**Apostila: O Caminho para o Profissional Raro em SQL**

* **Introdução**
  + Objetivo do Curso: O Analista "Raro"
  + Nosso Projeto: mercado\_cafe\_db
  + Nossas Ferramentas: MySQL e VS Code
* **Módulo 0: Preparando o Laboratório (Setup)**
  + Arquitetura: MySQL (Motor) vs. VS Code (Interface)
  + Comandos de Inspeção: SHOW TABLES, DESCRIBE, SELECT \*
  + A Lição Crítica: A importância do USE nome\_do\_banco;
* **Módulo 1: A Fundação (O SQL Básico - DQL)**
  + Os 3 Tipos de SQL: DDL, DML e DQL
  + DDL: CREATE TABLE (Nossas tabelas Fazendas, Lotes\_Producao, Vendas)
  + DML: INSERT INTO (Populando o banco)
  + DQL: SELECT e FROM (Pedindo dados)
  + Filtragem: WHERE (Texto vs. Números)
  + Ordenação: ORDER BY (ASC vs. DESC)
  + Filtros Avançados: AND, OR, IN, BETWEEN, LIKE
  + Lidando com Nulos: IS NULL vs. IS NOT NULL
  + Apelidos: AS (Para Colunas e Tabelas)
  + Desafio do Módulo 1 (Revisão de DQL básico)
* **Módulo 2: O Poder da Agregação (Resumindo Dados)**
  + O "Quinteto Fantástico": COUNT, SUM, AVG, MIN, MAX
  + Valores Únicos: DISTINCT
  + Agrupamento: GROUP BY (A lógica dos "baldes")
  + Filtrando Grupos: HAVING
  + A Diferença Crucial: Ordem de Execução (WHERE vs. HAVING)
  + Desafio do Módulo 2 (Combinando WHERE, GROUP BY, HAVING)
* **Módulo 3: Conectando Mundos (JOINs e UNION)**
  + Conceito: Chave Primária (PK) vs. Chave Estrangeira (FK)
  + Teoria: O Diagrama de Venn (INNER vs. LEFT)
  + INNER JOIN: Juntando 2 e 3 tabelas (Seguindo o "mapa")
  + LEFT JOIN: Garantindo todos os dados da tabela principal
  + RIGHT JOIN / FULL OUTER JOIN (Conceitos e Melhores Práticas)
  + UNION vs. UNION ALL: Empilhando resultados
  + Desafio do Módulo 3 (Combinando JOIN e GROUP BY)
* **Módulo 4: Transformação e Lógica Avançada**
  + Funções de Texto (A "Faxina"): UPPER, LOWER, TRIM, REPLACE, CONCAT, LEFT, RIGHT
  + Funções de Data (Análise Temporal): YEAR, MONTH, DATEDIFF, DATE\_ADD
  + Lógica Condicional: CASE WHEN ... THEN ... ELSE ... END (Criando categorias)
  + Subconsultas (Subqueries):
    - Uso no WHERE (Filtros baseados em agregações)
    - Uso no FROM (Agregação de agregação - "análise em duas etapas")
  + CTEs (Common Table Expressions): A cláusula WITH
  + Vantagens: CTEs vs. Subconsultas (Legibilidade, o "módulo" do SQL)
  + Prática: Múltiplas CTEs encadeadas
* **Módulo 5: O Nível Raro (Window Functions)**
  + Conceito: Agregação sem colapsar linhas (A cláusula OVER())
  + Particionamento: PARTITION BY (A "janela" por subgrupo)
  + Funções de Ranking: ROW\_NUMBER, RANK, DENSE\_RANK
  + Ranking de Distribuição: NTILE (Quartis)
  + Desafio Raro: O "Top N por Grupo" (CTE + PARTITION BY + ROW\_NUMBER)
  + Funções de Navegação: LAG (olhar para trás) e LEAD (olhar para frente)
  + Prática: Cálculo de Variação Percentual (com LAG)
  + Janelas Móveis (Definindo o Frame):
    - Somas Acumuladas (ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
    - Médias Móveis (ROWS BETWEEN N PRECEDING AND CURRENT ROW)
* **Módulo 6: O Ambiente Moderno (Nuvem e JSON)**
  + Conceitos: Data Warehouse (DW) vs. Data Lake
  + Arquitetura de Nuvem: BigQuery, Snowflake, Redshift (Separação de Armazenamento/Computação)
  + Dialetos SQL (Ex: ROLLUP vs. WITH ROLLUP no MySQL)
  + Otimização na Nuvem: Armazenamento Colunar (O perigo do SELECT \*) e Particionamento
  + Consultando JSON (Dados Semi-Estruturados):
    - Extraindo valores (->> '$.chave.aninhada')
    - Acessando Arrays ([indice])
  + "Desempacotando" Arrays: A lógica do JSON\_TABLE e PATH
  + Prática: JOIN entre dados JSON e tabelas relacionais
* **Módulo 7: Manipulação de Dados (CRUD)**
  + Revisão da Sintaxe: INSERT INTO (Create)
  + Revisão da Sintaxe: SELECT (Read)
  + Sintaxe Segura: UPDATE ... SET ... WHERE (Update)
  + Sintaxe Segura: DELETE FROM ... WHERE (Delete)
* **Conclusão**

**O Caminho para o Profissional Raro em SQL**

**Introdução**

Bem-vindo a este curso. O objetivo aqui não é apenas aprender a sintaxe do SQL, mas aprender a *pensar* como um Analista de Dados "Raro". Um profissional raro não apenas busca dados, mas entende a pergunta de negócio, sabe como limpar, transformar e conectar informações de múltiplas fontes para gerar insights que realmente agregam valor.

Nosso Projeto:

Ao longo deste curso, construiremos e analisaremos um banco de dados completo para um estudo de caso do agronegócio: mercado\_cafe\_db. Atuaremos como o principal analista de dados de um grupo de fazendas de café, respondendo a perguntas reais de gerentes de operação, qualidade, finanças e marketing.

**Nossas Ferramentas:**

* **Banco de Dados:** MySQL (um dos bancos relacionais mais populares do mundo).
* **Editor de Código:** Visual Studio Code (VS Code) com extensões de SQL, o ambiente de trabalho de um desenvolvedor moderno.

Prepare-se para ir do zero ao nível analítico avançado.

**Módulo 0: Preparando o Laboratório (Setup)**

Antes de analisar, precisamos preparar nosso ambiente. Este módulo garante que seu laboratório de dados esteja pronto.

**O que é MySQL vs. VS Code?**

* **MySQL:** É o **Servidor de Banco de Dados (SGBD)**. Pense nele como o motor de um carro ou o sistema de arquivos que armazena, organiza e protege seus dados. Ele funciona nos bastidores.
* **VS Code (com Extensões SQL):** É a sua **Interface Gráfica (GUI)** ou "painel do carro". É o editor de texto onde você escreve suas consultas (seus comandos) e as envia para o motor (MySQL) executar. Ele também exibe os resultados que o motor envia de volta.

**Comandos Essenciais de Verificação**

Existem três comandos que você usará o tempo todo para "inspecionar" seu banco de dados. É crucial não os confundir.

| **Comando** | **Pergunta que Responde** | **O que ele mostra?** | **Analogia** |
| --- | --- | --- | --- |
| **SHOW TABLES;** | "Quais tabelas existem neste banco?" | Uma lista de nomes de tabelas. | "O Mapa da Vizinhança" |
| **DESCRIBE nome\_da\_tabela;** | "Qual a estrutura desta tabela?" | As colunas e seus tipos de dados. | "A Planta Baixa da Casa" |
| **SELECT \* FROM nome\_da\_tabela;** | "O que há *dentro* desta tabela?" | Os dados reais (linhas/registros). | "Olhar os Móveis Dentro da Casa" |

**A Lição Mais Importante do Setup: USE**

Ao se conectar a um servidor MySQL, você pode ter acesso a vários bancos de dados (incluindo os do sistema, como mysql ou information\_schema).

**O Comando:**

SQL

USE mercado\_cafe\_db;

* **Para que serve:** Este comando diz ao SQL: "Todos os comandos que eu digitar a partir de agora devem ser executados **dentro** deste banco de dados específico".
* **Por que é crucial?** Sem ele, você pode acidentalmente executar seus comandos no banco de dados errado.
* **Lição Aprendida (Nossa 1ª Correção):** Descobrimos que, sem o USE, nossas tabelas Fazendas e Lotes\_Producao foram criadas no banco de dados do sistema, misturadas com tabelas internas do MySQL. Tivemos que usar DROP TABLE para apagá-las e recriá-las no lugar certo após usar USE mercado\_cafe\_db;.
* **Melhor Prática:** **Sempre** coloque USE nome\_do\_banco; no topo de cada novo arquivo .sql que você salvar. Isso torna seu script seguro, portátil e claro.

**Módulo 1: A Fundação (O SQL Básico)**

Com nosso laboratório pronto, começamos a fundação. Este módulo foca em criar nossas estruturas de dados e aprender a fazer as consultas mais fundamentais: ler, filtrar e ordenar.

**1.1 As "Caixas de Ferramentas" do SQL (DDL, DML, DQL)**

O SQL é dividido em "sub-linguagens" para diferentes tipos de tarefas:

1. **DDL (Data Definition Language):** Linguagem de **Definição** de Dados.
   * **Função:** O "Arquiteto". Usado para **criar e modificar a estrutura** do banco.
   * **Comandos:** CREATE TABLE, ALTER TABLE, DROP TABLE.
2. **DML (Data Manipulation Language):** Linguagem de **Manipulação** de Dados.
   * **Função:** O "Operador de Estoque". Usado para **manipular os dados *dentro* das tabelas**.
   * **Comandos:** INSERT INTO, UPDATE, DELETE.
3. **DQL (Data Query Language):** Linguagem de **Consulta** de Dados.
   * **Função:** O "Analista". Usado para **ler e perguntar** coisas aos dados.
   * **Comando:** SELECT.
   * **Nosso Foco:** Como analistas, 95% do nosso tempo será gasto dominando o SELECT.

**1.2 Criando Nossas Tabelas (DDL)**

Nossa primeira tarefa de DDL foi desenhar o "mapa" do nosso banco mercado\_cafe\_db. Decidimos criar três tabelas centrais:

**1. Fazendas (Quem produz):**

SQL

CREATE TABLE Fazendas (

ID\_Fazenda INT PRIMARY KEY AUTO\_INCREMENT,

Nome\_Fazenda VARCHAR(100) NOT NULL,

Municipio VARCHAR(100) NOT NULL,

Estado VARCHAR(2) NOT NULL,

Area\_Total\_Hectares DECIMAL(10, 2),

Altitude\_Metros INT

);

**2. Lotes\_Producao (O que é produzido):**

SQL

CREATE TABLE Lotes\_Producao (

ID\_Lote INT PRIMARY KEY AUTO\_INCREMENT,

ID\_Fazenda INT NOT NULL, -- <-- A Chave Estrangeira (ponte)

Ano\_Safra INT NOT NULL,

Especie\_Cafe VARCHAR(100) NOT NULL,

Classificacao\_Cafe VARCHAR(100) NOT NULL,

Volume\_Producao\_Sacas DECIMAL(10, 2),

Pontuacao\_Qualidade INT,

-- Definindo a "ponte" que liga Lotes à Fazendas:

CONSTRAINT FK\_Lotes\_Fazendas

FOREIGN KEY (ID\_Fazenda)

REFERENCES Fazendas(ID\_Fazenda)

);

**3. Vendas (O que é vendido):**

SQL

CREATE TABLE Vendas (

ID\_Venda INT PRIMARY KEY AUTO\_INCREMENT,

ID\_Lote INT NOT NULL, -- <-- A Chave Estrangeira (ponte)

Data\_Venda DATE NOT NULL,

Quantidade\_Vendida\_Sacas DECIMAL(10, 2) NOT NULL,

Preco\_Por\_Saca\_BRL DECIMAL(10, 2) NOT NULL,

Tipo\_Mercado VARCHAR(10),

-- Definindo a "ponte" que liga Vendas à Lotes:

CONSTRAINT FK\_Vendas\_Lotes

FOREIGN KEY (ID\_Lote)

REFERENCES Lotes\_Producao(ID\_Lote)

);

**1.3 Populando as Tabelas (DML)**

Depois de criar a estrutura (DDL), usamos o INSERT INTO (DML) para adicionar nossos dados fictícios e começar a análise.

SQL

-- Exemplo de como inserimos dados nas Fazendas

INSERT INTO Fazendas (Nome\_Fazenda, Municipio, Estado, Area\_Total\_Hectares, Altitude\_Metros)

VALUES ('Fazenda Monte Alegre', 'Alfenas', 'MG', 1250.50, 1100);

**1.4 Lendo os Dados (DQL): SELECT e FROM**

O coração da análise. SELECT escolhe as colunas (o "quê") e FROM escolhe a tabela (o "onde").

* **Buscando tudo (para exploração):** SELECT \* FROM Fazendas;
* **Buscando colunas específicas (para análise/relatórios):** SELECT Nome\_Fazenda, Estado, Altitude\_Metros FROM Fazendas;

**1.5 Filtrando Linhas: WHERE**

A cláusula WHERE é nosso filtro principal. Ela atua **linha por linha** e só mantém as que passam na condição.

* **Filtrando por texto (aspas simples são cruciais!):** SELECT \* FROM Fazendas WHERE Estado = 'MG';
* **Filtrando por números:** SELECT \* FROM Fazendas WHERE Altitude\_Metros > 1200;
* **Filtrando por diferença:** SELECT \* FROM Fazendas WHERE Estado != 'SP';

**1.6 Ordenando Resultados: ORDER BY**

A cláusula ORDER BY organiza a saída final do seu relatório. Ela é quase sempre a **última** cláusula da consulta.

* **Ordem Ascendente (padrão, A-Z, 0-9):** SELECT \* FROM Fazendas ORDER BY Nome\_Fazenda ASC; (O ASC é opcional)
* **Ordem Descendente (Z-A, 9-0):** SELECT \* FROM Fazendas ORDER BY Altitude\_Metros DESC;

**1.7 Filtros Avançados: AND, OR, IN, BETWEEN, LIKE**

Para responder perguntas de negócio mais complexas, combinamos filtros:

* **AND (Restritivo):** Ambas as condições devem ser verdadeiras. ... WHERE Estado = 'MG' AND Altitude\_Metros > 1150;
* **OR (Expansivo):** Pelo menos uma condição deve ser verdadeira. ... WHERE Estado = 'ES' OR Estado = 'SP';
* **IN (Atalho para múltiplos ORs):** O valor deve estar "dentro de uma lista". ... WHERE Estado IN ('MG', 'ES', 'BA');
* **BETWEEN (Para intervalos):** Inclui os limites inicial e final. ... WHERE Altitude\_Metros BETWEEN 900 AND 1200;
* **LIKE (Busca de Padrões):** Usa o % como curinga ("qualquer coisa"). ... WHERE Nome\_Fazenda LIKE '%Primavera%'; (Contém "Primavera") ... WHERE Nome\_Fazenda LIKE 'Fazenda%'; (Começa com "Fazenda")

**1.8 Lidando com Nulos (Dados Faltantes): IS NULL**

O NULL representa a ausência de um valor. Ele não é 0 nem um texto vazio. Não podemos usar = para compará-lo.

* **Lição Aprendida:** Descobrimos que WHERE coluna = NULL não funciona.
* **A Sintaxe Correta:** ... WHERE Pontuacao\_Qualidade IS NULL; (Para encontrar linhas onde o dado falta) ... WHERE Pontuacao\_Qualidade IS NOT NULL; (Para encontrar linhas onde o dado foi preenchido)

**1.9 Apelidos com AS: Relatórios Profissionais**

A cláusula AS nos permite renomear colunas ou tabelas temporariamente na nossa consulta.

**1. Apelidos de Coluna (Muda o cabeçalho do relatório):**

SQL

SELECT

Nome\_Fazenda AS "Nome da Propriedade",

Area\_Total\_Hectares AS "Área (ha)"

FROM Fazendas;

* **Lição Aprendida (Sintaxe):** Para apelidos com espaços ou acentos, usamos aspas duplas (" "). Para valores de texto (strings), usamos aspas simples (' ').

**2. Apelidos de Tabela (Para código mais limpo e JOINs futuros):**

SQL

SELECT fz.Nome\_Fazenda, fz.Estado

FROM Fazendas AS fz

WHERE fz.Estado = 'MG';

* **Melhor Prática:** Usar o apelido (fz.) em todas as colunas (SELECT e WHERE) torna o código mais claro e nos prepara para JOINs.

**1.10 Desafio do Módulo 1**

Para fechar o módulo, unimos todos esses conceitos em um desafio: "Encontrar lotes de café Arábica da safra 2024 que sejam da classificação Especial ou Gourmet, ordenados do maior volume para o menor."

**A Lógica da Solução:**

SQL

SELECT

ID\_Lote,

Classificacao\_Cafe AS "Classificação",

Volume\_Producao\_Sacas AS "Volume"

FROM Lotes\_Producao

WHERE

Especie\_Cafe = 'Arábica'

AND Ano\_Safra = 2024

AND Classificacao\_Cafe IN ('Especial', 'Gourmet')

ORDER BY

Volume\_Producao\_Sacas DESC;

* **Lição Aprendida (Correção):** Nosso primeiro erro foi tentar agrupar as colunas no WHERE (ex: WHERE (Especie\_Cafe, Ano\_Safra) IN (...)). A correção foi entender que o WHERE testa cada condição de forma independente, conectada pelo AND.

Perfeito. Vamos para o Módulo 2, onde paramos de apenas *listar* dados e começamos a *resumi-los* e *calculá-los*.

**Módulo 2: O Poder da Agregação (Resumindo Dados)**

No Módulo 1, nossas consultas sempre retornavam linhas que já existiam na tabela. No Módulo 2, aprendemos a fazer perguntas que nos retornam **um único resultado calculado** a partir de várias linhas.

**A Analogia:**

* **Filtrar (Módulo 1):** Pegar uma pilha de notas fiscais e selecionar apenas as de "Outubro". O resultado ainda é uma pilha de notas.
* **Agregar (Módulo 2):** Pegar a pilha de notas de "Outubro", somar todas na calculadora e anotar o total em um post-it. O resultado é o post-it.

**2.1 O "Quinteto Fantástico": Funções de Agregação**

Estas são as "calculadoras" do SQL. Elas sempre operam em um conjunto de linhas (ou na tabela inteira).

1. **COUNT(\*) (Contar):**
   * **O que faz:** Retorna o número total de linhas no grupo.
   * **Pergunta:** "Quantas fazendas eu tenho?"
   * **Exemplo:** SELECT COUNT(\*) AS "Total de Fazendas" FROM Fazendas;
2. **SUM(coluna) (Somar):**
   * **O que faz:** Soma todos os valores de uma coluna numérica.
   * **Pergunta:** "Qual o volume total de sacas produzidas em 2024?"
   * **Exemplo:** SELECT SUM(Volume\_Producao\_Sacas) FROM Lotes\_Producao WHERE Ano\_Safra = 2024;
3. **AVG(coluna) (Média):**
   * **O que faz:** Calcula a média de uma coluna numérica.
   * **Pergunta:** "Qual o preço médio de exportação?"
   * **Exemplo:** SELECT AVG(Preco\_Por\_Saca\_BRL) FROM Vendas WHERE Tipo\_Mercado = 'Externo';
4. **MIN(coluna) (Mínimo):**
   * **O que faz:** Encontra o menor valor em uma coluna.
   * **Pergunta:** "Qual foi o menor preço que já vendemos?"
   * **Exemplo:** SELECT MIN(Preco\_Por\_Saca\_BRL) FROM Vendas;
5. **MAX(coluna) (Máximo):**
   * **O que faz:** Encontra o maior valor em uma coluna.
   * **Pergunta:** "Qual foi o maior lote que já produzimos?"
   * **Exemplo:** SELECT MAX(Volume\_Producao\_Sacas) FROM Lotes\_Producao;

**2.2 Valores Únicos: DISTINCT**

O DISTINCT é um modificador usado dentro do SELECT para remover duplicatas do resultado.

* **Pergunta:** "Em quais estados *diferentes* eu tenho fazendas?"
* **Exemplo:** SELECT DISTINCT Estado FROM Fazendas;
* **Combinado com COUNT:** "Em *quantos* estados diferentes eu opero?" SELECT COUNT(DISTINCT Estado) FROM Fazendas;

**2.3 O Comando Essencial: GROUP BY**

Este é o comando que torna a agregação poderosa. Ele "cria baldes" ou "categorias" e aplica as funções de agregação (COUNT, SUM, etc.) a cada "balde" separadamente.

* **Pergunta:** "Quantas fazendas eu tenho **POR ESTADO**?"
* **Exemplo:**

SQL

SELECT

Estado, -- A coluna do "balde"

COUNT(\*) AS "Total de Fazendas"

FROM

Fazendas

GROUP BY

Estado; -- O comando que cria os baldes

* **Regra de Ouro (Lição Aprendida):** Quando você usa GROUP BY, qualquer coluna no seu SELECT que **não** seja uma função de agregação (como COUNT, SUM) **DEVE** estar listada na cláusula GROUP BY.

**2.4 Filtrando Grupos: HAVING**

Enquanto o WHERE filtra linhas individuais *antes* da agregação, o HAVING filtra os grupos *depois* que eles são calculados.

* **Pergunta:** "Quais estados têm **mais de 2 fazendas**?"
* **Exemplo:**

SQL

SELECT

Estado,

COUNT(\*) AS "Total de Fazendas"

FROM

Fazendas

GROUP BY

Estado

HAVING

COUNT(\*) > 2; -- Filtra os resultados do COUNT

* **Lição Aprendida:** Você não pode usar WHERE COUNT(\*) > 2. O WHERE não "enxerga" o resultado de agregações, apenas o HAVING pode fazer isso.

**2.5 A Diferença Crucial: WHERE vs. HAVING (Ordem de Execução)**

Esta foi uma das nossas lições teóricas mais importantes. O SQL não é executado na ordem em que o escrevemos.

* **Ordem de Escrita (Como lemos):** SELECT -> FROM -> WHERE -> GROUP BY -> HAVING
* **Ordem de Execução (Como o SQL pensa):**
  1. FROM: Pega as tabelas.
  2. WHERE: Filtra as linhas individuais (ex: só as de 2024).
  3. GROUP BY: Agrupa as linhas que sobraram (ex: agrupa por fazenda).
  4. HAVING: Filtra os grupos já formados (ex: só grupos com mais de 1 lote).
  5. SELECT: Calcula as funções (COUNT, SUM) e pega as colunas para mostrar.
  6. ORDER BY: Organiza o resultado final.

**Por que isso importa?** Isso explica por que o WHERE não pode filtrar por COUNT(\*) (o COUNT só é calculado no Passo 5, muito depois do WHERE no Passo 2).

**2.6 Desafio do Módulo 2**

Para fechar, combinamos WHERE, GROUP BY e HAVING para responder a uma pergunta complexa: "Considerando apenas os lotes de café Arábica da safra 2024 (WHERE), quais fazendas (GROUP BY ID\_Fazenda) tiveram mais de um lote de produção (HAVING COUNT(\*) > 1)?"

SQL

SELECT

ID\_Fazenda,

COUNT(\*) AS "Total de Lotes"

FROM

Lotes\_Producao

WHERE

Especie\_Cafe = 'Arábica'

AND Ano\_Safra = 2024

GROUP BY

ID\_Fazenda

HAVING

COUNT(\*) > 1;

Perfeito. Vamos construir o Módulo 3, que é indiscutivelmente o mais importante para um analista: conectar tabelas para criar relatórios ricos.

**Módulo 3: Conectando Mundos (JOINs e UNION)**

Os dados raramente vivem em um único lugar. A verdadeira inteligência de negócio vem de cruzar informações. Neste módulo, aprendemos a "costurar" nossas tabelas.

**3.1 O Conceito: Chaves Primárias (PK) e Estrangeiras (FK)**

A "costura" só é possível por causa das chaves que definimos no Módulo 1:

* **Chave Primária (PK):** O "RG" de uma linha. É um valor único que identifica inequivocamente um registro. (Ex: Fazendas.ID\_Fazenda).
* **Chave Estrangeira (FK):** A "ponte". É uma coluna em uma tabela que "aponta" para a PK de outra tabela, criando o relacionamento. (Ex: Lotes\_Producao.ID\_Fazenda aponta para Fazendas.ID\_Fazenda).

**Nosso Mapa de Conexões:** Vendas --(conecta-se via ID\_Lote)--> Lotes\_Producao --(conecta-se via ID\_Fazenda)--> Fazendas

**3.2 A Teoria: Diagrama de Venn (Quando usar qual JOIN?)**

A forma mais fácil de pensar em JOINs é visualizando dois círculos se cruzando (Tabela A e Tabela B).

* **INNER JOIN (A Interseção):**
  + **O que faz:** Retorna **apenas** as linhas que têm correspondência em **ambas** as tabelas.
  + **Quando usar:** Quando você quer um resultado garantido, onde a ligação existe. (Ex: "Mostre-me apenas os lotes que foram *efetivamente* vendidos").
* **LEFT JOIN (O Círculo da Esquerda):**
  + **O que faz:** Retorna **TODAS** as linhas da tabela da **esquerda** (a do FROM) e os dados correspondentes da tabela da direita (a do JOIN).
  + **Quando usar:** Quando a tabela da esquerda é a principal. Se não houver correspondência na direita, as colunas da direita virão como NULL. (Ex: "Mostre-me **todas** as fazendas e, *se elas tiverem*, seus lotes de produção").

**3.3 INNER JOIN: Juntando 2 e 3 Tabelas**

O INNER JOIN é nosso "cavalo de batalha" para cruzamentos. A sintaxe ON é onde especificamos a "ponte" (PK = FK).

**Juntando 2 Tabelas (Nosso Dia 28):**

* **Pergunta:** "Quais fazendas tiveram lotes registrados? Mostre o nome da fazenda e o ano da safra."
* **Lição Aprendida (Correção):** A condição ON deve ligar colunas que significam a mesma coisa. O erro comum foi ON f.ID\_Fazenda = lp.ID\_Lote. O correto é ligar ID\_Fazenda com ID\_Fazenda.
* **Código:**

SQL

SELECT

f.Nome\_Fazenda,

lp.Ano\_Safra

FROM

Fazendas AS f -- Tabela da Esquerda (A)

INNER JOIN

Lotes\_Producao AS lp -- Tabela da Direita (B)

ON f.ID\_Fazenda = lp.ID\_Fazenda; -- A "ponte" correta

**Juntando 3 Tabelas (Nosso Dia 29):** Para juntar 3 tabelas, apenas "empilhamos" os JOINs, seguindo o nosso mapa.

* **Pergunta:** "Para cada venda, qual o nome da fazenda, a classificação do lote e o preço da venda?"
* **Código:**

SQL

SELECT

f.Nome\_Fazenda,

lp.Classificacao\_Cafe,

v.Preco\_Por\_Saca\_BRL

FROM

Vendas AS v -- Começamos pela tabela mais transacional

INNER JOIN

Lotes\_Producao AS lp ON v.ID\_Lote = lp.ID\_Lote -- 1ª Ponte (Vendas -> Lotes)

INNER JOIN

Fazendas AS f ON lp.ID\_Fazenda = f.ID\_Fazenda; -- 2ª Ponte (Lotes -> Fazendas)

**3.4 LEFT JOIN: Buscando Dados Incompletos**

O LEFT JOIN é essencial para encontrar o que *está* e o que *não está* conectado.

* **Pergunta (Nosso Dia 31):** "Liste **TODAS** as fazendas, ordenadas por nome, e mostre o ano da safra dos lotes que ela produziu. Se não tiver lote, ela deve aparecer mesmo assim."
* **Lição Aprendida (Identificando as Tabelas):** Como o pedido foi "TODAS as fazendas", a tabela Fazendas é a nossa tabela principal (esquerda) e deve ir no FROM.
* **Código:**

SQL

SELECT

f.Nome\_Fazenda,

lp.Ano\_Safra

FROM

Fazendas AS f -- Tabela principal (esquerda)

LEFT JOIN

Lotes\_Producao AS lp -- Tabela secundária (direita)

ON f.ID\_Fazenda = lp.ID\_Fazenda

ORDER BY

f.Nome\_Fazenda;

*(O resultado mostrará NULL em Ano\_Safra para fazendas sem lotes registrados).*

**3.5 RIGHT JOIN e FULL OUTER JOIN**

* **RIGHT JOIN:** É o espelho do LEFT JOIN (traz tudo da tabela da direita).
  + **Melhor Prática:** Raramente usado. É mais claro e legível inverter a ordem das tabelas no FROM/JOIN e usar sempre LEFT JOIN.
* **FULL OUTER JOIN:** Traz tudo de ambas as tabelas (a união dos dois círculos).
  + **Observação:** O MySQL não tem o comando FULL OUTER JOIN. Simulamos o seu efeito usando LEFT JOIN e RIGHT JOIN juntos com um UNION.

**3.6 Empilhando Resultados: UNION vs. UNION ALL**

Enquanto JOINs adicionam colunas (horizontalmente), o UNION empilha linhas de duas consultas (verticalmente).

* **Regra:** As duas consultas SELECT devem ter o **mesmo número de colunas** e tipos de dados compatíveis.
* **UNION:** Empilha os resultados e **remove duplicatas**.
* **UNION ALL:** Empilha os resultados e **mantém tudo**, incluindo duplicatas. É mais rápido.

**Nosso Exercício (Dia 34):** Criamos uma tabela Compradores e a unimos com Fazendas para uma lista de contatos unificada.

SQL

-- Lição Aprendida (Sintaxe): Os nomes das colunas do resultado final

-- são definidos APENAS pelos apelidos do PRIMEIRO SELECT.

SELECT

Nome\_Fazenda AS "Parceiro",

Estado AS "Localização"

FROM

Fazendas

UNION ALL

SELECT

Nome\_Comprador, -- Apelido aqui seria ignorado

Pais\_Origem -- Apelido aqui seria ignorado

FROM

Compradores;

**3.7 Desafio do Módulo 3 (JOIN + Agregação)**

O desafio final do módulo foi combinar os Módulos 2 e 3 para responder: **"Qual foi o valor total vendido POR FAZENDA?"**

**Lição Aprendida (A Lógica):** A pergunta foi "POR FAZENDA", mas os dados do cálculo (Quantidade\_Vendida\_Sacas, Preco\_Por\_Saca\_BRL) estavam na tabela Vendas. Precisamos dos JOINs para "trazer" o Nome\_Fazenda para perto dos dados de cálculo, para então podermos usar o GROUP BY Nome\_Fazenda.

**A Solução:**

SQL

SELECT

f.Nome\_Fazenda,

SUM(v.Quantidade\_Vendida\_Sacas \* v.Preco\_Por\_Saca\_BRL) AS "Valor Total Vendido"

FROM

Vendas AS v

INNER JOIN

Lotes\_Producao AS lp ON v.ID\_Lote = lp.ID\_Lote

INNER JOIN

Fazendas AS f ON lp.ID\_Fazenda = f.ID\_Fazenda

GROUP BY

f.Nome\_Fazenda -- Agrupamos pelo nome que trouxemos

ORDER BY

"Valor Total Vendido" DESC;

**Módulo 4: Transformação e Lógica Avançada (Onde o analista brilha)**

Neste módulo, aprendemos a parar de ser "reféns" dos dados como eles chegam e começamos a limpá-los, transformá-los e criar novas informações a partir deles.

**4.1 Funções de Texto (A "Faxina" dos Dados)**

Dados de texto são notórios por inconsistências. Aprendemos a usar um kit de "limpeza":

* **UPPER() e LOWER()**: Convertem o texto para MAIÚSCULAS ou minúsculas, essencial para padronizar filtros e agrupamentos. SELECT UPPER(Nome\_Fazenda) FROM Fazendas;
* **LENGTH()**: Retorna o número de caracteres de um texto.
* **TRIM()**: Remove espaços em branco do início e do fim do texto. Crucial para limpar dados copiados de planilhas. SELECT TRIM(' MG '); -- Retorna 'MG'
* **REPLACE(coluna, 'procurar', 'substituir')**: O "Localizar e Substituir" do SQL. SELECT REPLACE(Classificacao\_Cafe, 'Spcial', 'Especial') ...
* **LEFT(coluna, N) e RIGHT(coluna, N)**: Extrai os N primeiros ou últimos caracteres de um texto. SELECT RIGHT(data\_safra, 4) ... -- Usamos para extrair o ano '2024' de textos como 'safra:2024'
* **CONCAT(texto1, texto2, ...)**: Junta (concatena) múltiplos textos em um só. SELECT CONCAT(Municipio, ' - ', Estado) AS Localidade FROM Fazendas;

**Lição Aprendida (Aninhamento de Funções):** O verdadeiro poder aparece quando combinamos (aninhamos) essas funções. Para criar uma "Etiqueta de Marketing" limpa, fizemos:

SQL

SELECT

UPPER( -- 3. No final, converte tudo para maiúsculas

CONCAT( -- 2. Junta os textos...

Especie\_Cafe,

' - ',

REPLACE(Classificacao\_Cafe, 'Spcial', 'Especial'), -- 1. Corrigindo o erro primeiro...

' (',

Ano\_Safra,

')'

)

) AS "Etiqueta de Marketing"

FROM Lotes\_Producao;

A lógica é construir a consulta "de dentro para fora".

**4.2 Funções de Data (Análise Temporal)**

Datas são um tipo especial de dado que podemos "desmontar" e usar em cálculos.

* **Extraindo Partes (YEAR, MONTH, DAY):** Usamos funções como YEAR(Data\_Venda) ou MONTH(Data\_Venda) para extrair partes de uma data, permitindo-nos agrupar por ano ou mês.

SQL

-- A base para um relatório de faturamento mensal (Módulo 2 + Módulo 4)

SELECT

YEAR(Data\_Venda) AS Ano,

MONTH(Data\_Venda) AS Mes,

SUM(Quantidade\_Vendida\_Sacas \* Preco\_Por\_Saca\_BRL) AS Faturamento

FROM Vendas

GROUP BY Ano, Mes

ORDER BY Ano, Mes;

* **Cálculos com Datas (DATEDIFF, DATE\_ADD):**
  + **DATEDIFF(data\_final, data\_inicial)**: Calcula a diferença em **dias** entre duas datas.
  + **DATE\_ADD(data\_inicial, INTERVAL N UNIDADE)**: Adiciona um intervalo a uma data (ex: INTERVAL 15 DAY).
* **Desafio (Dia 39):** Calculamos os "Dias em Estoque".
  + **Lição Aprendida (Correção):** A consulta DATEDIFF precisa de duas datas. Como Ano\_Safra era um número (ex: 2024), nós o transformamos em uma data ('2024-12-31') usando CONCAT para poder calcular a diferença.

SQL

SELECT

DATEDIFF(v.Data\_Venda, CONCAT(lp.Ano\_Safra, '-12-31')) AS "Dias em Estoque"

FROM Vendas AS v

INNER JOIN Lotes\_Producao AS lp ON v.ID\_Lote = lp.ID\_Lote;

**4.3 Lógica Condicional: CASE WHEN ... THEN ...**

O CASE é o "Se... Então... Senão..." do SQL. Ele cria uma nova coluna baseada em regras de negócio. Ele testa as condições WHEN em ordem e para na primeira que for verdadeira.

* **Pergunta:** "Crie faixas de qualidade de café (Categoria A, B, C, D) baseadas na pontuação."
* **Lição Aprendida (A Lógica Sequencial):** O erro comum é testar o intervalo completo (ex: WHEN pontuacao >= 85 AND pontuacao < 90). O CASE é mais inteligente. Se ele testa a segunda condição, é porque a primeira já falhou.
* **Código Correto (limpo e sequencial):**

SQL

SELECT

Pontuacao\_Qualidade,

CASE

WHEN Pontuacao\_Qualidade >= 90 THEN 'Categoria A - Premiado'

-- Se chegou aqui, JÁ SABEMOS que é < 90.

WHEN Pontuacao\_Qualidade >= 85 THEN 'Categoria B - Superior'

-- Se chegou aqui, JÁ SABEMOS que é < 85.

WHEN Pontuacao\_Qualidade >= 80 THEN 'Categoria C - Bom'

-- Se não for nenhum dos acima...

ELSE 'Categoria D - Padrão'

END AS "Categoria de Marketing"

FROM

Lotes\_Producao;

**4.4 Subconsultas (Subqueries)**

Uma Subconsulta (ou Subquery) é um SELECT completo que vive **dentro** de outra consulta.

* **Uso 1: No WHERE (para filtros dinâmicos):** Usado para filtrar com base em um valor agregado.
  + **Pergunta:** "Quais lotes tiveram produção acima da média geral?"
  + **Lição Aprendida:** Não podemos usar WHERE Volume\_Producao\_Sacas > AVG(...), pois o WHERE é executado antes do AVG (Ordem de Execução).
  + **Código Correto:**

SQL

SELECT ID\_Lote, Volume\_Producao\_Sacas

FROM Lotes\_Producao

WHERE Volume\_Producao\_Sacas > (

-- A subconsulta é executada primeiro e retorna um único número

SELECT AVG(Volume\_Producao\_Sacas) FROM Lotes\_Producao

);

* **Uso 2: No FROM (Tabelas Derivadas):** Usado para fazer uma agregação sobre outra agregação (análise em duas etapas).
  + **Pergunta:** "Qual a produção média por fazenda?"
  + **Lição Aprendida:** Primeiro calculamos o SUM por fazenda (consulta interna), depois calculamos o AVG desses totais (consulta externa).
  + **Código:**

SQL

SELECT

AVG(volume\_total\_da\_fazenda) AS "Produção média por fazenda"

FROM (

-- 1. Esta consulta interna cria uma tabela temporária:

SELECT

ID\_Fazenda,

SUM(Volume\_Producao\_Sacas) AS volume\_total\_da\_fazenda

FROM

Lotes\_Producao

GROUP BY

ID\_Fazenda

) AS relatorio\_de\_producao; -- 2. A consulta externa calcula o AVG dessa tabela.

**4.5 CTEs (Common Table Expressions) com WITH**

As CTEs são a alternativa moderna e profissional às Subconsultas no FROM. Elas permitem "modularizar" o código, tornando-o muito mais limpo e legível.

* **Lição Aprendida:** Vimos que as CTEs funcionam como "módulos" em programação: você define as "peças" primeiro e depois as usa.
* **Reescrevendo o Exemplo Acima com CTE:**

SQL

-- 1. A CTE "relatorio\_de\_producao" é definida PRIMEIRO

WITH relatorio\_de\_producao AS (

SELECT

ID\_Fazenda,

SUM(Volume\_Producao\_Sacas) AS volume\_total\_da\_fazenda

FROM

Lotes\_Producao

GROUP BY

ID\_Fazenda

)

-- 2. A consulta principal (limpa) é executada DEPOIS, lendo da CTE

SELECT

AVG(volume\_total\_da\_fazenda) AS "Produção média por fazenda"

FROM

relatorio\_de\_producao;

* **Múltiplas CTEs (Encadeadas):** Vimos que podemos criar uma "linha de produção" de dados, onde uma CTE lê da anterior, quebrando um problema complexo em etapas simples e legíveis.

**Módulo 5: O Nível Raro (SQL Analítico - Window Functions)**

Até agora, quando usamos agregações (SUM, COUNT), elas "esmagavam" várias linhas em um único resultado (com GROUP BY). O Módulo 5 introduz as **Window Functions**, que nos permitem fazer cálculos agregados **sem colapsar as linhas**.

**5.1 O Conceito: Agregação sem GROUP BY (A Cláusula OVER())**

Window Functions (Funções de Janela) realizam um cálculo sobre um conjunto de linhas (a "janela"), mas retornam o resultado **ao lado de cada linha original**.

* **A Diferença Crucial (Nossa Lição do Dia 52):**
  + **GROUP BY:** Retorna *uma linha por grupo*. Você perde os detalhes individuais.
  + **... OVER():** Retorna *todas as linhas originais*, mas com uma nova coluna de cálculo.
* **Exemplo (Média Geral vs. Valor Individual):** Usamos AVG() OVER() para comparar a altitude de *cada* fazenda com a média *geral* de todas as fazendas.

SQL

SELECT

Nome\_Fazenda,

Altitude\_Metros,

-- A "janela" OVER() vazia significa "a tabela inteira"

AVG(Altitude\_Metros) OVER() AS "Altitude Média Geral"

FROM

Fazendas;

* **Lição Aprendida (Correção):** Nossos cálculos de média estavam confusos até investigarmos. Vimos que nossos INSERTs duplicados resultaram em 15 fazendas, levando à média correta de 1144.00, que a Window Function calculou perfeitamente.

**5.2 Particionando a Janela: PARTITION BY**

O PARTITION BY é o "GROUP BY" das Window Functions. Ele divide a "janela" em subgrupos (partições) e calcula a função de agregação separadamente para cada um, mas ainda sem colapsar as linhas.

* **Pergunta:** "Qual a pontuação média *por classificação*?" (Comparar um lote 'Especial' com a média de *todos* os 'Especial').
* **Lição Aprendida (Correção):** Nosso primeiro erro foi PARTITION BY Pontuacao\_Qualidade, o que apenas espelhou o valor. O correto é particionar pela *categoria* que queremos agrupar.
* **Código Correto:**

SQL

SELECT

ID\_Lote,

Classificacao\_Cafe,

Pontuacao\_Qualidade,

-- Calcula a média APENAS para o grupo da Classificacao\_Cafe da linha atual

AVG(Pontuacao\_Qualidade) OVER(PARTITION BY Classificacao\_Cafe) AS "Média da Classificação"

FROM

Lotes\_Producao;

* **Particionando por Múltiplas Colunas:** Também aprendemos a particionar por mais de uma coluna (ex: PARTITION BY Ano\_Safra, Especie\_Cafe) para obter subtotais ainda mais granulares.

**5.3 Funções de Ranking (ROW\_NUMBER, RANK, DENSE\_RANK)**

Window Functions são perfeitas para criar rankings. Todas elas exigem um ORDER BY *dentro* do OVER() para saber como classificar.

* **ROW\_NUMBER()**: Atribui um número **único e sequencial** (1, 2, 3, 4...). Trata empates com números diferentes.
* **RANK()**: Dá o **mesmo rank** para empates, mas **pula** as posições seguintes. (Ex: 1, 2, 2, 4...).
* **DENSE\_RANK()**: Dá o **mesmo rank** para empates, mas **não pula** posições. (Ex: 1, 2, 2, 3...).

**Exemplo (Comparando os três):**

SQL

SELECT

ID\_Lote,

Pontuacao\_Qualidade,

ROW\_NUMBER() OVER(ORDER BY Pontuacao\_Qualidade DESC) AS "Row Number",

RANK() OVER(ORDER BY Pontuacao\_Qualidade DESC) AS "Rank (Com Pulos)",

DENSE\_RANK() OVER(ORDER BY Pontuacao\_Qualidade DESC) AS "Dense Rank (Contínuo)"

FROM

Lotes\_Producao;

**5.4 Desafio Raro: "Top N por Grupo"**

O desafio mais prático do módulo: "Quais são os 2 lotes com maior pontuação *dentro de cada classificação*?" A solução combinou CTEs (Módulo 4) com ROW\_NUMBER e PARTITION BY.

**A Solução:**

SQL

-- 1. CTE para criar o ranking interno de cada grupo

WITH ranking\_lotes\_por\_classificacao AS (

SELECT

ID\_Lote,

Classificacao\_Cafe,

Pontuacao\_Qualidade,

ROW\_NUMBER() OVER(

PARTITION BY Classificacao\_Cafe -- Reinicia o ranking para cada classificação

ORDER BY Pontuacao\_Qualidade DESC -- Ordena pela pontuação

) AS "Ranking\_na\_Classificacao"

FROM

Lotes\_Producao

)

-- 2. Consulta final para filtrar apenas os "Top 2" de cada grupo

SELECT \*

FROM ranking\_lotes\_por\_classificacao

WHERE

Ranking\_na\_Classificacao <= 2; -- Lição Aprendida: <= 2 (para pegar o 1º e 2º), não >= 2

**5.5 Funções de Navegação (LAG e LEAD)**

Aprendemos a "olhar" para linhas vizinhas (anterior ou seguinte) dentro da nossa janela ordenada.

* **LAG(coluna, N)**: "Olha para trás". Pega o valor da coluna de N linhas **anteriores**.
* **LEAD(coluna, N)**: "Olha para frente". Pega o valor da coluna de N linhas **seguintes**.

**Exemplo (Cálculo de Variação Percentual):** Este foi um desafio complexo que uniu LAG, NULLIF (para evitar divisão por zero/NULL) e formatação.

SQL

SELECT

Data\_Venda,

Preco\_Por\_Saca\_BRL,

LAG(Preco\_Por\_Saca\_BRL, 1) OVER(ORDER BY Data\_Venda ASC) AS "Preço Anterior",

-- ((Atual - Anterior) / Anterior) \* 100

CONCAT( -- Adicionamos formatação no final

CEILING( -- Arredondamos para cima

(

(Preco\_Por\_Saca\_BRL - LAG(Preco\_Por\_Saca\_BRL, 1) OVER(ORDER BY Data\_Venda ASC))

/

NULLIF(LAG(Preco\_Por\_Saca\_BRL, 1) OVER(ORDER BY Data\_Venda ASC), 0)

) \* 100

),

'%'

) AS "% Variação vs Anterior"

FROM

Vendas;

**5.6 Janelas Móveis (Frames de Janela)**

Aprendemos a definir a "janela" de cálculo com mais precisão usando ROWS BETWEEN ... AND ... dentro do OVER().

* **Soma Acumulada (Running Total):** A soma de tudo "desde o início até a linha atual".
  + **Sintaxe:** ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
  + **Código:**

SQL

SELECT

Data\_Venda,

(Quantidade\_Vendida\_Sacas \* Preco\_Por\_Saca\_BRL) AS "Faturamento da Venda",

SUM(Quantidade\_Vendida\_Sacas \* Preco\_Por\_Saca\_BRL) OVER(

ORDER BY Data\_Venda ASC

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

) AS "Faturamento Acumulado"

FROM Vendas;

* **Média Móvel (Moving Average):** A média de um número fixo de linhas anteriores (ex: 3 períodos).
  + **Sintaxe:** ROWS BETWEEN 2 PRECEDING AND CURRENT ROW (para 3 períodos = 2 anteriores + 1 atual)
  + **Código:**

SQL

SELECT

Data\_Venda,

Preco\_Por\_Saca\_BRL,

AVG(Preco\_Por\_Saca\_BRL) OVER(

ORDER BY Data\_Venda ASC

ROWS BETWEEN 2 PRECEDING AND CURRENT ROW

) AS "Média Móvel (3 Vendas)"

FROM Vendas;

**Módulo 6: O Ambiente Moderno (Nuvem e JSON)**

Neste módulo, saímos um pouco da escrita pura de SQL para entender o **contexto** onde nossas consultas rodarão em cenários de grande volume de dados (Big Data).

**6.1 Data Warehouse (DW) vs. Data Lake**

Entender a diferença entre essas duas "arquiteturas" é fundamental para um analista sênior.

* **Data Warehouse (DW):**
  + **Analogia:** Um "armazém organizado".
  + **O que é:** Um banco de dados otimizado para **análise (OLAP)**. Ele centraliza dados de várias fontes (vendas, marketing, etc.) que já foram **limpos, transformados e estruturados** em um formato padronizado (ex: esquema estrela).
  + **Foco:** Responder perguntas de negócio com dados históricos consistentes. É a fonte ideal para dashboards em Power BI ou Tableau.
  + **Exemplos:** Google BigQuery, Snowflake, Amazon Redshift.
* **Data Lake:**
  + **Analogia:** Um "lago de dados brutos".
  + **O que é:** Um repositório que armazena dados massivos em seu **formato original** (JSON, CSV, logs, vídeos, áudios). A estrutura é aplicada *na hora da leitura* (schema-on-read).
  + **Foco:** Armazenamento flexível e de baixo custo. É a base para Ciência de Dados e Machine Learning, que precisam dos dados brutos.
  + **Exemplos:** Amazon S3, Google Cloud Storage, Azure Data Lake Storage.
* **Relação:** Eles são complementares. É comum um Data Lake (dados brutos) alimentar um Data Warehouse (dados limpos e prontos para análise).

**6.2 Arquitetura de Nuvem (BigQuery, Snowflake, Redshift)**

Os DWs modernos rodam na nuvem (AWS, Google Cloud, Azure) e compartilham um conceito revolucionário: a **separação de armazenamento e computação**.

* **Armazenamento:** É barato e quase infinito (você paga pelo que guarda).
* **Computação:** É elástico (você "liga" o poder de processamento para rodar sua query e "desliga" depois, pagando só pelo uso).

Isso nos levou à nossa reflexão: plataformas como **Snowflake** são ideais para cargas de trabalho com picos (como fechamento de mês), pois permitem ligar poder computacional massivo por um curto período e depois desligá-lo, otimizando custos.

**6.3 Dialetos SQL (Por que eles existem?)**

Vimos na prática que o SQL não é 100% idêntico em todas as plataformas.

* **Lição Aprendida (Nossa Correção do Dia 82):** Tentamos usar GROUP BY ROLLUP (colunaA, colunaB), que é o padrão ANSI SQL. No entanto, o **MySQL** usa um dialeto diferente: GROUP BY colunaA, colunaB WITH ROLLUP.
* **Por que existem dialetos?**
  1. **Performance:** Para criar funções otimizadas para a arquitetura *interna* daquela plataforma.
  2. **Dados Modernos:** Para inovar e dar suporte rápido a novos tipos de dados (JSON, GEOGRAPHY) e funções (Machine Learning) que o padrão ANSI demora a adotar.
  3. **Diferenciação:** Para oferecer recursos exclusivos (ex: Time Travel do Snowflake).

**6.4 Otimização na Nuvem (Custo e Performance)**

Em DWs na nuvem, otimizar significa **minimizar a quantidade de dados lidos (escaneados)**, pois muitas vezes o custo está atrelado a isso.

* **Armazenamento Colunar:** DWs modernos armazenam dados por coluna, não por linha.
* **Lição Aprendida (Dia 74):** SELECT \* é um grande vilão na nuvem. Se uma tabela tem 100 colunas e você só precisa de 2, SELECT \* força o sistema a ler todas as 100 colunas do disco. SELECT colunaA, colunaB lê apenas 2.
* **Particionamento (por Data):** Dividir a tabela fisicamente em "gavetas" (ex: uma por mês). Uma consulta com WHERE Data\_Venda = '2025-10-01' só abre a gaveta de "Outubro de 2025", ignorando petabytes de dados.
* **Clusterização (por Chave):** Ordenar os dados dentro das partições por colunas comuns em filtros (ex: Estado), permitindo que o sistema "pule" blocos de dados irrelevantes.

**6.5 Consultando Dados Semi-Estruturados (JSON)**

Este foi um grande foco prático do módulo. Aprendemos a extrair dados de colunas JSON.

* **1. Criando a Tabela:** CREATE TABLE Vendas\_JSON (id INT, dados\_venda JSON);
* **2. Extraindo Valores Simples e Aninhados (O "Ponto"):** Usamos o atalho ->> (que já faz JSON\_UNQUOTE) para extrair texto limpo.

SQL

-- Caminho Raiz: '$.Status'

SELECT dados\_venda ->> '$.Status' AS Status\_Venda FROM Vendas\_JSON;

-- Caminho Aninhado: '$.Cliente.Contato.Email'

SELECT dados\_venda ->> '$.Cliente.Contato.Email' AS Email\_Contato FROM Vendas\_JSON;

* **3. Extraindo de Arrays (Os "Colchetes"):** Usamos [indice] (começando em 0) para pegar itens específicos de uma lista.

SQL

-- Pega o ID\_Lote do PRIMEIRO item (índice 0) da lista "Itens"

SELECT dados\_venda ->> '$.Itens[0].ID\_Lote' AS Primeiro\_Item FROM Vendas\_JSON;

* **4. "Desempacotando" Arrays (A JSON\_TABLE):** Para transformar *cada* item de um array JSON em uma *linha* SQL, usamos a poderosa (e complexa) JSON\_TABLE. **Lição Aprendida (A Lógica):**
  1. A vírgula no FROM (entre Vendas\_JSON e JSON\_TABLE) age como um CROSS JOIN LATERAL, aplicando a função para cada linha.
  2. '$.Itens[\*]' é o caminho para o array, onde [\*] significa "para cada item".
  3. A cláusula COLUMNS define a nova tabela virtual.
  4. O PATH '$.Chave' dentro de COLUMNS refere-se à chave *dentro* do elemento do array.

**Código (Dia 78):**

SQL

SELECT

v.id AS ID\_Registro\_Original,

jt.ID\_Lote\_Item,

jt.Sacas\_Item

FROM

Vendas\_JSON AS v,

JSON\_TABLE(

v.dados\_venda, -- 1. Fonte do JSON

'$.Itens[\*]' -- 2. Caminho para o Array

COLUMNS ( -- 3. Definição das colunas de saída

ID\_Lote\_Item INT PATH '$.ID\_Lote',

Sacas\_Item INT PATH '$.Sacas'

)

) AS jt; -- 4. Apelido da tabela virtual

* **5. O "Porquê" (Desafio Dia 79):** "Desempacotamos" o JSON para poder usar o jt.ID\_Lote\_Item (um dado que estava "preso" no texto) como chave para um INNER JOIN com nossa tabela Lotes\_Producao. Isso uniu o mundo semiestruturado (JSON) ao mundo relacional (SQL).

Perfeito. Vamos finalizar a apostila com o Módulo 7, que você corretamente identificou como essencial: a revisão objetiva das operações de manipulação de dados.

**[Início do Conteúdo da Apostila - Módulo 7]**

**Módulo 7: Manipulação de Dados (CRUD)**

Este módulo é um retorno aos fundamentos do DML (Data Manipulation Language). Enquanto a maior parte do nosso curso foi focada em **R**ead (Ler) com SELECT (o "R" do CRUD), esta seção revisa objetivamente como **C**riar, **A**tualizar e **A**pagar dados.

A regra mais importante deste módulo é: **UPDATE e DELETE sem WHERE são desastrosos.**

**1. C = CREATE (Criar) - INSERT INTO**

* **Para que serve:** Adicionar uma ou mais novas linhas a uma tabela.
* **Sintaxe Padrão (Recomendada):** Especificando as colunas para garantir que os valores corretos sejam inseridos nos lugares corretos.

SQL

INSERT INTO nome\_da\_tabela (coluna1, coluna2, coluna3)

VALUES (valor1, valor2, valor3);

* **Exemplo do Curso:**

SQL

INSERT INTO Fazendas (Nome\_Fazenda, Municipio, Estado, Area\_Total\_Hectares, Altitude\_Metros)

VALUES ('Fazenda Nova', 'Muzambinho', 'MG', 150.00, 1100);

* **Sintaxe de Múltiplas Linhas:** Para inserir vários registros de uma vez (como fizemos ao popular o banco):

SQL

INSERT INTO Lotes\_Producao (ID\_Fazenda, Ano\_Safra, Especie\_Cafe)

VALUES

(1, 2024, 'Arábica'),

(3, 2024, 'Arábica'),

(2, 2024, 'Conilon');

**2. R = READ (Ler) - SELECT**

* **Para que serve:** Consultar, buscar, filtrar, ordenar, agrupar, juntar e transformar dados.
* **Status:** **Módulos 1 a 6.** Esta é a ferramenta que dominamos ao longo de todo o curso.

**3. U = UPDATE (Atualizar) - UPDATE ... SET ... WHERE**

* **Para que serve:** Modificar dados em linhas que **já existem** na tabela.
* **A SINTAXE MAIS IMPORTANTE DO SQL:**

SQL

UPDATE nome\_da\_tabela

SET coluna\_a\_mudar = novo\_valor1,

coluna\_b\_mudar = novo\_valor2

WHERE

condicao\_que\_identifica\_a\_linha; -- SEM ISSO, VOCÊ ATUALIZA A TABELA INTEIRA!

* **Lição Aprendida:** Usamos o UPDATE no Dia 42 para preencher nossa coluna Pontuacao\_Qualidade recém-criada.
* **Exemplo de Correção de Dado:**

SQL

-- Corrigindo a pontuação de um lote específico

UPDATE Lotes\_Producao

SET Pontuacao\_Qualidade = 93,

Classificacao\_Cafe = 'Premiado' -- Pode atualizar múltiplas colunas

WHERE ID\_Lote = 4;

* **CUIDADO:** UPDATE Lotes\_Producao SET Pontuacao\_Qualidade = 0; (sem WHERE) colocaria a pontuação de **TODOS** os lotes para 0.

**4. D = DELETE (Apagar) - DELETE FROM ... WHERE**

* **Para que serve:** Remover uma ou mais linhas **inteiras** da tabela.
* **A SINTAXE MAIS PERIGOSA DO SQL:**

SQL

DELETE FROM nome\_da\_tabela

WHERE

condicao\_que\_identifica\_a\_linha; -- SEM ISSO, VOCÊ APAGA TODOS OS DADOS!

* **Exemplo de Remoção de Registro:**

SQL

-- Removendo a fazenda de teste que inserimos

DELETE FROM Fazendas

WHERE Nome\_Fazenda = 'Fazenda Nova';

* **CUIDADO MÁXIMO:** DELETE FROM Vendas; (sem WHERE) **apagaria permanentemente todo o seu histórico de vendas**. Sempre verifique seu WHERE com um SELECT primeiro.

**Conclusão da Apostila**

Com este módulo, você fechou o ciclo completo de gerenciamento de dados. Você sabe como **Criar** estruturas (CREATE TABLE), **Inserir** dados (INSERT), **Ler** e analisar dados de forma complexa (SELECT com JOIN, GROUP BY, Window Functions, CASE, CTEs) e, finalmente, **Atualizar** (UPDATE) e **Apagar** (DELETE) dados de forma segura.