

Views e Stored Procedures em SQL

DDL e DML Avançado

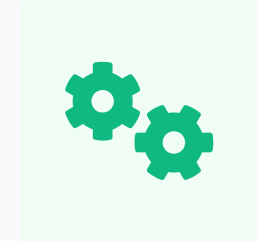
UC3 - Ciência de Dados

Aula 05

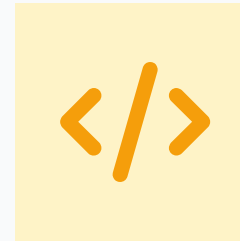
Automação e Otimização de Banco de Dados



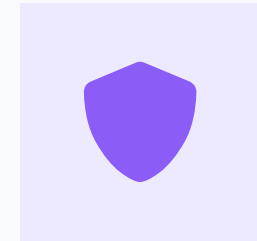
Views



Automação



Procedures



Segurança

"Um banco de dados complexo precisa de atalhos e automação"

Fundamentos de Views (Visões)

O que é uma View?

Uma **View (Visão)** é uma **tabela virtual** baseada no resultado de uma query (consulta) SQL. Ela não armazena dados fisicamente, mas apresenta dados de uma ou mais tabelas de forma simplificada e personalizada. Views funcionam como "janelas" ou "atalhos" para consultas complexas, permitindo que usuários acessem informações de maneira mais fácil e segura.

1. Simplificação

Simplifica consultas complexas, permitindo ao usuário buscar dados em **várias tabelas** usando apenas o nome da View. Em vez de escrever JOINS complexos repetidamente, basta usar `SELECT * FROM nome_da_view.`

2. Segurança

Restringe o acesso a **colunas sensíveis**. Você pode criar uma View que mostra os dados de clientes, mas **omite o CPF e o endereço**. Usuários veem apenas o que precisam, protegendo informações confidenciais.

3. Consistência

Garante que todos os usuários vejam o **mesmo conjunto de dados** (a mesma "janela") para uma análise específica. Regras de negócio e cálculos ficam centralizados na View, evitando inconsistências.

Sintaxe Básica (DDL): Criação de View

```
CREATE VIEW nome_da_view AS
SELECT coluna1, coluna2, coluna3
FROM tabela1
INNER JOIN tabela2 ON tabela1.id = tabela2.id_tabela1
WHERE condição;
```

Exemplo Prático: View para E-commerce

🕒 **Propósito da View:** Relatório dos **10 produtos mais vendidos no último mês**, mostrando nome do produto, preço, quantidade vendida e valor total de vendas (excluindo dados sensíveis dos clientes como CPF e endereço).

❌ ANTES: Query Base Complexa

```
-- Query complexa com múltiplos JOINS
SELECT
    p.id, p.nome_produto, p.preco,
    COUNT(ip.id_produto) AS qtd_vendida,
    SUM(ip.quantidade * ip.preco_unitario)
      AS valor_total
FROM Produto p
INNER JOIN ItemPedido ip
  ON p.id = ip.id_produto
INNER JOIN Pedido ped
  ON ip.id_pedido = ped.id
INNER JOIN Cliente c
  ON ped.id_cliente = c.id
WHERE
    ped.data_pedido >= DATE_SUB(
        CURRENT_DATE, INTERVAL 1 MONTH)
    AND ped.status = 'Concluído'
GROUP BY p.id, p.nome_produto, p.preco
ORDER BY qtd_vendida DESC
LIMIT 10;

-- Problema: Muito complexa!
-- Difícil de escrever e manter
```

✅ DEPOIS: View Simplificada

```
-- Criar a View (uma vez)
CREATE VIEW View_ProdutosMaisVendidos AS
SELECT
    p.id, p.nome_produto, p.preco,
    COUNT(ip.id_produto) AS qtd_vendida,
    SUM(ip.quantidade * ip.preco_unitario)
      AS valor_total
FROM Produto p
INNER JOIN ItemPedido ip
  ON p.id = ip.id_produto
INNER JOIN Pedido ped
  ON ip.id_pedido = ped.id
WHERE ped.status = 'Concluído'
GROUP BY p.id, p.nome_produto, p.preco
ORDER BY qtd_vendida DESC
LIMIT 10;

-- Usar a View (sempre)
SELECT * FROM View_ProdutosMaisVendidos;

-- Solução: Simples e reutilizável!
```


★ Benefícios da View

🔗 **Simplificação:** Query complexa se torna
SELECT
* FROM View_ProdutosMaisVendidos

🛡️ **Segurança:** Dados sensíveis dos clientes (CPF, endereço) não são expostos na View

🔄 **Consistência:** Todos os usuários veem os mesmos dados calculados da mesma forma

Atividade Prática 1 - Criação de Views (Parte 1)

 **Trabalho:** Em grupos |  **Ferramentas:** DBeaver, MySQL Workbench ou editor SQL online |  **Tempo:** 25 minutos

1 Definir o Propósito da View

O grupo deve escolher **qual relatório ou consulta** o cliente/gestor mais perguntaria. Pense em uma pergunta de negócio que seja **frequente e complexa**, envolvendo múltiplas tabelas. A View será criada para simplificar essa consulta.

Exemplo: E-commerce

"Quais foram os 10 produtos mais vendidos no último mês, com seus respectivos valores e nomes de clientes (excluindo dados sensíveis como CPF e endereço)?"

Exemplo: Biblioteca

"Quais são os livros atualmente atrasados, mostrando o nome do leitor, o título do livro, a data de empréstimo e a multa calculada?"

2 Escrever a Query Base (com JOINS)

Antes de criar a View, os alunos devem **escrever e testar a query SELECT completa**. Esta query provavelmente envolverá **múltiplos JOINS** (INNER JOIN, LEFT JOIN), **agregações** (COUNT, SUM, AVG), **filtros** (WHERE), e **ordenação** (ORDER BY).

Exemplo: E-commerce

```
SELECT p.nome_produto, COUNT(ip.id_produto) AS qtd_vendida,
SUM(ip.quantidade * ip.preco_unitario) AS valor_total FROM Produto p
INNER JOIN ItemPedido ip ON p.id = ip.id_produto INNER JOIN Pedido ped
ON ip.id_pedido = ped.id WHERE ped.data_pedido >=
DATE_SUB(CURRENT_DATE, INTERVAL 1 MONTH) GROUP BY p.id ORDER
BY qtd_vendida DESC LIMIT 10;
```

Exemplo: Biblioteca

```
SELECT l.titulo, le.nome, e.data_emprestimo, DATEDIFF(CURRENT_DATE,
e.data_devolucao_prevista) AS dias_atraso, (DATEDIFF(CURRENT_DATE,
e.data_devolucao_prevista) * 2.00) AS multa FROM Emprestimo e INNER
JOIN Livro l ON e.id_livro = l.id INNER JOIN Leitor le ON e.id_leitor = le.id
WHERE e.data_devolucao_real IS NULL AND e.data_devolucao_prevista <
CURRENT_DATE ORDER BY dias_atraso DESC;
```

Atividade Prática 1 - Criação de Views (Parte 2)

3 Criar a View (DDL)

Objetivo: Transformar a query testada em uma **View permanente** usando CREATE VIEW.

Instruções:

Use CREATE VIEW nome_da_view AS

Cole a query base após o AS

Escolha um nome descritivo

Exemplo: Criar View de Livros Atrasados

```
CREATE VIEW View_LivrosAtrasados AS
SELECT l.titulo, l.autor, le.nome AS nome_leitor,
       e.data_emprestimo, e.data_devolucao_prevista,
       DATEDIFF(CURRENT_DATE, e.data_devolucao_prevista)
       AS dias_atraso
FROM Emprestimo e
INNER JOIN Livro l ON e.id_livro = l.id
INNER JOIN Leitor le ON e.id_leitor = le.id
WHERE e.data_devolucao_real IS NULL
      AND e.data_devolucao_prevista < CURRENT_DATE
ORDER BY dias_atraso DESC;
```

4 Testar a View

Objetivo: Verificar se a View **simplifica a consulta complexa**.

Testes a realizar:

Execute SELECT * FROM nome_da_view;

Verifique se os dados estão corretos

Teste filtros adicionais

Exemplo: Testar a View Criada

```
SELECT * FROM View_LivrosAtrasados;

SELECT * FROM View_LivrosAtrasados
WHERE dias_atraso > 7;

SELECT COUNT(*) AS total_atrasados
FROM View_LivrosAtrasados;
```

💡 **Dica:** A View funciona como uma tabela normal em consultas SELECT, mas sempre busca os dados atualizados das tabelas base!

5 Documentar a View

Objetivo: Registrar informações importantes para **facilitar a manutenção futura**.

Informações a documentar:

Propósito: Para que serve?

Colunas omitidas: Dados excluídos

Benefícios: Como simplifica?

Exemplo: Documentação da View

```
-- =====
-- DOCUMENTAÇÃO DA VIEW
-- =====
-- Nome: View_LivrosAtrasados
-- Propósito: Acompanhar livros atrasados
-- Colunas omitidas: CPF e endereço
-- Benefícios:
-- 1. Simplificação: JOINS complexos
--    se tornam SELECT simples
-- 2. Segurança: Dados sensíveis
--    não são expostos
-- 3. Consistência: Cálculo sempre igual
-- =====
```

Fundamentos de Stored Procedures (Procedimentos Armazenados)

O que é uma Stored Procedure?

Uma **Stored Procedure (Procedimento Armazenado)** é um **bloco de código SQL (função)** pré-compilado e armazenado no servidor do banco de dados. Ela aceita **parâmetros**, executa uma série de instruções DML (INSERT, UPDATE, DELETE) e pode retornar resultados. Stored Procedures automatizam tarefas complexas e repetitivas, aumentando a eficiência e segurança do sistema.

1. Automação

Executam **tarefas repetitivas** automaticamente, como registrar uma transação complexa que envolve múltiplas tabelas. Em vez de executar vários comandos SQL manualmente, basta chamar a Stored Procedure com os parâmetros necessários.

2. Performance

Por serem **pré-compiladas**, são executadas mais rapidamente do que comandos SQL enviados individualmente. O servidor otimiza o plano de execução uma vez e o reutiliza, reduzindo o tempo de processamento.

3. Segurança

Permitem que os usuários executem **tarefas complexas sem terem acesso direto às tabelas**. Você pode conceder permissão para executar uma Stored Procedure sem dar permissão para INSERT, UPDATE ou DELETE direto nas tabelas.

Sintaxe Básica: Criação (DDL/DML)

```
DELIMITER $$

CREATE PROCEDURE nome_do_proc (
    IN parametro1 INT,
    IN parametro2 VARCHAR(100),
    OUT mensagem VARCHAR(200)
)
BEGIN
    -- Bloco de código SQL
    INSERT INTO tabela ...;
    UPDATE tabela ...;
    SET mensagem = 'Sucesso!';
END$$

DELIMITER ;
```

Sintaxe Básica: Execução

```
-- Executar a Stored Procedure
CALL nome_do_proc(valor1, valor2, @msg);

-- Ver o resultado do parâmetro OUT
SELECT @msg AS Resultado;

-- Exemplo prático:
CALL SP_RegistrarVenda(1, 5, @mensagem);
SELECT @mensagem;

-- Resultado: "Venda registrada!"
```

Exemplo Prático: SP para E-commerce

🎯 **Propósito da SP:** Automatizar o processo de **registro de venda**, incluindo validação de estoque, inserção de pedido e atualização de estoque.

⚠️ Problema Manual

Quando um cliente faz uma compra, precisamos executar **manualmente**:

- Verificar se há estoque suficiente
 - Inserir novo registro na tabela Pedido
 - Inserir itens na tabela ItemPedido
 - Atualizar estoque na tabela Produto
 - Garantir que TODAS as operações sejam concluídas ou NENHUMA (integridade)
- Fazer isso com vários comandos SQL é **trabalhoso e propenso a erros!**

✅ Solução Automatizada

Com uma **Stored Procedure**, todo o processo é automatizado:

- Um único comando** executa todas as operações
 - Validações automáticas** de estoque
 - Transações** garantem integridade
 - Segurança:** usuário não precisa acesso direto às tabelas
 - Performance:** código pré-compilado executa mais rápido
- Basta executar: `CALL SP_RegistrarVenda(...)`

Código da Stored Procedure: SP_RegistrarVenda

```
DELIMITER $$
CREATE PROCEDURE SP_RegistrarVenda(
    IN p_id_cliente INT, IN p_id_produto INT, IN p_quantidade INT,
    OUT p_mensagem VARCHAR(200)
) BEGIN
    DECLARE v_estoque INT; DECLARE v_preco DECIMAL(10,2); DECLARE v_id_pedido INT;
    -- Tratamento de erros
    DECLARE EXIT HANDLER FOR SQLEXCEPTION BEGIN ROLLBACK; SET p_mensagem = 'ERRO'; END;
    START TRANSACTION;
    -- PASSO 1: Verificar estoque e preço
    SELECT estoque, preco INTO v_estoque, v_preco FROM Produto WHERE id = p_id_produto;
    -- PASSO 2: Validar estoque
    IF v_estoque < p_quantidade THEN SET p_mensagem = 'Estoque insuficiente'; ROLLBACK;
    ELSE
        -- PASSO 3: Inserir pedido
        INSERT INTO Pedido (id_cliente, valor_total, status)
        VALUES (p_id_cliente, v_preco * p_quantidade, 'Pendente');
        SET v_id_pedido = LAST_INSERT_ID();
        -- PASSO 4: Inserir item do pedido
        INSERT INTO ItemPedido (id_pedido, id_produto, quantidade, preco_unitario)
        VALUES (v_id_pedido, p_id_produto, p_quantidade, v_preco);
        -- PASSO 5: Atualizar estoque
        UPDATE Produto SET estoque = estoque - p_quantidade WHERE id = p_id_produto;
        COMMIT;
        SET p_mensagem = CONCAT('SUCESSO: Pedido ', v_id_pedido);
    END IF;
END$$
DELIMITER ;
```

✅ Teste 1: Venda com Estoque Suficiente

```
CALL SP_RegistrarVenda(1, 5, 2, @msg); SELECT @msg AS Resultado;
```


→ **Resultado: "SUCESSO: Pedido 10"**

❌ Teste 2: Venda com Estoque Insuficiente

```
CALL SP_RegistrarVenda(1, 5, 1000, @msg); SELECT @msg AS Resultado;
```

→ **Resultado: "Estoque insuficiente"**

Atividade Prática 2 - Criação de SP (Parte 1)

 Trabalho: Em grupos |  Ferramentas: DBeaver, MySQL Workbench |  Tempo: 25 minutos

1 Estrutura Básica da SP

Toda Stored Procedure começa com **DELIMITER \$\$** para mudar o delimitador temporariamente, seguido por **CREATE PROCEDURE** com o nome do procedimento.

```
DELIMITER $$ CREATE PROCEDURE SP_NomeDoProcedimento() BEGIN -- Código SQL aqui END$$ DELIMITER ;
```

2 Declaração de Variáveis

Variáveis locais são declaradas dentro do bloco **BEGIN...END** usando **DECLARE**. Elas armazenam valores temporários durante a execução.

```
DECLARE v_estoque_atual INT; DECLARE v_preco DECIMAL(10,2); DECLARE v_mensagem VARCHAR(200); -- Prefixo 'v_' indica variável local
```

3 Parâmetros de Entrada (IN)

Parâmetros **IN** recebem valores do usuário quando a SP é chamada. São usados para passar informações necessárias para a execução.

```
CREATE PROCEDURE SP_RegistrarVenda( IN p_id_cliente INT, IN p_id_produto INT, IN p_quantidade INT ) -- Prefixo 'p_' indica parâmetro
```

4 Parâmetros de Saída (OUT)

Parâmetros **OUT** retornam valores para o usuário após a execução. São usados para enviar mensagens de sucesso/erro ou resultados.

```
CREATE PROCEDURE SP_RegistrarVenda( IN p_id_cliente INT, OUT p_mensagem VARCHAR(200) ) -- p_mensagem retornará resultado
```

Exemplo Completo: Estrutura Básica de uma Stored Procedure

```
DELIMITER $$
CREATE PROCEDURE SP_RegistrarVenda(
  -- Parâmetros de entrada
  IN p_id_cliente INT, IN p_id_produto INT, IN p_quantidade INT,
  -- Parâmetro de saída
  OUT p_mensagem VARCHAR(200)
) BEGIN
  -- Declaração de variáveis locais
  DECLARE v_estoque_atual INT;
  DECLARE v_preco_produto DECIMAL(10,2);
  DECLARE v_valor_total DECIMAL(10,2);
  -- Lógica da SP será implementada na Parte 2
  SET p_mensagem = 'Estrutura criada com sucesso!';
END$$
DELIMITER ;
```

Atividade Prática 2 - Criação de SP (Parte 2)

Implementação da Lógica DML (INSERT, UPDATE, DELETE)

Objetivo: Implementar a **lógica de negócio** dentro da Stored Procedure usando comandos DML.

Passos:

SELECT INTO: Buscar dados e armazenar em variáveis

INSERT/UPDATE/DELETE: Modificar registros

IF...THEN...ELSE: Adicionar validações

Exemplo: Lógica DML para Registrar Venda

```
SELECT estoque, preco INTO v_estoque_atual, v_preco_produto
FROM Produto WHERE id = p_id_produto;
IF v_estoque_atual < p_quantidade THEN
    SET p_mensagem = 'Estoque insuficiente';
ELSE
    INSERT INTO Pedido (id_cliente, valor_total)
    VALUES (p_id_cliente, v_preco_produto * p_quantidade);
    UPDATE Produto SET estoque = estoque - p_quantidade
    WHERE id = p_id_produto;
    SET p_mensagem = 'Venda registrada!';
END IF;
```



Uso de Transações (START TRANSACTION, COMMIT, ROLLBACK)

Objetivo: Garantir a **integridade dos dados** usando transações. Todas as operações devem ser concluídas ou nenhuma.

Conceitos:

START TRANSACTION: Inicia transação

COMMIT: Confirma operações

ROLLBACK: Desfaz em caso de erro

Exemplo: Uso de Transações

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
    ROLLBACK; SET p_mensagem = 'ERRO: Transação cancelada';
END;
START TRANSACTION;
INSERT INTO Pedido ...; UPDATE Produto ...;
COMMIT; SET p_mensagem = 'SUCESSO!';
```



Testes da Stored Procedure

Objetivo: Testar a Stored Procedure com **diferentes cenários** (sucesso e erro).

Cenários de teste:

Teste 1: Dados válidos (deve ter sucesso)

Teste 2: Dados inválidos (deve retornar erro)

Teste 3: Verificar se tabelas foram atualizadas

Exemplo: Testes da SP

```
CALL SP_RegistrarVenda(1, 5, 2, @msg);
SELECT @msg AS Resultado; -- "SUCESSO!"
CALL SP_RegistrarVenda(1, 5, 1000, @msg);
SELECT @msg AS Resultado; -- "Estoque insuficiente"
SELECT estoque FROM Produto WHERE id = 5;
-- Estoque deve ter diminuído
```

Validação Cruzada e Segurança

Validação Cruzada entre Grupos

Instruções para validação:

Trocar scripts: Cada grupo compartilha seus scripts de Views e Stored Procedures com outro grupo

Executar e testar: O grupo revisor executa os scripts no banco de dados e testa com diferentes cenários

Verificar documentação: Conferir se a documentação está clara e completa (propósito, parâmetros, benefícios)

Dar feedback: Apontar pontos fortes e sugestões de melhoria (validações, tratamento de erros, segurança)

Discussão em sala: Compartilhar os aprendizados e melhores práticas identificadas



Como Views e Stored Procedures Aumentam a Segurança

Síntese e Consolidação

Comparação: Views vs Stored Procedures

Views (Visões)

Propósito: Simplificar consultas complexas (SELECT) e restringir acesso a colunas sensíveis

Tipo de operação: Somente leitura (SELECT). Não modifica dados

Armazenamento: Não armazena dados fisicamente, apenas a definição da query

Uso típico: Relatórios, dashboards, consultas frequentes com múltiplos JOINs

Exemplo: `SELECT * FROM View_LivrosAtrasados;`

Stored Procedures

Propósito: Automatizar tarefas complexas que envolvem múltiplas operações DML

Tipo de operação: Leitura e escrita (SELECT, INSERT, UPDATE, DELETE)

Armazenamento: Armazena o código SQL pré-compilado no servidor

Uso típico: Transações complexas, validações de negócio, automação de processos

Exemplo: `CALL SP_RegistrarVenda(1, 5, 2, @msg);`

Checklist de Validação: Você aprendeu?

- ✓ Criar Views para simplificar consultas complexas com múltiplos JOINs
- ✓ Criar Stored Procedures com parâmetros IN e OUT
- ✓ Usar transações (START TRANSACTION, COMMIT, ROLLBACK) para garantir integridade
- ✓ Documentar Views e SPs para facilitar manutenção futura
- ✓ Usar Views para restringir acesso a colunas sensíveis (segurança)
- ✓ Implementar lógica DML (INSERT, UPDATE, DELETE) dentro de SPs
- ✓ Testar Views e SPs com diferentes cenários (sucesso e erro)
- ✓ Entender quando usar Views vs Stored Procedures

"Um banco de dados bem estruturado é mais fácil de manter e mais seguro"