

Slide - Introdução à banco de dados

1. Dados, Informação e Conhecimento

- **Dados:** São elementos brutos, sem significado isolado. Exemplo: "João", "25", "Engenharia".
 - **Informação:** Quando os dados são organizados de maneira significativa. Exemplo: "João tem 25 anos e estuda Engenharia".
 - **Conhecimento:** Quando se aplica a informação para tomar decisões. Exemplo: "Com base nos dados acadêmicos, João pode se candidatar a um estágio na área".
-

2. Banco de Dados vs. SGBD

- **Banco de Dados (BD):** Coleção organizada de dados estruturados, armazenados eletronicamente.
 - **SGBD (Sistema de Gerenciamento de Banco de Dados):** Software que gerencia e manipula bancos de dados.
Exemplo:
 - Banco de Dados: Contém informações sobre clientes, produtos e vendas.
 - SGBD: MySQL, PostgreSQL, Oracle, SQL Server, que gerenciam esses dados.
-

3. Modelos de Dados

- **Modelos Conceituais:** Representação abstrata do banco, focada no negócio.
 - Exemplo: **Diagrama Entidade-Relacionamento (DER)** com entidades como Cliente, Pedido e Produto.
- **Modelos Lógicos:** Adiciona detalhes técnicos, convertendo entidades em tabelas e atributos em colunas.
 - Exemplo: Cliente (id, nome, CPF).
- **Modelos Físicos:** Implementação concreta no SGBD escolhido, com scripts SQL.

Exemplo:

```
CREATE TABLE Cliente (  
  id INT PRIMARY KEY,  
  nome VARCHAR(100),  
  cpf VARCHAR(11) UNIQUE  
  );
```

4. Biblioteca Virtual - Exemplo Prático

Modelo Conceitual

- **Entidades:** Livro, Autor, Leitor, Empréstimo.
- **Relacionamentos:**
 - Livro-Escreve-Autor (Muitos-para-Muitos)
 - Leitor-Faz-Emprestimo (Um-para-Muitos)
 - Livro-EstaEm-Emprestimo (Um-para-Muitos)

Modelo Lógico

- **Tabelas:** Livro, Autor, Leitor, Empréstimo, Livro_Autor.
- **Campos:**
 - Livro: id (INT), titulo (TEXT), ano_publicacao (INT), genero (TEXT).
 - Emprestimo: id (INT), livro_id (INT), leitor_id (INT), data_emprestimo (DATE), data_devolucao (DATE).

Modelo Físico (SQL para MySQL)

```
CREATE TABLE Livro (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  titulo VARCHAR(255) NOT NULL,  
  ano_publicacao INT,  
  genero VARCHAR(100)  
);  
  
CREATE TABLE Autor (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  nome VARCHAR(255) NOT NULL,  
  nacionalidade VARCHAR(100)  
);  
  
CREATE TABLE Leitor (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  nome VARCHAR(255) NOT NULL,  
  email VARCHAR(255) UNIQUE NOT NULL  
);  
  
CREATE TABLE Emprestimo (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  livro_id INT NOT NULL,  
  leitor_id INT NOT NULL,  
  data_emprestimo DATE,
```

```
data_devolucao DATE,  
FOREIGN KEY (livro_id) REFERENCES Livro(id),  
FOREIGN KEY (leitor_id) REFERENCES Leitor(id)  
);
```

```
CREATE TABLE Livro_Autor (  
  livro_id INT NOT NULL,  
  autor_id INT NOT NULL,  
  PRIMARY KEY (livro_id, autor_id),  
  FOREIGN KEY (livro_id) REFERENCES Livro(id),  
  FOREIGN KEY (autor_id) REFERENCES Autor(id)  
);
```

Slide - Custo de Consultas SQL

1. Introdução ao Custo de Consultas SQL

- O **custo de uma consulta SQL** se refere ao impacto no tempo de execução e no uso de recursos como CPU, memória e operações de entrada/saída (I/O).
 - Os fatores que influenciam esse custo incluem:
 - **Tamanho das tabelas**
 - **Complexidade da consulta** (uso de **JOINS**, subconsultas, agregações)
 - **Existência de índices**
 - **Estratégia de execução escolhida pelo banco**
 - A **otimização de consultas** é essencial para melhorar a performance do sistema e reduzir o tempo de resposta, especialmente em bancos com grande volume de dados ou acessos concorrentes.
-

2. Estruturas que Influenciam o Custo de Consultas

2.1 Índices

- **Objetivo:** Acelerar a busca de dados.
- **Tipos de Índices:**
 - **B-Tree:** Bom para consultas de igualdade e intervalos (**=**, **>**, **<**, **BETWEEN**).
 - Exemplo: `CREATE INDEX idx_nome ON tabela (coluna);`
 - **Hash:** Melhor para comparações de igualdade exata (**=**).
 - Exemplo: `CREATE INDEX idx_hash ON tabela USING HASH (coluna);`
 - **Composto:** Útil para buscas filtrando múltiplas colunas.

- Exemplo: `CREATE INDEX idx_composto ON tabela (coluna1, coluna2);`

- **Custo dos Índices:**

- Melhoram buscas, mas impactam negativamente inserções, atualizações e exclusões, pois precisam ser atualizados.

2.2 Particionamento de Tabelas

- **O que é?** Divisão da tabela em partes menores com base nos valores de uma coluna específica.
- **Vantagens:** Reduz o escaneamento de dados e melhora consultas filtradas pela coluna de particionamento.
- **Desvantagens:** Aumenta a complexidade de manutenção.

2.3 Ordenação e Agrupamento

2.3 Ordenação e Agrupamento

- ◆ **Ordenação (`ORDER BY`)**

A ordenação (`ORDER BY`) é usada para classificar os resultados de uma consulta de acordo com uma ou mais colunas. O banco de dados pode fazer isso em memória ou no disco, dependendo do tamanho dos dados.

- ✓ **Exemplo Simples**

Suponha que temos uma tabela `clientes`:

n	id
---	----

A 2

P 3

M 2

Se quisermos listar os clientes ordenados por idade, usamos:

sql

CopiarEditar

- `SELECT * FROM clientes ORDER BY idade;`

♦ Saída ordenada:

n id

M 2

A 2

P 3

Se quisermos em **ordem decrescente**, adicionamos **DESC**:

sql

CopiarEditar

- `SELECT * FROM clientes ORDER BY idade DESC;`

♦ **Saída ordenada (decrescente):**

n id

P 3

A 2

♦ Custo do **ORDER BY**

O custo de ordenar os dados depende da quantidade de registros e se há um **índice** na coluna usada para ordenação.

- **Se houver um índice na coluna (**idade**)**, a ordenação é rápida, pois os dados já estão estruturados.
- **Se não houver índice**, o banco precisa fazer uma **ordenação manual**, usando um **algoritmo de ordenação eficiente**, como QuickSort ou MergeSort.

Complexidade de Tempo:

- O algoritmo usado tem custo médio de **$O(n \log n)$** (como o QuickSort).
- Se os dados forem muito grandes, o banco pode **usar o disco** para ordenação, tornando o processo mais lento.

♦ Agrupamento (**GROUP BY**)

O agrupamento (**GROUP BY**) serve para **agrupar linhas com valores repetidos** em uma coluna e aplicar **funções agregadoras** como **COUNT, SUM, AVG, MIN, MAX**.

✅ Exemplo Simples

Suponha que temos uma tabela **vendas** com os seguintes dados:

id	produto	quantidade
----	---------	------------

1	Celular	2
2	Notebook	1
3	Celular	3
4	TV	1
5	Celular	1

Se quisermos contar quantas vendas foram feitas por produto:

sql

CopiarEditar

- `SELECT produto, COUNT(*) AS total_vendas`
- `FROM vendas`
- `GROUP BY produto;`

♦ Saída agrupada:

produto	total_vendas
Celular	3
Notebook	1
TV	1

◆ Custo do GROUP BY

O custo do GROUP BY depende de dois fatores:

1. **Se a coluna já estiver ordenada (com índice)** → O agrupamento é mais rápido.
2. **Se a coluna não estiver ordenada** → O banco precisa primeiro ordenar os dados antes de agrupar.

Processo de agrupamento:

- O banco pode usar uma **estrutura de hash** para armazenar os grupos e contagens.
- Se precisar ordenar antes de agrupar, o custo pode ser **$O(n \log n)$** (ordenando antes de agrupar).
- Se os dados forem muitos, o banco pode usar **arquivos temporários no disco**, tornando a operação mais lenta.

Comparação ORDER BY vs GROUP BY

Operação	Objetivo	Complexidade	Impacto
ORDER BY	Ordenar os dados	$O(n \log n)$	Pode ser otimizado com índice
GROUP BY	Agrupar valores repetidos	$O(n \log n)$ ou $O(n)$ com índices	Pode precisar de ordenação antes de agrupar

Dicas de Otimização

- ✓ Criar índices nas colunas usadas em ORDER BY e GROUP BY
- ✓ Evitar agrupar ou ordenar grandes volumes de dados sem necessidade
- ✓ Filtrar (WHERE) antes de agrupar para reduzir o número de registros processados.

2.4 EXPLAIN

EXPLAIN: Analisando o Plano de Execução de uma Consulta

O comando **EXPLAIN** é usado para entender como o banco de dados executará uma consulta SQL. Ele mostra informações detalhadas sobre como os dados serão recuperados, quais índices serão utilizados e o custo estimado da consulta.

💡 **Objetivo:**

- Identificar gargalos de desempenho.
- Verificar se índices estão sendo usados corretamente.
- Melhorar consultas SQL para torná-las mais eficientes.

Como Usar o EXPLAIN?

1 Forma Básica do **EXPLAIN**

sql

CopiarEditar

- **EXPLAIN SELECT * FROM clientes WHERE idade > 30;**

♦ Isso retorna um plano de execução mostrando como a consulta será executada.

2 **EXPLAIN ANALYZE**: Analisando a Consulta em Tempo Real

O comando **EXPLAIN ANALYZE** executa a consulta e mostra estatísticas reais de desempenho.

sql

CopiarEditar

- **EXPLAIN ANALYZE SELECT * FROM clientes WHERE idade > 30;**

♦ Diferença do **EXPLAIN** normal:

- **EXPLAIN** apenas estima como a consulta será executada.
- **EXPLAIN ANALYZE** executa de fato a consulta e mostra os tempos reais de execução.

⚠ Atenção: Como **EXPLAIN ANALYZE** realmente executa a consulta, use com cuidado em comandos que modificam dados (**UPDATE**, **DELETE**, **INSERT**).



Como Interpretar o Resultado?

Suponha que rodamos:

sql

CopiarEditar

- **EXPLAIN SELECT * FROM clientes WHERE idade > 30;**

E o banco retorna:

pgsql

CopiarEditar

- **Seq Scan on clientes (cost=0.00..35.00 rows=10 width=50)**

Entendendo os Campos

Campo	Explicação
Seq Scan	O banco está fazendo uma varredura sequencial na tabela inteira (lento!).

cost=0.00..
35.00

Estimativa de custo: começa em 0.00 e pode chegar até 35.00. Quanto menor, melhor.

rows=10

Estima que 10 linhas serão retornadas.

width=50

Estima que cada linha ocupa 50 bytes na memória.

Problema:

O uso de "Seq Scan" indica que o banco não está usando índices, o que pode ser ruim para consultas em tabelas grandes.



Otimizando a Consulta

Se tivermos um índice na coluna **idade**, o banco pode usar um Index Scan (mais rápido):

sql

CopiarEditar

- **CREATE INDEX idx_idade ON clientes (idade);**

Agora, rodamos **EXPLAIN** novamente:

pgsql

CopiarEditar

- **Index Scan using idx_idade on clientes (cost=0.05..12.00 rows=10 width=50)**

Tipo de
Escaneamento

Quando Ocorre

Performance

Seq Scan (Varredura Sequencial)	Quando não há índice na coluna	Lento para tabelas grandes
Index Scan (Busca por Índice)	Quando há um índice adequado	Rápido
Bitmap Index Scan	Quando muitos registros precisam ser recuperados via índice	Médio

Exemplo com **EXPLAIN ANALYZE**

sql

CopiarEditar

- **EXPLAIN ANALYZE SELECT * FROM clientes WHERE idade = 30;**



Saída:

pgsql

CopiarEditar

- **Index Scan using idx_idade on clientes (cost=0.05..8.00 rows=1 width=50)**
- **(actual time=0.015..0.020 rows=1 loops=1)**

 Destaques:

- Index Scan indica que um índice foi usado 
- cost=0.05..8.00 → custo menor, pois o índice melhora a busca 

- actual time=0.015..0.020 → Tempo real de execução foi muito rápido 🕒

🔥 Resumo e Boas Práticas

- ✓ Use **EXPLAIN** para entender como suas consultas estão sendo executadas.
- ✓ Se um **Seq Scan** aparecer, veja se um índice pode ajudar.
- ✓ Use **EXPLAIN ANALYZE** para medir o tempo real da consulta.
- ✓ Evite consultas sem filtros (**WHERE**), pois elas podem forçar o banco a ler toda a tabela.

🚀 Conclusão: **EXPLAIN** é uma ferramenta essencial para otimizar consultas SQL e garantir que seu banco de dados esteja rodando da forma mais eficiente possível. Se precisar de mais detalhes, só avisar! 😊

- Comando usado para analisar o plano de execução de uma consulta.
 - Exemplo: **EXPLAIN ANALYZE SELECT * FROM tabela;**
- Ajuda a identificar gargalos e melhorar a performance.

2.5 Views Materializadas

Views Materializadas: O que são e como funcionam?

As **Views Materializadas** são uma técnica avançada de banco de dados para melhorar a performance de consultas que são repetitivas e custosas. Elas funcionam como **tabelas pré-computadas** que armazenam os resultados de uma consulta, reduzindo a necessidade de reexecutá-la frequentemente.



O que é uma View Materializada?

Uma **View Materializada** (ou MV – Materialized View) é um tipo especial de **tabela virtual armazenada** que guarda o resultado de uma consulta SQL no disco.

- ✓ Finalidade:

- Acelerar consultas complexas que envolvem agregações, junções e filtros.
 - Reduzir o tempo de processamento de relatórios e dashboards.
 - Melhorar a performance em consultas sobre grandes volumes de dados.
-

Exemplo Prático

Suponha que temos uma tabela **vendas**:

sql

CopiarEditar

- `CREATE TABLE vendas (`
- `id SERIAL PRIMARY KEY,`
- `data DATE NOT NULL,`
- `cliente VARCHAR(100),`
- `valor DECIMAL(10,2)`
- `);`

Agora, queremos **agrupar as vendas por data** para gerar relatórios rapidamente. Podemos criar uma **View Materializada** para armazenar essa informação.

sql









CopiarEditar

- `CREATE MATERIALIZED VIEW resumo_vendas AS`
- `SELECT data, SUM(valor) AS total`
- `FROM vendas`
- `GROUP BY data;`

♦ O que acontece aqui?

- O banco **executa a consulta** e **salva** os resultados em disco.
- Se consultarmos essa **View Materializada**, a resposta será **muito mais rápida** do que rodar `GROUP BY` sempre.

Diferença entre Views Normais e Views Materializadas

Característica	View Comum	View Materializada
Armazena dados?	 Não	 Sim
Sempre atualizada?	 Sim (dados em tempo real)	 Não (precisa de atualização)
Tempo de resposta	 Lento para consultas complexas	 Muito rápido
Uso de Disco	 Apenas lógica, sem uso extra	 Usa espaço em disco
Exemplo de criação	CREATE VIEW	CREATE MATERIALIZED VIEW

Atualização da View Materializada

Problema

A **View Materializada** não se atualiza sozinha. Se novos dados forem inseridos na tabela **vendas**, a view fica desatualizada.

Solução: REFRESH MATERIALIZED VIEW

Para atualizar os dados da View Materializada, usamos o comando:

sql

CopiarEditar

- `REFRESH MATERIALIZED VIEW resumo_vendas;`

- ♦ Isso **recalcula** os dados da view com base na tabela `vendas`.



Desvantagem:

Se a tabela de origem tiver muitos dados, a atualização pode ser **demorada**.



Otimizando com Atualizações Automáticas

1 Atualizar Somente Dados Modificados

O PostgreSQL permite a opção **CONCURRENTLY**, que atualiza a view sem bloquear leituras:

sql

CopiarEditar

- `REFRESH MATERIALIZED VIEW CONCURRENTLY resumo_vendas;`

- ♦ **Vantagem:** O banco continua acessível para leitura enquanto a atualização ocorre.
- ♦ **Requisito:** A view precisa ter um índice único (**UNIQUE INDEX**).

2 Agendar Atualizações Automáticas

Podemos programar a atualização da View Materializada periodicamente usando um **CRON JOB** ou um **EVENTO** no banco.

Exemplo no PostgreSQL usando um **agendador de tarefas (pg_cron)**:

sql

CopiarEditar

- `SELECT cron.schedule('0 * * * *', 'REFRESH MATERIALIZED VIEW resumo_vendas');`

- ♦ Isso atualiza a View a cada hora.
-



Quando Usar Views Materializadas?

✓ Use View Materializada quando:

- ✓ A consulta for muito complexa e demorada.
- ✓ Os dados não precisarem estar atualizados em tempo real.
- ✓ A mesma consulta for usada repetidamente.

✗ Não use se:

- ✗ Os dados precisam estar sempre atualizados.
 - ✗ Há necessidade de muita inserção/atualização na tabela base (pois exige constantes atualizações na view).
-



Conclusão

- **Views Materializadas** armazenam o resultado de uma consulta para melhorar a performance.
- **Devem ser atualizadas manualmente** ou com agendamentos automáticos.
- **Melhoram relatórios e dashboards** que precisam de **dados agregados** rapidamente.

Se precisar de mais exemplos ou detalhes, só perguntar! 😊🚀

40

Você disse:

e uma view normal?

O ChatGPT disse:



Views Normais: O que são e como funcionam?

Uma **View Normal** (ou simplesmente **View**) é uma **tabela virtual** baseada em uma consulta SQL. Ela **não armazena dados**, apenas exibe os resultados de uma consulta toda vez que é acessada.

✓ Finalidade:

- Melhorar a organização e reutilização de consultas SQL complexas.
 - Aumentar a segurança, ocultando colunas sensíveis da tabela original.
 - Facilitar a manutenção do banco de dados.
-

Exemplo Prático

Suponha que temos uma tabela **vendas**:

sql

CopiarEditar

- `CREATE TABLE vendas (`
- `id SERIAL PRIMARY KEY,`
- `data DATE NOT NULL,`
- `cliente VARCHAR(100),`
- `valor DECIMAL(10,2)`
- `);`

Agora, queremos **criar uma View** que mostre apenas as vendas feitas acima de R\$ 100,00.

sql








CopiarEditar

- `CREATE VIEW vendas_maiores_100 AS`
- `SELECT * FROM vendas WHERE valor > 100.00;`

♦ O que acontece aqui?

- Criamos uma **tabela virtual** chamada `vendas_maiores_100`.
- Quando rodamos `SELECT * FROM vendas_maiores_100`, o banco **executa a consulta original** em tempo real.

Diferença entre View Normal e View Materializada

Característica	View Comum	View Materializada
Armazena dados?	 Não	 Sim
Sempre atualizada?	 Sim (dados em tempo real)	 Não (precisa de atualização manual)
Tempo de resposta	 Mais lento para consultas complexas	 Muito rápido
Uso de Disco	 Apenas lógica, sem uso extra	 Usa espaço em disco
Exemplo de criação	CREATE VIEW	CREATE MATERIALIZED VIEW

Vantagens das Views Normais

1. Sempre atualizadas

Se novos dados forem inseridos na tabela **vendas**, a View **vendas_maiores_100** automaticamente refletirá essas mudanças.

2. Segurança

Podemos criar Views que mostram apenas **colunas específicas**, restringindo acesso a informações sensíveis.

sql

CopiarEditar

- `CREATE VIEW vendas_resumidas AS`
- `SELECT data, valor FROM vendas;`

- ♦ Aqui, a View **oculta** o nome do cliente da consulta.

✓ 3. Reutilização de Código

Se tivermos uma consulta SQL longa e complexa, podemos transformá-la em uma View e reutilizá-la facilmente.

sql

CopiarEditar

- `SELECT * FROM vendas_maiores_100 WHERE data = '2024-04-01';`

- ♦ Assim, evitamos reescrever a consulta toda vez.



Quando Usar Views Normais?

- ✓ Quando queremos simplificar consultas SQL complexas.
- ✓ Quando os dados precisam estar sempre atualizados.
- ✓ Quando queremos restringir colunas sensíveis.

✗ Não use se:

- ✗ A consulta for muito pesada e precisar ser acessada frequentemente (nesse caso, use uma **View Materializada**).



Conclusão

- **Views Normais** são consultas SQL armazenadas como tabelas virtuais.
- **Elas não armazenam dados**, apenas executam a consulta original em tempo real.
- **São úteis para organização, segurança e reutilização de código.**

3. Fatores que Impactam o Custo de Consultas

- **Tamanho das tabelas:** Maior volume de dados = maior custo de I/O.
- **Existência de índices:** Melhora buscas, mas impacta negativamente inserções/atualizações.
- **Complexidade da consulta:** **JOINs**, subconsultas e agregações aumentam o custo.
- **Hardware:** Discos **SSD** são mais rápidos para I/O do que **HDDs**.

Slide - Linguagem SQL

1. A Importância do SQL na Era Digital

A Structured Query Language (SQL) foi criada por Donald D. Chamberlin e Raymond F. Boyce nos anos 1970, sendo padronizada pela ANSI em 1986. Com o crescimento exponencial de dados, SQL se tornou essencial para armazenar, gerenciar e recuperar informações de maneira eficiente. Desde pequenas startups até grandes corporações, bancos de dados SQL são fundamentais para sistemas modernos.

2. Organização da Linguagem SQL

A SQL é dividida em cinco subconjuntos:

- **DDL (Data Definition Language):** Define a estrutura dos dados (CREATE, ALTER, DROP).
- **DML (Data Manipulation Language):** Manipula dados (INSERT, UPDATE, DELETE).
- **DQL (Data Query Language):** Consulta dados (SELECT).
- **DCL (Data Control Language):** Gerencia permissões (GRANT, REVOKE).

- **DTL (Data Transaction Language):** Controla transações (COMMIT, ROLLBACK).

3. DDL - Linguagem de Definição de Dados

3.1 Criando Banco de Dados e Tabelas

```
CREATE DATABASE Biblioteca;  
USE Biblioteca;
```

```
CREATE TABLE Livro (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    titulo VARCHAR(255) NOT NULL,  
    ano_publicacao INT,  
    genero VARCHAR(100)  
);
```

3.2 Modificando Estruturas

```
ALTER TABLE Livro ADD COLUMN editora VARCHAR(255);  
ALTER TABLE Livro MODIFY COLUMN titulo VARCHAR(500);  
ALTER TABLE Livro CHANGE COLUMN genero categoria VARCHAR(100);  
ALTER TABLE Livro ADD CONSTRAINT UNIQUE (titulo);  
ALTER TABLE Livro DROP COLUMN editora;
```

3.3 Removendo Estruturas

```
RENAME TABLE Livro TO Biblioteca_Livros;  
DROP TABLE Biblioteca_Livros;
```

4. DML - Linguagem de Manipulação de Dados

4.1 Inserindo Dados

```
INSERT INTO Livro (titulo, ano_publicacao, genero) VALUES ('Dom Quixote', 1605,  
'Romance');
```

4.2 Atualizando Dados

```
UPDATE Livro SET genero = 'Clássico' WHERE id = 1;
```

4.3 Removendo Dados

```
DELETE FROM Livro WHERE id = 1;
```

5. DQL - Linguagem de Consulta de Dados

5.1 Consultando Dados


```
SELECT titulo, ano_publicacao FROM Livro;
```

5.2 Filtrando Dados

```
SELECT titulo, genero FROM Livro WHERE ano_publicacao > 2000;
```

5.3 Ordenando Dados

```
SELECT titulo, ano_publicacao FROM Livro ORDER BY ano_publicacao DESC;
```

5.4 Funções Agregadas

```
SELECT genero, COUNT(*) AS total_livros FROM Livro GROUP BY genero;
```

5.5 Filtrando Grupos

```
SELECT genero, COUNT(*) AS total_livros FROM Livro GROUP BY genero HAVING  
COUNT(*) > 5;
```

6. Conclusão

A SQL é fundamental para qualquer aplicação que necessite de armazenamento e gerenciamento de dados. Com a compreensão de seus diferentes subconjuntos (DDL, DML, DQL, DCL e DTL), é possível criar, manipular e consultar informações de maneira eficiente. Dominar SQL é essencial para desenvolvedores e profissionais de banco de dados na era digital.

Slide - Junção com USING no SQL

A cláusula **USING** no SQL é uma forma simplificada de realizar junções (**JOIN**) entre tabelas quando existe uma coluna de mesmo nome em ambas as tabelas. Essa abordagem melhora a legibilidade da consulta, eliminando a necessidade de especificar explicitamente a relação entre as colunas das tabelas envolvidas.

Como o **USING** funciona em JOINS no SQL?

O **USING** é uma forma simplificada de escrever **JOIN** em SQL, utilizada quando as colunas usadas para junção **têm o mesmo nome** nas duas tabelas. Ele evita redundâncias no resultado final e deixa a consulta mais legível.

Diferença entre **USING** e **ON**

1 Com **ON** (Junção Tradicional)

- Especificamos a relação entre as colunas das tabelas manualmente.
- Se a coluna de junção for selecionada (**SELECT ***), ela aparece **duas vezes** no resultado.

2 Com **USING** (Forma Simplificada)

- Só pode ser usado quando a **coluna tem o mesmo nome** em ambas as tabelas.
 - O SQL sabe que a junção ocorre por essa coluna e a exibe **apenas uma vez** no resultado.
-

Exemplo Prático: **ON** vs. **USING**

Tabelas de Exemplo

sql

CopiarEditar

```
CREATE TABLE clientes (  
    id_cliente SERIAL PRIMARY KEY,  
    nome VARCHAR(100)  
);  
  
CREATE TABLE pedidos (  
    id_pedido SERIAL PRIMARY KEY,  
    id_cliente INT REFERENCES clientes(id_cliente),  
    total DECIMAL(10,2)
```

);

- ◆ Ambas as tabelas têm a coluna **id_cliente**, que relaciona os pedidos aos clientes.

Exemplo 1: Usando **ON**

sql

CopiarEditar

```
SELECT *  
  
FROM clientes  
  
JOIN pedidos ON clientes.id_cliente = pedidos.id_cliente;
```

- ◆ **Problema:** A coluna **id_cliente** aparece **duas vezes** no resultado: uma de cada tabela.

id_cliente (clientes)	nome	id_pedido	id_cliente (pedidos)	total
1	Maria	101	1	150.00
2	João	102	2	200.00

Exemplo 2: Usando **USING**

sql

CopiarEditar

```
SELECT *
```

```
FROM clientes
```

```
JOIN pedidos USING (id_cliente);
```

♦ **Vantagem:** O SQL sabe que `id_cliente` é a coluna de junção e a exibe **apenas uma vez** no resultado:

id_cliente	nome	id_pedido	total
1	Maria	101	150.00
2	João	102	200.00

Resumo: Quando usar **USING**?

✓ Use **USING** quando:

- ✓ A coluna usada na junção tem o **mesmo nome** nas duas tabelas.
- ✓ Você quer um **resultado mais limpo e sem duplicação**.

✗ Use **ON** quando:

- ✗ As colunas têm **nomes diferentes** em cada tabela.
- ✗ Precisa de **condições mais complexas** na junção.

Exemplo Extra: Junção com Múltiplas Colunas

Se houver mais de uma coluna com o mesmo nome para junção, podemos listar todas:

```
sql
```

CopiarEditar

```
SELECT *
```

```
FROM tabela1
```

```
JOIN tabela2 USING (coluna1, coluna2);
```

Isso garante que a junção ocorra em **todas** as colunas listadas, sem duplicá-las no resultado.



Conclusão

- **USING** é uma forma mais limpa e organizada de fazer JOINS, eliminando duplicações.
 - Funciona apenas quando as colunas têm o mesmo nome em ambas as tabelas.
 - **ON** ainda é necessário para junções mais complexas ou quando os nomes das colunas são diferentes.
-

Exemplo Prático 1: Biblioteca

Criação das Tabelas

sql

CopiarEditar

```
CREATE TABLE Livro (  
    id INT PRIMARY KEY,  
    titulo VARCHAR(100)  
);  
  
CREATE TABLE Emprestimo (  
    id INT PRIMARY KEY,  
    livro_id INT, -- Mesmo nome da coluna em Livro  
    data_emprestimo DATE,  
    FOREIGN KEY (livro_id) REFERENCES Livro(id)  
);
```

Consulta com **USING**

sql

CopiarEditar

```
SELECT Livro.titulo, Emprestimo.data_emprestimo  
FROM Livro  
LEFT JOIN Emprestimo USING (id);
```

Explicação:

- O **USING (id)** realiza a junção com base na coluna **id**, que está presente em ambas as tabelas.
- Como vantagem, a coluna **id** não aparece duplicada no resultado.

Consulta com **ON** (Alternativa)

sql

CopiarEditar

```
SELECT Livro.titulo, Emprestimo.data_emprestimo  
FROM Livro  
LEFT JOIN Emprestimo ON Livro.id = Emprestimo.livro_id;
```

- Aqui, a cláusula **ON** exige a especificação explícita da relação entre as colunas.

Exemplo Prático 2: Relacionamento Entre Atores e Filmes

No contexto de um sistema de banco de dados para um catálogo de filmes, há duas tabelas principais:

- **actor**: Armazena os dados dos atores.
- **film_actor**: Representa a relação entre atores e filmes.

Criação das Tabelas

sql

CopiarEditar

```
CREATE TABLE actor (  
    actor_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,  
    first_name VARCHAR(45) NOT NULL,
```

```
        last_name VARCHAR(45) NOT NULL,  
        last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON  
UPDATE CURRENT_TIMESTAMP,  
        PRIMARY KEY (actor_id)  
);  
  
CREATE TABLE film_actor (  
    actor_id SMALLINT UNSIGNED NOT NULL,  
    film_id SMALLINT UNSIGNED NOT NULL,  
    last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON  
UPDATE CURRENT_TIMESTAMP,  
    PRIMARY KEY (actor_id, film_id),  
    FOREIGN KEY (actor_id) REFERENCES actor(actor_id) ON DELETE  
RESTRICT ON UPDATE CASCADE,  
    FOREIGN KEY (film_id) REFERENCES film(film_id) ON DELETE  
RESTRICT ON UPDATE CASCADE  
);
```

Consulta com **USING**

sql

CopiarEditar

```
SELECT actor.first_name, actor.last_name, film.title,  
film.release_year, film.last_update  
FROM actor  
JOIN film_actor USING (actor_id)  
JOIN film USING (film_id);
```

Explicação:

- A junção ocorre automaticamente porque **actor_id** e **film_id** possuem o mesmo nome em ambas as tabelas.
- Essa abordagem simplifica o código, tornando a consulta mais clara.

Vantagens do **USING**

Código mais **limpo e legível**.

Evita **duplicação** de colunas no resultado.

Melhora a compreensão da estrutura da consulta.

Observação:

Se as colunas envolvidas na junção tiverem **nomes diferentes**, **USING** não pode ser utilizado, e a cláusula **ON** deve ser usada.

Desafio

Desafio SQL: Loja Virtual

O desafio SQL proposto envolve a criação, manipulação e consulta de um banco de dados para um sistema de e-commerce. O objetivo é estruturar e gerenciar as informações sobre produtos, clientes e vendas, utilizando comandos SQL no MySQL.

Etapa 1 - Criação do Banco de Dados

O primeiro passo é criar um banco de dados chamado **BDLOJA** para armazenar todas as informações necessárias.

```
sql
CopiarEditar
CREATE DATABASE BDLOJA;
USE BDLOJA;
```

Etapa 2 - Criação das Tabelas

O próximo passo é criar as tabelas **Produto**, **Cliente** e **Venda**, garantindo que elas tenham suas chaves primárias e estrangeiras corretamente definidas.

Tabela Produto

Armazena informações sobre os produtos disponíveis na loja.

```
sql
CopiarEditar
CREATE TABLE Produto (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    preco DECIMAL(10,2) NOT NULL,
    estoque INT NOT NULL
);
```

Tabela Cliente

Armazena os dados dos clientes cadastrados.

```
sql
CopiarEditar
CREATE TABLE Cliente (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
```

```
    telefone VARCHAR(20) NOT NULL  
);
```

Tabela Venda

Registra as compras realizadas pelos clientes.

sql

CopiarEditar

```
CREATE TABLE Venda (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    cliente_id INT,  
    produto_id INT,  
    quantidade INT NOT NULL,  
    data_venda DATE NOT NULL,  
    FOREIGN KEY (cliente_id) REFERENCES Cliente(id),  
    FOREIGN KEY (produto_id) REFERENCES Produto(id)  
);
```

Etapã 3 - Inserção de Dados

Agora que as tabelas foram criadas, precisamos inserir dados para simular um ambiente real.

Inserindo Produtos

sql

CopiarEditar

```
INSERT INTO Produto (nome, preco, estoque) VALUES  
( 'Notebook', 3500.00, 10),  
( 'Mouse', 150.00, 50),  
( 'Teclado', 200.00, 30),  
( 'Monitor', 1200.00, 15),  
( 'Impressora', 800.00, 20),  
( 'Cadeira Gamer', 900.00, 8),  
( 'HD Externo', 500.00, 25),  
( 'Pendrive', 60.00, 40),  
( 'Tablet', 1400.00, 12),  
( 'Fone de Ouvido', 250.00, 35);
```

Inserindo Clientes

sql

CopiarEditar

```
INSERT INTO Cliente (nome, email, telefone) VALUES
('Ana Silva', 'ana.silva@email.com', '11987654321'),
('Carlos Pereira', 'carlos.pereira@email.com', '11976543210'),
('Mariana Souza', 'mariana.souza@email.com', '11965432198'),
('João Mendes', 'joao.mendes@email.com', '11954321987'),
('Fernanda Rocha', 'fernanda.rocha@email.com', '11943219876'),
('Ricardo Alves', 'ricardo.alves@email.com', '11932198765'),
('Sofia Lima', 'sofia.lima@email.com', '11921098754'),
('Pedro Oliveira', 'pedro.oliveira@email.com', '11910987643'),
('Juliana Martins', 'juliana.martins@email.com', '11909876532'),
('Gabriel Costa', 'gabriel.costa@email.com', '11908765421');
```

Inserindo Vendas

sql

CopiarEditar

```
INSERT INTO Venda (cliente_id, produto_id, quantidade,
data_venda) VALUES
(1, 1, 1, '2024-03-01'),
(2, 2, 2, '2024-03-05'),
(3, 3, 1, '2024-03-07'),
(4, 4, 1, '2024-03-10'),
(5, 5, 2, '2024-03-15'),
(6, 6, 1, '2024-03-18'),
(7, 7, 1, '2024-03-20'),
(8, 8, 3, '2024-03-22'),
(9, 9, 1, '2024-03-25'),
(10, 10, 2, '2024-03-28');
```

Etapa 4 - Consultas no Banco de Dados

1. Listar todos os produtos ordenados pelo nome

sql

CopiarEditar

```
SELECT * FROM Produto ORDER BY nome;
```

2. Exibir nome dos clientes e produtos comprados (JOIN entre Cliente e Produto)

sql

CopiarEditar

```
SELECT Cliente.nome AS Cliente, Produto.nome AS Produto_Comprado
```

```
FROM Cliente
JOIN Venda ON Cliente.id = Venda.cliente_id
JOIN Produto ON Venda.produto_id = Produto.id;
```

3. Exibir a quantidade total de produtos vendidos e o valor total faturado

```
sql
CopiarEditar
SELECT SUM(Venda.quantidade) AS Total_Produtos_Vendidos,
       SUM(Venda.quantidade * Produto.preco) AS Total_Faturado
FROM Venda
JOIN Produto ON Venda.produto_id = Produto.id;
```

4. Listar os clientes que já compraram e os que ainda não compraram nada

```
sql
CopiarEditar
SELECT nome FROM Cliente WHERE id IN (SELECT cliente_id FROM
Venda)
UNION
SELECT nome FROM Cliente WHERE id NOT IN (SELECT cliente_id FROM
Venda);
```

5. Exibir vendas do último mês, ordenadas da mais recente para a mais antiga

```
sql
CopiarEditar
SELECT * FROM Venda
WHERE data_venda >= DATE_SUB(CURDATE(), INTERVAL 1 MONTH)
ORDER BY data_venda DESC;
```

Etapa 5 - Atualizações e Validação dos Dados

1. Aumentar o preço de um produto em 10%

```
sql
CopiarEditar
UPDATE Produto SET preco = preco * 1.10 WHERE id = 1; --
Exemplo: aumentar o preço do primeiro produto
```

2. Corrigir um e-mail digitado incorretamente

```
sql
CopiarEditar
```

```
UPDATE Cliente SET email = 'novo.email@email.com' WHERE id = 2;
```

3. Diminuir o estoque de um produto após uma nova venda

sql

CopiarEditar

```
UPDATE Produto SET estoque = estoque - 1 WHERE id = 3; --
```

Exemplo: reduzir estoque do terceiro produto

4. Excluir um cliente que solicitou remoção dos dados

sql

CopiarEditar

```
DELETE FROM Cliente WHERE id = 5;
```

5. Atualizar a data de uma venda para corrigir um erro

sql

CopiarEditar

```
UPDATE Venda SET data_venda = '2024-03-15' WHERE id = 2;
```

Revisão Atividade

QUESTÃO 1: Funções de Agregação

- **Funções de agregação** são usadas para calcular valores a partir de um conjunto de linhas. Exemplos incluem:
 - **AVG()**: calcula a média.
 - **SUM()**: soma os valores.
 - **COUNT()**: conta registros.
 - **MIN()** e **MAX()**: retornam o menor e o maior valor, respectivamente.
 - A alternativa correta é **b) AVG**, pois **AVG()** é uma função de agregação.
-

QUESTÃO 2: Categorias de SQL

A SQL é dividida em três categorias principais:

- **DDL (Data Definition Language)**: Define a estrutura do banco (**CREATE**, **ALTER**, **DROP**).
 - **DML (Data Manipulation Language)**: Manipula dados (**INSERT**, **UPDATE**, **DELETE**, **SELECT**).
 - **DCL (Data Control Language)**: Controla permissões (**GRANT**, **REVOKE**).
 - A alternativa correta é **b) DML, DDL e DCL**.
-

QUESTÃO 3: INNER JOIN

- **INNER JOIN** retorna apenas as linhas que possuem correspondência entre as tabelas.
- Se uma linha não tiver correspondência na outra tabela, ela será excluída do resultado.

- A alternativa correta é **e) Quando existir ao menos uma correspondência em ambas as tabelas.**
-

QUESTÃO 4: Sintaxe SQL incorreta

- Os erros mais comuns em SQL incluem:
 - Omitir **SELECT *** ao buscar todas as colunas.
 - Uso inadequado de aspas duplas em valores ("14/1/2007" deveria estar entre aspas simples '14/1/2007').
 - Faltando colunas na cláusula **SELECT FROM**.
 - A alternativa incorreta é **d) SELECT FROM RECURSO WHERE VALOR>10**, pois **SELECT** exige colunas.
-

QUESTÃO 5: CREATE TABLE pertence a qual categoria?

- **CREATE TABLE** define a estrutura da tabela, pertencendo à **DDL (Data Definition Language)**.
 - A alternativa correta é **a) DDL**.
-

QUESTÃO 6: LEFT JOIN

- **LEFT JOIN** retorna todas as linhas da tabela à esquerda e as correspondentes na tabela à direita. Se não houver correspondência, os valores da tabela à direita aparecem como **NULL**.
 - A alternativa correta é **e) Linhas e colunas da tabela da esquerda, mesmo que não haja correspondência na tabela da direita.**
-

QUESTÃO 7: Número de Instâncias em SELECTs com JOINS

- O resultado das consultas depende do tipo de **JOIN** utilizado:
 - **NATURAL JOIN**: une tabelas com base em colunas de mesmo nome.
 - **LEFT JOIN**: retorna todos da tabela da esquerda e os correspondentes da direita.
 - **RIGHT JOIN**: o contrário do **LEFT JOIN**.
 - **CROSS JOIN**: combina todas as linhas de ambas as tabelas.
 - A resposta correta depende dos dados, mas a alternativa correta é **e) 3 – 10 – 11 – 30**.
-

QUESTÃO 8: Cláusula JOIN

- **INNER JOIN**: retorna apenas registros correspondentes em ambas as tabelas.
 - **LEFT JOIN**: retorna todos os registros da tabela esquerda e os correspondentes na direita (ou **NULL** se não houver).
 - **FULL OUTER JOIN**: combina os registros de ambas as tabelas.
 - A alternativa correta é **e) São corretas as afirmativas 1, 2 e 3**.
-

QUESTÃO 9: Diferença no número de linhas entre LEFT JOIN e INNER JOIN

- **LEFT JOIN** inclui todas as linhas da tabela da esquerda, mesmo sem correspondência na tabela da direita.
 - **INNER JOIN** retorna apenas as correspondências.
 - A explicação correta é **c) T1 possui linhas cujo valor de CHAVE não está presente na T2**.
-

QUESTÃO 10: Tipos de JOINS

- **OUTER JOIN** adiciona tuplas extras ao resultado.
 - **FULL OUTER JOIN** combina **LEFT JOIN** e **RIGHT JOIN**.
 - **HIGHT OUTER JOIN** não existe.
 - A alternativa correta é **c) Apenas a III está correta**.
-

QUESTÃO 11: Diferença entre INNER JOIN e LEFT JOIN

- **INNER JOIN** retorna apenas linhas com correspondência.
 - **LEFT JOIN** retorna todas as linhas da tabela da esquerda e, se não houver correspondência, preenche com **NULL**.
 - A alternativa correta é **c) INNER JOIN retorna apenas as linhas com correspondência, enquanto LEFT JOIN retorna todas as linhas da tabela à esquerda, mesmo sem correspondência**.
-

QUESTÃO 12: Filtragem após GROUP BY

- **HAVING** é usado para filtrar grupos, enquanto **WHERE** filtra linhas individuais antes do **GROUP BY**.
- A alternativa correta é **b) HAVING**.