ChatGPT

Guia de Estudo – Funções de Janela em SQL (Melhoria de Nota dos Alunos)

Índice

1. Introdução

As **funções de janela** (Window Functions) em SQL são ferramentas poderosas para análises avançadas de dados. Diferentemente de uma agregação tradicional com `GROUP BY` — que comprime várias linhas em um único resultado por grupo — as funções de janela permitem calcular valores agregados ou rankings **sem** perder a granularidade das linhas originais [1] [2]. Em outras palavras, elas produzem um resultado para **cada linha** do conjunto, possivelmente repetindo valores de cálculo dentro de cada partição, ao invés de reduzir cada partição a uma única linha de resultado [2]. Isso viabiliza cálculos como somas cumulativas, médias móveis, diferenças entre linhas consecutivas,

rankings dentro de grupos e muito mais, tudo isso mantendo todas as linhas do conjunto de dados presentes no resultado [1].

Neste guia, exploraremos detalhadamente uma consulta SQL que analisa a melhoria da nota de alunos ao longo do tempo utilizando funções de janela. Usaremos esse exemplo prático para explicar passo a passo a estrutura da query, incluindo o uso de subconsultas, da função `FIRST_VALUE()`, das cláusulas `PARTITION BY` e `ORDER BY`, bem como da cláusula especial `WINDOW` para nomear janelas. Além disso, discutiremos a **ordem de execução** das cláusulas SQL e como isso afeta o comportamento das funções de janela.

Em seguida, faremos uma **comparação** entre a solução com funções de janela e uma solução alternativa tradicional (por exemplo, usando subconsultas correlacionadas ou junções) para entender questões de eficiência e legibilidade.

O guia também apresenta as principais funções de janela disponíveis nos SQL modernos – `FIRST_VALUE`, `LAST_VALUE`, `ROW_NUMBER`, `RANK`, `DENSE_RANK`, `LAG`, `LEAD` e `NTILE` – explicando para que serve cada uma, sua sintaxe e exemplos de uso prático em cenários simples (como dados de alunos, vendas, salários, etc.). Ilustraremos vários exemplos práticos de cada função com pequenos conjuntos de dados para fixar as ideias.

Por fim, incluímos uma seção de **exercícios resolvidos** (com soluções passo a passo) e **exercícios propostos** (com as soluções fornecidas) para que você possa praticar e consolidar seu entendimento sobre funções de janela em SQL. Este formato permite que você tente resolver os problemas e, em seguida, confira as soluções e explicações detalhadas para verificar seu raciocínio.

Vamos começar desvendando a consulta principal de análise de melhoria de notas dos alunos, passo a passo.

2. Explicação da Consulta SQL de Melhoria de Nota

Para contextualizar, imagine que temos uma tabela `NotasAlunos` com registros das notas que cada aluno tirou em diferentes provas ao longo do tempo. A estrutura simplificada da tabela é:

```
NotasAlunos (AlunoID, Nome, DataProva, Nota)
```

- **AlunoID:** Identificador do aluno (por exemplo, número de estudante)
- **Nome:** Nome do aluno
- **DataProva:** Data em que a prova foi realizada
- **Nota:** Nota obtida pelo aluno nessa prova

Nosso objetivo é analisar a **melhoria de nota** de cada aluno, comparando a nota da sua primeira prova com a nota da sua última prova. Em outras palavras, queremos saber para cada aluno qual era a nota inicial e qual foi a nota final, e possivelmente calcular a diferença entre elas (ou indicar se a nota aumentou, diminuiu ou manteve-se igual).

A consulta SQL fornecida pelo utilizador realiza exatamente essa tarefa usando funções de janela. Vamos apresentá-la e então dissecar cada parte:

2.1. Estrutura Geral da Query

Aqui está a consulta SQL principal que iremos explicar:

```sql
SELECT
    sub.AlunoID,
    sub.Nome,
    sub.Nota      AS Nota_Final,
    sub.Nota_Inicial,
    sub.Nota - sub.Nota_Inicial AS Diferenca_Nota
FROM (
    SELECT
        AlunoID,
        Nome,
        DataProva,
        Nota,
        FIRST_VALUE(Nota) OVER w AS Nota_Inicial,
        ROW_NUMBER()     OVER (PARTITION BY AlunoID ORDER BY DataProva DESC)
AS rn
    FROM NotasAlunos
    WINDOW w AS (PARTITION BY AlunoID ORDER BY DataProva ASC)
) AS sub
WHERE sub.rn = 1;
```

Vamos entender a lógica geral antes de detalhar cláusula por cláusula:

- A query externa (`SELECT ... FROM ( ... ) sub WHERE sub.rn = 1`) seleciona dados de uma subconsulta ali apelidada de `sub` , e filtra (`WHERE sub.rn = 1`) para considerar apenas determinadas linhas da subconsulta. A subconsulta, por sua vez, é onde ocorre a maior parte do cálculo usando funções de janela.
- A subconsulta interna (`SELECT ... FROM NotasAlunos WINDOW w AS (...)`) seleciona todos os registros da tabela `NotasAlunos` , mas adiciona duas colunas calculadas importantes:
- `Nota_Inicial` : calculada com `FIRST_VALUE(Nota) OVER w` , para obter a primeira nota do aluno (definiremos *w* adiante).
- `rn` : um número de linha gerado por `ROW_NUMBER() OVER (PARTITION BY AlunoID ORDER BY DataProva DESC)` , para identificar qual registro é o "último" de cada aluno.
- A cláusula `WINDOW w AS (PARTITION BY AlunoID ORDER BY DataProva ASC)` define uma janela nomeada `w` que é reutilizada dentro da subconsulta. Essa janela *w* especifica que as funções de janela associadas a ela serão calculadas **por aluno** (`PARTITION BY AlunoID`) e ordenadas pela data da prova em ordem ascendente (cronológica).
- No cálculo de `FIRST_VALUE(Nota) OVER w` , a janela *w* fará com que `FIRST_VALUE` (função de janela que retorna o primeiro valor de uma partição) nos dê a **primeira Nota de cada Aluno**, considerando a ordem ascendente de DataProva – ou seja, a nota da primeira prova que aquele aluno realizou.
- No cálculo de `ROW_NUMBER() OVER (PARTITION BY AlunoID ORDER BY DataProva DESC)` , não usamos a janela nomeada *w* porque a ordem é diferente (descendente). Em vez disso, definimos diretamente na função `ROW_NUMBER` que queremos numerar as linhas de cada aluno ordenando da prova mais recente para a mais antiga. Assim, para cada aluno, a prova mais

recente receberá `ROW_NUMBER = 1`, a segunda mais recente `ROW_NUMBER = 2`, e assim por diante.
- Depois de calculados esses valores na subconsulta, a query externa filtra `WHERE sub.rn = 1` para pegar **somente a linha de cada aluno que corresponde à sua prova mais recente** (pois definimos que `rn = 1` marca a última prova de cada aluno). Dessa forma, o resultado final terá no máximo uma linha por aluno.
- Por fim, a query externa seleciona o `AlunoID`, `Nome`, a `Nota` (renomeada como `Nota_Final` para indicar que é a nota da prova final) e `Nota_Inicial`. Calculamos também uma coluna `Diferenca_Nota` como `sub.Nota - sub.Nota_Inicial`, que representa o quanto a nota do aluno mudou (aumentou ou diminuiu) entre a primeira e a última prova. Se essa diferença for positiva, significa melhoria; se negativa, piora; se zero, manteve-se igual. (*Poderíamos até usar uma expressão* `CASE` *para traduzir isso em textos como "Melhorou"/"Piorou", mas aqui manteremos o valor numérico da diferença.*)

Agora vamos dissecar cada parte importante dessa consulta e entender os detalhes técnicos:

2.2. Subconsulta Interna e Cálculo da Nota Inicial (FIRST_VALUE)

Observe que toda a lógica de cálculo acontece dentro da subconsulta (o trecho entre parênteses, após o `FROM (` e antes do `) AS sub`). Essa subconsulta interna é essencialmente:

```
SELECT
    AlunoID,
    Nome,
    DataProva,
    Nota,
    FIRST_VALUE(Nota) OVER w AS Nota_Inicial,
    ROW_NUMBER()     OVER (PARTITION BY AlunoID ORDER BY DataProva DESC) AS
rn
FROM NotasAlunos
WINDOW w AS (PARTITION BY AlunoID ORDER BY DataProva ASC);
```

Vamos entender a coluna calculada `Nota_Inicial`. Ela utiliza a função de janela `FIRST_VALUE(Nota)`. A sintaxe geral de `FIRST_VALUE` é:

```
FIRST_VALUE(expr) OVER (PARTITION BY ... ORDER BY ... [FRAME])
```

- `FIRST_VALUE(expr)` retorna o primeiro valor de `expr` dentro da janela (ou partição) definida. Neste caso, `expr` é a coluna `Nota`. Ou seja, queremos o primeiro valor de nota dentro da partição de linhas do aluno.
- `OVER (...)` define a **janela** sobre a qual a função será aplicada. Dentro dos parênteses do OVER, podemos especificar cláusulas como `PARTITION BY` (para determinar os grupos/partições) e `ORDER BY` (para ordenar as linhas dentro de cada partição, determinando assim quem é "primeiro" ou "último").
- Em nosso caso, usamos `OVER w`, onde *w* é uma janela nomeada definida pela cláusula `WINDOW` (falaremos dela já a seguir). A definição de *w* é `(PARTITION BY AlunoID ORDER BY DataProva ASC)`. Isso significa que a partição é por

aluno (`AlunoID`) e, dentro de cada aluno, as linhas são ordenadas da data mais antiga para a mais recente (ASCendente).
- Portanto, `FIRST_VALUE(Nota) OVER w` para um dado aluno pegará a Nota da primeira linha, segundo a ordem ascendente de DataProva, ou seja, a **Nota na primeira prova daquele aluno** [3]. Esse valor será repetido em *todas* as linhas desse aluno na subconsulta (cada registro do aluno "sabe" qual foi sua primeira nota, graças à função de janela).

Exemplo conceitual: se para o aluno **João (ID 101)** as notas e datas forem: 10 (2021-01-10), 12 (2021-06-20), 15 (2022-02-15), o `FIRST_VALUE(Nota) OVER w` (particionado por João, ordenado por data asc) resultará em 10 para todas essas linhas, pois 10 é a primeira nota de João. Já para a aluna **Maria (ID 102)**, com notas 8 (2021-03-05) e 13 (2021-11-30), `FIRST_VALUE(Nota) OVER w` dará 8 em ambas as linhas de Maria.

Essa abordagem – obter o primeiro valor por partição e repeti-lo em cada linha – é típica do uso de funções de janela: **cada linha "enxerga" informações agregadas de seu grupo sem que as linhas sejam colapsadas em uma única** (como aconteceria num GROUP BY). Assim podemos comparar valores linha a linha, o que é útil para calcular diferenças, por exemplo, entre a nota atual e a primeira nota de cada aluno.

2.3. Uso de PARTITION BY e ORDER BY na Função de Janela

Vale reforçar a importância das cláusulas `PARTITION BY` e `ORDER BY` dentro da especificação da janela:

- `PARTITION BY AlunoID`: faz com que a função de janela seja **reiniciada para cada Aluno**. Cada valor diferente de `AlunoID` define uma partição separada. No nosso caso, isso garante que o cálculo de `FIRST_VALUE(Nota)` seja feito individualmente para cada aluno, e não misture notas de alunos distintos. Em outras palavras, cada aluno terá sua própria "janela" de notas.
- `ORDER BY DataProva ASC`: dentro de cada partição (ou seja, dentro de cada aluno), define a ordem das linhas por Data da prova, do mais antigo para o mais recente. Assim, "primeiro valor" corresponde à nota na menor Data (primeira cronologicamente). Se invertêssemos para DESC, o `FIRST_VALUE` nos daria a última nota (veremos a seguir uma abordagem alternativa para última nota).
- A **combinação** de PARTITION BY e ORDER BY é o que permite que funções como FIRST_VALUE, LAST_VALUE, LAG, LEAD etc. funcionem de forma significativa, pois define **o contexto** (partição) e **a posição** (ordem dentro do contexto) de cada linha [4]. Sem PARTITION BY, a janela seria toda a tabela; sem ORDER BY, "primeiro" ou "último" não fariam sentido determinístico.

No nosso exemplo, porque particionamos por AlunoID e ordenamos por Data crescente, para cada aluno a função FIRST_VALUE pega consistentemente a primeira nota no tempo.

2.4. Cláusula WINDOW Nomeada

Uma particularidade interessante da sintaxe usada é a cláusula `WINDOW` no final da subconsulta:

```
WINDOW w AS (PARTITION BY AlunoID ORDER BY DataProva ASC)
```

A cláusula WINDOW (introduzida no padrão SQL e presente em SGBDs como PostgreSQL e SQL Server 2022+) permite **nomear uma definição de janela** para reutilizá-la em múltiplas funções de janela na

mesma consulta [5] [6]. No nosso caso, definimos uma janela chamada `w`. Assim, poderíamos ter várias colunas calculadas usando `OVER w` sem repetir toda a cláusula `PARTITION BY AlunoID ORDER BY DataProva ASC` em cada uma delas.

Isso melhora a legibilidade e evita repetição. Imagine que além de `FIRST_VALUE(Nota)` quiséssemos também `LAST_VALUE(Nota)` na mesma partição e ordem – poderíamos usar o mesmo *w*. Sem a cláusula WINDOW nomeada, teríamos que escrever `FIRST_VALUE(Nota) OVER (PARTITION BY AlunoID ORDER BY DataProva ASC)` e depois `LAST_VALUE(Nota) OVER (PARTITION BY AlunoID ORDER BY DataProva ASC ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING)` e assim por diante repetindo a partição. Com a janela *w* definida, simplificamos para `FIRST_VALUE(Nota) OVER w` e poderíamos usar `LAST_VALUE(Nota) OVER w2` se definíssemos outra janela (por exemplo, w2 para a mesma partição mas com ORDER BY diferente, ou com frame diferente – mais sobre frames adiante). A cláusula WINDOW é, portanto, uma conveniência sintática que aumenta a clareza da consulta, especialmente quando se usam múltiplas funções de janela com a mesma especificação [7].

No nosso exemplo, *w* encapsula "por aluno, ordenado por data ascendente". Reutilizamos *w* em `FIRST_VALUE(Nota) OVER w`. Já para `ROW_NUMBER()`, optamos por não usar *w* pois a ordem precisava ser **diferente** (descendente). Poderíamos até definir outra janela, digamos `WINDOW w_desc AS (PARTITION BY AlunoID ORDER BY DataProva DESC)`, e então usar `ROW_NUMBER() OVER w_desc` para obter o mesmo efeito. Neste caso, para didática, a query explicitou o PARTITION BY/ORDER BY diretamente em `ROW_NUMBER()`.

Resumo até aqui: Dentro da subconsulta, para cada linha temos: - `Nota_Inicial`: a nota da primeira prova do aluno (constante em todas as linhas do mesmo aluno) graças a `FIRST_VALUE` na janela *w* (particionada por aluno, ordenada por data ascendente). - `rn`: um número sequencial dentro de cada aluno, começando em 1 na prova mais recente, graças a `ROW_NUMBER` com partição por aluno e ordenação descendente da data.

2.5. Filtrando a Última Prova de Cada Aluno (ROW_NUMBER e WHERE)

Agora entra o papel da coluna `rn`. Ela é derivada da função de janela `ROW_NUMBER()`. Essa função numera as linhas dentro de cada partição na ordem especificada: - `ROW_NUMBER() OVER (PARTITION BY AlunoID ORDER BY DataProva DESC)` gera uma numeração começando em 1 para a linha de maior DataProva (a prova mais recente) de cada aluno, 2 para a segunda mais recente, e assim por diante. Cada aluno recomeça a contagem a partir de 1, porque a partição é por AlunoID.

Ao calcular `rn` assim, nós marcamos explicitamente qual linha é a última prova de cada aluno (`rn = 1`). Na subconsulta, todas as linhas do aluno aparecerão (cada prova com sua respectiva rn e com a nota_inicial repetida), mas quando envolvemos isso numa subconsulta, podemos facilmente filtrar *apenas a última linha de cada aluno* usando `WHERE rn = 1` na query externa. Isso efetivamente nos dá um resultado agregado por aluno, mas notem que não usamos `GROUP BY` para isso – fizemos a filtragem após obter os valores por janela.

Por que não usar diretamente LAST_VALUE? – Uma dúvida comum seria: já que existe `FIRST_VALUE`, não existe `LAST_VALUE`? Poderíamos usar `LAST_VALUE(Nota) OVER (PARTITION BY AlunoID ORDER BY DataProva ASC)` para pegar a última nota? A função `LAST_VALUE()` existe e retorna o último valor dentro da janela [8], porém ela tem uma sutileza: por padrão, a janela (o *frame*) das funções de janela considera as linhas **desde o**

início da partição até a linha atual (se não especificarmos outro frame). Assim, `LAST_VALUE` sem ajustes, quando usada em uma ordenação ascendente, devolveria o valor da *linha atual* (porque até aquela linha, a última é ela mesma). Para que `LAST_VALUE` retorne o valor da última linha da partição inteira independente da linha atual, precisaríamos especificar um frame que vá da linha atual até o fim (por exemplo, `ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`). Outra alternativa é a que usamos: ordenar ao contrário e pegar o FIRST_VALUE da ordem inversa. De fato, `FIRST_VALUE(Nota) OVER (PARTITION BY AlunoID ORDER BY DataProva DESC)` nos daria a última nota diretamente (pois "primeiro" na ordem descendente é o maior data, ou seja, a última prova). Poderíamos ter feito isso para obter `Nota_Final`. Não o fizemos na subconsulta porque, de qualquer modo, precisaríamos filtrar para obter apenas uma linha por aluno.

Ao usar `ROW_NUMBER`, conseguimos tanto identificar a última prova (`rn = 1`) quanto eliminar as demais via filtro. Outra maneira comum seria usar `RANK()` ou `DENSE_RANK()` numa subconsulta e filtrar pelo rank = 1, ou ainda usar subconsultas correlacionadas para pegar MAX(Data) por aluno. Cada método tem seus méritos, mas `ROW_NUMBER` é frequentemente a opção mais direta para **filtrar a linha "top 1" dentro de cada partição** quando não queremos empates (aqui "top 1" por data, ou seja, o registro mais recente de cada aluno).

Portanto, com `WHERE sub.rn = 1` na query externa, ficamos somente com as linhas marcadas como 1 – isto é, apenas o registro de cada aluno que corresponde à última prova dele. Nessa linha, já calculamos anteriormente a `Nota_Inicial` do aluno (repetida nas demais, mas aqui haverá só essa linha visível) e temos também a `Nota` da prova final (renomeada como `Nota_Final` no SELECT externo para clareza).

2.6. Cálculo da Melhoria de Nota

A última parte da query externa seleciona e calcula os campos desejados no resultado final:

```
SELECT
    sub.AlunoID,
    sub.Nome,
    sub.Nota      AS Nota_Final,
    sub.Nota_Inicial,
    sub.Nota - sub.Nota_Inicial AS Diferenca_Nota
...
```

Aqui, `Nota_Final` é apenas um alias dado à nota da última prova (que no subselect era `Nota`). `Nota_Inicial` já foi calculada como explicado. E `Diferenca_Nota` é uma expressão que subtrai a nota inicial da nota final.

- Se `Diferenca_Nota` for **positivo**, significa que a Nota_Final é maior que a Nota_Inicial – ou seja, o aluno melhorou sua nota em relação à primeira prova.
- Se for **negativo**, a nota final ficou abaixo da inicial (desempenho piorou).
- Se for zero, o aluno tirou a mesma nota na primeira e na última prova.

Como exemplo hipotético, suponha que no resultado final tenhamos: AlunoID 101, Nome "João", Nota_Inicial = 10, Nota_Final = 15, Diferenca_Nota = 5. Isso indica que João começou com 10 e terminou com 15, uma melhoria de 5 pontos. Se outro aluno, Maria, teve Nota_Inicial = 8 e Nota_Final = 7, a diferença seria -1 (queda de um ponto).

Observação: Poderíamos apresentar a melhoria de forma categórica usando `CASE WHEN sub.Nota > sub.Nota_Inicial THEN 'Melhorou' WHEN sub.Nota < sub.Nota_Inicial THEN 'Piorou' ELSE 'Igual' END AS Situacao_Melhoria`. Isso não foi explicitamente solicitado, mas é uma possibilidade de pós-processamento com base nos valores calculados.

Recapitulando em termos de etapas lógicas, a consulta faz o seguinte para cada aluno: 1. Dentro da subconsulta, calcula a nota inicial (primeira) e anota cada prova com um número inverso da ordem cronológica. 2. Seleciona apenas a prova de número 1 (a mais recente). 3. Mostra essa prova com a nota inicial do aluno e calcula a diferença.

O resultado final é uma tabela onde cada linha corresponde a um aluno, mostrando sua nota inicial, nota final e a diferença entre elas.

3. Ordem de Execução do SQL e Impacto nas Funções de Janela

Para compreender por que a consulta acima funciona e como as funções de janela se encaixam no processamento SQL, é importante entender a **ordem lógica de execução** de uma query SQL. A ordem em que escrevemos as cláusulas (`SELECT`, `FROM`, `WHERE`, etc.) **não** é a ordem em que o SGBD as executa logicamente.

A ordem típica de execução de uma instrução `SELECT` é a seguinte [9] [10]:

1. **FROM** (incluindo `JOIN`s): determina as tabelas de origem e combinações (join) para formar o conjunto de dados inicial.
2. **WHERE**: filtra as linhas do conjunto resultante do FROM de acordo com as condições Booleanas.
3. **GROUP BY**: agrupa as linhas em conjuntos baseados em valores iguais de certas colunas.
4. **HAVING**: filtra grupos (após o GROUP BY) com base em condições sobre agregações.
5. **SELECT**: especifica quais colunas ou expressões serão retornadas. Aqui é onde as **funções de janela** são avaliadas – na mesma etapa lógica do SELECT, após processar os filtros acima [11] [12].
6. **WINDOW**: se houver uma cláusula WINDOW nomeada, ela é lida em conjunto com o SELECT (na verdade, a definição da janela já é conhecida e aplicada nas funções de janela na etapa 5). Conceitualmente podemos posicionar a aplicação das funções de janela aqui, após o SELECT/HAVING e antes de ORDER BY [9]. (Algumas descrições colocam "WINDOW" como etapa separada, mas pense que as funções de janela já foram computadas nas colunas selecionadas).
7. **QUALIFY** (cláusula disponível em alguns SGBDs como Snowflake, Teradata): aplica filtros após cálculo das funções de janela. *Nota:* Em sistemas que não suportam QUALIFY (como SQL Server, Postgres), consegue-se o mesmo efeito usando subconsultas ou CTEs – como fizemos ao filtrar `rn = 1` depois de calcular dentro da subconsulta [10].
8. **ORDER BY**: ordena o resultado final de acordo com as colunas especificadas (que podem incluir colunas derivadas ou até funções de janela já calculadas).
9. **LIMIT/OFFSET** (ou `FETCH FIRST n`): limita o número de linhas do resultado final (após toda a ordenação e cálculo).

Essa ordem explica por que, por exemplo, **não podemos usar diretamente uma função de janela em uma cláusula WHERE** – porque o WHERE é aplicado na etapa 2, antes de as funções de janela serem calculadas (etapa 5/6). Da mesma forma, não podemos usar alias definidos no SELECT dentro do WHERE, pois o SELECT vem logicamente depois do WHERE.

No contexto da nossa consulta: - A subconsulta interna primeiro faz o `FROM NotasAlunos`. - Em seguida (não havendo WHERE dentro dela), passa direto para o SELECT da subconsulta, onde calcula as colunas incluindo as funções de janela. Note que como não havia nenhum filtro anterior, as funções de janela veem **todas as linhas** da tabela, particionadas por Aluno. Se houvesse, por exemplo, um `WHERE DataProva >= '2022-01-01'` dentro da subconsulta, então a função FIRST_VALUE só consideraria as provas a partir de 2022 (ignorando as anteriores). Isso ilustra como um filtro antes de uma função de janela pode afetar o resultado dela – é fundamental ter claro quais linhas estão "visíveis" para a função de janela na etapa de SELECT [13] [12]. No nosso caso, queríamos todas as provas para pegar a nota inicial correta, então não colocamos filtro ali. - Depois que a subconsulta calcula tudo, ela entrega o conjunto de resultados (todas as provas de todos os alunos, cada uma anotada com Nota_Inicial e rn). - Agora, na query externa, o `FROM (subconsulta) AS sub` está pronto. Em seguida aplicamos o `WHERE sub.rn = 1`. Esse filtro é aplicado **após** termos as colunas calculadas pela subconsulta, inclusive as janelas. Então aqui não há problema em usar `rn` (que veio do SELECT interno) no WHERE externo. - Após o WHERE externo, passamos para o SELECT externo que seleciona e calcula `Diferenca_Nota`. - Se houvesse, por exemplo, um `ORDER BY Diferenca_Nota` no final, ele seria aplicado por último, usando já os valores calculados.

Resumindo: as funções de janela são calculadas **depois** dos filtros de WHERE e HAVING, mas **antes** do ORDER BY final [9]. Isso significa que: - Podemos usar colunas calculadas por janelas no ORDER BY ou SELECT final livremente. - Não podemos usá-las diretamente no WHERE/HAVING da mesma consulta em que são definidas. Em SGBDs que não têm QUALIFY, como mencionado, fazemos o truque de subconsulta ou CTE: primeiro calculamos a janela, depois filtramos externamente (como fizemos com `rn = 1`). - A cláusula WINDOW nomeada não muda a ordem de execução, ela é apenas uma ajuda para não repetir texto. O cálculo das funções de janela de uma cláusula WINDOW ocorre igualmente na etapa de SELECT.

Impacto prático: Ao escrever consultas com funções de janela, lembre-se de aplicar **primeiro** todos os filtros necessários (por exemplo, limitar o período de tempo, ou excluir certos casos) *antes* de usar a função de janela, caso contrário a partição pode incluir linhas indesejadas. No nosso exemplo, se quiséssemos analisar a melhoria apenas em 2022, poderíamos colocar `WHERE YEAR(DataProva) = 2022` dentro da subconsulta *antes* do SELECT interno – mas isso mudaria o significado de "Nota_Inicial" (passaria a ser a primeira nota de 2022, e não a primeira nota de sempre). Portanto, normalmente se deseja que funções como FIRST_VALUE ignorem quaisquer filtros temporais, devemos calculá-las sem filtrar, ou filtrar em um escopo apropriado.

Em suma, a ordem de execução é crucial para entender o comportamento: no nosso caso garantimos que todas as notas de cada aluno estavam presentes quando calculamos a primeira nota com FIRST_VALUE. Depois, marcamos a última linha e filtramos fora da subconsulta. Assim obtivemos o resultado correto. Essa técnica de calcular primeiro com janela e filtrar depois é comum devido à posição das janelas na ordem de execução.

4. Comparação: Query com Funções de Janela vs. Abordagem Alternativa

A solução apresentada aproveita as funções de janela (`FIRST_VALUE` e `ROW_NUMBER`) para obter a nota inicial e a nota final de cada aluno de forma elegante. Vamos agora comparar essa abordagem com uma possível solução alternativa **sem** usar funções de janela, destacando diferenças de eficiência e legibilidade.

Abordagem Alternativa (sem janela): Uma maneira clássica de resolver o mesmo problema (pegar a primeira e última nota de cada aluno) seria: - Encontrar, para cada aluno, qual é a data da primeira prova e a data da última prova. - Associar essas datas de volta às notas correspondentes. - Calcular a diferença.

Por exemplo, poderíamos escrever algo como:

```
SELECT
    A.AlunoID,
    A.Nome,
    F.Nota AS Nota_Inicial,
    L.Nota AS Nota_Final,
    L.Nota - F.Nota AS Diferenca_Nota
FROM
    (SELECT AlunoID,
            MIN(DataProva) AS DataPrimeira,
            MAX(DataProva) AS DataUltima
     FROM NotasAlunos
     GROUP BY AlunoID) AS Base
JOIN NotasAlunos F
     ON F.AlunoID = Base.AlunoID AND F.DataProva = Base.DataPrimeira
JOIN NotasAlunos L
     ON L.AlunoID = Base.AlunoID AND L.DataProva = Base.DataUltima;
```

Vamos entender essa alternativa: - A subconsulta `Base` usa agregação (`GROUP BY AlunoID`) para encontrar a menor data (primeira prova) e a maior data (última prova) de cada aluno. - Depois fazemos **dois JOINs** da tabela original: um para pegar o registro completo da primeira prova (`F` de "first") e outro para o registro da última prova (`L`). - Finalmente calculamos a diferença das notas.

Essa query daria o mesmo resultado pretendido. Contudo, existem diferenças importantes:

- **Legibilidade:** A consulta com funções de janela é, para muitos, mais direta de entender: ela diz “dê-me a primeira nota por aluno e marque o último registro por aluno”. A alternativa com JOINs requer entender que primeiro obtemos datas agregadas e depois ligamos de volta, o que é um pouco mais verboso. A presença de múltiplos JOINs e uma subconsulta de agregação torna a query mais longa e possivelmente menos intuitiva para quem não a escreveu, quando comparada à solução com janela que expressa o cálculo quase que declarativamente.
- **Eficiência:** Depende do SGBD e dos índices disponíveis. A abordagem com janela faz uma única varredura na tabela `NotasAlunos` para calcular `FIRST_VALUE` e `ROW_NUMBER` de todos os alunos de uma vez (possivelmente ordenando internamente por AlunoID, Data). A abordagem alternativa faz: uma varredura/agrupamento para obter min e max datas (isso é uma passagem pela tabela ou um índice de Data), depois dois JOINs adicionais que, na pior hipótese, podem ser duas novas buscas na tabela (embora possivelmente otimizadas por índices em (AlunoID, DataProva)). Em termos de complexidade, ambas as soluções devem ser **linear** no número de linhas da tabela, mas a versão com janelas muitas vezes pode ser executada em um único passo de varredura ordenada. Já a versão alternativa segmenta o trabalho em 3 etapas lógicas (group, join, join). Em bases pequenas, ambas terão desempenho similar; em bases grandes, a versão de janela pode ter vantagem se o otimizador conseguir utilizá-la de forma eficiente, pois evita join extras.

Uma comparação comum é entre janelas e subconsultas correlacionadas. Por exemplo, poderíamos pensar em usar subconsultas no SELECT para pegar a primeira e última nota:

```
SELECT
    AlunoID,
    Nome,
    /* subconsulta correlacionada para primeira nota */
    (SELECT Nota
     FROM NotasAlunos AS N2
     WHERE N2.AlunoID = N1.AlunoID
     ORDER BY DataProva ASC
     LIMIT 1) AS Nota_Inicial,
    /* subconsulta correlacionada para última nota */
    (SELECT Nota
     FROM NotasAlunos AS N2
     WHERE N2.AlunoID = N1.AlunoID
     ORDER BY DataProva DESC
     LIMIT 1) AS Nota_Final
FROM NotasAlunos AS N1
GROUP BY AlunoID, Nome;
```

Nota: A sintaxe `LIMIT 1` é de, por exemplo, MySQL/Postgres; em SQL Server usaríamos `TOP 1` e talvez `OUTER APPLY`. O importante é: subconsultas correlacionadas dentro do SELECT que buscam a primeira/última nota.

Essa abordagem evita os JOINs múltiplos e parece mais compacta, mas tem um problema: a subconsulta será executada para cada linha/agrupamento de `N1`. Com muitos alunos, isso pode ser ineficiente, pois é como um loop aninhado (cada aluno faz uma busca separada). As **funções de janela**, ao contrário, normalmente são otimizadas para calcular todos os valores em conjunto, possivelmente usando partições em memória ou ordenações únicas. Muitos SGBDs conseguem melhorar bastante a performance de queries com janela em comparação a um equivalente com subconsulta correlacionada, especialmente em conjuntos de dados grandes [14].

- **Manutenção e Extensibilidade:** Suponha que queiramos estender a análise – por exemplo, além de primeira e última nota, também calcular a **segunda** prova de cada aluno, ou a média de notas, ou a posição do aluno em cada prova. Com a abordagem de janela, basta adicionar mais funções de janela: `LAG(Nota)`, `AVG(Nota) OVER (PARTITION BY AlunoID)` etc., dentro da mesma subconsulta. Com a abordagem tradicional, teríamos que adicionar mais subconsultas ou junções, complicando ainda mais a query. As funções de janela oferecem um "kit" unificado para cálculos analíticos, tornando o ajuste da consulta mais simples conforme novas exigências aparecem.

Em resumo, a consulta usando funções de janela tende a ser **mais sucinta e declarativa**, facilitando a leitura (uma vez que se compreenda a sintaxe das janelas) e muitas vezes **mais eficiente** do que várias subconsultas correlacionadas ou junções, especialmente em cenários de análises complexas. Estudos de caso mostram que cálculos de totais acumulados ou classificação que antes exigiam subconsultas agora são resolvidos de forma **mais performática com window functions**, pois evitam cálculos repetidos para cada linha [14].

Quando usar qual abordagem? As funções de janela brilham em cenários de análise dentro de grupos e comparação entre linhas. A abordagem tradicional pode ser útil se o SGBD não suporta funções de janela ou por simplicidade em consultas muito básicas. Mas atualmente, dado o suporte amplo (SQL Server, Oracle, PostgreSQL, MySQL 8+, etc. todos suportam window functions) e as vantagens de legibilidade e performance, a solução com funções de janela é geralmente preferível para tarefas como a apresentada.

5. Principais Funções de Janela em SQL

Nesta seção, apresentamos as principais funções de janela disponíveis em SQL e explicadas em português de Portugal, conforme solicitado. Para cada função, explicaremos brevemente o que faz e daremos um exemplo prático simples usando conjuntos de dados fictícios (alunos, vendas, salários, etc.). As funções que abordaremos são:

- **FIRST_VALUE()** – Função de janela de *valor*.
- **LAST_VALUE()** – Função de janela de *valor*.
- **ROW_NUMBER()** – Função de janela de *ranqueamento*.
- **RANK()** – Função de janela de *ranqueamento*.
- **DENSE_RANK()** – Função de janela de *ranqueamento*.
- **NTILE()** – Função de janela de *ranqueamento/distribuição*.
- **LAG()** – Função de janela de *deslocamento (offset)*.
- **LEAD()** – Função de janela de *deslocamento (offset)*.

Essas funções podem ser divididas em três categorias: funções de valor, funções de ranking e funções de offset (deslocamento) [15] [16]. Veremos cada uma delas:

5.1. FIRST_VALUE()

A função `FIRST_VALUE(coluna)` retorna o **primeiro valor** de `coluna` dentro de cada partição da janela, de acordo com a ordem especificada [3]. Em termos simples, para cada linha, `FIRST_VALUE` olha para o grupo (partição) ao qual aquela linha pertence, identifica qual valor de interesse aparece primeiro quando esse grupo é ordenado, e devolve esse valor.

- **Sintaxe:** `FIRST_VALUE(expr) OVER (PARTITION BY ... ORDER BY ... [frame])`
  Onde `expr` é a expressão ou coluna que queremos obter. A partição e ordem funcionam como explicado antes.

- **Exemplo prático:** Considere uma tabela `AlunosProvas` com colunas (Aluno, DataProva, Nota). Vamos calcular a primeira nota obtida por cada aluno:

```
SELECT
    Aluno,
    DataProva,
    Nota,
    FIRST_VALUE(Nota) OVER (PARTITION BY Aluno ORDER BY DataProva ASC)
AS PrimeiraNota
FROM AlunosProvas;
```

Isso adicionaria uma coluna `PrimeiraNota` em cada linha, contendo a nota da primeira prova daquele aluno.

Suponha o seguinte conjunto de dados de exemplo para ilustrar: <table> <tr><th>Aluno</th><th>DataProva</th><th>Nota</th><th>PrimeiraNota (FIRST_VALUE)</th></tr> <tr><td>Ana</td><td>2021-01-15</td><td>14</td><td>14</td></tr> <tr><td>Ana</td><td>2021-06-20</td><td>16</td><td>14</td></tr> <tr><td>Ana</td><td>2022-03-10</td><td>18</td><td>14</td></tr> <tr><td>Bruno</td><td>2021-02-10</td><td>10</td><td>10</td></tr> <tr><td>Bruno</td><td>2021-12-05</td><td>12</td><td>10</td></tr> </table>

No exemplo acima, para Ana todas as linhas mostram `PrimeiraNota = 14`, que foi a primeira nota dela (na data 2021-01-15). Para Bruno, `PrimeiraNota = 10` em ambas as linhas, correspondendo à primeira nota dele.

Consideração sobre frame: Por padrão, `FIRST_VALUE` considera um frame que começa na primeira linha da partição e vai até a linha atual (inclusive). Isso significa que, na prática, mesmo sem especificar frame, ele **normalmente funciona como esperado**, porque o primeiro valor das linhas até a atual é sempre o primeiro da partição inteira (já que começamos do início). Contudo, se alterássemos o frame (coisa avançada), poderíamos restringir o olhar da função. Na maioria dos casos, usamos `FIRST_VALUE` sem frame customizado.

5.2. LAST_VALUE()

A função `LAST_VALUE(coluna)` retorna o **último valor** de `coluna` dentro da janela/partição [8]. A sintaxe é análoga a FIRST_VALUE, mas há um detalhe importante quanto ao frame, como já discutido:

- **Sintaxe:** `LAST_VALUE(expr) OVER (PARTITION BY ... ORDER BY ... [frame])`.
- **Uso correto:** Para obter o último valor da partição inteira, geralmente precisamos especificar que o frame vai até o fim da partição. Por exemplo:

```
LAST_VALUE(Nota)
  OVER (PARTITION BY Aluno ORDER BY DataProva ASC
        ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING)
AS UltimaNota
```

  A cláusula `ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING` garante que, para cada linha, a função veja desde a linha atual até o final da partição, de modo que o *last value* nesse frame seja efetivamente o último da partição [17] [18]. Sem isso, o default `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` faria com que `LAST_VALUE` retornasse o valor da linha atual (porque considera do início até a linha atual, cujo último é ela mesma).

- **Exemplo prático:** Usando a mesma tabela `AlunosProvas`, se quisermos trazer junto de cada registro a **última nota** daquele aluno (suponha que queremos comparar cada prova com a nota final do aluno):

```
SELECT
    Aluno,
    DataProva,
    Nota,
    LAST_VALUE(Nota)
      OVER (PARTITION BY Aluno ORDER BY DataProva ASC
            ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) AS
UltimaNota
FROM AlunosProvas;
```

Alternativamente, poderíamos ordenar por DataProva DESC e usar FIRST_VALUE, como discutido. Por exemplo:

```
FIRST_VALUE(Nota)
  OVER (PARTITION BY Aluno ORDER BY DataProva DESC) AS UltimaNota
```

Esta versão aproveita o fato de que a primeira na ordem descendente é a última cronológica.

Continuando o exemplo anterior dos alunos, se aplicarmos LAST_VALUE adequadamente: <table> <tr><th>Aluno</th><th>DataProva</th><th>Nota</th><th>UltimaNota (LAST_VALUE)</th></tr> <tr><td>Ana</td><td>2021-01-15</td><td>14</td><td>18</td></tr> <tr><td>Ana</td><td>2021-06-20</td><td>16</td><td>18</td></tr> <tr><td>Ana</td><td>2022-03-10</td><td>18</td><td>18</td></tr> <tr><td>Bruno</td><td>2021-02-10</td><td>10</td><td>12</td></tr> <tr><td>Bruno</td><td>2021-12-05</td><td>12</td><td>12</td></tr> </table>

Como esperado, para Ana todas as linhas mostram `UltimaNota = 18` (última nota dela), e para Bruno `UltimaNota = 12`.

Resumo FIRST vs LAST: `FIRST_VALUE` e `LAST_VALUE` são úteis para comparar cada linha com o primeiro ou último valor do grupo. Um caso prático comum é calcular a diferença relativa ao primeiro valor (como fizemos com notas, ou por exemplo diferença de preço desde o início do dia para ações) [3]. Apenas lembre-se de lidar corretamente com `LAST_VALUE` para que ele veja até o fim do conjunto. Em muitos SGDBs, usar o truque de inverter a ordem com FIRST_VALUE é mais simples.

5.3. ROW_NUMBER()

A função `ROW_NUMBER()` atribui um **número sequencial** a cada linha dentro da partição, de acordo com a ordem especificada [15] [19]. É provavelmente a função de janela de ranking mais utilizada, pois nos permite enumerar as linhas e depois filtrar ou selecionar posições específicas.

- **Sintaxe:** `ROW_NUMBER() OVER (PARTITION BY ... ORDER BY ...)`.
- **Comportamento:** Inicia em 1 para a primeira linha de cada partição e incrementa de 1 em 1. Se não for especificado PARTITION BY, numera sobre o resultset inteiro. Diferente de RANK, `ROW_NUMBER` **não lida com empates** – se duas linhas tiverem valores iguais na ordenação, a ordem de numeração entre elas será arbitrária (mas determinística baseado em critérios de desempate implícitos, por exemplo, internal ID), e uma receberá 1 e outra 2, sem "empate".

- **Exemplo prático 1:** Suponha uma tabela `Vendas` com colunas (Vendedor, ValorVenda). Podemos querer listar as vendas de cada vendedor da maior para a menor e enumerá-las:

```
SELECT
    Vendedor,
    ValorVenda,
    ROW_NUMBER() OVER (PARTITION BY Vendedor ORDER BY ValorVenda DESC)
AS RankValor
FROM Vendas;
```

Isso geraria algo como:

Vendedor	ValorVenda	RankValor (ROW_NUMBER)
Alice	500	1
Alice	300	2
Alice	100	3
Bob	400	1
Bob	200	2

Aqui, para Alice a maior venda foi 500 (rank 1), depois 300 (rank 2), etc. Para Bob, 400 foi rank 1, 200 rank 2. Note que se Alice tivesse duas vendas de 300, elas receberiam RankValor 2 e 3 (uma delas arbitrariamente seria considerada a segunda e outra a terceira, já que ROW_NUMBER não dá empate).

- **Exemplo prático 2:** Eliminar duplicados mantendo a primeira ocorrência – Um uso comum de ROW_NUMBER é em conjunto com um filtro para pegar, por exemplo, apenas o primeiro registro de cada grupo. Se tivéssemos uma tabela com registros duplicados e quiséssemos um de cada, poderíamos fazer `ROW_NUMBER() OVER (PARTITION BY camposDuplicados ORDER BY algo)` e filtrar `= 1`. Também, como vimos, para pegar top N por grupo (ex: top 3 vendas por categoria) faríamos similar e filtraríamos `WHERE ROW_NUMBER() <= 3`.

No contexto do nosso caso de estudo, usamos ROW_NUMBER para marcar a última prova de cada aluno. Em geral, a combinação de ROW_NUMBER e filtro é uma alternativa ao QUALIFY ou outras técnicas para obter linhas específicas dentro de grupos.

5.4. RANK()

A função `RANK()` também gera um número de ranking por ordem, porém ela trata **empates** de forma diferente do ROW_NUMBER [15] [19]. Quando valores estão empatados na ordenação, elas recebem o **mesmo rank**, e o próximo rank pula valores.

- **Sintaxe:** `RANK() OVER (PARTITION BY ... ORDER BY ...)`.
- **Comportamento:** Rank inicia em 1 para a primeira linha de cada partição (como row_number). Se duas ou mais linhas tiverem o mesmo valor de ordenação, elas recebem o mesmo número de rank. Entretanto, esse afeta o número subsequente: o rank seguinte será incrementado pelo número de linhas anteriores. Em outras palavras, se houver empate no 2º lugar entre duas

linhas, ambas terão rank 2, e o próximo rank emitido será 4 (não 3, pois duas linhas ocuparam o 2º lugar).

- **Exemplo prático:** Considere uma tabela `ResultadosProva` com colunas (ProvaID, Aluno, Nota). Queremos dar a classificação (colocação) dos alunos por Nota em cada prova. Usando RANK (descendente, porque maior nota -> rank 1):

```
SELECT
    ProvaID,
    Aluno,
    Nota,
    RANK() OVER (PARTITION BY ProvaID ORDER BY Nota DESC) AS
Classificacao
FROM ResultadosProva;
```

Suponha que na Prova 1 as notas foram: Ana 9, Bruno 8, Carla 8, David 7. Os ranks seriam:

ProvaID	Aluno	Nota	Classificacao (RANK)
1	Ana	9	1
1	Bruno	8	2
1	Carla	8	2
1	David	7	4

Observamos que Bruno e Carla têm a mesma nota 8, empatados no segundo lugar – ambos receberam `Classificacao = 2`. O próximo aluno, David, vem tecnicamente em quarto lugar (porque há três alunos à frente, embora um dos "lugares" esteja compartilhado). Por isso o RANK de David é 4, não 3.

5.5. DENSE_RANK()

A função `DENSE_RANK()` é similar ao RANK, mas atribui rankings **sem lacunas** (dense = denso) [20]. Em caso de empates, todos os empatados recebem o mesmo número, e o próximo rank é apenas incrementado em +1 (ao contrário do salto do RANK normal).

- **Sintaxe:** `DENSE_RANK() OVER (PARTITION BY ... ORDER BY ...)`.
- **Comportamento:** Como o RANK, mas "compactando" a sequência. Se duas linhas empataram no rank 2, o próximo terá rank 3 (em vez de 4).
- **Exemplo prático:** Usando o mesmo cenário da classificação da prova:

```
SELECT
    ProvaID,
    Aluno,
    Nota,
    DENSE_RANK() OVER (PARTITION BY ProvaID ORDER BY Nota DESC) AS
```

```
ClassificacaoDensa
FROM ResultadosProva;
```

Os resultados para o exemplo (Ana 9, Bruno 8, Carla 8, David 7) seriam:

ProvaID	Aluno	Nota	ClassificacaoDensa (DENSE_RANK)
1	Ana	9	1
1	Bruno	8	2
1	Carla	8	2
1	David	7	3

Aqui Bruno e Carla empatados têm 2, mas David passa a ser 3 (não pula para 4). DENSE_RANK, portanto, **não pula números**.

- **Quando usar RANK ou DENSE_RANK?** Depende do problema: se estamos atribuindo posições onde, se há empate, pulamos a posição seguinte (como "empate no segundo lugar, ninguém fica em terceiro"), usamos RANK. Se queremos enumerar posições consecutivas ignorando a contagem de empates (por exemplo, para "top 3 distintos", DENSE_RANK pode ser útil), usamos DENSE_RANK.

Diferença visual entre RANK e DENSE_RANK: Usando um pequeno exemplo ilustrativo:

Nome	Nota	RANK	DENSE_RANK
Alice	95	1	1
Bob	90	2	2
Carlos	90	2	2
Diana	85	4	3

Aqui, Bob e Carlos empatados em segundo lugar. RANK pula do 2 para 4; DENSE_RANK vai de 2 para 3.

5.6. NTILE()

A função `NTILE(n)` distribui as linhas de cada partição em **n grupos o mais equilibrados possível**, atribuindo a cada linha um número de 1 a n conforme seu grupo [21] . Em termos simples, é usada para dividir o conjunto ordenado em "quartis", "decilis" ou qualquer "n-ésimos" segmentos.

- **Sintaxe:** `NTILE(n) OVER (PARTITION BY ... ORDER BY ...)`. O `n` deve ser um inteiro positivo indicando em quantos compartimentos deseja dividir.
- **Comportamento:** Ordena as linhas pela cláusula ORDER BY dentro de cada partição e, em seguida, separa em n segmentos quase iguais em tamanho. Linhas extras irão nos primeiros segmentos (por exemplo, se 10 linhas em 3 grupos, um possível repartição é 4-3-3 linhas nos grupos 1, 2, 3).

- **Exemplo prático:** Suponha uma tabela `Funcionarios` com (Nome, Salario). Podemos dividir os funcionários em 4 faixas salariais (quartis):

```sql
SELECT
    Nome,
    Salario,
    NTILE(4) OVER (ORDER BY Salario DESC) AS QuartilSalario
FROM Funcionarios;
```

Isso atribuirá 1 para os ~25% com salários mais altos, 2 para o próximo quartil, e assim por diante até 4 para os com salários mais baixos (já que ordenamos descendentemente, 1 é maior salário). Se o número de funcionários não for múltiplo de 4, alguns quartis terão uma pessoa a mais que outros – normalmente os primeiros quartis recebem as linhas extras.

- **Exemplo prático 2:** Em um contexto de alunos, poderíamos querer dividir os alunos em 10 grupos iguais por nota para ver decil de desempenho. `NTILE(10) OVER (ORDER BY Nota DESC)` nos daria isso, onde decil 1 = top 10% de notas, decil 10 = bottom 10%.

Ilustração: Imagine 5 funcionários com salários [3000, 2500, 2000, 1500, 1200] e usamos `NTILE(3)` em ordem desc: - Ordenados desc: 3000, 2500, 2000, 1500, 1200. - Dividindo em 3 grupos (o mais equilibrado possível): precisamos 5/3 ≈ 1.66 por grupo, então teremos 2 no grupo 1, 2 no grupo 2, 1 no grupo 3 (tipicamente as linhas extras vão aos primeiros grupos). - Resultado: - Salário 3000 -> grupo 1 - Salário 2500 -> grupo 1 - Salário 2000 -> grupo 2 - Salário 1500 -> grupo 2 - Salário 1200 -> grupo 3

Grupos 1 e 2 têm 2 elementos, grupo 3 tem 1 elemento.

NTILE é útil para criação de percentis ou categorização contínua. Observe que se os dados tiverem muitos valores iguais, NTILE não garante separar valores iguais no mesmo grupo; ele simplesmente conta linhas, não entende do valor. Para percentis contínuos melhor usar mediana ou funções percentil específicas, mas NTILE dá uma aproximação simples.

5.7. LAG()

A função `LAG(coluna, offset)` retorna o valor da `coluna` na linha que está um certo **deslocamento para trás** em relação à linha atual dentro da mesma partição [22]. Em outras palavras, permite olhar para a linha "anterior".

- **Sintaxe:** `LAG(expr, offset, valor_padrao) OVER (PARTITION BY ... ORDER BY ...)`.
- O parâmetro `offset` (inteiro) indica quantas linhas para trás. `offset = 1` significa a linha imediatamente anterior; `offset = 2` a segunda linha anterior, etc. Se omitido, assume 1.
- O `valor_padrao` é opcional e serve para casos em que não haja linha anterior suficiente (por exemplo, a primeira linha não tem anterior). Se não fornecer, o padrão é NULL.
- **Exemplo prático 1:** Comparar cada registro de vendas com o anterior do mesmo produto:

```sql
SELECT
    ProdutoID,
    DataVenda,
    Valor,
    LAG(Valor, 1) OVER (PARTITION BY ProdutoID ORDER BY DataVenda) AS
```

```
ValorVendaAnterior
FROM Vendas;
```

Para cada venda de um produto, aparece a venda anterior (em ordem de data) daquele produto. Assim podemos calcular diferenças: `Valor - LAG(Valor,1) ... AS DiferencaValor` para saber quanto aumentou ou diminuiu em relação à venda anterior.

- **Exemplo prático 2:** Calcular a diferença de nota de uma prova para a prova anterior do mesmo aluno (um cenário similar ao de melhoria contínua):

```
SELECT
    Aluno,
    DataProva,
    Nota,
    LAG(Nota, 1) OVER (PARTITION BY Aluno ORDER BY DataProva) AS
NotaAnterior,
    Nota - LAG(Nota, 1) OVER (PARTITION BY Aluno ORDER BY DataProva) AS
VariacaoNota
FROM NotasAlunos;
```

A primeira prova de cada aluno terá NotaAnterior = NULL (ou valor_padrao se definido), e VariacaoNota = NULL, pois não existe prova anterior. As demais linhas mostrarão a nota anterior e a diferença.

- **Exemplo prático 3:** Suponha uma tabela de estoques diários: (Produto, Data, QtdeEmEstoque). Com LAG, podemos identificar dias em que o estoque diminuiu:

```
SELECT
    Produto,
    Data,
    QtdeEmEstoque,
    LAG(QtdeEmEstoque, 1) OVER (PARTITION BY Produto ORDER BY Data) AS
EstoqueDiaAnterior,
    CASE
      WHEN QtdeEmEstoque < LAG(QtdeEmEstoque, 1) OVER (PARTITION BY
Produto ORDER BY Data)
      THEN 'Diminuiu'
      WHEN QtdeEmEstoque > LAG(QtdeEmEstoque, 1) OVER (...)
      THEN 'Aumentou'
      ELSE 'Igual'
    END AS StatusVariacao
FROM Estoques;
```

Isso nos permite comparar dia a dia se o estoque subiu ou desceu.

LAG é muito versátil para qualquer situação de comparação temporal ou sequencial. É, como o nome sugere, um "atraso": traz valores passados para a linha corrente.

5.8. LEAD()

A função `LEAD(coluna, offset)` é o oposto do LAG: retorna o valor da `coluna` na linha que está um certo **deslocamento à frente** da linha atual na partição [22]. Ou seja, olha para a próxima linha.

- **Sintaxe:** `LEAD(expr, offset, valor_padrao) OVER (PARTITION BY ... ORDER BY ...)`.
- `offset` = 1 por padrão se não especificado.
- `valor_padrao` para quando não houver próxima linha (ex: última linha não tem próxima, resultaria NULL se não definido).

- **Exemplo prático 1:** Continuando no exemplo de vendas, poderíamos, para cada venda, olhar qual foi a **próxima** venda do mesmo produto:

```
SELECT
    ProdutoID,
    DataVenda,
    Valor,
    LEAD(Valor, 1) OVER (PARTITION BY ProdutoID ORDER BY DataVenda) AS
ProximoValorVenda
FROM Vendas;
```

A última venda de cada produto terá ProximoValorVenda = NULL (não há próxima). Outras linhas mostrarão o valor da venda seguinte. Isso é útil, por exemplo, se quisermos calcular intervalos entre vendas ou queda/aumento para a próxima.

- **Exemplo prático 2:** Em uma tabela de visitas de pacientes (Paciente, DataVisita, Peso), poderíamos querer saber o peso registrado na visita seguinte para calcular variações:

```
SELECT
    Paciente,
    DataVisita,
    Peso,
    LEAD(Peso, 1) OVER (PARTITION BY Paciente ORDER BY DataVisita) AS
PesoProximaVisita
FROM Pesagens;
```

Depois, podemos subtrair o peso atual do peso da próxima visita para ver quanto mudou até a próxima vez que veio.

- **Exemplo prático 3:** Calcular o tempo até a próxima compra de um cliente:

```
SELECT
    ClienteID,
    DataCompra,
    LEAD(DataCompra, 1) OVER (PARTITION BY ClienteID ORDER BY
DataCompra) AS ProximaCompra,
```

```
    DATEDIFF(day, DataCompra, LEAD(DataCompra, 1)
                      OVER (PARTITION BY ClienteID ORDER BY
DataCompra)) AS DiasAteProxima
FROM Compras;
```

Aqui estamos usando uma função de diferença de datas (como DATEDIFF no SQL Server) para contar dias entre a data atual e a próxima compra do mesmo cliente.

LEAD, assim como LAG, é fundamental para comparações entre linhas próximas. Sempre que precisamos "alinhar" valores seguintes para a linha corrente (por exemplo, comparar um valor com o próximo), LEAD resolve elegantemente.

Resumo LAG/LEAD: Ambos podem ser vistos como funções de deslocamento. O offset pode ser maior que 1 se quisermos pular mais de uma linha. Por exemplo, `LAG(x, 2)` pega o valor de duas posições atrás. Se não houver linhas suficientes (p.ex., LAG de 2 na segunda linha não tem duas anteriores), retorna NULL ou default.

Tanto LAG quanto LEAD são frequentemente usados para calcular **diferenças, variações, tendências**, sinalizar aumentos/quedas, calcular intervalos de tempo, etc., dentro de sequências ordenadas [23].

6. Exemplos Práticos

Nesta seção, apresentamos exemplos práticos e numerados que aplicam as funções de janela em cenários concretos simplificados. Os exemplos ajudarão a consolidar o entendimento do funcionamento e da sintaxe dessas funções.

Exemplo 1: Obtendo a Primeira e Última Nota de cada Aluno (FIRST_VALUE e LAST_VALUE)
Vamos retomar o cenário de alunos e notas. Suponha a seguinte tabela de notas (ordenada por aluno e data para facilitar visualização):

Aluno	DataProva	Nota
Ana	2021-03-01	8
Ana	2021-07-01	12
Ana	2022-01-15	15
Bruno	2021-05-10	10
Bruno	2022-02-20	13
Carlos	2021-01-20	14
Carlos	2021-12-30	14
Carlos	2022-07-07	16

Queremos para cada linha saber qual foi a primeira nota do aluno e a última nota do aluno.

Usando funções de janela:

```sql
SELECT
    Aluno,
    DataProva,
    Nota,
    FIRST_VALUE(Nota) OVER (PARTITION BY Aluno ORDER BY DataProva ASC)
        AS NotaPrimeira,
    LAST_VALUE(Nota) OVER (PARTITION BY Aluno ORDER BY DataProva ASC
                           ROWS BETWEEN CURRENT ROW AND UNBOUNDED
FOLLOWING)
        AS NotaUltima
FROM NotasAlunos;
```

No resultado, teríamos (colunas calculadas preenchidas):

Aluno	DataProva	Nota	NotaPrimeira	NotaUltima
Ana	2021-03-01	8	8	15
Ana	2021-07-01	12	8	15
Ana	2022-01-15	15	8	15
Bruno	2021-05-10	10	10	13
Bruno	2022-02-20	13	10	13
Carlos	2021-01-20	14	14	16
Carlos	2021-12-30	14	14	16
Carlos	2022-07-07	16	14	16

Observe como **NotaPrimeira** se repete para todos os registros do aluno (sempre 8 para Ana, 10 para Bruno, 14 para Carlos), e **NotaUltima** também se repete (15 para Ana, 13 Bruno, 16 Carlos), confirmando que essas funções estão capturando corretamente o primeiro e último valores dentro de cada partição de aluno. Poderíamos agora facilmente calcular colunas de diferença entre a nota atual e uma dessas se quisessemos.

Exemplo 2: Classificação de Alunos por Nota (RANK vs DENSE_RANK)
Imagine agora que queremos classificar alunos por sua nota em uma determinada prova. Suponha a **Prova 1** com os seguintes resultados:

Aluno	Nota
Alice	9.5
Bruno	7.0
Carolina	9.5
Daniel	6.0
Érica	7.0

Dois alunos (Alice e Carolina) tiraram 9.5, e outros dois (Bruno e Érica) tiraram 7.0, havendo empates. Vamos calcular o RANK e o DENSE_RANK desses alunos nesta prova:

```
SELECT
    Aluno,
    Nota,
    RANK()       OVER (ORDER BY Nota DESC)        AS RankNota,
    DENSE_RANK() OVER (ORDER BY Nota DESC)        AS DenseRankNota
FROM ResultadosProva1;
```

Isso gera:

Aluno	Nota	RankNota	DenseRankNota
Alice	9.5	1	1
Carolina	9.5	1	1
Bruno	7.0	3	2
Érica	7.0	3	2
Daniel	6.0	5	3

Explicação:
Alice e Carolina estão empatadas com a maior nota, ambas ficaram em 1º lugar (Rank 1). Bruno e Érica empatados na terceira posição por nota (havia dois à frente, então Rank 3). Daniel ficou em quinto no Rank normal (porque dois ficaram no Rank1, dois no Rank3, então o próximo rank seria 5). Já o DenseRankNota não pula números: depois do 1º lugar, considera Bruno e Érica como 2º lugar (DenseRank 2), e Daniel como 3º lugar.

Isso ilustra a diferença entre **RANK (que deixa lacunas)** e **DENSE_RANK (sem lacunas)** no tratamento de empates.

Exemplo 3: Comparando Vendas Mensais (LAG e LEAD)
Suponha a tabela `VendasMensais` com colunas (Mes, Vendas) contendo o total de vendas de uma loja a cada mês:

Mes	Vendas
Jan	100
Fev	120
Mar	90
Abr	150
Mai	150
Jun	130

Vamos usar LAG e LEAD para comparar cada mês com o anterior e o seguinte:

```sql
SELECT
    Mes,
    Vendas,
    LAG(Vendas, 1)  OVER (ORDER BY MesSequencial)   AS VendasMesAnterior,
    LEAD(Vendas, 1) OVER (ORDER BY MesSequencial)   AS VendasProximoMes
FROM VendasMensais;
```

Obs: Aqui assumimos que temos uma forma de ordenar corretamente por mês cronologicamente (pode ser um campo oculto MesSequencial = 1,2,3,... ou datas reais). Para simplicidade, considere Jan=1, Fev=2, etc.

O resultado seria:

Mês	Vendas	VendasMesAnterior	VendasProximoMes
Jan	100	NULL	120
Fev	120	100	90
Mar	90	120	150
Abr	150	90	150
Mai	150	150	130
Jun	130	150	NULL

Analisando:
- Janeiro não tem mês anterior, então LAG retorna NULL (podíamos definir um default, mas não o fizemos). Tem próximo mês com 120 (Fev).
- Fevereiro vê janeiro como anterior (100) e março como próximo (90). Poderíamos calcular variações: ex: 120 - 100 = +20, mas somente listamos valores.
- Março: anterior 120, próximo 150.
- Abril: anterior 90, próximo 150. (Note que as vendas subiram bastante de Mar para Abr, depois mantiveram igual em Mai).
- Maio: anterior 150, próximo 130.
- Junho: anterior 150, sem próximo (NULL).

Esse exemplo mostra bem como LAG e LEAD "deslocam" os valores na tabela, permitindo comparar facilmente períodos adjacentes. Com esses valores em colunas, podemos por exemplo adicionar:

```sql
CASE
 WHEN Vendas > LAG(Vendas,1) OVER (...) THEN 'Subiu'
 WHEN Vendas < LAG(Vendas,1) OVER (...) THEN 'Desceu'
 ELSE 'Igual'
END AS ComparativoAnterior
```

para sinalizar se subiu, desceu ou manteve em relação ao mês anterior. Da mesma forma, poderíamos ver previsão do próximo (mas aí não faria sentido de negócio, a menos que estivéssemos comparando com meta, etc.).

Estes exemplos cobrem casos de uso típicos: agregados iniciais/finais, rankings com e sem lacuna, e comparação entre linhas consecutivas.

7. Exercícios Resolvidos

Nesta seção, apresentamos alguns exercícios já resolvidos para exemplificar como aplicar as funções de janela em problemas concretos. Recomendamos ler o enunciado, tentar pensar na solução, e então conferir a resolução apresentada.

Exercício 1: Top 3 Vendas por Vendedor (ROW_NUMBER)

Enunciado: Suponha uma tabela `Vendas` com colunas `(Vendedor, DataVenda, Valor)`. Escreva uma consulta SQL para listar as **3 maiores vendas de cada vendedor**, mostrando o nome do vendedor, a data da venda e o valor, ordenadas do maior para o menor valor dentro de cada vendedor.

Resolução: Para resolver, podemos usar a função de janela `ROW_NUMBER()` para numerar as vendas de cada vendedor por ordem decrescente de valor, depois filtrar as três primeiras posições. Os passos lógicos: 1. Particionar por Vendedor, ordenar por Valor desc, e atribuir row_number. 2. Selecionar somente aquelas com row_number <= 3.

Vamos supor alguns dados de exemplo na tabela `Vendas` para ilustrar:

```
Vendas:
Vendedor | DataVenda   | Valor
-------- | ----------  | ------
Alice    | 2025-05-01  | 500
Alice    | 2025-04-20  | 300
Alice    | 2025-03-15  | 700
Alice    | 2025-01-30  | 250
Bruno    | 2025-02-10  | 400
Bruno    | 2025-03-05  | 200
Bruno    | 2025-03-30  | 450
```

No caso da Alice, as maiores vendas são 700, 500, 300; do Bruno, 450, 400, 200.

A consulta SQL:

```sql
SELECT
    Vendedor,
    DataVenda,
    Valor
FROM (
    SELECT
        Vendedor,
        DataVenda,
        Valor,
        ROW_NUMBER() OVER (PARTITION BY Vendedor
                           ORDER BY Valor DESC) AS rn
```

```
    FROM Vendas
) AS sub
WHERE rn <= 3
ORDER BY Vendedor, Valor DESC;
```

Explicação da Solução: A subconsulta `sub` calcula uma coluna `rn` que numera as vendas de cada vendedor por ordem de Valor decrescente (maior venda = 1). Em seguida, na query externa, filtramos `WHERE rn <= 3` para pegar as três primeiras de cada. Ordenamos o resultado final por Vendedor e Valor desc para apresentação.

Aplicando aos dados de exemplo, o resultado seria:

```
Vendedor | DataVenda  | Valor
Alice    | 2025-03-15 | 700   (rn=1)
Alice    | 2025-05-01 | 500   (rn=2)
Alice    | 2025-04-20 | 300   (rn=3)
Bruno    | 2025-03-30 | 450   (rn=1)
Bruno    | 2025-02-10 | 400   (rn=2)
Bruno    | 2025-03-05 | 200   (rn=3)
```

Todas as outras vendas (Alice 250, etc.) teriam rn > 3 e ficaram fora do resultado.

Checagem: Esta abordagem garante 3 vendas por vendedor, mesmo se os valores forem iguais (nesse caso, ROW_NUMBER desempata arbitrariamente, mas mantém 3 no total). Se quiséssemos incluir vendas empatadas na 3ª posição também, teríamos que usar RANK ou DENSE_RANK em vez de ROW_NUMBER para a lógica de filtro – mas como o enunciado pede "3 maiores vendas", provavelmente quer exatamente 3 linhas.

Exercício 2: Média Cumulativa das Notas dos Alunos (AVG com PARTITION BY e ORDER BY)

Enunciado: Considere uma tabela `HistoricoNotas` com colunas `(Aluno, DataProva, Nota)`, contendo as notas ao longo do tempo de cada aluno. Escreva uma consulta para, para cada registro, apresentar a **média cumulativa** das notas do aluno até aquela data. Ou seja, cada linha deve mostrar a nota da prova atual e a média de todas as notas anteriores e incluindo a atual do mesmo aluno.

Resolução: Podemos usar uma função de janela agregada `AVG(Nota)` combinada com `OVER (PARTITION BY Aluno ORDER BY DataProva ROWS UNBOUNDED PRECEDING)` para calcular a média acumulada até a linha atual. A cláusula de frame `ROWS UNBOUNDED PRECEDING` até `CURRENT ROW` é implícita para agregações, mas podemos deixá-la explícita para clareza. Essa janela indica: para cada aluno, conforme ordenamos por DataProva, calcular a média desde o início (unbounded preceding) até a linha corrente.

A consulta:

```
SELECT
    Aluno,
    DataProva,
```

```
    Nota,
    AVG(Nota)
      OVER (PARTITION BY Aluno
            ORDER BY DataProva
            ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
      AS MediaCumulativa
FROM HistoricoNotas
ORDER BY Aluno, DataProva;
```

Exemplo ilustrativo: Suponha os seguintes dados de `HistoricoNotas`:

```
Aluno | DataProva   | Nota
----- | ----------  | ----
Ana   | 2021-01-01  |  10
Ana   | 2021-06-01  |  14
Ana   | 2022-01-01  |  16
Bruno | 2021-03-10  |  8
Bruno | 2021-09-15  |  12
```

Resultado esperado:

```
Aluno | DataProva   | Nota | MediaCumulativa
Ana   | 2021-01-01  | 10   | 10.0   (média de [10])
Ana   | 2021-06-01  | 14   | 12.0   (média de [10,14] = 24/2)
Ana   | 2022-01-01  | 16   | ~13.33 (média de [10,14,16] = 40/3)
Bruno | 2021-03-10  |  8   | 8.0    (média de [8])
Bruno | 2021-09-15  | 12   | 10.0   (média de [8,12] = 20/2)
```

A consulta acima produziria exatamente isso.

Explicação da Solução: A função `AVG(Nota) OVER (PARTITION BY Aluno ORDER BY DataProva ...)` calcula a média incrementalmente: - Para a primeira prova de cada aluno, a média é a própria nota. - Para a segunda, é a média das duas primeiras, e assim por diante. - Usamos o frame `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` para deixar claro que a média considera desde o início da partição até a linha atual (isso já é o padrão para `AVG` com ORDER BY, mas é bom entender que podemos ajustar frames para outros comportamentos). - Ordenamos o resultado final por Aluno e DataProva para visualizar cronologicamente.

Verificação: Podemos testar mentalmente com os dados. Funciona inclusive se houver notas iguais ou variações – está simplesmente somando e dividindo progressivamente.

Este exercício demonstrou o uso de uma **função de janela com agregação** (AVG) para um cálculo acumulativo, algo impossível de fazer só com GROUP BY a menos que fizéssemos junções auto-correlacionadas ou subconsultas correlacionadas. Com janela, o código fica limpo e direto.

8. Exercícios Propostos (com Soluções)

Agora é sua vez! Abaixo, propomos alguns exercícios adicionais para praticar o uso de funções de janela. Tente resolvê-los e depois confira a solução apresentada. Cada exercício já vem acompanhado da solução para efeito de autoavaliação.

Exercício Proposto 1: Segunda Maior Nota de cada Aluno (DENSE_RANK)

Enunciado: Usando a tabela `NotasAlunos(AlunoID, Nome, DataProva, Nota)` do contexto inicial, escreva uma query que encontre a **segunda maior nota** de cada aluno. Ou seja, para cada aluno, identifique qual foi sua nota que é apenas menor que a máxima obtida por ele (pressupondo que cada aluno tem pelo menos duas notas distintas; se não tiver, esse aluno pode ficar de fora ou mostrar a nota repetida).

Dica: Considere usar uma função de rankeamento para ordenar as notas de cada aluno e escolher a posição correta.

Solução:

Uma forma conveniente é utilizar `DENSE_RANK()` para classificar as notas de cada aluno em ordem decrescente. A maior nota terá dense_rank = 1, a segunda maior terá dense_rank = 2 dentro da partição do aluno. Então podemos filtrar o rank = 2.

```
SELECT
    AlunoID,
    Nome,
    Nota AS SegundaMaiorNota
FROM (
    SELECT
        AlunoID,
        Nome,
        Nota,
        DENSE_RANK() OVER (PARTITION BY AlunoID ORDER BY Nota DESC) as dr
    FROM NotasAlunos
) AS sub
WHERE dr = 2;
```

Explicação da Solução: Na subconsulta, calculamos `DENSE_RANK` por aluno, ordenando as notas do maior para o menor. O maior valor(es) de nota recebe rank 1; o segundo valor distinto de nota recebe rank 2. Então filtramos `dr = 2` para obter as linhas que correspondem ao segundo maior valor de cada aluno. Usamos DENSE_RANK em vez de RANK porque, caso o aluno tenha tirado a mesma nota máxima em várias provas, todas ficariam rank 1 e a próxima nota distinta seria rank 2 – exatamente o que queremos (a **segunda nota distinta** mais alta).

Exemplo rápido: Se um aluno teve notas [5, 8, 8, 10], o DENSE_RANK desc dará: 10->rank1, 8->rank2, 5->rank3. Nosso resultado vai pegar nota 8 como segunda maior nota. (Com RANK normal, 10->1, 8->2, 8->2, 5->4; filtrar rank=2 ainda daria 8, mas duplicado; de qualquer forma poderíamos pegar DISTINCT no final. DENSE_RANK nos dá já a noção de “posição” na ordem de valores).

Exercício Proposto 2: Divisão de Alunos em Quartis por Nota Final (NTILE)

Enunciado: Suponha que temos uma tabela `AlunosDesempenho` com colunas `(AlunoID, Nome, NotaFinal)` representando a nota final de cada aluno em um curso. Queremos categorizar os alunos em **quartis de desempenho** com base em `NotaFinal`. Ou seja, os 25% com notas mais altas no 1º quartil, os seguintes 25% no 2º, e assim por diante até o 4º quartil (os 25% com notas mais baixas). Escreva uma query que acrescente uma coluna `Quartil` (de 1 a 4) indicando a qual quartil cada aluno pertence em termos de NotaFinal.

Solução:

Podemos usar `NTILE(4)` sobre a ordenação por NotaFinal descendente para criar 4 faixas aproximadamente iguais.

```
SELECT
    AlunoID,
    Nome,
    NotaFinal,
    NTILE(4) OVER (ORDER BY NotaFinal DESC) AS Quartil
FROM AlunosDesempenho;
```

Explicação da Solução: Aqui não temos PARTITION BY porque queremos o cálculo no contexto de todos os alunos (não dentro de grupos separados, assumindo que queremos os quartis da distribuição geral de notas finais). Ordenamos por NotaFinal DESC para que Quartil=1 corresponda ao topo. O SGBD dividirá o número total de alunos por 4 e distribuirá as linhas em 4 grupos. Se o número de alunos não for divisível por 4, alguns quartis (os primeiros) terão uma linha a mais que outros.

Exemplo: Se houver 100 alunos, cada quartil terá 25 alunos. Se houver 102 alunos, o NTILE(4) distribuirá possivelmente 26 no quartil1, 26 no quartil2, 25 no quartil3, 25 no quartil4 (totalizando 102). Assim, aproximadamente 25% em cada.

Após executar a query, cada aluno terá um número de 1 a 4. Poderíamos interpretá-los, por exemplo, como: 1 = "Excelente", 2 = "Bom", 3 = "Regular", 4 = "Precisando melhorar", dependendo do contexto – mas essa é uma classificação arbitrária com base nos quartis.

Exercício Proposto 3: Variação de Vendas em Relação ao Mês Anterior (LAG)

Enunciado: Considere uma tabela `VendasMensais(Periodo, Valor)` onde `Periodo` está em formato 'YYYY-MM' (ano-mês) ou algo similar, representando cada mês e o valor de vendas nesse mês. Escreva uma consulta que liste, para cada mês, o valor de vendas, o valor do mês anterior e uma coluna indicando a **diferença** (ou variação) em relação ao mês anterior. Ordene o resultado em ordem cronológica.

Solução:

Usaremos `LAG` para obter o valor do mês anterior e subtrair:

```sql
SELECT
    Periodo,
    Valor AS Vendas,
    LAG(Valor, 1) OVER (ORDER BY Periodo) AS VendasMesAnterior,
    Valor - LAG(Valor, 1) OVER (ORDER BY Periodo) AS Diferenca
FROM VendasMensais
ORDER BY Periodo;
```

Explicação da Solução: A função `LAG(Valor, 1)` com ORDER BY Periodo (partição única global, já que queremos a sequência de todos os meses) pega o valor do mês precedente. Subtraindo do valor atual, obtemos a diferença. Se `Periodo` for do tipo texto 'YYYY-MM', a ordem alfanumérica coincide com cronológica desde que o formato seja ano-mês; se for do tipo data ou numérico de mês sequencial, também funciona.

Exemplo de saída para ilustrar, suponha dados simplificados:

```
Periodo | Vendas | VendasMesAnterior | Diferenca
2023-01 | 100    | NULL              | NULL
2023-02 | 120    | 100               | 20
2023-03 | 90     | 120               | -30
2023-04 | 95     | 90                | 5
```

No primeiro mês não há anterior (NULL aparece para LAG e consequentemente Diferença também podemos considerar NULL ou tratá-la como não aplicável). A partir daí temos as variações mês a mês.

Caso quiséssemos, poderíamos adicionar uma coluna com `%` de variação, usando algo como:

```sql
ROUND(
    (Valor - LAG(Valor) OVER ... ) * 100.0 / LAG(Valor) OVER ... , 1)
AS PercentualVariacao
```

para indicar percentualmente a mudança, mas isso vai além do enunciado solicitado.

Com isso, concluímos o guia de estudo. Esperamos que os conceitos de funções de janela em SQL estejam agora mais claros. As funções de janela permitem realizar análises ricas de dados dentro de consultas SQL de forma concisa e eficiente, como vimos pelos exemplos e exercícios. Continue praticando com diferentes cenários (por exemplo, usando `SUM()` e `MAX()` em janelas, explorando `NTH_VALUE`, etc.) para aprofundar seu domínio nesse assunto. Boa análise de dados em SQL! [1] [2]

---

[1] [3] [8] Guia de funções de janela do SQL | LearnSQL.com.br
https://learnsql.com.br/blog/guia-de-funcoes-de-janela-do-sql/

2 4 15 16 17 18 19 20 21 22 23 WINDOWS FUNCTIONS: Desvendando o Poder das Funções de Janela no SQL. | by Elen Carvalho | comunidadeds | Medium
https://medium.com/comunidadeds/windows-functions-desvendando-o-poder-das-fun%C3%A7%C3%B5es-de-janela-no-sql-805cf12bfff2

5 6 7 Cálculos Avançados em SQL com a Cláusula WINDOW | iMasters
https://imasters.com.br/banco-de-dados/calculos-avancados-em-sql-com-a-clausula-window

9 10 Conhece o comando QUALIFY do SQL? | by Diego Itacolomy | Medium
https://diegoitacolomy.medium.com/conhece-o-comando-qualify-do-sql-91c6deb41209

11 12 13 sql server - What is the execution order of the PARTITION BY clause compared to other SQL clauses? - Stack Overflow
https://stackoverflow.com/questions/54357532/what-is-the-execution-order-of-the-partition-by-clause-compared-to-other-sql-cla

14 Window Functions vs Subqueries: A Battle of SQL Powerhouses | by Naresh Erukulla | Medium
https://medium.com/@nerukulla0719/window-functions-vs-subqueries-a-battle-of-sql-powerhouses-ca1759ee9c05