funcionarios”, multiplicando o valor atual do campo por “1.1” onde os registros tenham valores menores do que 99999 no campo “salario”

```
update funcionarios SET salario = (salario * 1.1) where salario < 99999;
```

Atualiza em 20% os salários dos funcionários do sexo feminino, multiplicando o valor atual do campo por “1.2” onde os registros tenham “F” no campo “sexo”

```
UPDATE funcionarios SET salario = (salario * 1.2) where sexo = 'F';
```

Troca o time dos funcionários com salário maior do que 4555555, atualiza o campo “time” para “GE Brasil” nos registros que possuem o campo “salario” com valor maior do que “4555555” na tabela “funcionarios”

```
update funcionarios set time='GE Brasil' where salario > 4555555;
```

Atualiza o ranking para “9” de todos os funcionários que começam com a letra "A” no nome, atualiza o campo “ranking” para o valor “9’ para todos os registros que começarem com a letra “A” no campo “nome” da tabela “funcionarios”

```
update funcionarios set ranking='9' where nome LIKE '%A';
```

Lista de forma única todos os nomes dos times, o parâmetro “distinct” retorna apenas uma vez cada valor da coluna “time” na tabela “funcionarios”

```
select distinct(time) from funcionarios;
```

Conta todos os funcionários que possuem o nome “Silva', seleciona todos os campos da tabela “funcionarios” onde contenha “Silva” em qualquer parte do campo “nome”

```
select count(*) from funcionarios where nome LIKE '%Silva%;
```

Mostra a quantidade de torcedores de cada time, seleciona o campo “nome”, cria um apelido como “torcedores” e o campo “time” da tabela “funcionarios” e agrupa pelo campo “time” com a contagem de “torcedores”

```
select count(nome) as torcedores, time from funcionarios group by time;
```

Mostra o número de funcionários por sexo, seleciona o campo “nome”, cria um apelido como “Qte” e o campo “sexo” da tabela “funcionarios” e agrupa pelo

campo “sexo” com a contagem de registros do campo “nome” e apresenta a coluna como “Qte”

```
select count(nome) as Qte, sexo from funcionarios group by sexo;
```

Insere na tabela “funcionarios” o nome, o time e o sexo do registro de um funcionário

```
INSERT INTO funcionarios (nome, time, sexo) VALUES (‘Fulano de Tal’, 'EC Pelotas', ‘M’);
```

Insere na tabela “funcionarios” o nome, o time e o sexo de três funcionários

```
INSERT INTO funcionarios (nome, time, sexo) VALUES
        (‘Cliclana de Tal’, 'EC Pelotas', ‘F’),
        (‘Beltrano de Tal’, 'Farroupilha', ‘M’);
        (‘Fulano de Tal’, 'EC Pelotas', ‘M’);
```

Inserção de dados usando arquivos

Insere dados na tabela “funcionarios” a partir de um arquivo em formato CSV

```
#Conteúdo do arquivo dados.csv
1,João Silva,joao.silva@empresa.com.br,2025-01-15
2,Maria Souza,maria.souza@empresa.com.br,2025-02-20
3,Carlos Pereira,carlos.pereira@empresa.com.br,2025-03-01
```

Criação da tabela “clientes” que receberá os dados importados do arquivo “dados.csv”

```
CREATE TABLE clientes (
    id INT PRIMARY KEY,
    nome VARCHAR(255),
```

```
    email VARCHAR(255),
    data_cadastro DATE
);
```

MySQL

*O arquivo "**dados.csv**" deve estar acessível ao usuário que executa o servidor MySQL. Por motivos de segurança, o arquivo precisa estar em um local onde o usuário do MySQL tenha permissão de leitura, ou no diretório de dados do MySQL.*

Importa os dados do arquivo "dados.csv" localizado no CLIENTE LOCAL em "/tmp" para a tabela "clientes", usando o final de linha do formato Linux "\n" e ignora a primeira linha do arquivo por ser o cabeçalho. Caso seja usado um arquivo que esteja armazenado no servidor deverá ser usada a diretiva "LOAD DATA INFILE" (sem o LOCAL).

```
LOAD DATA LOCAL INFILE '/tmp/dados.csv'
INTO TABLE clientes
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
IGNORE 1 LINES;
```

Importa os dados do arquivo "dados.csv" localizado no CLIENTE LOCAL em "/tmp", com os campos separados por vírgulas, para a tabela "clientes", usando o final de linha do formato Linux "\n" e ignora a primeira linha do arquivo por ser o cabeçalho. Neste caso serão importados apenas os campos "nome" e "email".

```
LOAD DATA LOCAL INFILE '/tmp/dados.csv'
INTO TABLE clientes
```

```
FIELDS TERMINATED BY ','

LINES TERMINATED BY '\n'

(nome, email);
```

No caso de importação parcial das colunas devem ser observadas as relações das chaves duplicadas e campos que não podem ser nulos.

PostgreSQL

Importa os dados do arquivo "/tmp/dados.csv", localizado no lado SERVIDOR, com os campos separados por vírgulas, para a tabela "clientes", ignorando a primeira linha "CSV HEADER". O arquivo "/tmp/dados.csv" deverá estar acessível para o usuário postgres no sistema de arquivos.

```
COPY clientes FROM '/tmp/dados.csv' DELIMITER ',' CSV HEADER;
```

Importa os dados do arquivo "/tmp/dados.csv", localizado no lado CLIENTE, com os campos separados por vírgulas, para a tabela "clientes", ignorando a primeira linha "CSV HEADER". O arquivo "/tmp/dados.csv" deverá estar acessível para o usuário postgres no sistema de arquivos.

```
\COPY clientes FROM '/home/aluno/dados.csv' DELIMITER ',' CSV HEADER;
```

Importa os dados do arquivo "dados.csv" localizado no SERVIDOR em "/tmp", com os campos separados por vírgulas, para a tabela "clientes", ignora a primeira linha do arquivo por ser o cabeçalho. Neste caso serão importados apenas os campos "nome" e "email".

```
COPY clientes (nome,email) FROM '/tmp/dados.csv' DELIMITER ',' CSV;
```

No caso de importação parcial das colunas devem ser observadas as relações das chaves duplicadas e campos que não podem ser nulos.

Exportação de Dados para Arquivos

MySQL

Seleciona os campos “id”, “nome”, “email” e “data_cadastro” da tabela “clientes” dos registros que tenham o ano de 2025 no campo “data_cadastro” e exporta para o arquivo “/tmp/clientes_2025.csv”. O arquivo terá os campos separados por vírgulas, os conteúdos dos campos entre aspas e as linhas terminadas com “\n”, padrão de arquivos textos no Linux.

```
SELECT id, nome, email, data_cadastro

FROM clientes

WHERE YEAR(data_cadastro) = 2025

INTO OUTFILE '/tmp/clientes_2025.csv'

FIELDS TERMINATED BY ','

ENCLOSED BY '"'

LINES TERMINATED BY '\n';
```

Exemplo de conteúdo do arquivo “clientes_2025.csv”

```
"1","João Silva","joao.silva@empresa.com.br","2025-01-15"
"2","Maria Souza","maria.souza@empresa.com.br","2025-02-20"
"3","Carlos Pereira","carlos.pereira@empresa.com.br","2025-03-01"
```

O caminho do arquivo é relativo ao servidor MySQL, não ao sistema de arquivos do sistema operacional. Por exemplo, o "/tmp" será uma pasta dentro da área privada do processo do MySQL, algo como ***"/tmp/systemd-private-8c0cf7bde4e4494b89fa99ac3d7175cd-mariadb.service-MTgwAn"****, sendo que a cada execução será gerado uma nova identificação da área privada do processo. Para gerar um novo arquivo, é preciso renomear ou remover o arquivo existente, pois ele não pode ser sobrescrito.*

PostgreSQL

Seleciona os campos "id", "nome", "email" e "data_cadastro" da tabela "clientes" dos registros que tenham o ano de 2025 no campo "data_cadastro" e exporta para o arquivo "/tmp/clientes_2025.csv". O arquivo terá a primeira linha com o cabeçalho (nome das colunas), os campos separados por vírgulas, os conteúdos dos campos entre aspas e as linhas terminadas com "\n", padrão de arquivos textos no Linux.

```
COPY (SELECT id, nome, email, data_cadastro

FROM clientes

WHERE EXTRACT(YEAR FROM data_cadastro) = 2025)

TO '/tmp/clientes_2025.csv'

WITH (FORMAT CSV, HEADER, DELIMITER ';', QUOTE '"');
```

Se houver privilégios de superusuário no banco de dados, ou se o arquivo precisa ser gravado na máquina cliente (onde está executando o psql), pode ser usado o meta-comando **\COPY** do utilitário psql.

A sintaxe é quase idêntica ao COPY TO, mas o "\" na frente indica que é um comando do psql e o caminho do arquivo é local (lado da máquina cliente).

```
\COPY (SELECT id, nome, email, data_cadastro

FROM clientes
```

```
WHERE EXTRACT(YEAR FROM data_cadastro) = 2025)

TO '/home/aluno/clientes_2025.csv'

WITH (FORMAT CSV, HEADER, DELIMITER ',', QUOTE '"');
```

Exemplo de conteúdo do arquivo “clientes_2025.csv”

```
id,nome,email,data_cadastro
1,João Silva,joao.silva@empresa.com.br,2025-01-15
2,Maria Souza,maria.souza@empresa.com.br,2025-02-20
3,Carlos Pereira,carlos.pereira@empresa.com.br,2025-03-01
```

5) Dicas de Administração de Bancos de Dados

A administração de bancos de dados envolve a infraestrutura dimensionada de forma adequada e os ajustes dos parâmetros disponíveis para melhor uso dos recursos do hardware. Os recursos computacionais necessários para obtenção de melhor desempenho dependerá do tipo de uso do banco de dados. Em sistemas de bancos de dados com milhares de transações por segundo os recursos necessários serão bem maiores do que em um sistema de cadastro simples com poucas transações por segundo.

Como regra geral, a utilização de servidores com processadores velozes, memória RAM abundante e armazenamento de alta velocidade, como SSDs, são essenciais para lidar com as demandas de um banco de dados. Em um ambiente virtualizado ou com o uso de storages em rede, a largura de banda disponível para acesso ao armazenamento poderá comprometer o desempenho do banco de dados. Uma das estratégias utilizadas para melhorar o desempenho é o uso de *cache* em memória.

A rede desempenha um papel crítico para o desempenho dos bancos de dados, exigindo alta velocidade e baixa latência para garantir a comunicação eficiente entre servidores e clientes, com segmentação e redundância para evitar gargalos e interrupções. O consumo de largura de banda não costuma ser alto em cada conexão, normalmente, as queries cabem em um pacote de rede e as respostas consomem poucos recursos da rede.

O tempo de atraso na espera das respostas das queries, na grande parte das vezes, está no tempo de processamento no servidor do banco de dados. Este tipo de problema pode ser demonstrado facilmente com o uso de ferramentas tais como Tcpdump ou Wireshark, onde poderá ser analisado o tráfego na rede. Para entender melhor este cenário, no vídeo disponível em O problema é na rede? (https://www.youtube.com/watch?v=HrMcFvldpG4) é mostrado um exemplo bem comum de atraso no acesso ao servidor do banco de dados na qual a rede leva a culpa e a origem do problema é no tempo de processamento da resposta no lado servidor.

A escolha de um sistema operacional estável e seguro, otimizado para bancos de dados, e a manutenção de software atualizado com patches de segurança são práticas indispensáveis. A primeira opção é o sistema operacional Linux para uso de banco de dados MySQL/MariaDB e PostgreSQL.

A segurança é primordial, principalmente, no gerenciamento de permissões de acesso por usuário e aplicações em bancos de dados e tabelas. Aliado a isto com a implementação de firewalls, sistemas de detecção e prevenção de intrusão, atualizações e auditorias em logs. O monitoramento contínuo da infraestrutura, com ferramentas que acompanham o desempenho de CPU, memória, disco e rede, permite identificar e resolver problemas proativamente. Alguns parâmetros importantes a serem monitorados em sistemas de bancos de dados são listados na Tabela 1.

Tabela 1. Parâmetros importantes a serem monitorados

Parâmetro	Observação
Uso de CPU (carga)	Monitorar o consumo de CPU do sistema operacional e dos processos do banco de dados
Uso de memória	Monitorar o consumo de memória do sistema operacional e dos processos do banco de dados
Número de processos	Monitorar o número de processos do sistema operacional e do banco de dados
Espaço em disco	Monitorar o espaço em disco do sistema operacional e das partições do banco de dados
Porta do serviço	Monitorar se a porta de comunicação do serviço do banco de dados está ativa
Uptime	Monitorar o tempo de atividade do sistema operacional e do banco de dados (importante para reinícios inesperados do processo do banco)
Validação de permissões e autenticação dos usuário no banco	Monitorar se as permissões dos usuários estão corretas e a autenticação no banco está funcional

Número de conexões	Monitorar a quantidade de conexões no banco de dados
Espaço em disco	Monitorar o espaço utilizado pelo banco e por tabelas
Transações por Segundo	Monitorar o número de transações por segundo realizadas no banco
Tempo de resposta - Query SQL	Monitorar por meio de alguma query conhecida o tempo de resposta. Usar o tempo de resposta da query como base para identificar anomalias nos tempos de resposta do servidor
Número de locks em tabelas	Monitorar o número de locks, que são bloqueios em tabelas que impedem a escrita. Em locks de maior duração poderá acarretar em bloqueios de conexões para os usuários e enfileiramento de processos
Queries Lentas	Monitorar a quantidade slow queries (consultas lentas) em logs
Taxa de acertos no cache em memória	Monitorar a porcentagem de acertos de dados que estão no cache da memória
Número de arquivos abertos	Monitorar o número de arquivos abertos pelos processos do banco de dados
Logs	Monitorar de forma automatizada os logs do banco de dados e buscar padrões de anomalias tais como tentativas de acessos, erros gerados pelo processo do banco, queries com alto consumo de recursos e alertas

6) Considerações Finais

Neste livro, foram explorados os fundamentos da administração de bancos de dados MySQL/MariaDB e PostgreSQL, ferramentas essenciais para profissionais de TI. Dominar os comandos SQL básicos e as rotinas de administração é uma competência crucial para administradores de rede e responsáveis pela infraestrutura. As dicas aqui apresentadas servem como base para garantir a eficiência, segurança e confiabilidade dos sistemas de informação, que se apoiam fortemente no uso de bancos de dados.

É importante reconhecer que a administração de bancos de dados compreende mais do que a mera execução de comandos; ela exige uma compreensão profunda da infraestrutura de TI, otimização de recursos e a capacidade de resolver problemas complexos. As rotinas apresentadas nesta obra são essenciais para a manutenção da integridade dos bancos de dados, desde a configuração inicial até o monitoramento contínuo e a implementação de backups e restaurações.

A distinção entre as áreas de infraestrutura e desenvolvimento tem se tornado cada vez mais reduzida, especialmente com a ascensão do conceito de DevOps. Este livro busca prepará-lo não apenas para administrar bancos de dados, mas também para colaborar efetivamente com desenvolvedores, integrando as melhores práticas de ambas as áreas.

Este material, no entanto, é apenas o ponto de partida. Os bancos de dados MySQL/MariaDB e PostgreSQL são vastos e estão em constante evolução, com novas funcionalidades e otimizações sendo introduzidas regularmente. É importante explorar a documentação oficial, participar de comunidades online e sempre estar atento às novas funcionalidades.

Realizar experimentos com os comandos e técnicas apresentados neste livro em ambientes de teste, simulação de cenários de falha e análises para diagnosticar e resolver problemas é essencial para o aprendizado. Para isto, é fundamental manter um ambiente de testes o mais próximo possível ao ambiente de produção para que sejam válidas as simulações e testes realizados.

Por fim, este livro foi concebido não apenas como um guia de aplicação direta, mas como um ponto de partida sólido para o aprofundamento dos conhecimentos. A continuidade dos estudos e o aprimoramento constante são fundamentais na jornada para dominar a administração de bancos de dados MySQL/MariaDB e PostgreSQL.

Referências bibliográficas

Estas referências bibliográficas aqui listadas foram consultadas para validar comandos entre versões diferentes do MySQL e PostgreSQL e para obtenção de exemplos com parâmetros mais comuns dos comandos. As principais referências utilizadas foram as documentações oficiais disponibilizadas pelos desenvolvedores dos sistemas de bancos de dados PostgreSQL, MySQL e MariaDB.

PostgreSQL

PostgreSQL Documentation

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

PostgreSQL ALTER DEFAULT PRIVILEGES - permissions explained

https://www.cybertec-postgresql.com/en/postgresql-alter-default-privileges-permissions-explained/

PGTune

https://pgtune.leopard.in.ua/

PostgreSQL Administration

https://neon.tech/postgresql/postgresql-administration

pg_activity

https://github.com/dalibo/pg_activity

pg_view

https://github.com/zalando/pg_view/

PostgreSQL - How to grant access to users?

https://tableplus.com/blog/2018/04/postgresql-how-to-grant-access-to-users.html

change_obj_ownership

https://github.com/trraghav/change_obj_ownership

User Management in PostgreSQL

https://hostman.com/tutorials/user-management-in-postgresql/

MySQL/MariaDB

MySQL Documentation

https://dev.mysql.com/doc/

MariaDB Server Documentation

https://mariadb.com/kb/en/documentation/

mysqldumpslow — Summarize Slow Query Log Files

https://dev.mysql.com/doc/refman/8.4/en/mysqldumpslow.html

MySQL Administration

https://www.mysqltutorial.org/mysql-administration/

mysqlcheck: Check and Repair Tables & Databases

https://hevodata.com/learn/mysqlcheck/

How To Reset Your MySQL or MariaDB Root Password

https://www.digitalocean.com/community/tutorials/how-to-reset-your-mysql-or-mariadb-root-password

MySQLTuner

https://github.com/major/MySQLTuner-perl

mysqlconfigurer

https://github.com/Releem/mysqlconfigurer

tuning-primer

https://github.com/mattiabasone/tuning-primer

mysqlmonitor-script

https://github.com/haydenjames/mysqlmonitor-script

Configuring MySQL to Use Encrypted Connections

https://dev.mysql.com/doc/refman/8.3/en/using-encrypted-connections.html