# 03_teoria

## QUESTÃO 10 - VIEWS

Em MySQL, uma "view" é uma **representação virtual** de uma tabela baseada no resultado de uma consulta SQL. Ela **não armazena dados fisicamente**, mas fornece uma maneira conveniente de estruturar e acessar dados de maneira específica.

As views são úteis para **simplificar consultas complexas**, ocultar detalhes da implementação do banco de dados e fornecer uma camada de **abstração para os usuários finais**.

Obs.: A criação de uma view envolve a definição de uma consulta SQL que extrai os dados desejados de uma ou mais tabelas existentes no banco de dados.

A view **pode ser referenciada** em consultas subsequentes como se fosse uma tabela real. Quando uma view é consultada, o sistema executa a consulta subjacente e retorna os resultados como se fossem provenientes de uma tabela física.

Para criar uma view no MySQL, é possível usar um único select para ter acesso a dados de diversas tabelas.

Segue abaixo o exemplo da “view” que implementamos no banco:

```
-- CRIAÇÃO
CREATE VIEW venda_detalhada AS
SELECT
    c.id AS compra_id,
    c.data_hora AS data_hora_compra,
    f.nome AS farmacia,
    cl.nome AS cliente,
    p.nome AS produto,
    pc.quantidade AS quantidade_comprada,
    pc.preco_unico AS preco_unitario,
    (pc.quantidade * pc.preco_unico) AS total
FROM
    Compras c
JOIN
    Farmacias f ON c.farmacia_id = f.id
JOIN
    Clientes cl ON c.cliente_id = cl.id
JOIN
    Produtos_comprados pc ON c.id = pc.compra_id
```

```
JOIN
    Produtos p ON pc.produto_id = p.id;


SELECT * FROM venda_detalhada; -- VISUALIZAR
```

Essa view dá um panorama geral da venda: mostrando a data e hora que foi feito a compra, em qual farmácia, por qual cliente, quais os produtos e suas respectivas quantidades e o preço total a parte de cada produto(quantidade comprada * valor do produto) simplificando o acesso aos dados e melhorando a legibilidade do código.

# QUESTÃO 11 - FUNCTIONS

Functions podem ser usadas para otimizar o desempenho. Elas podem **armazenar resultados intermediários**, reduzindo a necessidade de cálculos repetitivos.

Algumas observações:

**Encapsulação de lógica de negócios:**

- Funções permitem encapsular lógicas de negócios em um único bloco de código
- a função que criamos, `calcularTotalGasto`**,** encapsula a lógica de calcular o total de gastos com base em um ID de compra.
- a complexidade dessa lógica é isolada

**Manutenibilidade:**

- Centralização da lógica relacionada em locais específicos
- Se a lógica de cálculo do total de gastos precisar ser ajustada, você só modifica a função

```
--- COMANDOS PARA CRIAÇÃO E EXEMPLO DE USO DA FUNÇÃO ---

-- Desativa a verificação de segurança para criação de funções
SET GLOBAL log_bin_trust_function_creators = 1;
DELIMITER //
CREATE FUNCTION calcularTotalGasto(compraID INT)
RETURNS DECIMAL(10, 2)
BEGIN
    DECLARE total DECIMAL(10, 2);

    SELECT SUM(quantidade * preco_unico)
    INTO total
    FROM Produtos_comprados
```

```
    WHERE compra_id = compraID;

    RETURN total;
END //
DELIMITER ;
SET GLOBAL log_bin_trust_function_creators = 0;

/*SELECT calcularTotalGasto(1) AS total_gasto;*/

-- Cria uma tabela temporária para armazenar os resultados
CREATE TEMPORARY TABLE ResultadosTotais (
    compra_id INT,
    total_gasto DECIMAL(10, 2)
);

-- Preenche a tabela temporária usando a função
INSERT INTO ResultadosTotais (compra_id, total_gasto)
SELECT id, calcularTotalGasto(id) AS total_gasto
FROM Compras;

-- Exibe os resultados
SELECT * FROM ResultadosTotais;

-- Limpa a tabela temporária
DROP TEMPORARY TABLE IF EXISTS ResultadosTotais;
```

O objetivo da função `calcularTotalGasto` que criamos é fornecer o valor total de gastos em uma compra específica, com base na quantidade de produtos comprados e nos seus respectivos preços,  dessa forma podemos utilizá-la para criar uma tabela e adicionar o valor total de gastos em todas as compras tendo uma maior visualização, como assim fizemos.

Esse tipo de função também é útil em situações em que você precisa calcular o total de gastos associado a uma compra em diferentes partes do seu código. Promovendo a reutilização do código deixando a estrutura mais modulas e organizada.

# QUESTÃO 12 - STORED PROCEDURE

Código dentro do banco que foi usado como exemplo:

```
DELIMITER //
CREATE PROCEDURE ObterQuantidadeProdutoNaFarmaciass(
    IN farmacia_id_param INT,
    IN produto_id_param INT
)
BEGIN
```

```
    DECLARE quantidade_produto INT;
    DECLARE nome_produto VARCHAR(100);
    DECLARE nome_farmacia VARCHAR(100);

    SELECT nome INTO nome_farmacia
    FROM Farmacias
    WHERE id = farmacia_id_param;

    SELECT nome INTO nome_produto
    FROM Produtos
    WHERE id = produto_id_param;

    SELECT COALESCE(SUM(Estoque.quantidade), 0) INTO quantidade_produto
    FROM Estoque
    INNER JOIN EstoqueUnidade eu ON Estoque.id = eu.estoque_id
    INNER JOIN Unidades u ON eu.unidade_id = u.id
    WHERE u.farmacia_id = farmacia_id_param AND Estoque.produto_id = produto_id_param;

    SELECT nome_farmacia AS 'Nome da Farmácia', nome_produto AS 'Nome do Produto',
    quantidade_produto AS 'Quantidade Total do Produto no Estoque';
END //
DELIMITER ;
CALL ObterQuantidadeProdutoNaFarmaciass(1, 2);
```

## Propósito dessa Stored Procedure e como ela pode ajudar

A Stored Procedure `ObterQuantidadeProdutosNaFarmaciaa` tem como propósito calcular e retornar a quantidade total de um produto específico no estoque de uma farmácia específica. Ela recebe 2 parâmetros ( `farmacia_id_param` , `produto_id_param` ) que indica qual farmácia deve ser considerada no cálculo e qual o produto .

O propósito dos Stored Procedures em geral é encapsular e agrupar um conjunto de instruções SQL para realizar uma tarefa específica. No exemplo apresentado, ela utiliza um SELECT para calcular a quantidade total de produtos no estoque, com base nas tabelas relacionadas ( `Estoque` , `EstoqueUnidade` , `Unidades` ). O resultado é armazenado em uma variável ( `total_produtos` ) e, em seguida, é feito uma segunda SELECT para retornar essa quantidade total como resultado da execução do procedimento armazenado.

Desta forma, com a Stored Procedure a organização e reutilização lógica do banco de dados fica com uma maior eficiência e entendimento, além de deixa mais compacto devido a modularização, e facilitar a manutenção do código.

# QUESTAO 13- TRIGGERS

## Pesquisa sobre Triggers

```
Triggers também chamado de"gatilho"
- Associado a uma tabela.
- procedimento invocado quando um comando DML é executado.
DML - (inserção, atualização, exclusão)

Usos do trigger:
- Verificação de integridade dos dados
- Validação dos dados I
- rastreamento e registro de logs de atividades nas tabelas.
- Arquivamento de registros excluídos
- Um Trigger é associado a uma tabela
- Armazenado no BD como um arquivo separado.
- Não são chamados diretamente, são invocados automaticamente

SINTAXE dos Triggers:
CREATE TRIGGER nome timing operaão
ON tabela
FOR EACH ROW
declarações
timing = BEFORE | AFTER
operação = INSERT | UPDATE | DELETE

Explicação dos tipos:
  BEFORE -> Executados antes da operação que acionou o evento,
usados para validar ou modificar dados antes que sejam gravados na tabela.

  AFTER -> Executados após a conclusão da operação que acionou o evento,
usado para realizar ações automáticas pós-gravação.

Explicação dos eventos que são acionados:
INSERT -> após a inserção de novos dados em uma tabela.
UPDATE -> após a atualização de dados em uma tabela.
DELETE -> após a exclusão de dados de uma tabela.

Nível de Declaração:
Row -> Acionados uma vez para cada linha afetada pelo evento,
Podem acessar e modificar os dados da linha que o acionou.
Statement -> Acionados uma vez para cada operação, não têm acesso
direto aos dados da linha.
```

## Explicação como a trigger funciona e como ela pode ser usada

```
DELIMITER //
CREATE TRIGGER before_update_Produtos_comprados
BEFORE UPDATE ON Produtos_comprados
FOR EACH ROW
BEGIN
```

```
    -- Calcula o desconto de 15% no novo preço unitário
    -- e fornece o valor do produto com o desconto
    SET NEW.desconto_preco_unico = NEW.preco_unico * 0.85;
END;
//
DELIMITER ;
```

A trigger criada ela funciona da seguinte forma, toda vez que os dados da tabela produtos for atualizado, antes de realizar a atualização a trigger vai ser acionada e irá fornecer o preço do produto com um desconto de 15% em cima do preço único dele, guardando esse valor no campo desconto_preco_unico que se encontra na tabela produtos_comprados. Podemos usar essa trigger para automatizar ações , como por exemplo em um evento de black friday onde todos os produtos vão ser atualizados no banco e receberam 15% de desconto em cima do preco_unico deles.