# Consultas Avançadas em SQL

## JOINs Múltiplos, Subqueries e Funções de Janela

UC3 - AULA 09

JOINs

Subqueries

Window Functions

# Abertura e Contexto: Consultando Dados Complexos

Nas aulas anteriores, aprendemos a **criar tabelas, inserir dados e fazer consultas simples**. Agora enfrentamos um desafio maior: **combinar informações de múltiplas tabelas de forma eficiente**.

### O Desafio: Dados Residem em Três ou Mais Tabelas Interligadas

Um pedido não existe isoladamente. Ele conecta um cliente, múltiplos produtos, datas, valores e histórico. Para responder perguntas de negócio reais (ex: "Qual é o cliente que mais gastou em eletrônicos nos últimos 3 meses?"), precisamos **costurar dados de várias tabelas** usando **JOINs, Subqueries e Funções de Janela**.

### JOINs Múltiplos

Combinam dados de múltiplas tabelas baseado em relacionamentos. Eficientes e diretos.

**Quando usar:**

Listar dados de múltiplas tabelas com uma única consulta. Ex: Cliente + Pedido + Produto.

### Subqueries

Consultas aninhadas que retornam valores para análise condicional. Poderosas para lógica complexa.

**Quando usar:**

Comparar valores com agregações dinâmicas. Ex: Clientes acima da média.

### Window Functions

Cálculos em janelas de dados sem agrupar. Mantêm detalhe de cada linha.

**Quando usar:**

Análise temporal, ranqueamento e comparações. Ex: Soma cumulativa de vendas.

Dominar essas três técnicas é o que **distingue um operador SQL de um analista de dados profissional**. Você não apenas consulta dados, você **extrai insights, identifica padrões e responde perguntas complexas de negócio**.

**Vamos começar!** 🚀

# O Fluxo Lógico do SELECT: Ordem de Execução

```
1. FROM
    ↓
2. JOIN
    ↓
3. WHERE
    ↓
4. GROUP BY
    ↓
5. HAVING
    ↓
6. SELECT
    ↓
7. ORDER BY
```

## O Que Acontece em Cada Etapa

PASSO 1

### FROM

Identifica a tabela principal. O SQL **começa aqui**, não no SELECT!

PASSO 2

### JOIN

Combina dados de outras tabelas baseado em condições (ON). Múltiplos JOINs executam sequencialmente.

PASSO 3

### WHERE

Filtra linhas **ANTES de agrupar**. Reduz o volume de dados rapidamente.

PASSO 4

### GROUP BY

Agrupa linhas por colunas especificadas. Necessário quando há funções de agregação (SUM, COUNT, AVG).

PASSO 5

### HAVING

Filtra grupos **DEPOIS de agrupar**. Usa resultados de agregações.

PASSO 6

### SELECT

Escolhe quais colunas retornar. Executa **DEPOIS** de todas as filtragens!

PASSO 7

### ORDER BY

Ordena o resultado final. Última operação executada.

# JOINs Múltiplos: Conceito e Sintaxe

**JOINs combinam dados de múltiplas tabelas** baseado em **relacionamentos definidos nas cláusulas ON**. Para unir **três tabelas**, você precisa de **dois JOINs sequenciais**. A tabela B liga A e C. A ordem importa: A JOIN B JOIN C (ou A JOIN C JOIN B).

## INNER JOIN

Retorna **apenas registros que existem em AMBAS as tabelas**. Perfeito para faturamento (só quer clientes com pedidos).

```
SELECT c.nome, pd.valor
FROM Clientes c
INNER JOIN Pedidos pd
  ON c.id = pd.id_cliente
```

**Resultado:** Apenas clientes que fizeram pedidos aparecem.

## ← LEFT JOIN

Retorna **todos os registros da tabela esquerda** + correspondências da direita. Útil para análise completa (todos os clientes, com ou sem pedidos).

```
SELECT c.nome, pd.valor
FROM Clientes c
LEFT JOIN Pedidos pd
  ON c.id = pd.id_cliente
```

**Resultado:** Todos os clientes aparecem. Clientes sem pedidos têm NULL.

## Estratégias de Eficiência com Múltiplos JOINs

### 1. Use Aliases Obrigatoriamente

c.nome deixa claro que é de Clientes. Sem alias, SQL fica confuso e lento. Exemplo: **FROM Clientes c**

### 2. Filtre Cedo com WHERE

Adicione WHERE **ANTES de JOINs** quando possível. Reduz volume de dados rapidamente. Menos linhas = JOINs mais rápidos.

### 3. Ordem Correta de JOINs

Comece com a tabela maior. Se Clientes tem 10k linhas e Pedidos tem 100k, comece com Pedidos. Menos comparações = mais rápido.

### 4. Evite JOINs Desnecessários

Cada JOIN adiciona complexidade. Se precisa de dados de 5 tabelas, pergunte: **Realmente preciso de todas?** Às vezes, uma subquery é mais eficiente.

# JOINs Múltiplos: Exemplos Práticos

### Exemplo 1: Dois JOINs Sequenciais (Cliente → Pedido → Produto)

```
SELECT
  c.nome AS Cliente,
  pd.valor_total AS Valor_Pedido,
  p.nome AS Produto
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 >= '2025-01-01'
ORDER BY c.nome;
```

**Objetivo:** Listar cliente, valor do pedido e produto. **Fluxo:** Começa com Clientes (FROM), une com Pedidos (1º JOIN), une com Produtos (2º JOIN), filtra por data (WHERE), ordena por nome (ORDER BY).

**✓ Resultado: 1 linha por pedido com dados do cliente e produto**

### Exemplo 2: LEFT JOIN com Agregação (Manter Todos os Clientes)

```
SELECT
  c.nome AS Cliente,
  COUNT(pd.id) AS Total_Pedidos,
  COALESCE(SUM(pd.valor_total), 0) AS Valor_Total
FROM Clientes c
LEFT JOIN Pedidos pd ON c.id = pd.id_cliente
GROUP BY c.id, c.nome
HAVING COUNT(pd.id) >= 0
ORDER BY Valor_Total DESC;
```

**Objetivo:** Listar TODOS os clientes com total de pedidos (mesmo quem não tem). **Diferença:** LEFT JOIN mantém clientes sem pedidos (com NULL), COUNT retorna 0. COALESCE converte NULL em 0.

**✓ Resultado: 1 linha por cliente com agregações (clientes sem pedidos aparecem com 0)**

### Exemplo 3: Três Tabelas com Múltiplos Filtros (Análise Completa)

```
SELECT
  c.nome AS Cliente,
  c.cidade AS Cidade,
  pd.data_pedido AS Data,
  pd.valor_total AS Valor,
  p.nome AS Produto,
  p.categoria AS Categoria
FROM Clientes c
INNER JOIN Pedidos pd ON c.id = pd.id_cliente
INNER JOIN Produtos p ON pd.id_produto = p.id
WHERE p.categoria = 'Eletrônicos'
  AND pd.data_pedido >= DATE_SUB(CURDATE(), INTERVAL 90 DAY)
  AND c.cidade = 'São Paulo'
ORDER BY pd.data_pedido DESC, pd.valor_total DESC;
```

**Objetivo:** Análise completa de vendas de eletrônicos em São Paulo nos últimos 90 dias. **Filtros:** Categoria (WHERE), data (WHERE), cidade (WHERE). **Ordem:** Mais recente primeiro, depois maior valor.

**✓ Resultado: Linhas detalhadas de vendas com informações de cliente, pedido e produto**

# Atividade Prática 1: JOINs Múltiplos - 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: JOINs Múltiplos - 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 valor total do pedido e o nome do produto mais caro dentro daquele pedido."**

## Passo a Passo Detalhado (5 Passos)

PASSO 1

### Identifique as Tabelas Necessárias

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

PASSO 2

### Defina os JOINs e Condições ON

JOIN 1: Pedidos com Clientes (via id_cliente)
JOIN 2: Pedidos com Produtos (via id_produto)
Use INNER JOIN ou LEFT JOIN? (Resposta: INNER JOIN, porque queremos apenas pedidos com cliente e produto válidos)

PASSO 3

### Escreva a Estrutura Base com JOINs

Comece com a estrutura básica do SELECT com os JOINs:

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

PASSO 4

### Adicione Filtros (WHERE) se Necessário

Você quer filtrar por data, região ou categoria? Adicione WHERE se necessário. Exemplo:

```
WHERE pd.data_pedido >= '2025-01-01' AND p.categoria = 'Eletrônicos'
```

PASSO 5

### Ordene e Teste

Adicione **ORDER BY** para organizar os resultados. Teste a consulta e verifique se os resultados fazem sentido. Há duplicatas? Os valores estão corretos?

DICA IMPORTANTE

### Use Aliases para Tabelas e Colunas

Use **c** para Clientes, **pd** para Pedidos, **p** para Produtos. Dê nomes significativos às colunas com **AS**. Exemplo: **c.nome AS Cliente**

# Subqueries: Conceito e Tipos

Uma **subquery (consulta aninhada)** é uma consulta SQL dentro de outra consulta SQL. A subquery interna executa primeiro e retorna um valor (ou múltiplos valores) que a consulta externa usa para filtrar, comparar ou calcular. Subqueries são poderosas para **análise condicional dinâmica**, permitindo comparar valores com agregações que mudam conforme os dados.

## Subquery de Linha Única

Retorna **exatamente uma linha e uma coluna**. Resultado é um único valor que pode ser comparado com operadores simples.

**Operadores Permitidos:**

`=` Igual a

`>` Maior que

`<` Menor que

`>=, <=, <>` Outros comparadores

**Exemplo de Sintaxe:**

```
SELECT nome, salario
FROM Funcionarios
WHERE salario > (
  SELECT AVG(salario)
  FROM Funcionarios
);
```

**Caso de Uso:**

Comparar cada valor com uma agregação dinâmica (média, máximo, mínimo).

## Subquery de Múltiplas Linhas

Retorna **múltiplas linhas e/ou colunas**. Resultado é um conjunto de valores que requer operadores especiais.

**Operadores Permitidos:**

`IN` Pertence ao conjunto

`ANY` Compara com qualquer valor

`ALL` Compara com todos os valores

`EXISTS` Verifica existência

**Exemplo de Sintaxe:**

```
SELECT nome, email
FROM Clientes
WHERE id IN (
  SELECT id_cliente
  FROM Pedidos
  WHERE valor > 500
);
```

**Caso de Uso:**

Verificar se um valor pertence a um conjunto de resultados (ex: clientes que fizeram pedidos).

# Subqueries: Exemplos Práticos - Parte 1

## 1 Subquery de Linha Única: Comparação com Média

```
SELECT nome, salario
FROM Funcionarios
WHERE salario > (
  SELECT AVG(salario) FROM Funcionarios
);
```

**Objetivo:** Listar funcionários com salário acima da média geral. A **subquery interna** calcula a média (ex: 3000), depois a **consulta externa** compara cada salário com esse valor.

✓ **Resultado: Funcionários com salário > 3000**

## 2 Subquery com IN: Verificar Pertencimento a Conjunto

```
SELECT nome, email
FROM Clientes
WHERE id IN (
  SELECT id_cliente FROM Pedidos
  WHERE valor_total > 500
);
```

**Objetivo:** Listar clientes que fizeram pedidos acima de R$ 500. A **subquery retorna múltiplos IDs** (ex: 1, 3, 5), e IN verifica se cada cliente.id pertence a esse conjunto.

✓ **Resultado: Clientes cujos IDs estão na lista de pedidos > 500**

## 3 Subquery com ANY: Comparar com Qualquer Valor

```
SELECT nome, valor_total
FROM Pedidos
WHERE valor_total > ANY (
  SELECT valor_total FROM Pedidos
  WHERE id_cliente = 5
);
```

**Objetivo:** Listar pedidos com valor maior que QUALQUER pedido do cliente 5. Se cliente 5 tem pedidos de 100, 200, 300, retorna pedidos > 100 (o menor). **ANY = "maior que o mínimo"**.

✓ **Resultado: Pedidos com valor > mínimo do cliente 5**

**Próximo slide:** Veja os dois últimos exemplos (ALL para comparar com todos os valores, e Subquery no FROM para criar tabelas temporárias/derivadas) na continuação desta aula.

# Subqueries: Exemplos Práticos - Parte 2

## 4 Subquery com ALL: Comparar com Todos os Valores

```
SELECT nome, valor_total
FROM Pedidos
WHERE valor_total > ALL (
  SELECT valor_total FROM Pedidos
  WHERE id_cliente = 5
);
```

**Objetivo:** Listar pedidos com valor maior que TODOS os pedidos do cliente 5. Se cliente 5 tem pedidos de 100, 200, 300, retorna pedidos > 300 (o maior). **ALL = "maior que o máximo"**.

**✓ Resultado: Pedidos com valor > máximo do cliente 5 (muito restritivo!)**

## 5 Subquery no FROM: Tabela Temporária/Derivada

```
SELECT cliente_nome, total_pedidos, valor_medio
FROM (
  SELECT c.nome AS cliente_nome,
         COUNT(pd.id) AS total_pedidos,
         AVG(pd.valor_total) AS valor_medio
  FROM Clientes c
  LEFT JOIN Pedidos pd ON c.id = pd.id_cliente
  GROUP BY c.id, c.nome
) AS cliente_stats
WHERE total_pedidos > 0;
```

**Objetivo:** Análise em múltiplas etapas. A **subquery cria uma tabela temporária** com agregações (total_pedidos, valor_medio), depois a consulta externa filtra essa tabela. Permite lógica complexa em camadas.

**✓ Resultado: Clientes com agregações, filtrados por total_pedidos > 0**

# Atividade Prática 2: Subqueries - Parte 1

## Tarefa Principal

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

**"Listar todos os clientes que fizeram um valor total de compra ACIMA da média geral de todos os pedidos."**

## Passo a Passo: Primeiros 3 Passos

PASSO 1

### Calcule a Média Geral

Primeiro, descubra qual é a média de valor_total de TODOS os pedidos. Execute esta consulta isoladamente:

```
SELECT AVG(valor_total)
FROM Pedidos;
```

PASSO 2

### Estrutura Base: Agrupe Clientes

Crie uma consulta que agrupa clientes e calcula o valor total de compras:

```
SELECT c.nome, SUM(pd.valor_total)
FROM Clientes c
JOIN Pedidos pd ON c.id = pd.id_cliente
GROUP BY c.id, c.nome;
```

PASSO 3

### Adicione a Subquery no HAVING

Use a subquery para filtrar apenas clientes com valor acima da média:

```
HAVING SUM(pd.valor_total) > (
  SELECT AVG(valor_total)
  FROM Pedidos
)
```

→ **Próximo slide:** Veja a continuação com os Passos 4-6 (Consulta Completa, Teste e Validação, Documentação) e o Desafio Extra para alunos avançados.

# Atividade Prática 2: Subqueries - Parte 2

## Passo a Passo Detalhado (Continuação: Passos 4-6)

PASSO 4

### Consulta Completa

Combine tudo em uma consulta completa com ORDER BY:

```
SELECT c.nome,
       SUM(pd.valor_total) AS Total_Compras
FROM Clientes c
JOIN Pedidos pd ON c.id = pd.id_cliente
GROUP BY c.id, c.nome
HAVING SUM(pd.valor_total) > (
  SELECT AVG(valor_total) FROM Pedidos
)
ORDER BY Total_Compras DESC;
```

PASSO 5

### Teste e Valide

Execute a consulta. Verifique: Quantos clientes aparecem? Os valores fazem sentido? Compare com a média calculada no Passo 1. Todos os valores devem ser maiores que a média.

PASSO 6

### Documente e Reflita

Adicione comentários no seu código explicando cada parte. Reflita: **Por que usar HAVING em vez de WHERE?** (Resposta: Porque SUM é uma agregação, e WHERE não funciona com agregações)

## ⭐ Desafio Extra para Alunos Avançados

**Modificação 1:** Altere a consulta para listar clientes com valor ACIMA da média de sua categoria de cliente (ex: clientes de São Paulo vs Rio de Janeiro).

**Modificação 2:** Crie uma subquery adicional que mostre a diferença entre o valor total do cliente e a média geral:

```
SELECT c.nome,
       SUM(pd.valor_total) AS Total_Compras,
       (SELECT AVG(valor_total) FROM Pedidos) AS Media_Geral,
       (SUM(pd.valor_total) -
        (SELECT AVG(valor_total) FROM Pedidos)) AS Diferenca
FROM Clientes c
JOIN Pedidos pd ON c.id = pd.id_cliente
GROUP BY c.id, c.nome
HAVING SUM(pd.valor_total) > (
  SELECT AVG(valor_total) FROM Pedidos
)
ORDER BY Diferenca DESC;
```

# Funções de Janela: Conceito e Sintaxe

Uma **função de janela (window function)** é uma função SQL que opera sobre um **subconjunto de linhas (janela)** e retorna um valor para cada linha, **mantendo todas as linhas originais**. Diferente de GROUP BY que agrupa e reduz linhas, window functions calculam valores agregados ou ranqueados **sem perder detalhe de cada linha**.

**Exemplo:** Você quer saber o valor total de vendas acumulado até cada data, mas mantendo cada venda como uma linha separada. Window functions fazem isso. GROUP BY não consegue.

## Sintaxe Básica: OVER()

```
SELECT coluna1, coluna2,
  FUNÇÃO_JANELA() OVER (
    PARTITION BY coluna_agrupamento
    ORDER BY coluna_ordenacao
  ) AS resultado_janela
FROM tabela;
```

**PARTITION BY:** Divide dados em grupos. Sem ele, toda tabela é uma janela.
**ORDER BY:** Define ordem dentro da janela. Essencial para LAG/LEAD e somas cumulativas.

### Ranqueamento

Atribui posição a cada linha.

`ROW_NUMBER()`
Sequencial (1,2,3,4...)
`RANK()`
Com empates (1,2,2,4...)
`DENSE_RANK()`
Sem saltos (1,2,2,3...)

### Agregação

Calcula agregações mantendo linhas.

`SUM()`
Soma cumulativa
`AVG()`
Média em janela
`COUNT()`
Contagem em janela

### Deslocamento

Acessa valores de outras linhas.

`LAG()`
Linha anterior
`LEAD()`
Próxima linha
`FIRST_VALUE()`
Primeiro da janela

## Window Functions vs GROUP BY: Diferença Crítica

# Funções de Janela: Exemplos Práticos

### 1. ROW_NUMBER() - Numeração Sequencial

```
SELECT
  p.categoria,
  p.nome,
  COUNT(pd.id) AS vendas,
  ROW_NUMBER() OVER (
    PARTITION BY p.categoria
    ORDER BY COUNT(pd.id) DESC
  ) AS posicao
FROM Produtos p
LEFT JOIN Pedidos pd ON p.id = pd.id_produto
GROUP BY p.id, p.nome, p.categoria
```

**ROW_NUMBER()** numera sequencialmente cada linha dentro de cada categoria. Sem saltos mesmo com empates.

💡 **Insight: Identifica o 1º, 2º, 3º produto mais vendido em cada categoria.**

### 2. RANK() - Ranqueamento com Empates

```
SELECT
  c.nome AS cliente,
  SUM(pd.valor_total) AS total,
  RANK() OVER (
    ORDER BY SUM(pd.valor_total) DESC
  ) AS posicao_geral
FROM Clientes c
LEFT JOIN Pedidos pd ON c.id = pd.id_cliente
GROUP BY c.id, c.nome
ORDER BY total DESC
```

**RANK()** permite empates. Se dois clientes têm o mesmo total, ambos recebem posição 1, e o próximo é 3.

💡 **Insight: Ranqueia clientes por valor total de compras, respeitando empates.**

### 3. SUM() OVER - Soma Cumulativa

```
SELECT
  pd.data_pedido,
  pd.valor_total,
  SUM(pd.valor_total) OVER (
    ORDER BY pd.data_pedido
    ROWS BETWEEN UNBOUNDED PRECEDING
      AND CURRENT ROW
  ) AS vendas_acumuladas
FROM Pedidos pd
ORDER BY pd.data_pedido
```

**SUM() OVER com ROWS** calcula a soma acumulada até a linha atual. Mostra crescimento ao longo do tempo.

💡 **Insight: Visualiza como as vendas crescem dia a dia. Tendência de crescimento?**

### 4. AVG() OVER - Comparação com Média

```
SELECT
  c.nome,
  pd.data_pedido,
  pd.valor_total,
  AVG(pd.valor_total) OVER (
    PARTITION BY c.id
  ) AS media_cliente,
  CASE
    WHEN pd.valor_total > AVG(pd.valor_total)
      OVER (PARTITION BY c.id)
    THEN 'Acima'
    ELSE 'Abaixo'
  END AS categoria
FROM Clientes c
JOIN Pedidos pd ON c.id = pd.id_cliente
```

**AVG() OVER com PARTITION BY** calcula a média por cliente e compara cada pedido com essa média.

💡 **Insight: Identifica pedidos anormais (muito acima ou abaixo da média do cliente).**

### 5. LAG() e LEAD() - Comparação Temporal

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

**LAG()** retorna valor anterior. **LEAD()** retorna próximo. Compara pedidos sequenciais.

💡 **Insight: Detecta tendências (pedidos crescentes/decrescentes) e padrões de compra.**

### 6. DENSE_RANK() - Ranqueamento Sem Saltos

```
SELECT
  p.categoria,
  p.nome,
  COUNT(pd.id) AS vendas,
  DENSE_RANK() OVER (
    PARTITION BY p.categoria
    ORDER BY COUNT(pd.id) DESC
  ) AS rank_denso
FROM Produtos p
LEFT JOIN Pedidos pd ON p.id = pd.id_produto
GROUP BY p.id, p.nome, p.categoria
ORDER BY p.categoria, rank_denso
```

**DENSE_RANK()** como RANK(), mas sem saltos. Se há empate em 1º, próximo é 2º (não 3º).

💡 **Insight: Ranqueia produtos por categoria sem lacunas numéricas. Mais limpo visualmente.**

# Atividade Prática 3: Funções de Janela

Trabalhem em **duplas** para completar duas tarefas práticas usando **Window Functions**. Essas funções são poderosas para análise temporal, ranqueamento e comparações sem perder o detalhe de cada linha.

## Tarefa 1: Ranqueamento com RANK

Ranquear **todos os produtos por volume de vendas** dentro de cada categoria. Produtos com mesmo volume recebem mesmo rank.

### Passo a Passo:

- **Passo 1:** Contar quantas vezes cada produto foi vendido (COUNT de pedidos por produto)
- **Passo 2:** Usar **RANK() OVER (PARTITION BY categoria ORDER BY COUNT DESC)** para ranquear dentro de cada categoria
- **Passo 3:** Selecionar: categoria, nome do produto, total de vendas, rank
- **Passo 4:** Testar: há produtos com mesmo rank? (significa empate em vendas)

**💡 Dica: PARTITION BY vs GROUP BY**

**PARTITION BY** divide a janela mas **mantém todas as linhas**. **GROUP BY** agrupa e reduz linhas. Aqui usamos PARTITION BY porque queremos ver cada produto com seu rank.

## Tarefa 2: Soma Cumulativa com SUM

Calcular a **soma cumulativa de vendas ao longo do tempo**. Mostrar como as vendas crescem dia a dia.

### Passo a Passo:

- **Passo 1:** Selecionar data do pedido, valor do pedido de Pedidos
- **Passo 2:** Usar **SUM(valor) OVER (ORDER BY data_pedido)** para calcular soma cumulativa
- **Passo 3:** Adicionar coluna com a soma cumulativa ao lado do valor individual
- **Passo 4:** Testar: a soma cumulativa sempre aumenta? (ou fica igual se não há vendas naquele dia?)

**💡 Dica: ORDER BY Define a Ordem da Janela**

**ORDER BY data_pedido** faz a janela "deslizar" cronologicamente. Sem ORDER BY, a janela seria toda a tabela (soma total). Com ORDER BY, a janela cresce a cada linha.

# Validação Cruzada e Consolidação Final

## Atividade de Validação Cruzada Entre Duplas

Agora que todas as duplas completaram as três atividades práticas (JOINs, Subqueries, Window Functions), vamos validar o trabalho uns dos outros:

- **Passo 1:** Troque suas consultas SQL com outra dupla. Cada dupla recebe o trabalho de outra.
- **Passo 2:** Leia o código da outra dupla e verifique se está correto usando o checklist abaixo.
- **Passo 3:** Teste o código no seu banco de dados. Os resultados fazem sentido?
- **Passo 4:** Forneça feedback construtivo: "Achei bom porque...", "Poderia melhorar em...", "Uma alternativa seria..."

## Checklist de Erros Comuns a Procurar

### Sintaxe SQL

Há erros de digitação? Parênteses balanceados? Vírgulas nos lugares certos? Nomes de tabelas/colunas corretos?

### Lógica da Consulta

A consulta responde à pergunta? Os JOINs estão corretos? A subquery está no lugar certo? O PARTITION BY faz sentido?

### Performance

A consulta é eficiente? Há JOINs desnecessários? Filtros (WHERE) estão no lugar certo? Índices poderiam ajudar?

### Documentação

Há comentários explicando o código? É fácil entender o que cada parte faz? Um colega conseguiria manter esse código?

### Dados e Resultados

Os resultados fazem sentido? Há valores nulos inesperados? Duplicatas? A quantidade de linhas está correta?

### Legibilidade

O código está bem formatado? Aliases são significativos? Nomes de colunas são claros? Fácil de ler?

**Próximo slide:** Veja a importância de documentação em SQL, síntese final das três ferramentas aprendidas (JOINs, Subqueries, Window Functions) e conclusão da aula.

# Consolidação Final: Síntese e Conclusão

## A Importância de Documentação em SQL

**Código bem documentado é código profissional.** Sempre adicione comentários explicando o "por quê" e o "como" de sua consulta. Exemplo:

```
-- Objetivo: Listar clientes com valor total acima da média
-- Método: Subquery de linha única para calcular média dinâmica
-- Data: 2025-01-15 | Autor: João Silva

SELECT c.nome,
       SUM(pd.valor_total) AS total_compras
FROM Clientes c
INNER JOIN Pedidos pd ON c.id = pd.id_cliente
GROUP BY c.id, c.nome
HAVING SUM(pd.valor_total) > (
  -- Subquery: Calcula média geral de todos os pedidos
  SELECT AVG(valor_total) FROM Pedidos
)
ORDER BY total_compras DESC;
```

## Síntese Final: O Que Você Aprendeu Hoje

Você dominou **três ferramentas SQL avançadas**: **JOINs Múltiplos** para combinar dados de múltiplas tabelas, **Subqueries** para análise condicional dinâmica, e **Funções de Janela** para insights analíticos mantendo detalhe de cada linha.

Essas três ferramentas são o que **distingue um operador SQL de um analista de dados profissional**. Um operador executa consultas simples. Um analista de dados **extrai insights, identifica padrões, responde perguntas complexas de negócio** e comunica descobertas de forma clara.

Você agora tem as habilidades para fazer análises profissionais. Continue praticando, documentando seu código, e sempre perguntando: **"Que insight essa consulta me traz?"**

🎓 **Parabéns!** Você completou a Aula 09 da UC3 sobre Consultas Avançadas em SQL (JOINs Múltiplos, Subqueries e Funções de Janela).

✨ **Próximo passo: Aula 10 sobre Otimização de Consultas e Índices. Você está no caminho para se tornar um analista de dados profissional!** 🚀