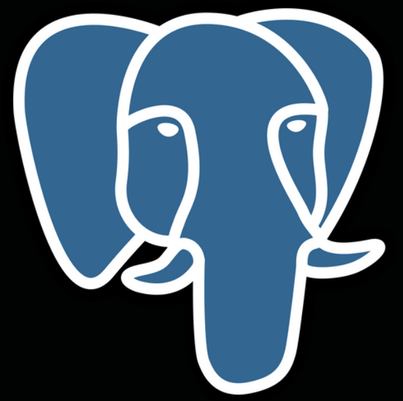
Unisenac
Campus Pelotas

BD2 – Banco de Dados 2



Apresentando o PostgreSQL

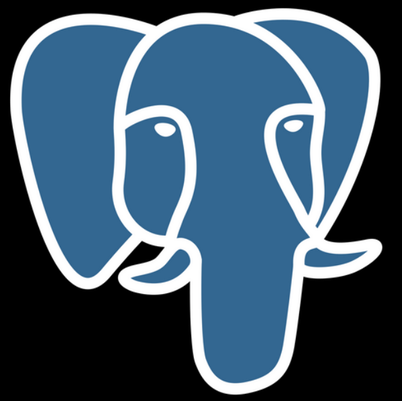
```
-- O que é PostgreSQL?

/*
Sistema Gerenciador de Banco de Dados (SGBD) relacional de código
aberto.

Altamente compatível com SQL padrão e com suporte a extensões avançadas.

Suporta extensões avançadas: JSON, XML, Procedimentos, etc.

Popular em sistemas corporativos, acadêmicos e cloud.
*/
```



```
-- MySQL vs PostgreSQL (Resumo Geral)

+--------------------+------------------+---------------------------+
| Característica     | MySQL            | PostgreSQL                |
|--------------------|------------------|---------------------------|
| Licença            | GPL              | PostgreSQL License (MIT)  |
| Ponto forte        | Velocidade       | Conformidade com padrão   |
| JSON               | Parcial          | Completo                  |
| Escalabilidade     | Alta             | Alta                      |
| Terminal padrão    | mysql -u root -p | psql -U postgres          |
| Criação de usuário | CREATE USER      | CREATE ROLE               |
+--------------------+------------------+---------------------------+
```

```
-- Conectando via Terminal

-- MySQL
mysql -u root -p

-- PostgreSQL
psql -U postgres
```

```sql
-- Criando o banco de dados


-- MySQL
DROP DATABASE IF EXISTS loja;
CREATE DATABASE IF NOT EXISTS loja;

-- Acessa (usa) o banco de dados criado
USE loja;




-- PostgreSQL
DROP DATABASE IF EXISTS loja;
CREATE DATABASE loja;

-- conecta ao banco de dados criado
\c loja
```

```sql
-- MySQL e PostgreSQL

-- Criando tabela clientes
CREATE TABLE clientes (
  id INT PRIMARY KEY,
  nome CHAR(30)
);

-- Criando tabela pedidos
CREATE TABLE pedidos (
  id INT PRIMARY KEY,
  cliente_id INT,
  valor NUMERIC(10,2)
);


-- Dica:
No PostgreSQL, é mais comum usar NUMERIC, no MySQL é mais comum o DECIMAL.
O funcionamento de ambos é idêntico.
```

```sql
-- MySQL e PostgreSQL

-- Inserindo dados nas tabelas
INSERT INTO clientes (id, nome) VALUES
(1, 'João'),
(2, 'Maria'),
(3, 'Carlos'),
(4, 'Fernanda'),
(5, 'Ana');

INSERT INTO pedidos (id, cliente_id, valor) VALUES
(1, 1, 150.00),
(2, 2, 220.50),
(3, 2, 99.90),
(4, 3, 305.00),
(5, 1, 50.00),
(6, 4, 400.00),
(7, 6, 120.00);  -- cliente_id 6 não existe
```

```sql
-- MySQL e PostgreSQL

/* INNER JOIN – Somente clientes com pedidos
e pedidos que pertencem a cliente */
SELECT clientes.nome, pedidos.valor
FROM clientes
INNER JOIN pedidos ON clientes.id = pedidos.cliente_id;

-- LEFT JOIN - Todos os clientes (com ou sem pedido)
SELECT clientes.nome, pedidos.valor
FROM clientes
LEFT JOIN pedidos ON clientes.id = pedidos.cliente_id;

-- RIGHT JOIN - Todos os pedidos (com ou sem cliente)
SELECT clientes.nome, pedidos.valor
FROM clientes
RIGHT JOIN pedidos ON clientes.id = pedidos.cliente_id;
```

```sql
-- MySQL e PostgreSQL

-- LEFT Excluding JOIN - Clientes sem pedidos
SELECT clientes.nome
FROM clientes
LEFT JOIN pedidos ON clientes.id = pedidos.cliente_id
WHERE pedidos.id IS NULL;

-- RIGHT Excluding JOIN - Pedidos sem cliente
SELECT clientes.nome
FROM clientes
RIGHT JOIN pedidos ON clientes.id = pedidos.cliente_id
WHERE pedidos.id IS NULL;
```

```sql
-- MySQL e PostgreSQL

-- Clientes com ou sem pedidos (agrupados) - Qtd de pedidos por cliente
SELECT clientes.nome, COUNT(pedidos.id) AS total_pedidos
FROM clientes
LEFT JOIN pedidos ON clientes.id = pedidos.cliente_id
GROUP BY clientes.nome;
```

```sql
-- MySQL e PostgreSQL

-- Clientes com pedidos (agrupados) - Qtd de pedidos por cliente
SELECT clientes.nome, COUNT(pedidos.id) AS total_pedidos
FROM clientes
INNER JOIN pedidos ON clientes.id = pedidos.cliente_id
GROUP BY clientes.nome;
```

```sql
-- FULL JOIN – Todos os clientes e todos os pedidos
-- (nativo PostgreSQL, simulado no MySQL)

-- PostgreSQL
SELECT clientes.nome, pedidos.valor
FROM clientes FULL OUTER JOIN pedidos ON clientes.id = pedidos.cliente_id
ORDER BY nome;



-- MySQL e PostgreSQL
SELECT clientes.nome, pedidos.valor
FROM clientes
LEFT JOIN pedidos ON clientes.id = pedidos.cliente_id
UNION
SELECT clientes.nome, pedidos.valor
FROM clientes
RIGHT JOIN pedidos ON clientes.id = pedidos.cliente_id
ORDER BY nome;
```

```sql
-- NATURAL JOIN

-- PostgreSQL (somente PostgreSQL)
SELECT *
FROM clientes NATURAL JOIN pedidos;
-- Dispensa o uso do ON clientes.cliente_id = pedidos.cliente_id
-- É tentador de tão fácil, né?

O NATURAL JOIN realiza a junção automaticamente com base em todas as colunas
que possuem o mesmo nome e tipo nas duas tabelas.*/
```

```sql
-- Assim não funciona

CREATE TABLE clientes (
 id INT PRIMARY KEY,
 nome CHAR(30)
);

CREATE TABLE pedidos (
 id INT PRIMARY KEY,
 cliente_id INT,
 valor NUMERIC(10,2)
);
```

```sql
-- Assim funciona

CREATE TABLE clientes (
 cliente_id INT PRIMARY KEY,
 nome CHAR(30)
);

CREATE TABLE pedidos (
 id INT PRIMARY KEY,
 cliente_id INT,
 valor NUMERIC(10,2)
);
```

```sql
-- NATURAL JOIN

-- PostgreSQL (somente PostgreSQL)
SELECT *
FROM clientes NATURAL JOIN pedidos;

-- É tentador de tão fácil, né?
/* ALERTA: Uso de NATURAL JOIN
O NATURAL JOIN realiza a junção automaticamente com base em todas as colunas
que possuem o mesmo nome e tipo nas duas tabelas.*/

-- Problemas que podem ocorrer:
-- Pode gerar resultados inesperados se novas colunas com o mesmo nome forem adicionadas.
-- É difícil de ler e entender o critério de junção (não explícito).
-- Pode afetar a performance ou retornar resultados incorretos silenciosamente.

-- Boa prática:
-- Prefira sempre usar JOIN com cláusula ON explícita, para garantir clareza e controle.

-- Exemplo mais seguro:
SELECT *
FROM clientes
JOIN pedidos ON clientes.id = pedidos.cliente_id;
```

```sql
-- SHOW DATABASES

-- MySQL
SHOW DATABASES;

-- PostgreSQL
\l
SELECT datname FROM pg_database;
```

```
-- Conceito de SCHEMAS (somente no PostgreSQL)

/* O que é um SCHEMA?
Um SCHEMA no PostgreSQL é como uma "pasta" ou "espaço de nomes" dentro do banco de dados.

Ele permite:
- Organizar melhor as tabelas, views e funções.
- Separar lógicas por módulo (ex: vendas, financeiro, estoque).
- Ter tabelas com o mesmo nome em schemas diferentes (sem conflito).

Cada banco de dados já tem um schema padrão chamado: public

Você pode criar novos schemas assim:
*/

CREATE SCHEMA vendas;
```

```sql
-- Você pode criar novos schemas assim:
CREATE SCHEMA vendas;

-- Criando tabela dentro do schema vendas:
CREATE TABLE vendas.clientes (
  id SERIAL PRIMARY KEY,
  nome TEXT,
  cidade TEXT
);

-- Criando outra tabela com o mesmo nome no schema 'financeiro':
CREATE SCHEMA financeiro;

CREATE TABLE financeiro.clientes (
  id SERIAL PRIMARY KEY,
  nome TEXT,
  limite_credito NUMERIC(10,2)
);
```

```sql
-- Inserindo dados em vendas.clientes
INSERT INTO vendas.clientes (nome, cidade) VALUES
  ('João Silva', 'Porto Alegre'),
  ('Maria Souza', 'Pelotas'),
  ('Carlos Lima', 'Canoas');

-- Inserindo dados em financeiro.clientes
INSERT INTO financeiro.clientes (nome, limite_credito) VALUES
  ('Angelo Luz', 1000.00),
  ('Gladimir Catarino', 2500.50),
  ('Pablo Rosa', 500.00);
```

```sql
-- Acesso às tabelas com schema explícito:
SELECT * FROM vendas.clientes;
SELECT * FROM financeiro.clientes;

-- Dica:
-- Você pode configurar o search_path para priorizar um schema por padrão:
SET search_path TO vendas;

-- Agora, essa consulta vai buscar em vendas.clientes por padrão:
SELECT * FROM clientes;
```

```sql
-- VSUALIZAR TABELAS

-- MySQL
SHOW TABLES;

-- PostgreSQL
\dt
```

```sql
-- VER ESTRUTURA DA TABELA

-- MySQL
DESC clientes;

-- PostgreSQL
\d clientes
```

```sql
-- EXCLUIR TABELA e EXCLUIR BANCO DE DADOS

-- MySQL e PostgreSQL
DROP TABLE IF EXISTS pedidos;
DROP TABLE IF EXISTS clientes;
DROP DATABASE IF EXISTS loja;
```

```sql
-- TRUNCATE vs DELETE

-- MySQL e PostgreSQL
TRUNCATE TABLE pedidos;
DELETE FROM pedidos WHERE cliente_id = 2;

-- Qual a diferença?
-- DELETE:
- Remove registros um a um.
- Pode usar cláusula WHERE para filtrar.
- Pode ser revertido com ROLLBACK (se estiver em transação).
- Aciona triggers (gatilhos), se existirem.

-- TRUNCATE:
- Remove todos os registros da tabela de forma imediata.
- Não pode usar WHERE.
- Mais rápido e consome menos recursos.
- Em muitos SGBDs, não aciona triggers.
- Em PostgreSQL, pode ou não ser transacional (depende do contexto).

-- ATENÇÃO:
-- 'TRUNCATE' é mais perigoso: não tem volta se usado fora de transação.
-- Ideal para apagar tudo antes de um reprocessamento, não para exclusões parciais.

-- Dica:
-- Use 'DELETE' quando precisar de controle.
-- Use 'TRUNCATE' quando tiver certeza que quer limpar tudo e não precisa registrar cada exclusão.
```

```
-- Comandos exclusivos
-- Alguns comandos são específicos de cada SGBD e não funcionam no outro.
```

```sql
-- Comandos exclusivos

-- MySQL:

-- Mostra o status dos engines de armazenamento
SHOW ENGINE INNODB STATUS;

-- Retorna a quantidade de linhas de uma query sem LIMIT (requer uso específico)
SELECT SQL_CALC_FOUND_ROWS * FROM pedidos LIMIT 5;
-- SQL_CALC_FOUND_ROWS armazena o total de linhas que seriam retornadas sem o LIMIT
SELECT FOUND_ROWS();
```

```sql
-- Comandos exclusivos

-- PostgreSQL:
-- Usa RETURNING para obter valores de colunas após INSERT/UPDATE/DELETE
UPDATE pedidos
SET valor = valor * 1.1
WHERE cliente_id = 1
RETURNING id, valor;

-- Cria SCHEMA para organização (não existe como comando no MySQL)
CREATE SCHEMA financeiro;

-- Exemplo de função com linguagem procedural (PL/pgSQL)
CREATE FUNCTION saudacao(nome TEXT) RETURNS TEXT AS $$
BEGIN
  RETURN 'Olá, ' || nome || '!';
END;
$$ LANGUAGE plpgsql;

-- Chamando a função criada
SELECT saudacao('Gladimir');
```

```
-- Comandos exclusivos
-- Alguns comandos são específicos de cada SGBD e não funcionam no outro.


-- Observações:
-- Esses recursos mostram como cada banco oferece funcionalidades distintas.

-- PostgreSQL tem mais recursos para programação avançada e suporte a JSONB,
CTEs e funções de janela.

-- MySQL tende a ser mais simples e direto, com foco em performance de leitura e
facilidade de uso.

-- Conhecer os comandos exclusivos ajuda na migração entre bancos e na escolha
do SGBD ideal para cada projeto.
```

```sql
-- CTE vs Subconsulta - Introdução

-- MySQL e postgreSQL
/*
CTE (Common Table Expression) é uma subconsulta nomeada e reutilizável,
que melhora a legibilidade e organização de consultas complexas.
Disponível em PostgreSQL e MySQL 8+
*/
```

```sql
-- Subconsulta tradicional (sem CTE)

-- MySQL e PostgreSQL
SELECT clientes.nome,
       (SELECT COUNT(*)
        FROM pedidos
        WHERE pedidos.cliente_id = clientes.id) AS total_pedidos
FROM clientes;
```

```sql
-- A mesma consulta do slide anterior, usando CTE
-- MySQL 8+ e PostgreSQL

WITH total_pedidos AS (
  SELECT cliente_id, COUNT(*) AS total
  FROM pedidos
  GROUP BY cliente_id
)
SELECT clientes.nome, total_pedidos.total
FROM clientes
JOIN total_pedidos ON clientes.id = total_pedidos.cliente_id;
```

```sql
-- Várias CTEs juntas

-- MySQL 8+ e PostgreSQL
WITH total_pedidos AS (
  SELECT cliente_id, COUNT(*) AS total
  FROM pedidos
  GROUP BY cliente_id
),
valores_totais AS (
  SELECT cliente_id, SUM(valor) AS soma
  FROM pedidos
  GROUP BY cliente_id
)
SELECT c.nome, p.total, v.soma
FROM clientes c
JOIN total_pedidos p ON c.id = p.cliente_id
JOIN valores_totais v ON c.id = v.cliente_id;
```

```sql
-- CTE Recursiva (exemplo em hierarquia)

-- PostgreSQL
-- Exemplo de estrutura de cargos onde cada funcionário pode ter um superior
-- (auto-relacionamento)
CREATE TABLE funcionarios (
 id SERIAL PRIMARY KEY,
 nome TEXT,
 chefe_id INT REFERENCES funcionarios(id)
);

-- Inserindo dados com hierarquia
INSERT INTO funcionarios (nome, chefe_id) VALUES
('Angelo', NULL),  -- Angelo é o topo da hierarquia
('Bruna', 1),      -- Bruna é subordinada de Angelo
('Pablo', 2),      -- Pablo é subordinado de Bruna
('Gladimir', 2),   -- Gladimir também é subordinado de Bruna
('Wagner', 3);     -- Wagner é subordinado de Pablo
```

```sql
-- CTE Recursiva (exemplo em hierarquia)

-- PostgreSQL
-- Consulta hierárquica: encontra todos os subordinados de João
WITH RECURSIVE hierarquia AS (
  -- Caso base: seleciona João
  SELECT id, nome, chefe_id, 1 AS nivel
  FROM funcionarios
  WHERE nome = 'Angelo'

  UNION ALL

  -- Passo recursivo: pega os subordinados do último nível
  SELECT f.id, f.nome, f.chefe_id, h.nivel + 1
  FROM funcionarios f
  INNER JOIN hierarquia h ON f.chefe_id = h.id
)
SELECT * FROM hierarquia;

/* Observação: CTE recursiva é suportada apenas no PostgreSQL (e alguns outros
SGBDs avançados). */
```

```sql
-- Exercício 1 - INNER JOIN
-- Liste o nome dos clientes e os valores de seus pedidos.

-- veja a solução no próximo slide
```

```sql
-- Exercício 1 - INNER JOIN
-- Liste o nome dos clientes e os valores de seus pedidos.

-- MySQL e PostgreSQL
SELECT clientes.nome, pedidos.valor
FROM clientes
INNER JOIN pedidos ON clientes.id = pedidos.cliente_id;
```

```sql
-- Exercício 2
-- Liste todos os clientes, mesmo aqueles que não têm pedidos.

-- veja a solução no próximo slide
```

```sql
-- Exercício 2
-- Liste todos os clientes, mesmo aqueles que não têm pedidos.

-- MySQL e PostgreSQL
SELECT clientes.nome, pedidos.valor
FROM clientes
LEFT JOIN pedidos ON clientes.id = pedidos.cliente_id;
```

```sql
-- Exercício 3
-- Liste apenas os clientes que não realizaram nenhum pedido.

-- veja a solução no próximo slide
```

```sql
-- Exercício 3
-- Liste apenas os clientes que não realizaram nenhum pedido.

-- MySQL e PostgreSQL
SELECT clientes.nome
FROM clientes
LEFT JOIN pedidos ON clientes.id = pedidos.cliente_id
WHERE pedidos.id IS NULL;
```

```sql
-- Exercício 4
-- Liste o nome dos clientes e a quantidade de pedidos realizados por cada um.

-- veja a solução no próximo slide
```

```sql
-- Exercício 4
-- Liste o nome dos clientes e a quantidade de pedidos realizados por cada um.

-- MySQL e PostgreSQL
SELECT clientes.nome, COUNT(pedidos.id) AS total
FROM clientes
LEFT JOIN pedidos ON clientes.id = pedidos.cliente_id
GROUP BY clientes.nome;
```

```sql
-- Exercício 5
-- Liste os clientes que fizeram mais de um pedido.

-- veja a solução no próximo slide
```

```sql
-- Exercício 5
-- Liste os clientes que fizeram mais de um pedido.

-- MySQL e PostgreSQL
SELECT clientes.nome, COUNT(pedidos.id) AS total
FROM clientes
INNER JOIN pedidos ON clientes.id = pedidos.cliente_id
GROUP BY clientes.nome
HAVING COUNT(pedidos.id) > 1;
```

```sql
-- CTE - Common Table Expression
-- Calcular a quantidade de pedidos por cliente com melhor legibilidade

-- veja a solução no próximo slide
```

```sql
-- CTE - Common Table Expression
-- Calcular a quantidade de pedidos por cliente com melhor legibilidade

-- PostgreSQL e MySQL 8+
WITH total_pedidos AS (
  SELECT cliente_id, COUNT(*) AS total
  FROM pedidos
  GROUP BY cliente_id
)
SELECT clientes.nome, total_pedidos.total
FROM clientes
JOIN total_pedidos ON clientes.id = total_pedidos.cliente_id;
```

```sql
-- WINDOW FUNCTIONS - Funções de Janela
-- Mostra o ranking de valor de pedidos por cliente

-- PostgreSQL e MySQL 8+
SELECT cliente_id, valor,
       RANK() OVER (PARTITION BY cliente_id ORDER BY valor DESC) AS ranking
FROM pedidos;
```

```sql
-- PostgreSQL e MySQL 8+
-- Inserts adicionais

-- cliente_id 10 fará 4 pedidos (com 2 valores iguais)
INSERT INTO pedidos (id, cliente_id, valor) VALUES
  (100, 10, 200.00),
  (101, 10, 200.00),
  (102, 10, 150.00),
  (103, 10, 100.00);

-- cliente_id 11 fará 3 pedidos com valores diferentes
INSERT INTO pedidos (id, cliente_id, valor) VALUES
  (104, 11, 300.00),
  (105, 11, 200.00),
  (106, 11, 100.00);

-- cliente_id 12 fará 3 pedidos com todos os valores iguais
INSERT INTO pedidos (id, cliente_id, valor) VALUES
  (107, 12, 500.00),
  (108, 12, 500.00),
  (109, 12, 500.00);
```

```
-- PostgreSQL e MySQL 8+
Funções de Janela – Ranking em SQL

RANK():
  - Atribui a mesma posição para valores empatados.
  - Pula as posições seguintes em caso de empate.

DENSE_RANK():
  - Também atribui o mesmo ranking em empates.
  - Mas NÃO pula posições.

ROW_NUMBER():
  - Atribui um número único para cada linha, mesmo em empates.
  - Ordena sem considerar empates.

As três funções são úteis para gerar rankings, mas com comportamentos diferentes
*/
```

```sql
-- PostgreSQL e MySQL 8+
Funções de Janela – Ranking em SQL

-- RANK(): Mostra o ranking dos pedidos por cliente usando
SELECT cliente_id, valor,
       RANK() OVER (
         PARTITION BY cliente_id       -- Grupos por cliente
         ORDER BY valor DESC           -- Do maior para o menor valor
       ) AS ranking
FROM pedidos
WHERE cliente_id IN (10, 11, 12);
```

```sql
-- PostgreSQL e MySQL 8+
Funções de Janela – Ranking em SQL

-- DENSE_RANK: ranking contínuo mesmo em empates
SELECT cliente_id, valor,
       DENSE_RANK() OVER (
         PARTITION BY cliente_id
         ORDER BY valor DESC
       ) AS ranking
FROM pedidos
WHERE cliente_id IN (10, 11, 12);

-- ROW_NUMBER: numeração única por linha
SELECT cliente_id, valor,
       ROW_NUMBER() OVER (
         PARTITION BY cliente_id
         ORDER BY valor DESC
       ) AS ranking
FROM pedidos
WHERE cliente_id IN (10, 11, 12);
```

```sql
-- JSON e JSONB
-- Armazenando e consultando dados em formato JSON

-- PostgreSQL
CREATE TABLE produtos (
  id SERIAL PRIMARY KEY,          -- ID gerado automaticamente
  nome TEXT,                      -- Nome do produto
  atributos JSONB                 -- Campo que armazena um objeto JSON (formato binário
otimizado)
);

-- Inserindo um produto com atributos armazenados em JSONB
INSERT INTO produtos (nome, atributos)
VALUES (
  'Notebook',
  '{"ram": "16GB", "ssd": "512GB"}'  -- Objeto JSON com duas chaves: ram e ssd
);

-- Consulta no PostgreSQL
-- atributos->>'ram': acessa o valor da chave "ram" como texto (sem aspas)
SELECT atributos->>'ram' FROM produtos;
```

```sql
-- JSON e JSONB
-- Armazenando e consultando dados em formato JSON

-- MySQL
CREATE TABLE produtos (
  id INT AUTO_INCREMENT PRIMARY KEY,  -- ID autoincrementável
  nome VARCHAR(100),                  -- Nome do produto
  atributos JSON                      -- Campo JSON (texto em formato estruturado)
);

-- Inserindo dados no campo JSON
INSERT INTO produtos (nome, atributos)
VALUES (
  'Notebook',
  '{"ram": "16GB", "ssd": "512GB"}'
);

-- Consulta no MySQL
-- JSON_EXTRACT: extrai o valor do campo JSON usando caminho (path)
-- '$.ram' significa: chave "ram" na raiz do JSON
-- JSON_UNQUOTE: remove as aspas do resultado JSON
SELECT JSON_UNQUOTE(JSON_EXTRACT(atributos, '$.ram')) FROM produtos;
```

```
/* Desafio Avançado 1
Calcular o valor total dos pedidos feitos por cada cliente,
utilizando CTE para organizar a consulta em etapas.
*/
```

```sql
/* Desafio Avançado 1
Calcular o valor total dos pedidos feitos por cada cliente,
utilizando CTE para organizar a consulta em etapas.
*/

-- PostgreSQL e MySQL 8+
WITH soma_pedidos AS (
  SELECT cliente_id, SUM(valor) AS total
  FROM pedidos
  GROUP BY cliente_id
)
SELECT clientes.nome, soma_pedidos.total
FROM clientes
JOIN soma_pedidos ON clientes.id = soma_pedidos.cliente_id;
```

```
/* Desafio Avançado 2
Objetivo: Para cada cliente, mostrar seus pedidos ranqueados do maior para o menor valor.
Usa a função de janela RANK().
*/
```

```sql
/* Desafio Avançado 2
Objetivo: Para cada cliente, mostrar seus pedidos ranqueados do maior para o menor valor.
Usa a função de janela RANK().
*/

-- PostgreSQL e MySQL 8+
SELECT cliente_id, valor,
       RANK() OVER (PARTITION BY cliente_id ORDER BY valor DESC) AS ranking
FROM pedidos;
```

```
/* Desafio Avançado 3
Criar uma tabela de produtos com um campo JSON para armazenar
informações dinâmicas como RAM, SSD, cor, etc. Em seguida, consultar um campo específico.
*/
```

```sql
/* Desafio Avançado 3
Criar uma tabela de produtos com um campo JSON para armazenar
informações dinâmicas como RAM, SSD, cor, etc. Em seguida, consultar um campo específico.
*/

-- MySQL
CREATE TABLE produtos (
  id INT AUTO_INCREMENT PRIMARY KEY,
  nome VARCHAR(100),
  atributos JSON
);

INSERT INTO produtos (nome, atributos)
VALUES ('Notebook', '{"ram": "16GB", "ssd": "512GB", "cor": "cinza"}');

-- Consulta: pegar o valor da RAM
SELECT JSON_UNQUOTE(JSON_EXTRACT(atributos, '$.ram')) AS ram FROM produtos;
```

```sql
/* Desafio Avançado 3
Criar uma tabela de produtos com um campo JSON para armazenar
informações dinâmicas como RAM, SSD, cor, etc. Em seguida, consultar um campo específico.
*/

-- PostgreSQL
CREATE TABLE produtos (
  id SERIAL PRIMARY KEY,
  nome TEXT,
  atributos JSONB
);

INSERT INTO produtos (nome, atributos)
VALUES ('Notebook', '{"ram": "16GB", "ssd": "512GB", "cor": "cinza"}');

-- Consulta: pegar o valor da RAM
SELECT atributos->>'ram' AS ram FROM produtos;
```