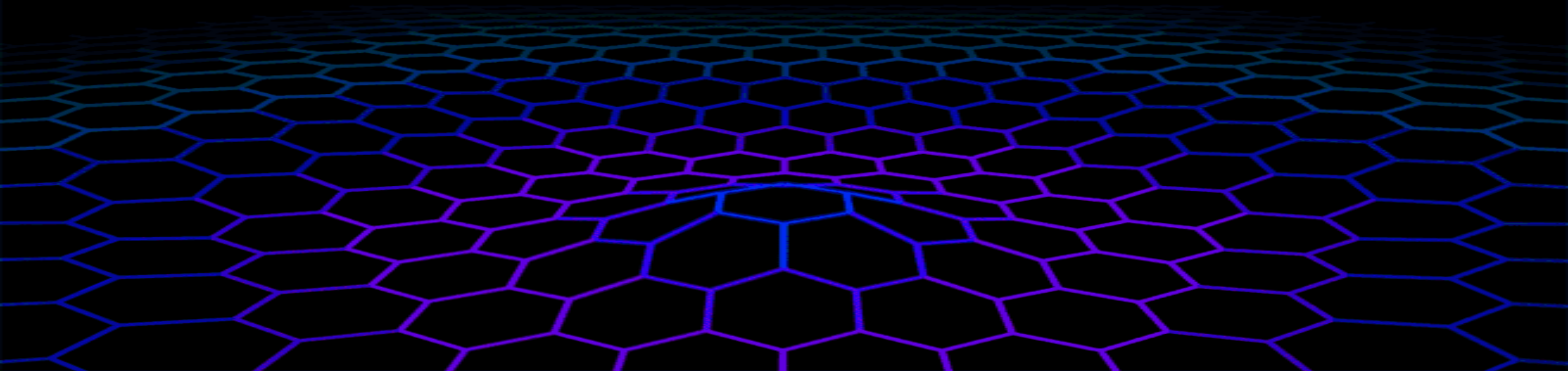
Unisenac
Campus Pelotas

Banco de Dados 2



Banco de Dados 2

```sql
-- Banco de dados para testes

-- PostgreSQL e MySQL
CREATE DATABASE aula04m;
-- Se for do turno da noite use aula04n

-- Selecionando o banco de dados

-- PostgreSQL
\c aula04m
-- Se for do turno da noite use aula04n

-- MySQL
USE aula04m;
-- Se for do turno da noite use aula04n
```

```sql
-- PostgreSQL
CREATE TABLE clientes (
  id    SERIAL,
  nome  VARCHAR(100),
  email VARCHAR(100),
  PRIMARY KEY(id)
);
```

```sql
-- MySQL
CREATE TABLE clientes (
  id    INT AUTO_INCREMENT,
  nome  VARCHAR(100),
  email VARCHAR(100),
  PRIMARY KEY(id)
);
```

```sql
-- PostgreSQL
CREATE TABLE clientes (
  id INT GENERATED ALWAYS AS IDENTITY,
  nome VARCHAR(100),
  email VARCHAR(100),
  PRIMARY KEY(id)
);
```

```sql
-- PostgreSQL
CREATE TABLE clientes (
  id INT GENERATED BY DEFAULT AS IDENTITY,
  nome VARCHAR(100),
  email VARCHAR(100),
  PRIMARY KEY(id)
);
```

```sql
-- PostgreSQL
CREATE TABLE pedidos (
  id          SERIAL,
  cliente_id  INT,
  valor       NUMERIC(10,2),
  data_pedido DATE DEFAULT CURRENT_DATE,
  PRIMARY KEY(id),
  FOREIGN KEY (cliente_id) REFERENCES clientes(id)
);

-- MySQL
CREATE TABLE pedidos (
  id          INT AUTO_INCREMENT,
  cliente_id  INT,
  valor       DECIMAL(10,2),
  data_pedido DATE,
  PRIMARY KEY (id),
  FOREIGN KEY (cliente_id) REFERENCES clientes(id)
);
```

```sql
-- PostgreSQL
CREATE TABLE log_pedidos (
  id        SERIAL,
  mensagem TEXT,
  data_log TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY(id)
);



-- MySQL
CREATE TABLE log_pedidos (
  id        INT AUTO_INCREMENT,
  mensagem TEXT,
  data_log TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY(id)
);
```

```sql
-- PostgreSQL e MySQL

-- Inserindo clientes
INSERT INTO clientes (nome, email) VALUES
  ('João Silva', 'joao@email.com'),
  ('Maria Souza', 'maria@email.com');

-- Inserindo pedidos
INSERT INTO pedidos (cliente_id, valor, data_pedido) VALUES
  (1, 199.90, '2025-03-01'),
  (1, 49.90, '2025-03-10'),
  (2, 99.90, '2025-03-15');
```

```
/*
O que são delimitadores?

Em PostgreSQL e MySQL, delimitadores são usados para
definir blocos de código.

PostgreSQL: recomenda-se usar $$ como delimitador

MySQL: usa-se DELIMITER para definir outro
delimitador temporário como $$ ou //
*/
```

```
/* O que é uma TRIGGER?

Uma TRIGGER (gatilho) é um conjunto de instruções SQL que é automaticamente
executado em resposta a certos eventos em uma tabela ou view.
Eventos possíveis:
    - BEFORE INSERT
    - AFTER INSERT
    - BEFORE UPDATE
    - AFTER UPDATE
    - BEFORE DELETE
    - AFTER DELETE

Triggers são úteis para:
    - Garantir integridade dos dados
    - Criar logs de alterações
    - Aplicar regras de negócio automáticas

Padrão de nomenclatura sugerido:   "trg_" + nome_tabela + código_evento
Ex:
- trg_pedidos_AI -> Trigger em pedidos, After Insert
- trg_clientes_BU -> Trigger em clientes, Before Update
*/
```

```
-- MySQL & PostgreSQL

/* Quando usamos TRIGGERS, podemos acessar os valores antes e depois da
operação (INSERT, UPDATE, DELETE):

OLD -> representa os dados antigos (antes da operação).
NEW -> representa os dados novos (após a operação).

| Tipo de Trigger      | OLD disponível | NEW disponível |
|----------------------|----------------|----------------|
| BEFORE INSERT        | ❌             | ✅             |
| AFTER INSERT         | ❌             | ✅             |
| BEFORE UPDATE        | ✅             | ✅             |
| AFTER UPDATE         | ✅             | ✅             |
| BEFORE DELETE        | ✅             | ❌             |
| AFTER DELETE         | ✅             | ❌             |
*/
```

```
-- Exemplos:

-- MySQL & PostgreSQL
-- Acessando valor novo (INSERT)
NEW.quantidade

-- MySQL & PostgreSQL
-- Comparando valor antigo com novo (UPDATE)
OLD.quantidade - NEW.quantidade

-- MySQL & PostgreSQL
-- Logando nome antes de excluir (DELETE)
OLD.nome

-- Atualização (UPDATE): comparar antigo e novo
-- PostgreSQL
OLD.valor || ' -> ' || NEW.valor

-- MySQL
CONCAT(OLD.valor, ' -> ', NEW.valor)
```

```sql
-- TRIGGER AFTER INSERT (MySQL)

DELIMITER $$

CREATE TRIGGER trg_pedidos_AI
AFTER INSERT ON pedidos
FOR EACH ROW
BEGIN
  INSERT INTO log_pedidos (mensagem)
  VALUES (
    CONCAT('Novo pedido inserido para o cliente ID ', NEW.cliente_id)
  );
END $$

DELIMITER ;
```

```sql
-- TRIGGER BEFORE DELETE (MySQL)

DELIMITER $$

CREATE TRIGGER trg_clientes_BD
BEFORE DELETE ON clientes
FOR EACH ROW
BEGIN
  INSERT INTO log_pedidos (mensagem)
  VALUES (CONCAT('Cliente ID ', OLD.id, ' será removido: ', OLD.nome));
END$$

DELIMITER ;
```

```sql
-- TRIGGER AFTER UPDATE (MySQL)

DELIMITER $$

CREATE TRIGGER trg_pedidos_AU
AFTER UPDATE ON pedidos
FOR EACH ROW
BEGIN
  INSERT INTO log_pedidos (mensagem)
  VALUES (
    CONCAT('Pedido ID ', OLD.id, ' atualizado. Valor anterior: ', OLD.valor, ', novo
valor: ', NEW.valor)
  );
END $$

DELIMITER ;
```

```
-- MySQL
/* A trigger já contém diretamente o que será
executado (BEGIN...END;) */

-- PostgreSQL
/* A trigger precisa de uma função externa que será
executada quando o evento ocorrer */
```

```sql
-- TRIGGER AFTER INSERT (PostgreSQL)

CREATE OR REPLACE FUNCTION log_pedido_insert()
RETURNS TRIGGER AS $$
BEGIN
  INSERT INTO log_pedidos (mensagem)
  VALUES ('Novo pedido inserido para o cliente ID ' || NEW.cliente_id);
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_pedidos_AI
AFTER INSERT ON pedidos
FOR EACH ROW
EXECUTE FUNCTION log_pedido_insert();
```

```sql
-- TRIGGER BEFORE DELETE (PostgreSQL)

CREATE OR REPLACE FUNCTION log_delete_cliente()
RETURNS TRIGGER AS $$
BEGIN
  INSERT INTO log_pedidos (mensagem)
  VALUES ('Cliente ID ' || OLD.id || ' será removido: ' || OLD.nome);
  RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_clientes_BD
BEFORE DELETE ON clientes
FOR EACH ROW
EXECUTE FUNCTION log_delete_cliente();
```

```sql
-- TRIGGER AFTER UPDATE (PostgreSQL)

CREATE OR REPLACE FUNCTION log_update_pedido()
RETURNS TRIGGER AS $$
BEGIN
  INSERT INTO log_pedidos (mensagem)
  VALUES (
    'Pedido ID ' || OLD.id || ' atualizado. Valor anterior: ' || OLD.valor || ', novo
valor: ' || NEW.valor
  );
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_pedidos_AU
AFTER UPDATE ON pedidos
FOR EACH ROW
EXECUTE FUNCTION log_update_pedido();
```

/* PROCEDURE e FUNCTION */

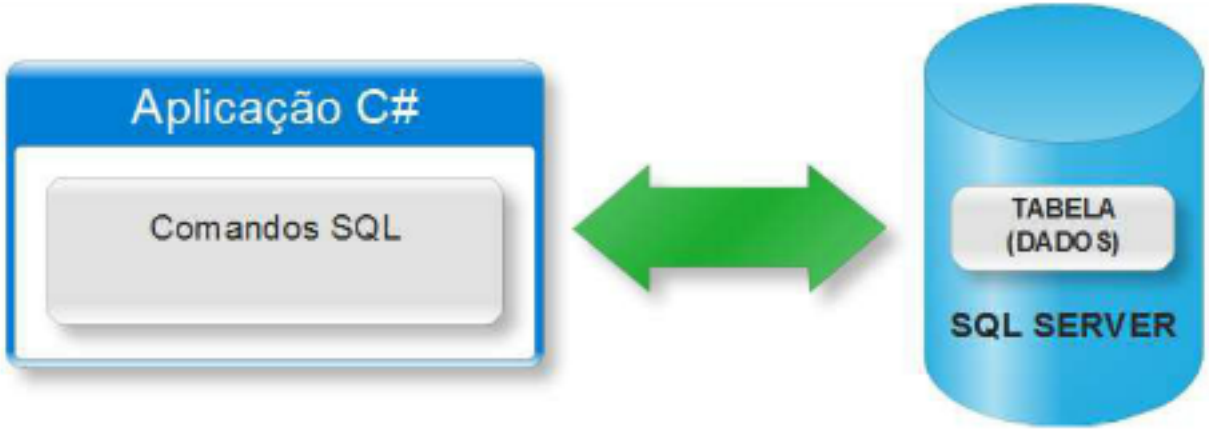
PROCEDURE:
- Bloco reutilizável de comandos SQL
- Pode receber parâmetros de entrada (e saída)
- Não retorna um valor diretamente (usa OUT/INOUT)

FUNCTION:
- Retorna um valor diretamente (ou uma tabela no PostgreSQL)
- Pode ser usada em SELECTs, WHERE, etc.
- Excelente para encapsular lógica de negócio

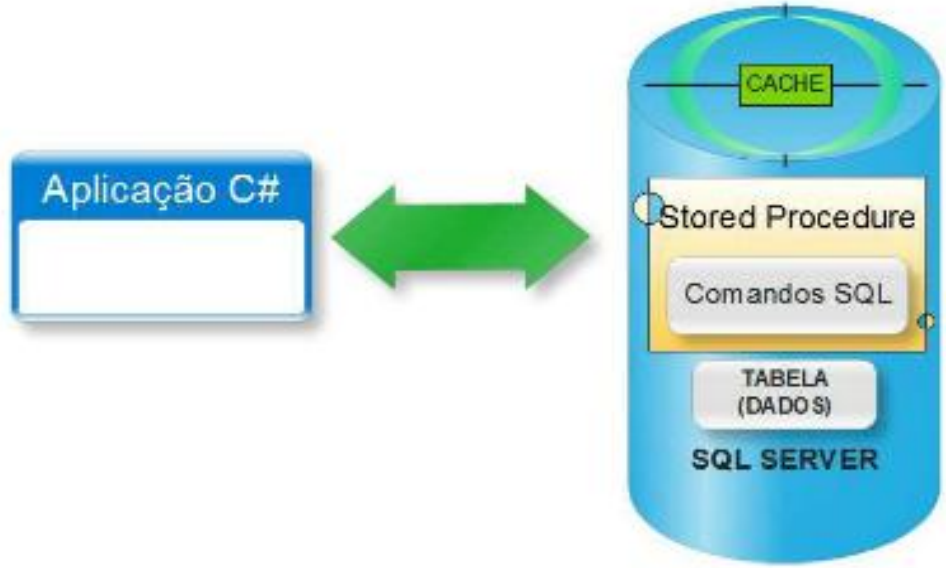
Diferenças:
- Função sempre retorna valor; procedure não.
- Função pode ser chamada dentro de SELECT; procedure não.
- Procedures são mais utilizadas para automação e rotinas administrativas.

/* PROCEDURE e FUNCTION */

Modelo de Acesso ao Banco de Dados sem utilização de Stored Procedures



Modelo de Acesso ao Banco de Dados utilizando Stored Procedures



```sql
-- PROCEDURE para inserir novo cliente (MySQL)

DELIMITER $$
CREATE PROCEDURE inserir_cliente(IN nome_cli VARCHAR(100), IN
email_cli VARCHAR(100))
BEGIN
  INSERT INTO clientes (nome, email)
  VALUES (nome_cli, email_cli);
END $$
DELIMITER ;



-- Chamando a procedure
CALL inserir_cliente('Carlos Mendes', 'carlos@email.com');
```

```sql
-- PROCEDURE para inserir novo cliente (PostgreSQL)

CREATE OR REPLACE PROCEDURE inserir_cliente(
  IN nome_cli TEXT,
  IN email_cli TEXT
)
LANGUAGE plpgsql
AS $$
BEGIN
  INSERT INTO clientes (nome, email)
  VALUES (nome_cli, email_cli);
END;
$$;


-- Chamando a procedure
CALL inserir_cliente('Carlos Mendes', 'carlos@email.com');
```

```sql
-- FUNÇÃO para verificar total de pedidos por cliente (MySQL)

DELIMITER $$
CREATE FUNCTION total_pedidos(id_cliente INT)
RETURNS INT
DETERMINISTIC
BEGIN
  DECLARE total INT;
  SELECT COUNT(*) INTO total
  FROM pedidos
  WHERE cliente_id = id_cliente;
  RETURN total;
END$$
DELIMITER ;


-- Chamando a função
SELECT total_pedidos(1);
```

```sql
-- FUNÇÃO para verificar total de pedidos por cliente (PostgreSQL)]

CREATE OR REPLACE FUNCTION total_pedidos(id_cliente INT)
RETURNS INT AS $$
DECLARE
  total INT;
BEGIN
  SELECT COUNT(*) INTO total
  FROM pedidos
  WHERE cliente_id = id_cliente;
  RETURN total;
END;
$$ LANGUAGE plpgsql;



-- Chamando a função
SELECT total_pedidos(1);
```

```sql
-- FUNÇÃO com retorno de múltiplas colunas
-- Retorna nome, email e total de pedidos de um cliente

-- PostgreSQL
CREATE OR REPLACE FUNCTION dados_cliente(id_cliente INT)
RETURNS TABLE(nome TEXT, email TEXT, total_pedidos INT) AS $$
BEGIN
  RETURN QUERY
  SELECT c.nome, c.email, COUNT(p.id)
  FROM clientes c
  LEFT JOIN pedidos p ON c.id = p.cliente_id
  WHERE c.id = id_cliente
  GROUP BY c.nome, c.email;
END;
$$ LANGUAGE plpgsql;

-- Chamando a função
SELECT * FROM dados_cliente(1);
```

```sql
-- PostgreSQL permite criar funções que retornam múltiplas colunas usando RETURN TABLE.
-- MySQL, por outro lado, não suporta funções com RETURN TABLE da mesma forma.
-- Para simular esse comportamento no MySQL, utiliza-se uma VIEW.

-- VIEW que retorna nome, email e total de pedidos por cliente
-- PostgreSQL e MySQL
CREATE VIEW dados_cliente_view AS
SELECT
  c.id,
  c.nome,
  c.email,
  COUNT(p.id) AS total_pedidos
FROM
  clientes c
LEFT JOIN
  pedidos p ON c.id = p.cliente_id
GROUP BY
  c.id, c.nome, c.email;

-- Consultando a VIEW
SELECT * FROM dados_cliente_view WHERE id = 1;
```

/* EVENTOS são tarefas agendadas para serem executadas automaticamente no banco de dados.

-- MySQL:
- Suporta eventos nativamente.
- Permite criar rotinas automatizadas diretamente no banco.
- Ex: apagar registros antigos, gerar backups, enviar notificações.

-- PostgreSQL:
- Não possui suporte nativo a eventos.
- Utiliza-se ferramentas do sistema operacional (como o CRON no Linux).

-- Vantagens:
- Automatização de tarefas de manutenção.
- Redução de carga administrativa.
- Maior segurança ao centralizar a lógica no banco.

```sql
-- EVENTOS NO MYSQL - Execução agendada

-- Habilitando agendador de eventos (se não estiver ativo)
SET GLOBAL event_scheduler = ON;

-- EVENTO DIÁRIO - EXECUTA NA HORA EM QUE FOI CRIADO
-- Evento que limpa pedidos antigos
DELIMITER $$
CREATE EVENT IF NOT EXISTS limpa_pedidos_antigos
ON SCHEDULE EVERY 1 DAY
DO
BEGIN
  DELETE FROM pedidos WHERE data_pedido < CURDATE() - INTERVAL 365 DAY;
END $$
DELIMITER ;

-- Este evento será executado a cada 24 horas, contando a partir do momento da sua criação.
-- Exemplo: Se for criado às 15h, ele será executado diariamente às 15h.
-- Útil para testes ou execuções simples sem necessidade de horário fixo.
```

```sql
-- EVENTO DIÁRIO EM HORÁRIO ESPECÍFICO

CREATE EVENT limpa_pedidos_antigos
ON SCHEDULE
  EVERY 1 DAY
  STARTS TIMESTAMP(CURDATE() + INTERVAL 2 HOUR) -- exemplo: todos os dias às 2h
DO
BEGIN
  DELETE FROM pedidos WHERE data_pedido < CURDATE() - INTERVAL 365 DAY;
END;

-- Este evento será executado todos os dias no horário definido com STARTS.
-- Exemplo: STARTS TIMESTAMP(CURDATE() + INTERVAL 2 HOUR) -> executa diariamente às 02h00 da manhã.
■ Útil para rotinas noturnas ou tarefas que devem rodar em horário de menor uso do sistema.
-- A sintaxe STARTS TIMESTAMP(...) define o horário do primeiro disparo.
-- O evento será repetido exatamente no mesmo horário nos dias seguintes.
```

```
/* VIEW (Visão)

Uma VIEW é uma "tabela virtual" baseada em uma consulta (SELECT).
Ela não armazena dados fisicamente, mas sim a lógica da consulta.

Sempre que você acessa a VIEW, o SGBD executa a consulta por trás dela.
Você pode tratar uma VIEW como uma tabela comum para fazer SELECT, JOINs, etc.

Principais vantagens:
--  Reutilização de consultas complexas
--  Organização e clareza no código
--  Restrição de acesso a dados sensíveis (ex: criar views que ocultam colunas)
--  Padronização de relatórios e análises
--  Abstração da estrutura real das tabelas

Empresas usam views para:
- Sumarizar dados (ex: total de vendas por cliente)
- Ocultar complexidade de joins e filtros
- Criar "relatórios" reutilizáveis
- Reduzir repetição de código
*/
```

```sql
-- Visão (VIEW) simples com JOIN

-- MySQL e PostgreSQL
CREATE VIEW resumo_pedidos AS
SELECT c.nome, COUNT(p.id) AS total_pedidos, SUM(p.valor) AS
valor_total
FROM clientes c
LEFT JOIN pedidos p ON c.id = p.cliente_id
GROUP BY c.nome;

-- Consultando a visão
SELECT * FROM resumo_pedidos;
```

```sql
-- View com filtro

CREATE VIEW clientes_com_pedidos AS
SELECT c.*
FROM clientes c
WHERE EXISTS (
  SELECT 1 FROM pedidos p WHERE p.cliente_id = c.id
);

SELECT * FROM clientes_com_pedidos;
```

```sql
-- ALTERAR e EXCLUIR PROCEDURES e VIEWS (MySQL)

-- EXCLUIR uma PROCEDURE
DROP PROCEDURE nome_da_procedure;

-- ALTERAR uma PROCEDURE
-- No MySQL, não é possível alterar diretamente o conteúdo de uma procedure.
-- É necessário excluir (DROP) e criar novamente (CREATE).
DROP PROCEDURE IF EXISTS inserir_cliente;
-- Em seguida, recrie a procedure com o novo conteúdo.


-- EXCLUIR uma VIEW
DROP VIEW nome_da_view;

-- ALTERAR uma VIEW
CREATE OR REPLACE VIEW resumo_pedidos AS
SELECT nome, COUNT(*) AS total
FROM clientes
GROUP BY nome;
```

```sql
-- EXCLUIR FUNÇÕES (PosgreSQL e MySQL)

-- PostgreSQL
DROP FUNCTION IF EXISTS total_pedidos(INT);
DROP FUNCTION IF EXISTS dados_cliente(INT); -- Pergunte ao Gladimir o motivo de precisar informar o parâmetro para excluir.

-- MySQL
DROP FUNCTION IF EXISTS total_pedidos;
```

```sql
-- CONSULTAS ÚTEIS PARA DEBUG (MySQL)

-- Exibe todas as procedures do banco de dados atual
SHOW PROCEDURE STATUS WHERE Db = 'aula04m';

-- Exibe o código SQL de uma procedure específica
SHOW CREATE PROCEDURE nome_da_procedure;

-- Exibe TRIGGERS de forma compacta (sem o código)
SELECT TRIGGER_NAME, EVENT_OBJECT_TABLE, EVENT_MANIPULATION, ACTION_TIMING
FROM INFORMATION_SCHEMA.TRIGGERS
WHERE TRIGGER_SCHEMA = 'aula04m';

-- Exibe TRIGGERS com o código (ACTION_STATEMENT)
SELECT TRIGGER_NAME, EVENT_OBJECT_TABLE, ACTION_STATEMENT,
EVENT_MANIPULATION, ACTION_TIMING
FROM INFORMATION_SCHEMA.TRIGGERS
WHERE TRIGGER_SCHEMA = 'aula04m';

-- Se for do turno da noite use aula04n
```

```sql
-- LISTANDO PROCEDURES (PostgreSQL)

SELECT proname, proargtypes, prosrc
FROM pg_proc
WHERE pronamespace IN (SELECT oid FROM pg_namespace WHERE nspname = 'public');
```

```
-- EXERCÍCIOS

/*
Pratique seus conhecimentos em:
- FUNÇÕES
- VIEWS
- EVENTOS
- TRIGGERS

Use os desafios abaixo como inspiração para praticar os comandos!

-- Crie:
1. Uma função que retorne nome, total de pedidos e valor total por cliente
   (No PostgreSQL, use RETURN TABLE. No MySQL, pode usar uma VIEW para simular.)

2. Uma VIEW que exiba nome do cliente e valor médio dos pedidos

3. Um EVENTO que execute semanalmente e remova pedidos com valor abaixo de R$ 10,00

4. Uma função que retorne nome, total de pedidos e a data do último pedido do
cliente */
```

```
-- EXERCÍCIOS

/*
Crie TRIGGERS para manter o controle automático do estoque de produtos com base nas
operações da tabela de vendas. As triggers devem:
 - AFTER INSERT em vendas: subtrair do estoque a quantidade vendida.
 - AFTER DELETE em vendas: retornar a quantidade ao estoque (caso uma venda seja
cancelada).
 - AFTER UPDATE em vendas: recalcular o estoque ajustando a diferença entre a nova e
a antiga quantidade vendida.


Veja script no slide seguinte */
```

```sql
-- EXERCÍCIOS

-- MySQL
-- Tabela de produtos
CREATE TABLE produtos (
  id      INT AUTO_INCREMENT,
  nome    VARCHAR(100),
  estoque INT,
  PRIMARY KEY(id)
);

-- Tabela de vendas
CREATE TABLE vendas (
  id         INT AUTO_INCREMENT,
  produto_id INT,
  quantidade INT,
  data_venda DATE,
  PRIMARY KEY(id),
  FOREIGN KEY (produto_id) REFERENCES produtos(id)
);
```

```sql
-- EXERCÍCIOS

-- Inserindo produtos
INSERT INTO produtos (nome, estoque) VALUES
('Mouse Gamer', 50),
('Teclado RGB', 30),
('Monitor 240Hz', 20);

-- Inserindo vendas
INSERT INTO vendas (produto_id, quantidade) VALUES
(1, 2),  -- Mouse Gamer
(2, 1);  -- Teclado RGB
```