Apostila Técnica

Curso Completo de SQL: Do Básico ao Avançado

Uma jornada completa pelos fundamentos e práticas avançadas de SQL com comparações entre Firebird, MySQL e PostgreSQL

Sumário

1. Introdução ao SQL

2. Consultas Básicas (SELECT)

3. Filtros e Funções

4. Junções de Tabelas (JOIN)

5. Agrupamentos e Agregações

6. Subconsultas (Subqueries)

7. Modificações de Dados

8. Criação e Estrutura de Tabelas

9. Consultas Avançadas

10. SQL para Relatórios

11. Desempenho e Boas Práticas

1. Introdução ao SQL

1.1 Apresentação da linguagem SQL

SQL (Structured Query Language) é uma linguagem de programação padronizada desenvolvida para gerenciar dados em sistemas de gerenciamento de banco de dados relacionais (SGBDRs). Criada pela IBM nos anos 1970, tornou-se uma linguagem padrão ANSI/ISO para manipulação de dados em bancos de dados relacionais.

Diferente de outras linguagens de programação, SQL é uma linguagem declarativa, onde você especifica *o que* deseja obter, e não *como* obtê-lo. O SGBD é responsável por determinar a melhor maneira de executar a consulta.

A linguagem SQL é dividida em subconjuntos:

DDL (Data Definition Language): Comandos para criar e modificar estruturas de banco de dados (CREATE, ALTER, DROP)

DML (Data Manipulation Language): Comandos para manipular dados (SELECT, INSERT, UPDATE, DELETE)

DCL (Data Control Language): Comandos para controle de acesso (GRANT, REVOKE)

TCL (Transaction Control Language): Comandos para controle de transações (COMMIT, ROLLBACK)

1.2 Banco de dados vs. SGBDs

É importante entender a diferença entre um banco de dados e um Sistema Gerenciador de Banco de Dados (SGBD):

Banco de dados é uma coleção organizada de dados estruturados, armazenados eletronicamente em um sistema de computador. É o repositório onde as informações são armazenadas.

SGBD é o software que permite criar, gerenciar e manipular bancos de dados. Ele fornece uma interface entre os dados armazenados e os usuários ou aplicações que precisam acessá-los.

Principais funcionalidades de um SGBD:

Criação e manutenção de estruturas de dados

Inserção, atualização, exclusão e consulta de dados

Controle de concorrência (múltiplos acessos simultâneos)

Segurança e controle de acesso

Integridade dos dados

Backup e recuperação

Otimização de consultas

Neste curso, abordaremos três populares SGBDs relacionais: **Firebird**, **MySQL** e **PostgreSQL**.

Firebird

SGBD relacional de código aberto, descendente do Interbase. Destaca-se pela simplicidade, baixo consumo de recursos e facilidade de instalação.

Características: Estrutura de arquivo único, stored procedures, triggers, gerenciamento de transações ACID.

MySQL

Um dos SGBDs mais populares do mundo, conhecido por sua velocidade e robustez. Amplamente utilizado em aplicações web.

Características: Múltiplos mecanismos de armazenamento (InnoDB, MyISAM),

PostgreSQL

SGBD relacional-objeto avançado de código aberto, conhecido por sua conformidade com padrões e recursos extensíveis.

Características: Tipos de dados customizáveis, índices avançados, recursos de integridade, extensibilidade,

replicação, particionamento, alta performance.

procedimentos armazenados em múltiplas linguagens.

1.3 Modelos Relacionais vs. Não Relacionais

Os bancos de dados podem ser classificados em dois grandes grupos:

Bancos de Dados Relacionais

Baseados no modelo relacional proposto por E.F. Codd em 1970, que organiza dados em tabelas (relações) compostas por linhas (tuplas) e colunas (atributos). Utilizam SQL como linguagem padrão.

Características principais:

Estrutura baseada em tabelas com esquema rígido

Relações entre tabelas por meio de chaves primárias e estrangeiras

Suporte a transações ACID (Atomicidade, Consistência, Isolamento, Durabilidade)

Normalização para reduzir redundância e melhorar integridade

Bancos de Dados Não Relacionais (NoSQL)

Surgiram como alternativa aos bancos relacionais, oferecendo maior flexibilidade e escalabilidade. São projetados para lidar com grandes volumes de dados não estruturados ou semiestruturados.

Principais tipos:

Documentos: Armazenam dados como documentos (geralmente JSON ou BSON) - Ex: MongoDB

Chave-valor: Armazenam dados como pares de chave-valor - Ex: Redis

Colunar: Organizam dados por colunas em vez de linhas - Ex: Cassandra

Grafos: Otimizados para relacionamentos complexos - Ex: Neo4j

> ### Quando usar cada tipo:
>
> **Relacional:** Ideal para dados estruturados, onde as relações entre entidades são importantes e a integridade dos dados é crítica (sistemas financeiros, ERPs, CRMs).
>
> **NoSQL:** Adequado para grandes volumes de dados, requisitos de escalabilidade horizontal, esquemas flexíveis ou em evolução (redes sociais, IoT, análise em tempo real).

1.4 Objetivo do curso

Este curso foi projetado para fornecer uma formação sólida e prática em SQL, partindo dos fundamentos até técnicas avançadas. Ao final do curso, você será capaz de:

Compreender os conceitos fundamentais de bancos de dados relacionais

Escrever consultas SQL eficientes, desde as mais simples até as mais complexas

Manipular dados com precisão usando comandos DML

Criar e gerenciar estruturas de banco de dados utilizando DDL

Otimizar consultas para melhor desempenho

Identificar e aplicar as diferenças de sintaxe entre Firebird, MySQL e PostgreSQL

Desenvolver relatórios e análises baseados em dados usando SQL

Aplicar boas práticas de desenvolvimento e otimização em ambientes de produção

Para um aprendizado prático, utilizaremos um cenário de negócio fictício de uma prateleira de livros, que nos permitirá explorar diversos aspectos do SQL em um contexto real e aplicado.

> ### Nosso Cenário: Sistema de Biblioteca Virtual
>
> Ao longo do curso, trabalharemos com um banco de dados para gerenciar uma biblioteca virtual, que inclui:
>
> Cadastro de autores
>
> Cadastro de livros
>
> Cadastro de clientes/usuários
>
> Registro de vendas

> Estantes personalizadas para cada cliente
>
> Este cenário nos permitirá explorar relacionamentos complexos, consultas multi-tabelas e técnicas avançadas de SQL.

2. Consultas Básicas (SELECT)

2.1 Sintaxe básica

A instrução SELECT é a mais utilizada em SQL e serve para recuperar dados de uma ou mais tabelas. A forma mais básica de uma consulta SELECT é:

```
SELECT coluna1, coluna2, ... FROM tabela;
```

Para selecionar todas as colunas, utilize o caractere asterisco (*):

```
SELECT * FROM tabela;
```

Vamos implementar nosso modelo para a biblioteca virtual com o seguinte esquema simplificado:

```sql
-- Autores
CREATE TABLE Autores (
    autor_id INT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    biografia TEXT,
    data_nascimento DATE
);

-- Gêneros
CREATE TABLE Generos (
    genero_id INT PRIMARY KEY,
    nome VARCHAR(50) NOT NULL,
    descricao TEXT
);

-- Livros
CREATE TABLE Livros (
    livro_id INT PRIMARY KEY,
    titulo VARCHAR(200) NOT NULL,
    autor_id INT,
    genero_id INT,
    ano_publicacao INT,
    preco DECIMAL(10, 2),
    estoque INT,
    FOREIGN KEY (autor_id) REFERENCES Autores(autor_id),
    FOREIGN KEY (genero_id) REFERENCES Generos(genero_id)
);

-- Clientes
CREATE TABLE Clientes (
```

```
    cliente_id INT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE,
    data_cadastro DATE,
    telefone VARCHAR(20)
);

-- Estantes
CREATE TABLE Estantes (
    estante_id INT PRIMARY KEY,
    cliente_id INT,
    nome VARCHAR(100),
    data_criacao DATE,
    FOREIGN KEY (cliente_id) REFERENCES Clientes(cliente_id)
);

-- Itens da Estante
CREATE TABLE ItensEstante (
    estante_id INT,
    livro_id INT,
    data_adicao DATE,
    notas TEXT,
    PRIMARY KEY (estante_id, livro_id),
    FOREIGN KEY (estante_id) REFERENCES Estantes(estante_id),
    FOREIGN KEY (livro_id) REFERENCES Livros(livro_id)
);

-- Vendas
CREATE TABLE Vendas (
    venda_id INT PRIMARY KEY,
    cliente_id INT,
```

```
    data_venda DATE,
    valor_total DECIMAL(10, 2),
    FOREIGN KEY (cliente_id) REFERENCES Clientes(cliente_id)
);

-- Itens da Venda
CREATE TABLE ItensVenda (
    venda_id INT,
    livro_id INT,
    quantidade INT,
    preco_unitario DECIMAL(10, 2),
    PRIMARY KEY (venda_id, livro_id),
    FOREIGN KEY (venda_id) REFERENCES Vendas(venda_id),
    FOREIGN KEY (livro_id) REFERENCES Livros(livro_id)
);
```

Agora, vamos realizar algumas consultas básicas utilizando o comando SELECT:

```
-- Listar todos os autores SELECT * FROM autores; -- Listar apenas nomes e nacionalidades dos autores SELECT nome, nacionalidade FROM autores; -- Listar todos os livros SELECT * FROM livros; -- Listar títulos e preços dos livros SELECT titulo, preco FROM livros;
```

2.2 Alias (AS), ordenações (ORDER BY), limite (FIRST, LIMIT, TOP)

Alias (AS)

Alias são nomes temporários atribuídos a tabelas ou colunas para tornar o código mais legível ou para renomear colunas no resultado da consulta:

```
-- Renomeando colunas no resultado SELECT titulo AS "Título do Livro", preco AS "Preço (R$)" FROM livros; -- Alias para tabelas SELECT a.nome AS "Nome do Autor", l.titulo AS "Título do Livro" FROM autores a JOIN livros l ON a.autor_id = l.autor_id;
```

> ### Diferenças entre SGBDs:
>
> No MySQL e PostgreSQL, o uso da palavra-chave AS é opcional.
>
> No Firebird, o uso de AS é recomendado para manter compatibilidade com versões futuras.
>
> PostgreSQL é case-sensitive para nomes de colunas em resultados, exigindo aspas duplas para preservar maiúsculas/minúsculas.

Ordenações (ORDER BY)

A cláusula ORDER BY permite ordenar os resultados com base em uma ou mais colunas:

```
-- Ordenar livros por preço (do mais barato ao mais caro) SELECT titulo, preco FROM livros ORDER BY preco ASC; -- Ordenar livros por preço (do mais caro ao mais barato) SELECT titulo, preco FROM livros
```

```
ORDER BY preco DESC; -- Ordenação por múltiplas colunas SELECT titulo, genero, preco FROM livros
ORDER BY genero ASC, preco DESC;
```

A ordenação padrão é ASC (ascendente). Se você omitir ASC ou DESC, a ordenação será ascendente.

Limitando resultados (FIRST, LIMIT, TOP)

Para limitar o número de linhas retornadas por uma consulta, cada SGBD tem sua própria sintaxe:

MySQL

```
-- Retornar apenas os 5 primeiros livros SELECT * FROM livros LIMIT 5; -- Pular os primeiros 10 e retornar os próximos 5 (paginação) SELECT * FROM livros LIMIT 10, 5; -- Sintaxe alternativa para paginação SELECT * FROM livros LIMIT 5 OFFSET 10;
```

PostgreSQL

```
-- Retornar apenas os 5 primeiros livros SELECT * FROM livros LIMIT 5; -- Pular os primeiros 10 e retornar os próximos 5 SELECT * FROM livros LIMIT 5 OFFSET 10;
```

Firebird

```
-- Retornar apenas os 5 primeiros livros SELECT FIRST 5 * FROM livros; -- Pular os primeiros 10 e retornar os próximos 5 SELECT FIRST 5 SKIP 10 * FROM livros;
```

Importante:

Ao usar limitadores de resultados com ORDER BY, certifique-se de que a ordenação seja aplicada antes da limitação. Caso contrário, você pode obter resultados inesperados.

Exemplo correto: `SELECT * FROM livros ORDER BY preco DESC LIMIT 5;`

Exercício 2.1:

Utilizando nosso banco de dados da biblioteca virtual:

Escreva uma consulta que retorne os títulos dos livros e seus respectivos preços, ordenados do mais caro para o mais barato.

Modifique a consulta anterior para mostrar apenas os 3 livros mais caros.

Crie uma consulta que mostre o nome do cliente, o nome da estante e a data de criação, renomeando as colunas para "Nome do Cliente", "Estante" e "Data de Criação".

Dica: Utilize os recursos de alias, ORDER BY e limitadores apropriados para cada SGBD que estiver usando.

3. Filtros e Funções

3.1 WHERE, operadores de comparação

A cláusula WHERE permite filtrar registros com base em condições específicas, retornando apenas as linhas que atendem aos critérios definidos.

```
-- Sintaxe básica SELECT colunas FROM tabela WHERE condição;
```

Operadores de comparação

Operador	Descrição	Exemplo
=	Igual a	`WHERE preco = 50.00`
!=, <>	Diferente de	`WHERE genero != 'Ficção'`
>	Maior que	`WHERE preco > 30.00`
<	Menor que	`WHERE estoque < 10`
>=	Maior ou igual a	`WHERE ano_publicacao >= 2000`
<=	Menor ou igual a	`WHERE ano_publicacao <= 2020`

Podemos combinar condições usando operadores lógicos:

AND: Ambas condições devem ser verdadeiras

OR: Pelo menos uma condição deve ser verdadeira

NOT: Inverte o resultado da condição

```
-- Livros de ficção com preço acima de R$ 30,00 SELECT titulo, genero, preco FROM livros WHERE genero = 'Ficção' AND preco > 30.00; -- Livros de ficção ou biografia SELECT titulo, genero FROM livros WHERE genero = 'Ficção' OR genero = 'Biografia'; -- Livros que não são de ficção SELECT titulo, genero FROM livros WHERE NOT genero = 'Ficção'; -- Ou de forma equivalente: SELECT titulo, genero FROM livros WHERE genero != 'Ficção';
```

3.2 LIKE, IN, BETWEEN

Operador LIKE

O operador LIKE é utilizado para buscar padrões em textos, utilizando caracteres curinga:

% - Representa zero ou mais caracteres

_ - Representa exatamente um caractere

```
-- Livros que começam com "O" SELECT titulo FROM livros WHERE titulo LIKE 'O%'; -- Livros que contêm a palavra "amor" em qualquer posição do título SELECT titulo FROM livros WHERE titulo LIKE '%amor%';
```

```
-- Autores com nomes de 5 letras SELECT nome FROM autores WHERE nome LIKE '_____';
```

> ### Diferenças entre SGBDs:
>
> PostgreSQL: Oferece ILIKE para buscas case-insensitive.
>
> MySQL: Por padrão, LIKE é case-insensitive (depende da configuração de collation).
>
> Firebird: LIKE é case-sensitive. Para buscas case-insensitive, utilize UPPER() ou LOWER().

Operador IN

O operador IN verifica se um valor corresponde a qualquer valor em uma lista:

```
-- Livros de determinados gêneros SELECT titulo, genero FROM livros WHERE genero IN ('Ficção', 'Biografia', 'Romance'); -- Autores de determinadas nacionalidades SELECT nome, nacionalidade FROM autores WHERE nacionalidade IN ('Brasileiro', 'Português', 'Americano');
```

A mesma consulta poderia ser escrita com múltiplos OR, mas o uso de IN torna o código mais limpo e eficiente:

```
SELECT titulo, genero FROM livros WHERE genero = 'Ficção' OR genero = 'Biografia' OR genero = 'Romance';
```

Operador BETWEEN

O operador BETWEEN seleciona valores dentro de um intervalo (inclusive):

```
-- Livros publicados entre 2010 e 2020 SELECT titulo, ano_publicacao FROM livros WHERE ano_publicacao BETWEEN 2010 AND 2020; -- Livros com preço entre R$ 20,00 e R$ 50,00 SELECT titulo, preco FROM livros WHERE preco BETWEEN 20.00 AND 50.00; -- Vendas realizadas em um período específico SELECT venda_id, data_venda, valor_total FROM vendas WHERE data_venda BETWEEN '2023-01-01' AND '2023-12-31';
```

Equivalente usando operadores de comparação:

```
SELECT titulo, ano_publicacao FROM livros WHERE ano_publicacao >= 2010 AND ano_publicacao <= 2020;
```

3.3 Funções nativas (matemáticas, texto, data/hora)

Funções matemáticas

Função	Descrição	Exemplo
ABS(x)	Valor absoluto	`SELECT ABS(-10)` → 10
ROUND(x, d)	Arredonda x para d casas decimais	`SELECT ROUND(10.456, 2)` → 10.46

CEILING(x) ou CEIL(x)	Menor inteiro maior ou igual a x	`SELECT CEILING(10.1)` → 11
FLOOR(x)	Maior inteiro menor ou igual a x	`SELECT FLOOR(10.9)` → 10
POWER(x, y)	x elevado a y	`SELECT POWER(2, 3)` → 8
SQRT(x)	Raiz quadrada de x	`SELECT SQRT(16)` → 4

```
-- Arredondando preços para 1 casa decimal SELECT titulo, ROUND(preco, 1) AS preco_arredondado FROM livros; -- Calculando desconto de 10% SELECT titulo, preco, ROUND(preco * 0.9, 2) AS preco_com_desconto FROM livros;
```

Funções de texto

Função	Descrição	MySQL	PostgreSQL	Firebird
Comprimento	Retorna o número de caracteres	LENGTH()	LENGTH()	CHAR_LENGTH()
Maiúsculas	Converte para maiúsculas	UPPER()	UPPER()	UPPER()
Minúsculas	Converte para minúsculas	LOWER()	LOWER()	LOWER()
Substring	Extrai parte de uma string	SUBSTRING()	SUBSTRING()	SUBSTRING()
Concatenação	Une strings	CONCAT()	|| (operador)	|| (operador)

Remoção de espaços	Remove espaços em branco	TRIM()	TRIM()	TRIM()

```
-- MySQL: Concatenando nome e nacionalidade do autor SELECT CONCAT(nome, ' (', nacionalidade, ')') AS autor_info FROM autores; -- PostgreSQL/Firebird: Concatenando nome e nacionalidade do autor SELECT nome || ' (' || nacionalidade || ')' AS autor_info FROM autores; -- Convertendo títulos para maiúsculas SELECT UPPER(titulo) AS titulo_maiusculo FROM livros; -- Extraindo os primeiros 10 caracteres do título SELECT titulo, SUBSTRING(titulo, 1, 10) || '...' AS titulo_curto FROM livros WHERE LENGTH(titulo) > 10;
```

Funções de data e hora

Operação	MySQL	PostgreSQL	Firebird
Data atual	CURDATE()	CURRENT_DATE	CURRENT_DATE
Hora atual	CURTIME()	CURRENT_TIME	CURRENT_TIME
Data e hora atuais	NOW()	CURRENT_TIMESTAMP	CURRENT_TIMESTAMP
Extrair ano	YEAR()	EXTRACT(YEAR FROM)	EXTRACT(YEAR FROM)
Extrair mês	MONTH()	EXTRACT(MONTH FROM)	EXTRACT(MONTH FROM)
Extrair dia	DAY()	EXTRACT(DAY FROM)	EXTRACT(DAY FROM)

Adicionar intervalo	DATE_ADD()	+ INTERVAL	DATEADD()
Diferença entre datas	DATEDIFF()	- (operador)	DATEDIFF()
Formatar data	DATE_FORMAT()	TO_CHAR()	Usar funções de concatenação

MySQL

```
-- Calculando idade do autor SELECT nome, data_nascimento, TIMESTAMPDIFF(YEAR, data_nascimento, CURDATE()) AS idade FROM autores; -- Formatando datas SELECT venda_id, DATE_FORMAT(data_venda, '%d/%m/%Y') AS data_formatada FROM vendas; -- Vendas dos últimos 30 dias SELECT * FROM vendas WHERE data_venda >= DATE_SUB(CURDATE(), INTERVAL 30 DAY);
```

PostgreSQL

```
-- Calculando idade do autor SELECT nome, data_nascimento, EXTRACT(YEAR FROM AGE(CURRENT_DATE, data_nascimento)) AS idade FROM autores; -- Formatando datas SELECT venda_id, TO_CHAR(data_venda, 'DD/MM/YYYY') AS data_formatada FROM vendas; -- Vendas dos últimos 30 dias SELECT * FROM vendas WHERE data_venda >= CURRENT_DATE - INTERVAL '30 days';
```

Firebird

```
-- Calculando idade do autor SELECT nome, data_nascimento, EXTRACT(YEAR FROM CURRENT_DATE) - EXTRACT(YEAR FROM data_nascimento) AS idade FROM autores; -- Extraindo componentes de data SELECT venda_id, EXTRACT(DAY FROM data_venda) || '/' || EXTRACT(MONTH FROM data_venda) || '/' || EXTRACT(YEAR FROM data_venda) AS data_formatada FROM vendas; -- Vendas dos últimos 30 dias SELECT * FROM vendas
```

```
WHERE data_venda >=
DATEADD(DAY, -30,
CURRENT_DATE);
```

Exercício 3.1:

Com base no nosso banco de dados da biblioteca virtual, escreva consultas SQL para:

Encontrar livros com títulos que contenham a palavra "Segredo".

Listar todos os autores brasileiros, portugueses ou espanhóis.

Encontrar livros publicados na última década (entre 10 anos atrás e hoje).

Calcular um desconto de 15% nos livros com estoque superior a 50 unidades.

Mostrar o título de cada livro com a primeira letra em maiúscula e o restante em minúsculas.

Listar vendas realizadas no último mês, mostrando a data no formato DD/MM/YYYY.

Dica: Adapte as consultas de acordo com o SGBD que você está utilizando, observando as diferenças de sintaxe para funções de data e texto.

4. Junções de Tabelas (JOIN)

4.1 INNER JOIN, LEFT JOIN, RIGHT JOIN

Junções (JOINs) permitem combinar registros de duas ou mais tabelas com base em colunas relacionadas. Existem diferentes tipos de junções para diferentes cenários.

INNER JOIN

Retorna registros quando há correspondência em ambas as tabelas. Registros sem correspondência são excluídos do resultado.

```
-- Sintaxe básica SELECT colunas FROM tabela1 INNER JOIN tabela2 ON tabela1.coluna = tabela2.coluna;
-- Exemplo: Listar livros com seus respectivos autores SELECT l.titulo, a.nome AS autor FROM livros l
INNER JOIN autores a ON l.autor_id = a.autor_id;
```

No exemplo acima, apenas livros que têm um autor correspondente na tabela de autores serão incluídos no resultado.

LEFT JOIN (LEFT OUTER JOIN)

Retorna todos os registros da tabela à esquerda (primeira tabela) e os registros correspondentes da tabela à direita. Se não houver correspondência, os resultados da tabela da direita serão NULL.

```
-- Sintaxe básica SELECT colunas FROM tabela1 LEFT JOIN tabela2 ON tabela1.coluna = tabela2.coluna; -
- Exemplo: Listar todos os autores e seus livros (incluindo autores sem livros) SELECT a.nome AS
autor, l.titulo FROM autores a LEFT JOIN livros l ON a.autor_id = l.autor_id;
```

No exemplo acima, todos os autores serão incluídos no resultado, mesmo aqueles que não têm livros cadastrados. Nesses casos, a coluna "titulo" terá valor NULL.

RIGHT JOIN (RIGHT OUTER JOIN)

Retorna todos os registros da tabela à direita (segunda tabela) e os registros correspondentes da tabela à esquerda. Se não houver correspondência, os resultados da tabela da esquerda serão NULL.

```
-- Sintaxe básica SELECT colunas FROM tabela1 RIGHT JOIN tabela2 ON tabela1.coluna = tabela2.coluna;
-- Exemplo: Listar todos os livros e seus autores (incluindo livros sem autor) SELECT l.titulo,
a.nome AS autor FROM autores a RIGHT JOIN livros l ON a.autor_id = l.autor_id;
```

> ### Nota:
>
> É comum haver confusão sobre LEFT e RIGHT JOIN. Uma forma de lembrar é:
>
> LEFT JOIN mantém todos os registros da tabela mencionada antes do JOIN (à esquerda)
>
> RIGHT JOIN mantém todos os registros da tabela mencionada depois do JOIN (à direita)

> Na prática, muitos desenvolvedores preferem usar apenas LEFT JOIN e trocar a ordem das tabelas quando necessário, para maior clareza no código.

> ### Diferenças entre SGBDs:
>
> **Firebird:** Suporta INNER JOIN, LEFT JOIN e RIGHT JOIN normalmente.
>
> **MySQL:** Suporta INNER JOIN, LEFT JOIN e RIGHT JOIN normalmente.
>
> **PostgreSQL:** Suporta INNER JOIN, LEFT JOIN e RIGHT JOIN normalmente, mas oferece suporte mais avançado a junções como FULL OUTER JOIN.
>
> Todos os três SGBDs usam a palavra-chave JOIN, embora alguns permitam sintaxes alternativas.

4.2 LEFT OUTER JOIN e diferenças entre tipos

Além dos tipos básicos de JOIN, existem outros tipos que podem ser úteis em cenários específicos:

FULL OUTER JOIN

Retorna todos os registros quando há uma correspondência em qualquer uma das tabelas. Combina o efeito de LEFT e RIGHT JOIN.

PostgreSQL

```
SELECT a.nome AS autor, l.titulo FROM autores a FULL OUTER JOIN livros l ON a.autor_id = l.autor_id;
```

MySQL

```
-- MySQL não suporta FULL OUTER JOIN diretamente -- É necessário simular com UNION SELECT a.nome AS autor, l.titulo FROM autores a LEFT JOIN livros l ON a.autor_id = l.autor_id UNION SELECT a.nome AS autor, l.titulo FROM autores a RIGHT JOIN livros l ON a.autor_id = l.autor_id WHERE a.autor_id IS NULL;
```

Firebird

```
SELECT a.nome AS autor, l.titulo FROM autores a FULL JOIN livros l ON a.autor_id = l.autor_id;
```

CROSS JOIN

Retorna o produto cartesiano de duas tabelas, ou seja, combina cada linha da primeira tabela com cada linha da segunda tabela. Não requer cláusula ON.

```
-- Exemplo: Combinando todos os livros com todos os clientes (geralmente não é útil) SELECT l.titulo, c.nome AS cliente FROM livros l CROSS JOIN clientes c;
```

O CROSS JOIN raramente é usado em aplicações reais, pois gera um grande número de linhas (produto do número de linhas das duas tabelas) e geralmente não tem significado lógico.

SELF JOIN

Não é um tipo específico de JOIN, mas uma técnica em que uma tabela é associada a si mesma. É útil para consultar relações hierárquicas ou comparações dentro da mesma tabela.

```
-- Exemplo: Estrutura de funcionários e gerentes (não está em nosso modelo, apenas ilustrativo)
SELECT e1.nome AS funcionario, e2.nome AS gerente FROM funcionarios e1 LEFT JOIN funcionarios e2 ON
e1.gerente_id = e2.funcionario_id;
```

4.3 ON vs USING

Há duas formas principais de especificar a condição de junção: usando ON ou USING.

Cláusula ON

Permite especificar qualquer condição de junção, não apenas igualdade entre colunas. É mais flexível e pode conter múltiplas condições.

```
-- Junção com ON SELECT l.titulo, a.nome AS autor FROM livros l INNER JOIN autores a ON l.autor_id =
a.autor_id; -- Múltiplas condições SELECT l.titulo, a.nome AS autor FROM livros l INNER JOIN autores
```

```
a ON l.autor_id = a.autor_id AND a.nacionalidade = 'Brasileiro';
```

Cláusula USING

Uma alternativa mais concisa quando as colunas de junção têm o mesmo nome em ambas as tabelas.

```
-- Junção com USING SELECT l.titulo, a.nome AS autor FROM livros l INNER JOIN autores a USING
(autor_id);
```

A cláusula USING é uma forma mais simples quando:

A junção é feita com base em igualdade entre colunas

As colunas de junção têm o mesmo nome nas tabelas

> ### Importante:
>
> Quando você usa USING, a coluna de junção aparece apenas uma vez no resultado, enquanto com ON, ambas as colunas aparecem no resultado (uma para cada tabela).

> ### Compatibilidade:
>
> **PostgreSQL:** Suporta tanto ON quanto USING.

MySQL: Suporta tanto ON quanto USING.

Firebird: Suporta ON, mas o suporte a USING pode variar de acordo com a versão.

Para maior compatibilidade entre SGBDs, é recomendável usar ON.

4.4 Diferenças entre bancos na sintaxe de joins

Embora a sintaxe padrão ANSI SQL seja amplamente suportada, existem diferenças e recursos específicos em cada SGBD.

Sintaxe alternativa (antiga) para junções

Antes da sintaxe ANSI JOIN, era comum usar uma forma implícita de junções, que ainda é suportada em muitos SGBDs:

```
-- Sintaxe implícita (antiga) para INNER JOIN SELECT l.titulo, a.nome AS autor FROM livros l, autores a WHERE l.autor_id = a.autor_id;
```

Aviso:

A sintaxe implícita não é recomendada por várias razões:

É menos legível, especialmente com múltiplas junções

É mais propensa a erros (se esquecer a condição WHERE, resulta em um CROSS JOIN)

Mistura condições de junção com filtros, tornando a consulta menos clara

Não suporta explicitamente LEFT, RIGHT ou FULL JOINs
É sempre preferível usar a sintaxe ANSI JOIN com ON ou USING.

Diferenças específicas entre SGBDs

Recurso	PostgreSQL	MySQL	Firebird
FULL OUTER JOIN	Suportado nativamente	Requer UNION de LEFT e RIGHT JOIN	Suportado como FULL JOIN
NATURAL JOIN	Suportado	Suportado	Suporte limitado
Junções com USING	Suportado	Suportado	Suporte variável por versão

Dica de desempenho:

Independentemente do SGBD, ao escrever consultas com junções:

Mencione a tabela menor primeiro em LEFT JOINs quando possível

Certifique-se de que as colunas de junção estão indexadas

Use INNER JOIN em vez de LEFT JOIN quando souber que todas as linhas têm correspondência

> Evite junções desnecessárias; verifique se a consulta realmente precisa dos dados de todas as tabelas

Exercício 4.1:

Com base no nosso banco de dados da biblioteca virtual, escreva consultas SQL para:

Listar todos os livros com seus respectivos autores usando INNER JOIN.

Listar todos os autores e seus livros, incluindo autores que não têm livros cadastrados.

Listar todos os clientes e suas estantes, incluindo clientes que não têm estantes.

Listar todos os livros, os autores e as estantes em que os livros estão presentes.

Reescrever uma das consultas acima usando a cláusula USING em vez de ON (caso seu SGBD suporte).

Desafio: Escreva uma consulta para encontrar pares de livros do mesmo autor que estão na mesma estante (dica: use SELF JOIN).

5. Agrupamentos e Agregações

5.1 GROUP BY, HAVING

A cláusula GROUP BY permite agrupar registros com base em uma ou mais colunas, geralmente para aplicar funções de agregação a cada grupo.

```
-- Sintaxe básica SELECT coluna1, coluna2, FUNÇÃO_AGREGAÇÃO(coluna3) FROM tabela GROUP BY coluna1, coluna2;
```

Exemplos de GROUP BY

```
-- Contar quantos livros cada autor tem SELECT a.nome AS autor, COUNT(l.livro_id) AS quantidade_livros FROM autores a LEFT JOIN livros l ON a.autor_id = l.autor_id GROUP BY a.nome; -- Calcular o valor total de livros por gênero SELECT genero, COUNT(*) AS quantidade, SUM(preco) AS valor_total, AVG(preco) AS preco_medio FROM livros GROUP BY genero;
```

Cláusula HAVING

A cláusula HAVING é usada para filtrar grupos, assim como WHERE filtra linhas. A diferença é que HAVING é aplicada após o agrupamento e pode usar funções de agregação.

```
-- Sintaxe básica SELECT coluna1, FUNÇÃO_AGREGAÇÃO(coluna2) FROM tabela GROUP BY coluna1 HAVING condição_com_função_agregação;
```

Exemplos de HAVING

```
-- Encontrar autores com mais de 3 livros SELECT a.nome AS autor, COUNT(l.livro_id) AS quantidade_livros FROM autores a LEFT JOIN livros l ON a.autor_id = l.autor_id GROUP BY a.nome HAVING COUNT(l.livro_id) > 3; -- Encontrar gêneros com preço médio acima de R$ 40,00 SELECT genero, AVG(preco) AS preco_medio FROM livros GROUP BY genero HAVING AVG(preco) > 40.00;
```

WHERE vs HAVING:

WHERE: Filtra linhas antes do agrupamento

HAVING: Filtra grupos após o agrupamento

Para melhor desempenho, use WHERE para filtrar linhas antes de agrupá-las, e HAVING apenas para condições que precisam de funções de agregação.

```
-- Combinando WHERE e HAVING SELECT genero, COUNT(*) AS quantidade, AVG(preco) AS preco_medio FROM livros WHERE ano_publicacao >= 2000 -- Filtro antes do agrupamento GROUP BY genero HAVING COUNT(*) > 5; -- Filtro após o agrupamento
```

5.2 Funções agregadoras (SUM, COUNT, AVG, etc.)

Funções de agregação realizam cálculos em um conjunto de valores e retornam um único valor. São comumente usadas com GROUP BY, mas também podem ser usadas sem agrupamento.

Função	Descrição	Exemplo
COUNT()	Conta o número de linhas ou valores não nulos	`COUNT(livro_id)`, `COUNT(*)`
SUM()	Calcula a soma dos valores	`SUM(preco)`
AVG()	Calcula a média dos valores	`AVG(preco)`
MIN()	Retorna o menor valor	`MIN(preco)`
MAX()	Retorna o maior valor	`MAX(preco)`

Exemplos sem GROUP BY

```
-- Estatísticas gerais sobre a biblioteca SELECT COUNT(*) AS total_livros, AVG(preco) AS preco_medio, MIN(preco) AS livro_mais_barato, MAX(preco) AS livro_mais_caro, SUM(preco * estoque) AS valor_total_estoque FROM livros;
```

Exemplos com GROUP BY

```
-- Vendas por mês SELECT EXTRACT(YEAR FROM data_venda) AS ano, EXTRACT(MONTH FROM data_venda) AS mes, COUNT(*) AS total_vendas, SUM(valor_total) AS receita_total FROM vendas GROUP BY EXTRACT(YEAR FROM data_venda), EXTRACT(MONTH FROM data_venda) ORDER BY ano, mes; -- Livros mais vendidos SELECT l.titulo, SUM(iv.quantidade) AS total_vendido FROM livros l JOIN itens_venda iv ON l.livro_id = iv.livro_id GROUP BY l.titulo ORDER BY total_vendido DESC;
```

Considerações importantes:

COUNT(*) conta todas as linhas, incluindo nulos

COUNT(coluna) conta apenas valores não nulos na coluna

AVG, SUM, MIN e MAX ignoram valores NULL

DISTINCT pode ser usado com funções de agregação: `COUNT(DISTINCT autor_id)`

5.3 OVER() e janela de funções (com comparação entre bancos)

As funções de janela (window functions) permitem realizar cálculos em um conjunto de linhas relacionadas à linha atual, sem a necessidade de agrupar as linhas em um único resultado como acontece com GROUP BY.

A cláusula OVER()

A cláusula OVER() define a "janela" ou conjunto de linhas em que a função é aplicada. Pode incluir particionamento, ordenação e delimitação da janela.

```
-- Sintaxe básica SELECT coluna1, coluna2, FUNÇÃO_JANELA() OVER([PARTITION BY coluna] [ORDER BY coluna] [frame_clause]) AS resultado FROM tabela;
```

Funções de janela comuns

Função	Descrição
ROW_NUMBER()	Número sequencial único para cada linha
RANK()	Classificação com lacunas para valores duplicados
DENSE_RANK()	Classificação sem lacunas para valores duplicados
NTILE(n)	Divide as linhas em n grupos de tamanhos aproximadamente iguais
LAG(expr, n)	Valor da expressão n linhas antes da linha atual
LEAD(expr, n)	Valor da expressão n linhas após a linha atual
SUM(), AVG(), COUNT(), etc.	Funções de agregação usadas como funções de janela

PostgreSQL

MySQL

Firebird

```
-- Listar livros com seu preço e preço médio do gênero SELECT titulo, genero, preco, AVG(preco) OVER(PARTITION BY genero) AS preco_medio_genero FROM livros; -- Classificar livros por preço dentro de cada gênero SELECT titulo, genero, preco, RANK() OVER(PARTITION BY genero ORDER BY preco DESC) AS rank_preco FROM livros; -- Calcular receita acumulada por mês SELECT EXTRACT(YEAR FROM data_venda) AS ano, EXTRACT(MONTH FROM data_venda) AS mes, SUM(valor_total) AS receita_mensal, SUM(SUM(valor_total)) OVER( PARTITION BY EXTRACT(YEAR FROM data_venda) ORDER BY EXTRACT(MONTH FROM data_venda) ROWS UNBOUNDED PRECEDING ) AS receita_acumulada FROM
```

```
-- MySQL 8.0+ suporta todas as funções de janela -- Listar livros com seu preço e preço médio do gênero SELECT titulo, genero, preco, AVG(preco) OVER(PARTITION BY genero) AS preco_medio_genero FROM livros; -- Classificar livros por preço dentro de cada gênero SELECT titulo, genero, preco, RANK() OVER(PARTITION BY genero ORDER BY preco DESC) AS rank_preco FROM livros; -- Calcular receita acumulada por mês SELECT YEAR(data_venda) AS ano, MONTH(data_venda) AS mes, SUM(valor_total) AS receita_mensal, SUM(SUM(valor_total)) OVER( PARTITION BY YEAR(data_venda) ORDER BY MONTH(data_venda) ROWS UNBOUNDED PRECEDING ) AS receita_acumulada FROM
```

```
-- Firebird 3.0+ suporta funções de janela -- Listar livros com seu preço e preço médio do gênero SELECT titulo, genero, preco, AVG(preco) OVER(PARTITION BY genero) AS preco_medio_genero FROM livros; -- Classificar livros por preço dentro de cada gênero SELECT titulo, genero, preco, RANK() OVER(PARTITION BY genero ORDER BY preco DESC) AS rank_preco FROM livros; -- Para versões anteriores, é necessário usar subconsultas SELECT l.titulo, l.genero, l.preco, (SELECT AVG(preco) FROM livros WHERE genero = l.genero) AS preco_medio_genero FROM livros l;
```

```
vendas GROUP BY
EXTRACT(YEAR FROM
data_venda), EXTRACT(MONTH
FROM data_venda) ORDER BY
ano, mes;
```

```
vendas GROUP BY
YEAR(data_venda),
MONTH(data_venda) ORDER BY
ano, mes;
```

Compatibilidade:

PostgreSQL: Suporte completo a funções de janela desde a versão 8.4

MySQL: Suporte completo a funções de janela a partir da versão 8.0

Firebird: Suporte completo a funções de janela a partir da versão 3.0

Para versões mais antigas, muitas operações de janela podem ser simuladas com subconsultas ou auto-joins, mas com menos eficiência.

Casos de uso comuns para funções de janela

Comparar valores atuais com médias, máximos ou mínimos de grupos

Calcular rankings, percentis ou quartis

Calcular valores cumulativos ou médias móveis

Comparar valores com linhas anteriores ou seguintes (análise de tendências)

Paginar resultados com controle preciso

Dica de desempenho:

Funções de janela são executadas após WHERE, GROUP BY e HAVING, mas antes de ORDER BY na cláusula externa. Isso significa que você não pode filtrar com base nos resultados de uma função de janela diretamente em WHERE. Se precisar filtrar com base nesses resultados, use uma subconsulta ou CTE.

Exercício 5.1:

Com base no nosso banco de dados da biblioteca virtual, escreva consultas SQL para:

Calcular o número total de livros e o valor médio por gênero.

Encontrar os 3 autores com mais livros cadastrados.

Listar os clientes e o valor total de suas compras, mostrando apenas aqueles que gastaram mais de R$ 200,00.

Usando funções de janela, classificar os livros por preço dentro de cada gênero e mostrar apenas os 2 mais caros de cada gênero.

Calcular o percentual que cada livro representa no valor total do estoque.

Desafio: Para cada venda, mostrar o valor da venda, o valor total de todas as vendas do mesmo cliente, e o percentual que essa venda representa no total do cliente.

6. Subconsultas (Subqueries)

6.1 Subqueries em SELECT, WHERE e FROM

Subconsultas (também chamadas de subqueries) são consultas SQL aninhadas dentro de outra consulta. Elas podem aparecer em diferentes partes da consulta principal e servem para fornecer dados que serão usados pela consulta externa.

Subconsultas em SELECT

Permitem incluir valores calculados por outra consulta no resultado de cada linha.

```
-- Listar livros com o número de vezes que foram vendidos SELECT l.titulo, l.preco, (SELECT SUM(quantidade) FROM itens_venda iv WHERE iv.livro_id = l.livro_id) AS total_vendido FROM livros l; -- Listar autores com o número de livros e o preço médio SELECT a.nome AS autor, (SELECT COUNT(*) FROM livros l WHERE l.autor_id = a.autor_id) AS num_livros, (SELECT AVG(preco) FROM livros l WHERE l.autor_id = a.autor_id) AS preco_medio FROM autores a;
```

> Importante:

> Subconsultas em SELECT devem retornar apenas um valor (escalar) para cada linha da consulta externa. Se a subconsulta retornar múltiplas linhas, ocorrerá um erro.

Subconsultas em FROM

Permitem usar o resultado de uma consulta como uma tabela virtual (derivada) que pode ser consultada e unida a outras tabelas.

```
-- Encontrar os 3 gêneros mais vendidos SELECT g.genero, g.total_vendido FROM ( SELECT l.genero, SUM(iv.quantidade) AS total_vendido FROM livros l JOIN itens_venda iv ON l.livro_id = iv.livro_id GROUP BY l.genero ) AS g ORDER BY g.total_vendido DESC LIMIT 3; -- Calcular o valor médio das vendas por cliente SELECT c.nome AS cliente, v.total_vendas, v.valor_total, v.valor_total / v.total_vendas AS valor_medio FROM clientes c JOIN ( SELECT cliente_id, COUNT(*) AS total_vendas, SUM(valor_total) AS valor_total FROM vendas GROUP BY cliente_id ) AS v ON c.cliente_id = v.cliente_id;
```

Subconsultas em WHERE

Usadas para criar condições de filtro com base nos resultados de outra consulta.

```
-- Encontrar livros com preço acima da média SELECT titulo, preco FROM livros WHERE preco > (SELECT AVG(preco) FROM livros); -- Encontrar autores que têm livros de ficção SELECT nome FROM autores WHERE autor_id IN ( SELECT DISTINCT autor_id FROM livros WHERE genero = 'Ficção' );
```

6.2 Comparações com operadores (EXISTS, IN, = ANY)

Existem diferentes operadores que podem ser usados com subconsultas, dependendo do tipo de comparação que se deseja fazer.

Operador IN

Verifica se um valor está presente em um conjunto de valores retornados por uma subconsulta.

```
-- Encontrar clientes que compraram livros de ficção SELECT DISTINCT c.nome FROM clientes c WHERE c.cliente_id IN ( SELECT v.cliente_id FROM vendas v JOIN itens_venda iv ON v.venda_id = iv.venda_id JOIN livros l ON iv.livro_id = l.livro_id WHERE l.genero = 'Ficção' );
```

Operador EXISTS

Verifica se a subconsulta retorna pelo menos uma linha. É frequentemente mais eficiente que IN quando a subconsulta retornaria muitas linhas.

```
-- Encontrar autores que têm pelo menos um livro com preço acima de R$ 50,00 SELECT a.nome FROM autores a WHERE EXISTS ( SELECT 1 FROM livros l WHERE l.autor_id = a.autor_id AND l.preco > 50.00 );
-- Encontrar clientes que nunca compraram SELECT c.nome FROM clientes c WHERE NOT EXISTS ( SELECT 1 FROM vendas v WHERE v.cliente_id = c.cliente_id );
```

> **Dica de desempenho:**
>
> Em consultas EXISTS, o valor selecionado (SELECT 1, SELECT *, etc.) é irrelevante, pois o que importa é se a consulta retorna linhas, não quais valores ela retorna. Usar SELECT 1 é uma convenção para indicar isso claramente.

Operadores de comparação com subconsultas

Operador	Descrição
= ANY	Verdadeiro se o valor for igual a qualquer valor retornado pela subconsulta
< ANY	Verdadeiro se o valor for menor que qualquer valor retornado pela subconsulta
> ANY	Verdadeiro se o valor for maior que qualquer valor retornado pela subconsulta
= ALL	Verdadeiro se o valor for igual a todos os valores retornados pela subconsulta
< ALL	Verdadeiro se o valor for menor que todos os valores retornados pela subconsulta
> ALL	Verdadeiro se o valor for maior que todos os valores retornados pela subconsulta

```
-- Encontrar livros mais caros que qualquer livro de romance SELECT titulo, preco FROM livros WHERE preco > ANY ( SELECT preco FROM livros WHERE genero = 'Romance' ); -- Encontrar livros mais caros que
```

```
todos os livros de romance SELECT titulo, preco FROM livros WHERE preco > ALL ( SELECT preco FROM livros WHERE genero = 'Romance' );
```

PostgreSQL

Suporta todos os operadores de subconsulta mencionados (IN, EXISTS, ANY, ALL) e oferece suporte adicional a expressões de tabela comuns (CTEs) com a cláusula WITH, que pode tornar subconsultas complexas mais legíveis e eficientes.

MySQL

Suporta IN, EXISTS, ANY, ALL e também oferece suporte a CTEs com WITH desde a versão 8.0. Versões anteriores requerem subconsultas aninhadas ou tabelas temporárias.

Firebird

Suporta IN, EXISTS e operadores de comparação com subconsultas. O suporte a CTEs com WITH está disponível a partir da versão 2.1.

6.3 Subconsultas correlacionadas

Subconsultas correlacionadas são aquelas que fazem referência a colunas da consulta externa. Elas são avaliadas uma vez para cada linha da consulta externa.

```
-- Encontrar livros com preço acima da média de seu gênero SELECT l.titulo, l.genero, l.preco FROM livros l WHERE l.preco > ( SELECT AVG(l2.preco) FROM livros l2 WHERE l2.genero = l.genero ); -- Encontrar o livro mais caro de cada autor SELECT a.nome AS autor, l.titulo, l.preco FROM autores a
```

```
JOIN livros l ON a.autor_id = l.autor_id WHERE l.preco = ( SELECT MAX(l2.preco) FROM livros l2 WHERE l2.autor_id = a.autor_id );
```

Considerações de desempenho:

Subconsultas correlacionadas podem ser menos eficientes, pois são executadas repetidamente para cada linha da consulta externa. Em muitos casos, é possível reescrever a consulta usando JOIN, funções de janela ou CTEs para melhor desempenho.

Reescrevendo com funções de janela

```
-- Reescrevendo o exemplo anterior com função de janela SELECT autor, titulo, preco FROM ( SELECT a.nome AS autor, l.titulo, l.preco, MAX(l.preco) OVER(PARTITION BY a.autor_id) AS max_preco FROM autores a JOIN livros l ON a.autor_id = l.autor_id ) AS subq WHERE preco = max_preco;
```

Exercício 6.1:

Com base no nosso banco de dados da biblioteca virtual, escreva consultas SQL para:

Encontrar livros que nunca foram vendidos.

Listar autores cujo preço médio dos livros está acima da média geral.

Encontrar clientes que compraram todos os livros de um determinado autor.

Para cada livro, mostrar quanto seu preço está acima ou abaixo da média do seu gênero (em percentual).

Encontrar estantes que contêm pelo menos um livro de cada gênero existente.
Desafio: Encontrar pares de autores que têm livros na mesma estante, mas nunca tiveram livros vendidos na mesma venda.

7. Modificações de Dados

7.1 INSERT, UPDATE, DELETE

Além de consultar dados, o SQL permite inserir, atualizar e excluir registros. Essas operações são realizadas com os comandos INSERT, UPDATE e DELETE, respectivamente.

INSERT

O comando INSERT adiciona novos registros a uma tabela. Existem várias formas de utilizá-lo:

```
-- Inserção básica de um único registro INSERT INTO autores (autor_id, nome, nacionalidade) VALUES
(1, 'Carlos Drummond de Andrade', 'Brasileiro'); -- Inserção com todos os campos (na ordem das
colunas da tabela) INSERT INTO autores VALUES (2, 'José Saramago', '1922-11-16', 'Português',
'Escritor português, vencedor do Prêmio Nobel de Literatura.'); -- Inserção de múltiplos registros de
```

```
uma só vez INSERT INTO autores (autor_id, nome, nacionalidade) VALUES (3, 'Gabriel García Márquez', 'Colombiano'), (4, 'Jane Austen', 'Inglesa'), (5, 'Machado de Assis', 'Brasileiro');
```

Também é possível inserir dados com base em uma consulta:

```
-- Inserir clientes na tabela de clientes_premium com base em critérios INSERT INTO clientes_premium (cliente_id, nome, nivel) SELECT c.cliente_id, c.nome, 'Ouro' AS nivel FROM clientes c JOIN vendas v ON c.cliente_id = v.cliente_id GROUP BY c.cliente_id, c.nome HAVING SUM(v.valor_total) > 1000.00;
```

Diferenças entre SGBDs:

PostgreSQL: Oferece a cláusula RETURNING para retornar dados inseridos.

MySQL: Para chaves auto incrementais, pode-se obter o ID gerado com LAST_INSERT_ID().

Firebird: Possui construções como RETURNING para obter valores gerados por triggers ou sequências.

UPDATE

O comando UPDATE modifica registros existentes em uma tabela:

```
-- Atualização básica UPDATE livros SET preco = 39.90 WHERE livro_id = 101; -- Atualização de múltiplas colunas UPDATE livros SET preco = 45.50, estoque = estoque - 5 WHERE genero = 'Ficção'; --
```

```
Atualização com base em outras tabelas UPDATE livros l SET preco = l.preco * 1.1 -- aumento de 10%
WHERE l.autor_id IN ( SELECT autor_id FROM autores WHERE nacionalidade = 'Brasileiro' );
```

PostgreSQL

```
-- Atualizações em massa com FROM UPDATE livros l SET preco = l.preco * 1.1 FROM autores a WHERE l.autor_id = a.autor_id AND a.nacionalidade = 'Brasileiro'; -- Retornando dados atualizados UPDATE livros SET estoque = estoque - 1 WHERE livro_id = 101 RETURNING titulo, estoque AS novo_estoque;
```

MySQL

```
-- Atualizações em massa com JOIN UPDATE livros l JOIN autores a ON l.autor_id = a.autor_id SET l.preco = l.preco * 1.1 WHERE a.nacionalidade = 'Brasileiro'; -- Atualizações com ORDER BY e LIMIT UPDATE livros SET preco = preco * 0.8 -- desconto de 20% WHERE genero = 'Romance' ORDER BY preco DESC LIMIT 5; -- apenas os 5 mais caros
```

Firebird

```
-- Atualizações com base em condições UPDATE livros l SET preco = l.preco * 1.1 WHERE l.autor_id IN ( SELECT a.autor_id FROM autores a WHERE a.nacionalidade = 'Brasileiro' ); -- Retornando dados atualizados UPDATE livros SET estoque = estoque - 1 WHERE livro_id = 101 RETURNING titulo, estoque AS novo_estoque;
```

DELETE

O comando DELETE remove registros de uma tabela:

```
-- Exclusão básica DELETE FROM itens_venda WHERE venda_id = 500; -- Exclusão com subconsulta DELETE FROM livros WHERE estoque = 0 AND livro_id NOT IN ( SELECT DISTINCT livro_id FROM itens_venda );
```

Cuidado!

O comando DELETE sem uma cláusula WHERE excluirá TODAS as linhas da tabela. Sempre verifique sua condição WHERE antes de executar um DELETE.

Em ambientes de produção, considere iniciar com uma consulta SELECT para verificar quais registros serão afetados:

```
-- Verificar primeiro SELECT * FROM livros WHERE estoque = 0; -- Somente então excluir DELETE FROM livros WHERE estoque = 0;
```

7.2 Inserções múltiplas

Inserir múltiplos registros em uma única operação é mais eficiente do que realizar várias inserções individuais.

```
-- Inserir múltiplos livros de uma vez INSERT INTO livros (livro_id, titulo, autor_id, genero, preco, estoque) VALUES (201, 'O Alquimista', 10, 'Ficção', 29.90, 50), (202, 'Brida', 10, 'Ficção', 27.50, 30), (203, 'Onze Minutos', 10, 'Romance', 32.80, 25);
```

Também podemos usar INSERT com múltiplos VALUES em conjunto com funções e expressões:

```
-- Inserir registros de venda com cálculos INSERT INTO vendas (venda_id, cliente_id, data_venda, valor_total) VALUES (1001, 50, CURRENT_DATE, 120.50), (1002, 35, CURRENT_DATE - INTERVAL '1 day', 85.75), (1003, 22, CURRENT_DATE - INTERVAL '2 day', 255.30);
```

Recursos avançados para inserções em massa

PostgreSQL

```
-- COPY: Carregar dados de um arquivo COPY livros(livro_id, titulo, autor_id, preco) FROM '/caminho/para/livros.csv' WITH (FORMAT CSV, HEADER);
-- ON CONFLICT: Inserir ou atualizar (upsert) INSERT INTO autores (autor_id, nome) VALUES (10, 'Paulo Coelho') ON CONFLICT (autor_id) DO UPDATE SET nome = EXCLUDED.nome;
```

MySQL

```
-- LOAD DATA: Carregar dados de um arquivo LOAD DATA INFILE '/caminho/para/livros.csv' INTO TABLE livros FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n' IGNORE 1 LINES (livro_id, titulo, autor_id, preco); -- INSERT ... ON DUPLICATE KEY UPDATE INSERT INTO autores (autor_id, nome) VALUES (10, 'Paulo Coelho') ON
```

Firebird

```
-- Inserções com base em SELECT INSERT INTO autores_brasileiros (autor_id, nome) SELECT autor_id, nome FROM autores WHERE nacionalidade = 'Brasileiro'; -- Para importações em massa, frequentemente -- utiliza-se ferramentas externas como gbak -- ou APIs específicas
```

```
DUPLICATE KEY UPDATE nome =
VALUES(nome);
```

7.3 Cuidados com WHERE

A cláusula WHERE em operações de modificação de dados (UPDATE e DELETE) define quais registros serão afetados. É crucial usá-la corretamente para evitar modificações indesejadas.

Práticas de segurança:

Sempre inclua uma cláusula WHERE específica para limitar o escopo das modificações.

Teste sua condição WHERE primeiro com uma consulta SELECT para verificar quais registros serão afetados.

Use chaves primárias ou identificadores únicos sempre que possível nas condições.

Realize as operações dentro de transações para poder reverter em caso de erro.

Mantenha backups regulares do banco de dados.

```
-- Abordagem segura: testar primeiro com SELECT SELECT * FROM livros WHERE genero = 'Ficção' AND
preco < 30.00; -- Após verificar os resultados, proceder com UPDATE UPDATE livros SET preco = preco *
```

```
1.05 WHERE genero = 'Ficção' AND preco < 30.00;
```

Usando transações:

```
-- Iniciar transação BEGIN TRANSACTION; -- Realizar operações UPDATE livros SET preco = preco *
1.10 WHERE genero = 'Romance'; -- Verificar resultados (opcional) SELECT * FROM livros WHERE
genero = 'Romance'; -- Se tudo estiver correto: COMMIT; -- Se houver problemas: ROLLBACK;
```

7.4 Integridade referencial

A integridade referencial assegura que relações entre tabelas permaneçam consistentes, impedindo que registros relacionados fiquem orfãos ou inconsistentes após operações de exclusão ou atualização.

Chaves estrangeiras e ações referenciais

Ao definir uma chave estrangeira, é possível especificar o que deve acontecer quando o registro referenciado é atualizado ou excluído:

```
CREATE TABLE itens_venda ( venda_id INT, livro_id INT, quantidade INT, PRIMARY KEY (venda_id,
livro_id), FOREIGN KEY (venda_id) REFERENCES vendas(venda_id) ON DELETE CASCADE ON UPDATE CASCADE,
```

```
FOREIGN KEY (livro_id) REFERENCES livros(livro_id) ON DELETE RESTRICT ON UPDATE CASCADE );
```

Opções de ação referencial

Ação	Descrição
CASCADE	Propaga a alteração/exclusão para os registros relacionados
RESTRICT	Impede a alteração/exclusão se existirem registros relacionados
SET NULL	Define como NULL a chave estrangeira nos registros relacionados
SET DEFAULT	Define como o valor padrão a chave estrangeira nos registros relacionados
NO ACTION	Similar a RESTRICT, mas verifica ao final da transação

Exemplos de uso:

CASCADE: Se um cliente for excluído, todas as suas estantes também serão excluídas.

RESTRICT: Impedir a exclusão de um livro que está em alguma venda ou estante.

SET NULL: Se um autor for excluído, os livros ficam sem autor (autor_id = NULL).

SET DEFAULT: Se um autor for excluído, atribuir um autor padrão aos livros.

PostgreSQL

Suporta todas as opções de ação referencial (CASCADE, RESTRICT, SET NULL, SET DEFAULT, NO ACTION).

MySQL

O engine InnoDB suporta todas as ações referenciais, mas o MyISAM não implementa integridade referencial.

Firebird

Suporta CASCADE, RESTRICT, SET NULL e SET DEFAULT. O comportamento padrão é RESTRICT.

Exercício 7.1:

Com base no nosso banco de dados da biblioteca virtual, escreva consultas SQL para:

Inserir um novo autor e, em seguida, três livros desse autor.

Atualizar o preço de todos os livros do gênero "Suspense", aumentando em 8%.

Excluir todas as estantes que não contêm nenhum livro.

Inserir uma nova venda com três itens diferentes, calculando o valor total automaticamente.

Transferir todos os livros de uma estante para outra, e em seguida excluir a estante vazia.

Desafio: Escreva um script SQL que realize uma "venda" completa: atualize o estoque dos livros vendidos, crie o registro de venda, insira os itens da venda e atualize o valor total da venda com base nos itens. Use transações para garantir a integridade da operação.

8. Criação e Estrutura de Tabelas

8.1 CREATE TABLE

O comando CREATE TABLE é usado para criar novas tabelas no banco de dados, definindo sua estrutura, colunas, tipos de dados e restrições.

```
-- Sintaxe básica CREATE TABLE nome_tabela ( coluna1 tipo_dado [restricoes], coluna2 tipo_dado [restricoes], ..., [restricoes_tabela] );
```

Exemplo completo de criação de tabela:

```
CREATE TABLE livros ( livro_id INT PRIMARY KEY, titulo VARCHAR(200) NOT NULL, subtitulo VARCHAR(300), autor_id INT, editora VARCHAR(100), ano_publicacao INT CHECK (ano_publicacao > 1450), isbn VARCHAR(20) UNIQUE, paginas INT, genero VARCHAR(50), idioma CHAR(2) DEFAULT 'PT', preco DECIMAL(10,2) NOT NULL, estoque INT NOT NULL DEFAULT 0, data_cadastro DATE DEFAULT CURRENT_DATE, FOREIGN KEY (autor_id) REFERENCES autores(autor_id) );
```

Restrições comuns

Restrição	Descrição

PRIMARY KEY	Identifica exclusivamente cada registro (combina NOT NULL e UNIQUE)
FOREIGN KEY	Estabelece link com outra tabela, garantindo integridade referencial
NOT NULL	Garante que a coluna não aceite valores nulos
UNIQUE	Garante que todos os valores na coluna sejam diferentes
CHECK	Garante que os valores em uma coluna satisfaçam uma condição
DEFAULT	Define um valor padrão para a coluna quando não especificado

PostgreSQL

```
-- Tabela com SERIAL (auto incremento) CREATE TABLE clientes ( cliente_id SERIAL PRIMARY KEY, nome VARCHAR(100) NOT NULL, email VARCHAR(100) UNIQUE, data_cadastro TIMESTAMP DEFAULT CURRENT_TIMESTAMP ); -- Tabela com herança (específico do PostgreSQL) CREATE TABLE produtos_fisicos ( peso
```

MySQL

```
-- Tabela com AUTO_INCREMENT CREATE TABLE clientes ( cliente_id INT AUTO_INCREMENT PRIMARY KEY, nome VARCHAR(100) NOT NULL, email VARCHAR(100) UNIQUE, data_cadastro TIMESTAMP DEFAULT CURRENT_TIMESTAMP, INDEX idx_nome (nome) ) ENGINE=InnoDB; -- Com definição de charset e collation CREATE TABLE
```

Firebird

```
-- Tabela com generator para auto incremento CREATE GENERATOR GEN_CLIENTE_ID; CREATE TABLE clientes ( cliente_id INTEGER NOT NULL PRIMARY KEY, nome VARCHAR(100) NOT NULL, email VARCHAR(100) UNIQUE, data_cadastro TIMESTAMP DEFAULT CURRENT_TIMESTAMP ); -- Trigger para auto incremento CREATE TRIGGER
```

```
DECIMAL(10,2), dimensoes
VARCHAR(50) ) INHERITS
(produtos);
```

```
livros ( livro_id INT
PRIMARY KEY, titulo
VARCHAR(200) NOT NULL,
descricao TEXT )
ENGINE=InnoDB CHARACTER SET
utf8mb4 COLLATE
utf8mb4_unicode_ci;
```

```
trg_cliente_bi FOR clientes
BEFORE INSERT AS BEGIN IF
(NEW.cliente_id IS NULL)
THEN NEW.cliente_id =
GEN_ID(GEN_CLIENTE_ID, 1);
END;
```

8.2 Tipos de dados nos bancos (comparativos)

Cada SGBD implementa seus próprios tipos de dados, embora muitos sejam semelhantes entre diferentes sistemas. Abaixo, uma comparação dos tipos mais comuns:

Tipos numéricos

Conceito	PostgreSQL	MySQL	Firebird
Inteiros	SMALLINT, INTEGER, BIGINT	TINYINT, SMALLINT, INT, BIGINT	SMALLINT, INTEGER, BIGINT
Auto incremento	SERIAL, BIGSERIAL	AUTO_INCREMENT	Generators + Triggers
Decimais fixos	NUMERIC(p,s), DECIMAL(p,s)	NUMERIC(p,s), DECIMAL(p,s)	NUMERIC(p,s), DECIMAL(p,s)

Ponto flutuante	REAL, DOUBLE PRECISION	FLOAT, DOUBLE	FLOAT, DOUBLE PRECISION
Monetário	MONEY	DECIMAL	NUMERIC(15,2)

Tipos de texto

Conceito	PostgreSQL	MySQL	Firebird
Tamanho fixo	CHAR(n)	CHAR(n)	CHAR(n)
Tamanho variável	VARCHAR(n), TEXT	VARCHAR(n), TEXT	VARCHAR(n), BLOB SUB_TYPE TEXT
Texto longo	TEXT	TEXT, MEDIUMTEXT, LONGTEXT	BLOB SUB_TYPE TEXT

Tipos de data e hora

Conceito	PostgreSQL	MySQL	Firebird
Data	DATE	DATE	DATE
Hora	TIME, TIME WITH TIME ZONE	TIME	TIME
Data e hora	TIMESTAMP, TIMESTAMP WITH TIME ZONE	DATETIME, TIMESTAMP	TIMESTAMP
Intervalo	INTERVAL	Não tem tipo nativo	Não tem tipo nativo

Outros tipos

Conceito	PostgreSQL	MySQL	Firebird
Booleano	BOOLEAN	BOOLEAN, TINYINT(1)	BOOLEAN (Firebird 3.0+)
Binário	BYTEA	BINARY, VARBINARY, BLOB	BLOB
JSON	JSON, JSONB	JSON	Não tem tipo nativo
UUID	UUID	CHAR(36)	CHAR(36)
Enumeração	ENUM	ENUM	Não tem tipo nativo
Array	Qualquer tipo[]	Não tem tipo nativo	Não tem tipo nativo

Considerações na escolha dos tipos:

Espaço de armazenamento: Escolha tipos que ocupem apenas o espaço necessário.

Precisão e escala: Para valores monetários, use NUMERIC/DECIMAL com precisão adequada.

Performance: Tipos menores são processados mais rapidamente.

Compatibilidade: Considere a portabilidade entre SGBDs se for um requisito.

Validação: Alguns tipos têm validação automática (ex: DATE, BOOLEAN).

8.3 Chaves primárias, estrangeiras, índices e constraints

Chaves primárias (PRIMARY KEY)

Uma chave primária é uma coluna ou conjunto de colunas que identifica unicamente cada registro em uma tabela. Cada tabela pode ter apenas uma chave primária.

```
-- Definição na criação da tabela CREATE TABLE autores ( autor_id INT PRIMARY KEY, nome VARCHAR(100) NOT NULL ); -- Chave primária composta CREATE TABLE estante_livros ( estante_id INT, livro_id INT, data_adicao DATE, PRIMARY KEY (estante_id, livro_id) ); -- Adição posterior com ALTER TABLE CREATE TABLE clientes ( cliente_id INT, nome VARCHAR(100) NOT NULL ); ALTER TABLE clientes ADD PRIMARY KEY (cliente_id);
```

Características de uma chave primária:

Deve ser única para cada registro

Não pode ser NULL

Deve ser estável (não mudar com frequência)

É automaticamente indexada para rápida recuperação