Consultas Avançadas em SQL

JOINs, Subqueries e Funções de Janela



JOINs Múltiplos **Subqueries** **Window Functions**

UC3 - Aula 08

Contexto e Abertura

> **"A magia do SQL está na capacidade de costurar dados de múltiplas tabelas."**
>
> Esta aula é focada em habilidades de consulta avançada, essenciais para extrair **insights complexos** de um banco de dados relacional. Dominar JOINs, Subqueries e Funções de Janela é o que diferencia um analista de dados competente de um excelente.

Por que Consultas Avançadas Importam?

Dados brutos não contam histórias. Consultas avançadas permitem combinar dados de múltiplas tabelas, aplicar lógica condicional complexa e realizar análises temporais que revelam padrões, tendências e oportunidades de negócio que dados simples nunca mostrariam.

JOINs Múltiplos

Use quando você precisa **combinar dados de múltiplas tabelas** baseado em relacionamentos.

Exemplo: Listar clientes com seus pedidos e produtos comprados.

Subqueries

Use quando você precisa **comparar valores com agregações** ou aplicar lógica condicional complexa.

Exemplo: Produtos cujo preço está acima da média.

Window Functions

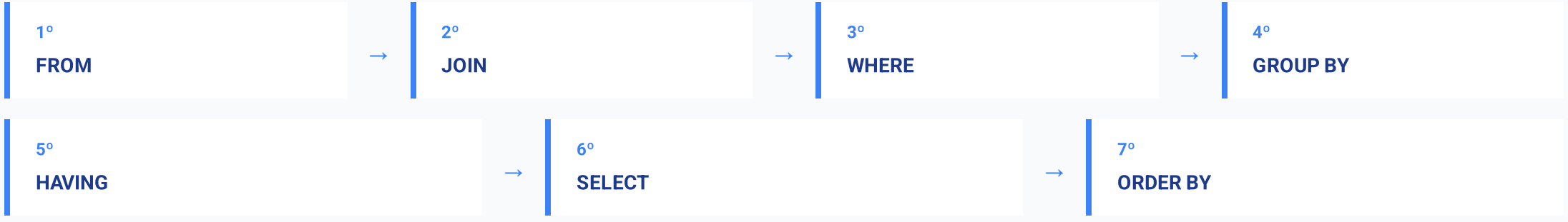
Use quando você precisa **realizar cálculos em janelas de dados** mantendo o detalhe de cada linha.

Exemplo: Ranking de vendas, soma cumulativa, comparação com período anterior.

O Fluxo Lógico do SELECT

Embora a sintaxe SQL comece com **SELECT**, a execução segue uma ordem lógica completamente diferente. **Entender essa ordem é crucial** para escrever consultas corretas e evitar erros comuns.

Ordem Lógica de Execução do SELECT



1-2. FROM e JOIN

Definem as tabelas de origem e como combiná-las. Os dados são carregados e ligados primeiro.

```
FROM Clientes JOIN Pedidos
```

3. WHERE

Filtra as linhas ANTES de qualquer agregação. Reduz o conjunto de dados.

```
WHERE data_pedido > '2025-01-01'
```

4-5. GROUP BY e HAVING

Agrupam dados e filtram grupos (não linhas individuais como WHERE).

```
GROUP BY cliente HAVING COUNT(*) > 5
```

6-7. SELECT e ORDER BY

SELECT escolhe as colunas finais. ORDER BY ordena o resultado final.

```
SELECT nome ORDER BY nome ASC
```

JOINs Múltiplos: Conceito e Sintaxe

Um JOIN sempre liga duas tabelas por vez. Para unir três tabelas (A, B, C), você precisa de **dois comandos JOIN**. Isso é fundamental para entender consultas complexas com múltiplas tabelas.

INNER JOIN (Interseção)

```
SELECT c.nome, p.id
FROM Clientes c
INNER JOIN Pedidos p
  ON c.id = p.id_cliente;
```

Retorna **apenas registros que existem em ambas as tabelas**. Se um cliente não tem pedidos, não aparece no resultado.

LEFT JOIN (Inclusão)

```
SELECT c.nome, p.id
FROM Clientes c
LEFT JOIN Pedidos p
  ON c.id = p.id_cliente;
```

Retorna **todos os registros da tabela esquerda** (Clientes), mesmo que não tenham correspondência. Pedidos será NULL se o cliente não tiver pedidos.

Aliases (Apelidos) para Tabelas

❌ Sem Alias (Verboso)

```
SELECT Clientes.nome, Pedidos.id FROM Clientes INNER JOIN Pedidos ON
Clientes.id = Pedidos.id_cliente;
```

✓ Com Alias (Recomendado)

```
SELECT c.nome, p.id FROM Clientes c INNER JOIN Pedidos p ON c.id =
p.id_cliente;
```

Ligando Três Tabelas (Dois JOINs)

```
SELECT c.nome, p.nome, pr.preco
FROM Clientes c
INNER JOIN Pedidos p ON c.id = p.id_cliente
INNER JOIN Produtos pr ON p.id_produto = pr.id;

Ordem de execução:
1. Ligar Clientes com Pedidos (primeira JOIN)
2. Ligar resultado com Produtos (segunda JOIN)
3. Selecionar colunas desejadas
```

JOINs Múltiplos: Exemplos Práticos

Exemplo 1: 2 JOINs

Listar cliente, produto e data do pedido para o último trimestre.

```
SELECT
  c.nome AS Cliente,
  p.nome AS Produto,
  pd.data_pedido AS Data
FROM Clientes c
INNER JOIN Pedidos pd
  ON c.id = pd.id_cliente
INNER JOIN Produtos p
  ON pd.id_produto = p.id
WHERE pd.data_pedido >=
  DATE_SUB(CURDATE(),
    INTERVAL 3 MONTH)
ORDER BY pd.data_pedido DESC;
```

Fluxo:

- FROM Clientes: tabela principal
- JOIN Pedidos: liga via id_cliente
- JOIN Produtos: liga via id_produto
- WHERE: filtra últimos 3 meses

✓ **Resultado: Pedidos recentes com detalhes**

Exemplo 2: LEFT JOIN

Listar todos os clientes e seus pedidos (mesmo sem pedidos).

```
SELECT
  c.id,
  c.nome AS Cliente,
  COUNT(pd.id) AS Qtd_Pedidos,
  SUM(pd.valor_total)
    AS Valor_Total
FROM Clientes c
LEFT JOIN Pedidos pd
  ON c.id = pd.id_cliente
GROUP BY c.id, c.nome
ORDER BY Valor_Total DESC;
```

Diferença:

- LEFT JOIN: mantém todos clientes
- COUNT(pd.id): conta pedidos
- SUM(): soma valores
- Clientes sem pedidos: COUNT=0

✓ **Resultado: Todos clientes + resumo**

Exemplo 3: 3 Tabelas

Listar cliente, produto, categoria e preço para cada pedido.

```
SELECT
  c.nome AS Cliente,
  p.nome AS Produto,
  cat.nome AS Categoria,
  p.preco AS Preco,
  (p.preco * pd.quantidade)
    AS Subtotal
FROM Clientes c
INNER JOIN Pedidos pd
  ON c.id = pd.id_cliente
INNER JOIN Produtos p
  ON pd.id_produto = p.id
INNER JOIN Categorias cat
  ON p.id_categoria = cat.id
WHERE YEAR(pd.data_pedido) = 2025
ORDER BY c.nome;
```

Sequência:

- Três JOINs = 4 tabelas
- Cada JOIN liga 2 tabelas
- Cálculo de subtotal
- Filtro por ano 2025

✓ **Resultado: Análise completa**

Atividade Prática 1: Múltiplos JOINs - Parte 1

Trabalhem em **duplas** para completar esta atividade. Vocês devem escrever uma consulta SQL com múltiplos JOINs que responda a uma pergunta de negócio complexa usando as tabelas fornecidas. Esta primeira parte apresenta o esquema de dados e os relacionamentos entre as tabelas.

Esquema de Dados Disponível

Clientes

- **id** (PK)
- nome
- email
- telefone
- cidade

Pedidos

- **id** (PK)
- **id_cliente** (FK)
- **id_produto** (FK)
- data_pedido
- valor_total

Produtos

- **id** (PK)
- nome
- preco
- categoria
- estoque

Relacionamentos Entre Tabelas

```
Clientes (1) ──→ (N) Pedidos ←──(N) Produtos

Explicação:
• Um cliente pode ter MÚLTIPLOS pedidos (1:N)
• Um produto pode estar em MÚLTIPLOS pedidos (N:M)
• Pedidos é a tabela central que liga Clientes e Produtos

Chaves Estrangeiras (FK):
• Pedidos.id_cliente → Clientes.id
• Pedidos.id_produto → Produtos.id
```

Atividade Prática 1: Múltiplos JOINs - Parte 2

Tarefa Principal

Desenvolver uma consulta SQL com **pelo menos dois JOINs** que responda à seguinte pergunta de negócio:

"Listar o nome do cliente, o nome do produto comprado e a data do pedido para todos os pedidos realizados no último trimestre."

Passo a Passo Detalhado

PASSO 1

Identifique as Tabelas

Você precisa de **Clientes**, **Pedidos** e **Produtos**. Qual é a tabela principal? (Resposta: Pedidos)

PASSO 2

Defina os JOINs

JOIN 1: Pedidos com Clientes (via id_cliente)
JOIN 2: Pedidos com Produtos (via id_produto)
Use INNER JOIN ou LEFT JOIN?

PASSO 3

Escreva a Estrutura Base

Comece com a estrutura básica do SELECT com os JOINs:

```
SELECT c.nome, p.nome, pd.data_pedido FROM Pedidos pd JOIN Clientes c ON pd.id_cliente = c.id JOIN Produtos p ON pd.id_produto = p.id
```

PASSO 4

Adicione a Cláusula WHERE

Filtre para o último trimestre:

```
WHERE pd.data_pedido >= DATE_SUB(CURDATE(), INTERVAL 3 MONTH)
```

PASSO 5

Ordene e Teste

Adicione **ORDER BY pd.data_pedido DESC**. Teste a consulta e verifique se os resultados fazem sentido.

DICA IMPORTANTE

Use Aliases para Tabelas

Use **c** para Clientes, **pd** para Pedidos, **p** para Produtos. Isso torna o código mais legível e evita ambiguidade.

Subqueries: Conceito e Tipos

O que é uma Subquery?

Uma **subquery** (ou subconsulta) é uma consulta **SELECT aninhada** dentro de outra consulta. A subquery executa primeiro e retorna um conjunto de resultados que é usado pela consulta externa para filtrar, comparar ou calcular valores. Subqueries são poderosas para lógica condicional complexa.

Linha Única

A subquery retorna **UM ÚNICO valor** (uma linha, uma coluna). Usada com operadores simples de comparação.

Operadores Usados:

- **=** (igual a)
- **>** (maior que)
- **<** (menor que)
- **>=, <=, <>** (variações)

Caso de Uso:

Pergunta: "Quais produtos têm preço acima da média?"
Subquery: SELECT AVG(preco) FROM Produtos
Comparação: WHERE preco > (resultado da subquery)

Múltiplas Linhas

A subquery retorna **MÚLTIPLOS valores** (múltiplas linhas). Usada com operadores especiais que lidam com conjuntos.

Operadores Usados:

- **IN** (está em uma lista)
- **ANY** (maior/menor que qualquer um)
- **ALL** (maior/menor que todos)
- **EXISTS** (verifica existência)

Caso de Uso:

Pergunta: "Quais clientes fizeram pedidos em 2025?"
Subquery: SELECT id_cliente FROM Pedidos WHERE YEAR(data)=2025
Comparação: WHERE id IN (lista de IDs)

Subqueries: Exemplos Práticos

Atividade Prática 2: Subqueries

Trabalhem em **duplas** para completar esta atividade. Vocês devem escrever uma **subquery de linha única** que compara um valor da tabela principal com uma agregação (média, máximo, mínimo) calculada na subquery.

Tarefa Principal

Desenvolver uma consulta SQL com **subquery de linha única** que responda à seguinte pergunta de negócio:

"Listar os produtos cujo preço está acima do preço médio de todos os produtos no estoque. Mostrar também a diferença entre o preço do produto e a média."

Passo a Passo Detalhado

PASSO 1

Calcule a Média

Primeiro, escreva a **subquery** que calcula a média de preços:

```
SELECT AVG(preco) FROM Produtos
```

PASSO 2

Escreva a Consulta Externa

Escreva a consulta principal que seleciona produtos:

```
SELECT nome, preco FROM Produtos
```

PASSO 3

Adicione a Cláusula WHERE

Use **WHERE preco >** para comparar com a subquery:

```
WHERE preco > (SELECT AVG(preco) FROM
Produtos)
```

PASSO 4

Calcule a Diferença

Adicione uma coluna que mostra a diferença do preço médio:

```
(preco - (SELECT AVG(preco) FROM Produtos)) AS
Diferenca
```

PASSO 5

Ordene os Resultados

Ordene por preço (maior para menor) para visualizar melhor:

```
ORDER BY preco DESC
```

PASSO 6

Teste e Valide

Execute a consulta completa. Verifique se os resultados fazem sentido. Todos os produtos listados têm preço > média?

Funções de Janela: Conceito e Sintaxe

O que é uma Função de Janela?

Funções de Janela realizam **cálculos em um conjunto de linhas (a janela)** relacionado à linha atual, **sem agrupar o resultado**. O resultado é anexado a cada linha, **mantendo o detalhe**. Diferente de GROUP BY que reduz linhas, Window Functions preservam todas as linhas originais e adicionam cálculos.

Sintaxe Essencial: OVER()

```
SELECT coluna1, coluna2,
  FUNCAO_JANELA() OVER (
    PARTITION BY coluna_grupo
    ORDER BY coluna_ordem
  ) AS resultado_janela
FROM tabela;
```

Componentes: **FUNCAO_JANELA():** ROW_NUMBER(), RANK(), SUM(), AVG(), LAG(), LEAD(), etc. | **PARTITION BY:** Divide a janela em grupos (opcional) | **ORDER BY:** Define a ordem dentro da janela (opcional, mas geralmente necessário)

Ranqueamento

Atribui **posições ou números** às linhas dentro da janela.

Funções:

- **ROW_NUMBER()** - Número sequencial
- **RANK()** - Rank com empates
- **DENSE_RANK()** - Rank sem saltos

```
ROW_NUMBER() OVER (
  ORDER BY valor DESC
) AS posicao
```

Agregação

Realiza **cálculos agregados** (soma, média, contagem) em janelas sem agrupar.

Funções:

- **SUM()** - Soma cumulativa
- **AVG()** - Média em janela
- **COUNT()** - Contagem em janela

```
SUM(valor) OVER (
  ORDER BY data
) AS soma_acumulada
```

Deslocamento

Compara a **linha atual com linhas anteriores/próximas** na janela.

Funções:

- **LAG()** - Linha anterior
- **LEAD()** - Linha próxima
- **FIRST_VALUE()** - Primeira linha

```
LAG(valor) OVER (
  ORDER BY data
) AS valor_anterior
```

Funções de Janela: Exemplos Práticos - Parte 1

Exemplo 1: ROW_NUMBER()

```
SELECT
  c.nome AS Cliente,
  pd.valor_total,
  ROW_NUMBER() OVER
    (PARTITION BY c.id
     ORDER BY pd.valor_total DESC)
    AS Posicao
FROM Clientes c
JOIN Pedidos pd
  ON c.id = pd.id_cliente;
```

Objetivo: Numerar sequencialmente os pedidos de cada cliente (maior para menor).

✓ Cada pedido recebe um número sequencial por cliente

Exemplo 2: RANK()

```
SELECT
  nome,
  preco,
  RANK() OVER
    (ORDER BY preco DESC)
    AS Posicao,
  ROW_NUMBER() OVER
    (ORDER BY preco DESC)
    AS Numero
FROM Produtos;
```

Objetivo: Ranquear produtos por preço, mostrando diferença entre RANK e ROW_NUMBER.

✓ RANK tem saltos (1,2,2,4); ROW_NUMBER é sequencial (1,2,3,4)

Exemplo 3: SUM() - Soma Cumulativa

```
SELECT
  data_pedido,
  valor_total,
  SUM(valor_total) OVER
    (ORDER BY data_pedido)
    AS Venda_Acumulada
FROM Pedidos
ORDER BY data_pedido;
```

Objetivo: Calcular o total de vendas acumulado até cada data.

✓ Cada linha mostra a soma de todos os pedidos até essa data

Próximo slide: Exemplos 4, 5 e 6 (AVG em Janela, LAG/LEAD, DENSE_RANK)

Funções de Janela: Exemplos Práticos - Parte 2

Exemplo 4: AVG() - Média em Janela

```
SELECT
  c.nome,
  pd.valor_total,
  AVG(pd.valor_total) OVER
    (PARTITION BY c.id)
    AS Media_Cliente,
  (pd.valor_total -
   AVG(pd.valor_total) OVER
   (PARTITION BY c.id))
    AS Diferenca
FROM Clientes c
JOIN Pedidos pd
  ON c.id = pd.id_cliente;
```

Objetivo: Comparar cada pedido com a média de pedidos do cliente.

✓ **Cada pedido mostra quanto está acima/abaixo da média do cliente**

Exemplo 5: LAG() / LEAD()

```
SELECT
  c.nome,
  pd.data_pedido,
  pd.valor_total,
  LAG(pd.valor_total) OVER
    (PARTITION BY c.id
     ORDER BY pd.data_pedido)
    AS Valor_Anterior,
  LEAD(pd.valor_total) OVER
    (PARTITION BY c.id
     ORDER BY pd.data_pedido)
    AS Valor_Proximo
FROM Clientes c
JOIN Pedidos pd
  ON c.id = pd.id_cliente;
```

Objetivo: Comparar cada pedido com o anterior e o próximo do mesmo cliente.

✓ **Análise de tendência: pedidos aumentando ou diminuindo?**

Exemplo 6: DENSE_RANK()

```
SELECT
  c.nome,
  SUM(pd.valor_total)
    AS Total_Vendas,
  DENSE_RANK() OVER
    (ORDER BY
     SUM(pd.valor_total) DESC)
    AS Posicao
FROM Clientes c
JOIN Pedidos pd
  ON c.id = pd.id_cliente
GROUP BY c.id, c.nome;
```

Objetivo: Ranquear clientes por valor total de vendas sem saltos.

✓ **DENSE_RANK: 1,2,2,3 (sem saltos como RANK)**

Atividade Prática 3: Funções de Janela

Trabalhem em **duplas** para completar esta atividade. Vocês devem implementar duas tarefas usando **Window Functions**: uma com ranqueamento e outra com soma cumulativa. Usem o esquema de dados com Clientes, Pedidos e Produtos.

Tarefa 1: Ranqueamento

Ranquear o valor total dos pedidos de cada cliente do maior para o menor.

Passo a Passo:

Passo 1: Selecione cliente, valor do pedido e a função RANK()

Passo 2: Use RANK() OVER (ORDER BY valor_total DESC)

Passo 3: Teste com diferentes clientes

Passo 4: Observe como pedidos com mesmo valor recebem mesmo rank

```
SELECT
  c.nome AS Cliente,
  pd.valor_total AS Valor,
  RANK() OVER (
    ORDER BY pd.valor_total DESC
  ) AS Posicao_Vendas
FROM Clientes c
INNER JOIN Pedidos pd
  ON c.id = pd.id_cliente
ORDER BY Posicao_Vendas;
```

Tarefa 2: Soma Cumulativa

Calcular o total de vendas acumulado a cada dia (crescimento ao longo do tempo).

Passo a Passo:

Passo 1: Selecione data do pedido, valor e SUM()

Passo 2: Use SUM(valor_total) OVER (ORDER BY data_pedido)

Passo 3: Observe como cada linha mostra a soma até aquela data

Passo 4: Compare com SUM() sem OVER (resultado diferente!)

```
SELECT
  pd.data_pedido AS Data,
  pd.valor_total AS Valor_Dia,
  SUM(pd.valor_total) OVER (
    ORDER BY pd.data_pedido
  ) AS Venda_Acumulada
FROM Pedidos pd
ORDER BY pd.data_pedido;
```

Dicas Importantes

PARTITION BY

Use **PARTITION BY** para reiniciar a janela para cada grupo (ex: PARTITION BY id_cliente para ranquear por cliente separadamente).

ORDER BY

Use **ORDER BY** dentro de OVER() para definir a ordem de processamento da janela (não é o ORDER BY final).

RANK vs ROW_NUMBER

RANK() permite empates (1, 2, 2, 4). **ROW_NUMBER()** sempre sequencial (1, 2, 3, 4).

Mantém Detalhe

Window Functions **não agrupam dados** como GROUP BY. Cada linha original é mantida no resultado.

Validação Cruzada e Documentação - Parte 1

Validação cruzada é quando **outras pessoas revisam seu código** para identificar erros de sintaxe, lógica ou estilo. É uma prática profissional essencial que melhora a qualidade do código e garante que as consultas SQL funcionem corretamente em diferentes cenários.

Atividade: Troca de Scripts e Validação

PASSO 1

Troque Scripts com Outra Dupla

Cada dupla entrega seus três scripts SQL (JOINs, Subquery, Window Function) para outra dupla revisar. Não é sobre encontrar culpados, é sobre **melhorar a qualidade**.

PASSO 2

Identifique Erros de Sintaxe

Execute os scripts na sua máquina. Há erros de sintaxe? Parênteses faltando? Nomes de tabelas incorretos? Aliases não definidos? Documente cada erro encontrado.

PASSO 3

Verifique a Lógica

O script retorna o resultado esperado? A lógica está correta? Há condições WHERE desnecessárias? Os JOINs estão corretos? A window function faz sentido?

PASSO 4

Discuta os Resultados

Reúna-se com a dupla original. Apresente os erros encontrados de forma **construtiva**. Explique por que é um erro. Sugira melhorias. Aprenda juntos.

Checklist: Erros Comuns a Procurar

- ❌ **Aliases não definidos:** Usar coluna sem alias (ex: c.nome quando c não foi definido)
- ❌ **JOINs incorretos:** ON conditions erradas ou tabelas invertidas
- ❌ **WHERE vs HAVING:** Usar HAVING para filtrar linhas (deveria ser WHERE)
- ❌ **Subqueries faltando parênteses:** (SELECT ...) precisa de parênteses
- ❌ **Window Functions sem OVER():** Toda window function precisa de OVER()
- ❌ **ORDER BY ambíguo:** Qual coluna? De qual tabela? Use alias.

Próximo slide: Importância da Documentação em SQL com exemplos práticos

Validação Cruzada e Documentação - Parte 2

Importância da Documentação em SQL

Por Que Documentar?

Código sem comentários é **difícil de manter**. Você mesmo pode esquecer o que fez em 2 meses. Colegas não entendem a lógica. Documentação profissional é **obrigatória** em qualquer empresa. Comentários bem escritos economizam tempo e evitam erros.

Exemplo: Script Bem Documentado

```
-- Objetivo: Listar produtos
-- acima da média de preço
-- Autor: João Silva
-- Data: 2025-10-22

SELECT
  nome,
  preco,
  -- Diferença do preço médio
  (preco -
   (SELECT AVG(preco)
    FROM Produtos)
  ) AS diferenca
FROM Produtos
WHERE preco >
  (SELECT AVG(preco)
   FROM Produtos)
ORDER BY preco DESC;
```

Boas Práticas de Documentação Profissional

Cabeçalho do Script: Sempre inclua objetivo, autor, data e versão no topo do script

Comentários Inline: Explique **por que** você faz algo, não apenas o que faz

Nomes Significativos: Use aliases e nomes de coluna que deixem claro o propósito

Indentação Consistente: Organize o código com indentação clara para legibilidade

Documentar Mudanças: Se modificar um script, adicione comentário com data e motivo

Explicar Lógica Complexa: Subqueries e Window Functions precisam de explicação

Documentação Externa: Mantenha arquivo README.md descrevendo todos os scripts

Exemplos de Saída: Documente o resultado esperado para facilitar testes

Síntese e Consolidação Final

Resumo dos Três Tópicos Principais

JOINs Múltiplos

Combinam dados de múltiplas tabelas baseado em relacionamentos. **Dois JOINs ligam 3 tabelas.** Use INNER JOIN para interseção, LEFT JOIN para inclusão. Sempre use aliases para clareza.

Subqueries

Consultas aninhadas que retornam valores para a consulta externa. **Linha única:** operadores simples (=, >, <). **Múltiplas linhas:** operadores IN, ANY, ALL. Poderosas para lógica condicional.

Window Functions

Cálculos em janelas de dados sem agrupar. **Três categorias:** ranqueamento (ROW_NUMBER, RANK), agregação (SUM, AVG), deslocamento (LAG, LEAD). Mantém detalhe de cada linha.

Checklist de Boas Práticas Essenciais

- ✓ **Use aliases para tabelas** - Torna o código legível e evita ambiguidade
- ✓ **Sempre use ON conditions corretas** - JOINs incorretos causam resultados errados
- ✓ **Teste com SELECT antes de DELETE** - Verifique o que será deletado
- ✓ **Documente seu código** - Comentários explicam a lógica para você e colegas
- ✓ **Entenda a ordem de execução** - FROM → JOIN → WHERE → GROUP BY → HAVING → SELECT → ORDER BY
- ✓ **Use PARTITION BY em Window Functions** - Reinicia a janela para cada grupo
- ✓ **Prefira JOINs a Subqueries** - JOINs geralmente são mais eficientes
- ✓ **Valide seus resultados** - Peça para colegas revisarem seu código (validação cruzada)

Reflexão Final: Proficiência em Consultas Avançadas

Dominar **JOINs, Subqueries e Window Functions** é a diferença entre um usuário de banco de dados e um **analista de dados profissional**. Essas habilidades permitem extrair insights complexos, automatizar análises e responder perguntas de negócio que dados simples nunca revelariam.

A jornada não termina aqui. Continue praticando, explorando novos SGBDs (MySQL, PostgreSQL, SQL Server), e aprofundando em otimização de performance. **Cada consulta que você escreve é uma oportunidade de aprender.**

🎓 Parabéns! Você completou a Aula 08 de Consultas Avançadas em SQL. Agora você tem as ferramentas para trabalhar com dados de forma profissional e confiante!