Apostila de Banco de Dados

Felipe Tagawa Reis

22 de fevereiro de 2026

Sumário

SEÇÃO 1

Introdução

1.1. Conceitos Iniciais.— A linguagem SQL possui recursos para definição da estrutura de dados, para modificar dados no banco de dados e recursos para especificar restrições de segurança e integridade.

1.1.1. Introdução Histórica. – A linguagem Structured English Query Language (SEQUEL) foi desenvolvida pela IBM Corporation, Inc., para usar o modelo de Codd. SEQUEL mais tarde se tornou SQL (ainda pronunciado ”sequel”). Em 1979, a Relational Software, Inc. (agora Oracle) introduziu a primeira implementação de SQL disponível comercialmente. Hoje, SQL é aceito como a linguagem RDBMS padrão.

1.1.2. Prós e Contras da Linguagem. – Como toda linguagem, SQL possui vantagens e desvantagens em sua utilização. Como prós, pode-se citar:

i. Garantia ACID [1.2];

ii. Junções complexas e consultas poderosas [2.6];

iii. Padronização e maturidade.

Já como Contras, é possível citar:

iv. Escalabilidade horizontal mais difícil;

v. Esquema rígido;

vi. Menor desempenho em cenários de Big Data.

1.1.3. Tipos de Dados. – Os tipos de dados SQL são divididos em: Numéricos, *Strings*, Data/Hora e Lógicos.

Tabela 1: Tipos de Dados

Numéricos	**Strings**	**Data**	**Lógico**
INTEGER(INT)	CHAR(n)	DATE	BOOLEAN
DECIMAL(i,j)	VARCHAR(n)	TIME	TINYINT(1)
FLOAT(j)	TEXT	DATETIME	
DOUBLE			

Legenda

i: quantidade de números totais; j: quantidade de números flutuantes; n: quantidade de caracteres.

1.2. ACID.— Em ciência da computação, ACID (acrônimo de Atomicidade, Consistência, Isolamento e Durabilidade - do inglês: Atomicity, Consistency, Isolation, Durability) é um conjunto de propriedades de transação em banco de dados.

1.2.1. Atomicidade. – Primeira propriedade ACID, a Atomicidade trata de transações serem indivisíveis (atômicas) - todas as operações devem ser realizadas com sucesso, ou deverão ser totalmente desfeitas.
Como exemplo, pode-se considerar uma operação de banco: havendo qualquer tipo de problema, como falha segurança ou não chegada do valor na conta destino, toda a operação deve ser revertida.

1.2.2. Consistência. – A consistência deve ser mantida de um estado de Banco para outro seguinte. Ela é mantida pela manutenção da integridade geral dos dados - todos os dados devem ser válidos de acordo com as regras especificadas, restrições (constraints) [1.3], operações em cascata [1.4.2] e triggers [3.3].

1.2.3. Isolamento. – Conjunto de técnicas utilizadas visando impossibilitar que transações possam interferir uma nas outras - idealmente evitar concorrência.

1.2.4. Durabilidade. – A Durabilidade é a garantia de disponibilidade dos dados, mesmo em situações adversas, como quedas de internet e de energia, travamentos ou erros.

1.3. Constraints.— SQL *constraints* são utilizadas para especificar regras para os dados nas tabelas. Além disso, são usadas a fim de limitar o tipo de dado que pode entrar em uma tabela, garantindo precisão e confiabilidade dos dados. Se houver qualquer tipo de violação entre a restrição e a ação de dados, a ação será abortada.

1.3.1. Chaves Primárias. – Chaves Primárias (PK) são a base de toda tabela relacional, uma vez que desempenham o papel de identificar unicamente um registro dentro de uma tabela. Esta chave pode compreender uma ou mais colunas, sendo neste último caso, a chave primária chamada de composta, se necessário.

```
CREATE TABLE Pessoa(
    idPessoa INT Primary Key NOT NULL -- NÃO pode ser NULL
);
```

1.3.2. Chaves Estrangeiras. – Chaves Estrangeiras (FK) são a base de toda relação entre tabelas, uma vez que desempenham o papel de referenciar a chave primária de outra tabela. Esta chave estabelece um link efetivo entre os registros de duas tabelas, mantendo a integridade referencial dentro do Banco de Dados.

```
CREATE TABLE Pessoa(
    idPessoa INT Primary Key NOT NULL,
    Carro_idCarro INT NOT NULL,
    CONSTRAINT FKCarro
    FOREIGN KEY (Carro_idCarro)
    REFERENCES Carro (idCarro)
);
```

1.3.3. NOT NULL. – Garante que uma coluna não possa ter o valor *NULL* (*NULL* é diferente de 0(zero)).

```
CREATE TABLE Funcionarios (
idFuncionario INT PRIMARY KEY,
-- O nome e o cargo são obrigatórios (NOT NULL)
nome VARCHAR(100) NOT NULL,
cargo VARCHAR(50) NOT NULL,
-- O telefone pode ficar vazio (NULL por padrão)
telefone VARCHAR(15)
);

-- Este comando causará ERRO porque o campo 'nome' não foi preenchido
INSERT INTO Funcionarios (idFuncionario, cargo) VALUES (1, 'Engenheiro');
```

1.3.4. UNIQUE. – Garante que todos os valores de uma coluna sejam diferentes.

```
CREATE TABLE Pessoa(
    idPessoa INT Primary Key NOT NULL,
    Nome VARCHAR(45) UNIQUE -- Nomes não poderão ser iguais
);
```

1.3.5. CHECK. – Garante que todos os valores de uma coluna satisfaçam uma condição específica, como por exemplo, inserir somente clientes com idade ¿ 18 anos.

```
CREATE TABLE Pessoa(
    idPessoa INT Primary Key NOT NULL,
    Nome VARCHAR(45) UNIQUE,
    Idade INT CHECK (idade > 18) -- Apenas maiores de idade
);
```

1.3.6. DEFAULT. – Garante um valor padrão para uma coluna se nenhum valor for fornecido.

```sql
CREATE TABLE Pessoa(
    idPessoa INT Primary Key NOT NULL,
    Nome VARCHAR(45) UNIQUE,
    Idade INT CHECK (idade > 18),
    Saldo DOUBLE DEFAULT 20.0
);
```

1.4. Modificadores.— Controlam o comportamento funcional e administrativo dos comandos ou das colunas. Podem determinar como o banco deve gerar novos valores automaticamente (AUTO_INCREMENT) ou até qual motor de armazenamento será utilizado para gerir aquela tabela (ENGINE).

1.4.1. AUTO_INCREMENT. – Utilizado geralmente em chaves primárias para gerar uma sequência numérica automática.

```sql
CREATE TABLE Alunos (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL
);
```

1.4.2. ON DELETE / UPDATE CASCADE. – Este é um modificador de Integridade Referencial usado em Chaves Estrangeiras (FOREIGN KEY). Ele define o que acontece com os dados filhos quando o registro pai é alterado ou removido.

- CASCADE: Se você deletar um cliente, todos os pedidos dele também serão deletados automaticamente.

```sql
CREATE TABLE Pedidos (
    idPedido INT PRIMARY KEY,
    id_cliente INT,
    FOREIGN KEY (id_cliente) REFERENCES Clientes(id)
    ON DELETE CASCADE -- Remove pedidos se o cliente for deletado
);
```

1.4.3. ENGINE. – Define o motor de armazenamento da tabela. Para garantir as propriedades ACID citadas na Seção [1.2], utilizamos o `InnoDB`.

InnoDB : Suporta transações (ACID) e chaves estrangeiras - Mais Comum;

MyISAM : Mais rápido para leituras, mas não suporta transações.

```sql
CREATE TABLE Log_Sistema (
    id INT PRIMARY KEY,
    mensagem TEXT
) ENGINE=InnoDB;
```

1.5. Relacionamentos.— Os relacionamentos de banco de dados são associações entre tabelas que são criadas usando instruções de junção para recuperar dados.

1.5.1. Relacionamento 1:1. – Ambas tabelas podem ter somente um registro de cada lado do relacionamento.

```sql
-- Exemplo: Pessoa e seu Curriculum
CREATE TABLE Curriculum (
idCurr INT PRIMARY KEY,
descricao TEXT,
idPessoa INT UNIQUE, -- UNIQUE garante que uma pessoa só tenha um currículo
FOREIGN KEY (idPessoa) REFERENCES Pessoa(idPessoa)
);
```

1.5.2. Relacionamento 1:N. – A tabela de chave primária contém somente um registro relacionado a nenhum, a um ou a muitos registros da tabela relacionada.

```sql
-- Exemplo: Departamento e Funcionários
CREATE TABLE Funcionarios (
idFunc INT PRIMARY KEY,
nome VARCHAR(45),
idDepto INT, -- Vários funcionários pertencem a um departamento
FOREIGN KEY (idDepto) REFERENCES Departamento(idDepto)
ON DELETE CASCADE -- Se o depto for excluído, os funcionários também serão
);
```

1.5.3. Relacionamento N:M. – Cada registro em ambas as tabelas pode se relacionar a nenhum ou a qualquer número de registros na outra tabela. Esses relacionamentos requerem uma terceira tabela, chamada de tabela associada ou de associação, pois os sistemas relacionais não podem acomodar diretamente o relacionamento.

```sql
-- Exemplo: Alunos e Disciplinas
-- Um aluno faz várias disciplinas e uma disciplina tem vários alunos.

CREATE TABLE Aluno_Disciplina (
idAluno INT,
idDisc INT,
PRIMARY KEY (idAluno, idDisc), -- Chave primária composta
FOREIGN KEY (idAluno) REFERENCES Aluno(idAluno),
FOREIGN KEY (idDisc) REFERENCES Disciplina(idDisc)
);
```

1.5.4. Relacionamento Recursivo (Auto-Relacionamento). – O auto-relacionamento ocorre quando uma tabela se relaciona com ela mesma. Utilizado para representar hierarquias como categorias e subcategorias.

```sql
CREATE TABLE Categoria (
    id        INT PRIMARY KEY,
    nome      VARCHAR(100),
    parent_id INT NULL,  -- NULL = categoria raiz
    FOREIGN KEY (parent_id) REFERENCES Categoria(id)
);

-- Inserindo categorias e subcategorias
INSERT INTO Categoria VALUES (1, 'Eletronicos', NULL);
INSERT INTO Categoria VALUES (2, 'Celulares',   1);
INSERT INTO Categoria VALUES (3, 'Notebooks',   1);
```

A `FOREIGN KEY` aponta para a própria tabela. Categorias com `parent_id NULL` são as principais; as demais são subcategorias.

1.6. Models.— Os Models (Modelos) são classes que representam as entidades do mundo real dentro do código. Eles funcionam como um ”espelho”das tabelas do banco de dados, onde cada classe corresponde a uma tabela e cada atributo da classe corresponde a uma coluna.

1.6.1. Entidades. – As classes são definidas de forma simples, contendo apenas os dados e os métodos de acesso (quando linkadas a Java(POJOs) ou Python(POPOs)).

1.6.2. Mapeamento Objeto-Relacional. – Técnica que permite converter dados entre sistemas incompatíveis: a orientação a objetos (código) e o modelo relacional (banco de dados).

Abstração de Tabelas: Cada classe no código representa uma tabela no SQL, onde os atributos da classe mapeiam as colunas da tabela;

Conversão de Tipos: Resolve as diferenças entre tipos, como converter um VARCHAR para uma String.

Gerenciamento de Chaves: O mapeamento define como as chaves primárias (@Id) e estrangeiras são tratadas para manter a integridade referencial.

1.6.3. Validação de Dados. – A validação de dados nos Models funciona como uma primeira linha de defesa antes que a informação chegue ao Banco de Dados. Ela garante que os dados sigam as regras de negócio definidas, evitando erros de execução no SQL.

Validação de Presença: Verifica se campos marcados como NOT NULL no banco de dados foram preenchidos no objeto antes do envio ao DAO.

Consistência de Tipos: Garante que valores numéricos não contenham caracteres alfanuméricos, prevenindo falhas de inserção em colunas INT ou DOUBLE.

SEÇÃO 2

Comandos da Linguagem

2.1. Comandos DDL.— A DDL fornece comandos para definição e modificação de esquemas de relação, remoção de relações e criação de índices. Os principais comandos que fazem parte da DDL são: CREATE, ALTER, DROP e TRUNCATE.

2.1.1. Comando CREATE. – CREATE é o comando de **Criação** de objetos em um Banco de Dados, sendo responsável pela criação do Banco de Dados em si, usuários, tabelas e estruturas.

```sql
CREATE TABLE Clientes (
    id INT PRIMARY KEY,
    nome VARCHAR(100) NOT NULL
);
```

Tabela 2: Exemplo de Tabela de Clientes

ID Cliente	Nome	Idade	Cidade
101	João Silva	35	São Paulo
102	Maria Oliveira	28	Rio de Janeiro
103	Pedro Souza	42	Belo Horizonte
104	Ana Costa	22	Curitiba

2.1.2. Comando DROP. – DROP é o comando de **Remoção** de objetos em um Banco de Dados, sendo responsável pela eliminação do Banco de Dados, usuários, tabelas e estruturas.

```sql
DROP TABLE Clientes; -- Removendo a tabela Clientes do Banco;
DROP USER Professor'@'%; -- Removendo o usuário Professor do Banco;
DROP DATABASE BD; -- Removendo o Banco BD;
DROP PROCEDURE nomeProcedure; -- Removendo a Procedure.
```

2.1.3. Comando ALTER. – ALTER é o comando de **Modificação** de objetos em um Banco de Dados, sendo responsável pela alteração de usuários, atributos de tabelas.

```sql
ALTER TABLE Clientes ADD Saldo Double; -- Adicionando novo Atributo;
ALTER TABLE Clientes DROP COLUMN Cidade; -- Eliminando o Atributo 'Cidade';
ALTER TABLE Clientes MODIFY Nome VARCHAR(35) NOT NULL; -- Alterando um Atributo (tamanho e/ou restrição).
```

Tabela após as modificações (considerando que R$ 20,00 é o saldo de cada um):

Tabela 3: Tabela de Clientes Modificada

ID Cliente	Nome	Idade	Saldo
101	João Silva	35	R$ 20,00
102	Maria Oliveira	28	R$ 20,00
103	Pedro Souza	42	R$ 20,00
104	Ana Costa	22	R$ 20,00

2.1.4. Comando TRUNCATE. – TRUNCATE é o comando de **Limpeza** de tabelas dentro de um Banco de Dados, sendo responsável pela remoção de todas as instâncias anteriormente inseridas. A diferença do TRUNCATE para o DELETE está principalmente no reinício do identificador para 1.

```sql
TRUNCATE TABLE Clientes; -- Deleta todos os registros, resetando o ID.
```

Resultado do comando TRUNCATE na tabela 'Clientes':

Tabela 4: Tabela de Clientes Truncada

ID Cliente	Nome	Idade	Saldo
1			
2			
3			
4			

2.1.5. Comando CREATE INDEX. – Usado para criar e recuperar dados do Banco de Dados de forma mais eficiente.

```sql
-- Criando um índice para acelerar buscas pelo nome da pessoa
CREATE INDEX idx_nome ON Pessoa(Nome);

-- Exemplo de busca que será otimizada pelo índice acima
SELECT * FROM Pessoa WHERE Nome = 'Felipe Reis';
```

Dica de Performance

Embora acelerem as consultas (`SELECT`), os índices podem deixar as operações de escrita (`INSERT`, `UPDATE`) um pouco mais lentas, pois o banco precisa atualizar o índice toda vez que os dados mudam. Use-os em colunas que aparecem frequentemente em cláusulas `WHERE`.

2.2. *Modificadores de Definição.*—

2.2.1. IF EXISTS. – É utilizado nos comandos de remoção (DROP) ou alteração (ALTER). Ele garante que o comando só tente agir sobre um objeto que realmente exista no banco.

```sql
-- Remove a tabela apenas se ela for encontrada no banco
DROP TABLE IF EXISTS Departamentos;

-- Também pode ser usado para remover outros objetos, como bancos de dados
DROP DATABASE IF EXISTS sistema_escolar;
```

2.2.2. IF NOT EXISTS. – É utilizado principalmente no comando CREATE. Ele instrui o banco de dados a criar o objeto apenas se ele ainda não estiver presente no esquema.

```sql
-- Cria a tabela apenas se ela não existir, evitando erros de duplicidade
CREATE TABLE IF NOT EXISTS Departamentos (
idDept INT PRIMARY KEY,
nomeDept VARCHAR(50) NOT NULL
);
```

2.3. *Comandos DML.*— A SQL DML compreende comandos para inserir, remover e modificar informações em um banco de dados. Os comandos básicos da DML são: INSERT, UPDATE, DELETE.

2.3.1. Comando INSERT. – Para inserir novos registros dentro de uma tabela, deve-se usar o comando INSERT INTO e fornecer os valores especificados para cada campo. Considerando a tabela previamente criada [2]:

```
INSERT INTO Clientes VALUES (
105, 'Felipe Reis',
21, 'Poços de Caldas'
);
```

Tabela 5: Tabela de Clientes com Novo Registro

ID Cliente	Nome	Idade	Cidade
101	João Silva	35	São Paulo
102	Maria Oliveira	28	Rio de Janeiro
103	Pedro Souza	42	Belo Horizonte
104	Ana Costa	22	Curitiba
105	Felipe Reis	21	Poços de Caldas

2.3.2. Comando UPDATE. – A fim de modificar algum registro (instância) dentro de uma tabela, o comando UPDATE é utilizado. Suponha que o cliente 'Maria Oliveira' mude da cidade 'Rio de Janeiro' para 'Santa Rita do Sapucaí':

```
UPDATE Clientes
SET Cidade = 'Santa Rita do Sapucaí'
WHERE Nome = 'Maria Oliveira';
```

Tabela 6: Tabela de Clientes com Registro Atualizado

ID Cliente	Nome	Idade	Cidade
101	João Silva	35	São Paulo
102	Maria Oliveira	28	Santa Rita do Sapucaí
103	Pedro Souza	42	Belo Horizonte
104	Ana Costa	22	Curitiba
105	Felipe Reis	21	Poços de Caldas

2.3.3. Comando DELETE. – Dados naturalmente precisam ser removidos das tabelas criadas, por motivos diversos. Nesse caso, utiliza-se o comando `DELETE` para deletar algum registro por completo. Levando em consideração a última tabela [6], suponha que o cliente 'Ana Costa' não seja mais um cliente:

```
DELETE FROM Clientes
WHERE ID = 104; -- ID do cliente 'Ana Costa'.
```

Tabela 7: Tabela de Clientes com Registro Removido

ID Cliente	Nome	Idade	Cidade
101	João Silva	35	São Paulo
102	Maria Oliveira	28	Santa Rita do Sapucaí
103	Pedro Souza	42	Belo Horizonte
105	Felipe Reis	21	Poços de Caldas

Note que o cliente 'Felipe Reis' permaneceu no ID '105' e não se tornou o novo '104'.

Importante!!

Não confunda o comando DELETE com o TRUNCATE, aquele remove apenas registros especificados e **NÃO RESETA O IDENTIFICADOR**, enquanto este remove TODOS os registros de uma só vez, resetando o ID.

2.4. Comandos DCL.— É o conjunto de comandos que fazem o cadastramento de usuários e determinam seu nível de privilégio para os objetos do banco de dados. Os principais comandos são: GRANT, REVOKE.

2.4.1. Comando GRANT. – A declaração GRANT em MySQL é usada para conceder permissões a usuários ou grupos de usuários para executar determinadas ações em um banco de dados específico.

1. A sintaxe básica da declaração GRANT no MySQL é a seguinte:

```
GRANT permissão(s) ON nome_banco.nome_tabela TO 'usuario'@'host' IDENTIFIED BY 'senha';
```

2. Conceder acesso total ao banco de dados db_biblioteca para o usuário felipe:

```
GRANT ALL PRIVILEGES ON db_biblioteca.* TO 'felipe'@'localhost' IDENTIFIED BY '123';
```

3. Conceder acesso somente inserção e leitura a um banco de dados para o usuário na tabela autores:

```
GRANT INSERT, SELECT ON db_biblioteca.tbl_autores TO 'felipe'@'localhost' IDENTIFIED BY '123';
```

2.4.2. Comando REVOKE. – A declaração REVOKE em MySQL é usada para revogar as permissões concedidas anteriormente a um usuário ou grupo de usuários em um banco de dados ou tabela específica.

1. A sintaxe básica da declaração REVOKE no MySQL é a seguinte:

```
REVOKE permissão(s) ON nome_banco_dados.nome_tabela FROM 'usuario'@'host';
```

2. Revogar todas as permissões do usuário felipe no banco de dados db_biblioteca:

```
REVOKE ALL PRIVILEGES ON db_biblioteca.* FROM 'felipe'@'localhost';
```

3. Revogar a permissão de inserção na tabela de autores para um grupo de usuários:

```
REVOKE INSERT ON db_biblioteca.tbl_autores FROM 'grupo_usuarios'@'localhost';
```

2.4.3. Comando FLUSH PRIVILEGES. – O comando `FLUSH PRIVILEGES` recarrega as tabelas de permissões do MySQL, aplicando imediatamente qualquer alteração feita manualmente nas tabelas de controle de acesso do banco de dados.

```
-- Concedendo privilégios a um usuário
GRANT ALL PRIVILEGES ON banco_exemplo.* TO 'usuario'@'localhost';

-- Aplicando as alterações imediatamente
FLUSH PRIVILEGES;
```

Observação

Na prática, ao usar `GRANT` e `REVOKE` diretamente, o MySQL já aplica as mudanças automaticamente. O `FLUSH PRIVILEGES` é necessário apenas quando as tabelas de permissão são alteradas manualmente via `INSERT`, `UPDATE` ou `DELETE`.

2.5. Comandos TCL.— A TCL compreende comandos que gerenciam as mudanças feitas por comandos DML. Elas permitem que várias operações sejam agrupadas em uma única unidade lógica chamada **transação**.

2.5.1. Comando COMMIT. – O comando `COMMIT` é utilizado para salvar permanentemente todas as alterações feitas durante a transação atual no banco de dados. Após o COMMIT, as alterações não podem ser desfeitas pelo banco.

```
START TRANSACTION;
INSERT INTO Clientes (id, nome, idade, cidade)
VALUES (106, 'Maria Silva', 19, 'Santa Rita do Sapucaí');
COMMIT; -- Salva a nova cliente permanentemente
```

2.5.2. Comando ROLLBACK. – O `ROLLBACK` é o comando de ”arrependimento”. Ele desfaz todas as alterações da transação atual que ainda não foram salvas pelo COMMIT. É vital para manter a integridade caso ocorra um erro.

Suponha que você deletou todos os clientes por engano sem a cláusula `WHERE`:

```
START TRANSACTION;
DELETE FROM Clientes; -- Erro gravíssimo: deletou tudo!
ROLLBACK; -- Desfaz a remoção e restaura os dados abaixo
```

Tabela 8: Estado da Tabela Restaurada após ROLLBACK

ID Cliente	Nome	Idade	Cidade
101	João Silva	35	São Paulo
102	Maria Oliveira	28	Santa Rita do Sapucaí
103	Pedro Souza	42	Belo Horizonte
105	Felipe Reis	21	Poços de Caldas
106	Maria Silva	19	Santa Rita do Sapucaí

2.5.3. Comando SAVEPOINT. – O `SAVEPOINT` cria um ”ponto de restauração”dentro de uma transação. Isso permite que você desfaça apenas parte das operações, voltando para um estado específico sem cancelar a transação inteira.

```
START TRANSACTION;
INSERT INTO Clientes VALUES (107, 'Monitoria Inatel', 25, 'SRS');
SAVEPOINT ponto1; -- Ponto de segurança criado

DELETE FROM Clientes WHERE id = 101; -- Deleta o João
ROLLBACK TO SAVEPOINT ponto1; -- O João volta, mas o ID 107 permanece!
COMMIT;
```

Dica de Transações

Lembre-se: no MySQL, comandos DDL (como `CREATE` ou `DROP`) causam um **Commit Implicito**. Ou seja, você não consegue dar `ROLLBACK` na criação de uma tabela!

2.6. *Comandos DQL.*—

A SQL DQL inclui uma linguagem de consulta baseada na álgebra relacional e no cálculo relacional.

2.6.1. Busca Simples: SELECT sem JOIN. – Este tipo de busca aborda apenas uma tabela.

a. Buscar todos os clientes, ordenados por nome:

```
SELECT *
FROM Clientes
ORDER BY nome ASC;
```

b. Retornar apenas um subconjunto de colunas:

```
SELECT id,
    nome,
    cidade AS Localidade
FROM Clientes;
```

c. Retornar apenas clientes de uma cidade específica e com idade menor que 30:

```
SELECT nome,
    idade,
    cidade
FROM Clientes
WHERE cidade = 'Santa Rita do Sapucaí'
    AND idade < 30;
```

2.6.2. Busca Complexa: SELECT com JOIN. – Este tipo de busca aborda duas ou mais tabelas.

a. **JOIN em Relacionamento 1:1 (Pessoa e Currículo):** Utilizado para buscar dados complementares que foram separados por questões de organização ou performance.

```
SELECT P.Nome, C.descricao AS Curriculo
FROM Pessoa P
JOIN Curriculum C ON P.idPessoa = C.idPessoa;
```

b. **JOIN em Relacionamento 1:N (Departamentos e Funcionários):** É o uso mais comum, onde buscamos a descrição de um atributo que está representado apenas por um ID na tabela de origem.

```
SELECT F.nome AS Funcionario, D.nomeDept AS Departamento
FROM Funcionarios F
INNER JOIN Departamentos D ON F.idDepto = D.idDepto;
```

c. **JOIN em Relacionamento N:M (Alunos e Disciplinas):** Para este caso, são necessários **dois JOINs**, pois precisamos passar pela tabela intermediária para conectar as duas pontas.

```
SELECT A.nome AS Aluno, D.nomeDisc AS Disciplina
FROM Aluno A
JOIN Aluno_Disciplina AD ON A.idAluno = AD.idAluno
JOIN Disciplina D ON AD.idDisc = D.idDisc;
```

2.6.3. Filtros e Parâmetros de Busca. – I. Filtros (Seleção de Registros)

1. **Lógica WHERE:** É a cláusula fundamental para filtrar registros baseada em uma condição.

```
SELECT * FROM Clientes WHERE idade >= 18;
```

2. **AND, OR e NOT:** Operadores lógicos que permitem combinar múltiplas condições.

```
SELECT * FROM Produtos
WHERE preco < 100 AND NOT categoria = 'Luxo';
```

3. **Filtro de Texto (% e _):** Utilizados com o operador `LIKE` para buscas parciais. O `%` representa qualquer sequência de caracteres, enquanto o `_` representa exatamente um caractere.

```
SELECT * FROM Alunos WHERE nome LIKE 'Mar%'; -- Começa com Mar
SELECT * FROM Alunos WHERE nome LIKE '_a%';  -- Segunda letra é 'a'
```

4. **LIMIT:** Restringe a quantidade de linhas retornadas pela consulta. Essencial para paginação em sistemas Java/Python.

```
SELECT * FROM Logs ORDER BY data DESC LIMIT 10;
```

5. **DISTINCT:** Elimina linhas duplicadas no resultado do `SELECT`.

```
SELECT DISTINCT cidade FROM Clientes;
```

6. **Funções de Agregação (AVG, SUM, COUNT):** Realizam cálculos sobre um conjunto de valores.

```
SELECT AVG(salario) AS MediaSalarial FROM Funcionarios;
```

II. Parâmetros de Organização e Agrupamento

1. **GROUP BY:** Agrupa linhas que possuem valores iguais em colunas específicas, geralmente usado com funções de agregação.

```
SELECT cidade, COUNT(*) FROM Clientes GROUP BY cidade;
```

2. **HAVING:** Semelhante ao `WHERE`, mas aplicado após o agrupamento (`GROUP BY`).

```
SELECT cidade, AVG(idade) FROM Clientes
GROUP BY cidade HAVING AVG(idade) > 25;
```

3. **ORDER BY:** Ordena o conjunto de resultados em ordem ascendente (`ASC`) ou descendente (`DESC`).

```
SELECT * FROM Produtos ORDER BY preco DESC;
```

4. **BETWEEN:** Seleciona valores dentro de um intervalo inclusivo.

```
SELECT * FROM Vendas WHERE data BETWEEN '2026-01-01' AND '2026-02-01';
```

5. **LIKE:** Operador de comparação de strings (visto anteriormente com curingas).

6. **IN (e relação com UNION):** O `IN` permite especificar múltiplos valores em uma cláusula `WHERE`. Funciona como uma alternativa mais limpa ao uso de vários `OR` ou de um `UNION` de queries simples.

```
-- Busca clientes de cidades específicas
SELECT * FROM Clientes WHERE cidade IN ('São Paulo', 'Poços de Caldas', 'SRS');
```

2.6.4. Comando SHOW. – SHOW é o comando de recuperação de metadados, detalhes de configuração do servidor e estrutura do Banco de Dados.

```
SHOW DATABASES; -- Mostra os diversos Banco de Dados criados;
SHOW TABLES; -- Mostra as tabelas criadas dentro do Banco de Dados;
SHOW COLUMNS FROM Clientes; -- Mostra as colunas da tabela 'Clientes'.
```

SEÇÃO 3

Objetos Programáveis

Além de armazenar dados, os bancos de dados modernos permitem armazenar lógica e automações. Essas estruturas ajudam a reduzir o tráfego de rede e garantem a integridade dos dados diretamente no servidor.

3.1. Functions.— São blocos de código que realizam cálculos e obrigatoriamente retornam um valor.

```
DELIMITER $$

DROP FUNCTION IF EXISTS soma $$

CREATE FUNCTION soma(a FLOAT, b FLOAT) RETURNS FLOAT
DETERMINISTIC
BEGIN
RETURN a + b;
END $$

DELIMITER ;

-- Chamando a função diretamente em uma consulta
SELECT soma(10.5, 4.25) AS Resultado;
```

Note que a cláusula `RETURNS FLOAT` define o tipo de dado que será devolvido pela função.

3.2. Stored Procedures.— As *Stored Procedures* (Procedimentos Armazenados) permitem encapsular uma sequência de comandos lógicos no servidor de banco de dados, podendo ser chamadas por aplicações Java ou Python para executar tarefas complexas com um único comando.

Diferente das funções, as Procedures executam uma série de comandos SQL (INSERT, UPDATE, DELETE) e não precisam obrigatoriamente retornar um valor. Elas aceitam parâmetros de entrada (IN) e saída (OUT).

```sql
DELIMITER $$

DROP PROCEDURE IF EXISTS AtualizarSalario$$

CREATE PROCEDURE AtualizarSalario(
    IN funcionario_id INT,
    IN percentual_aumento DECIMAL(5,2)
)$$

BEGIN
    -- Atualiza o salário baseado no ID e percentual fornecidos
    UPDATE Funcionarios
    SET salario = salario * (1 + percentual_aumento / 100)
    WHERE id = funcionario_id;

    SELECT 'Salário atualizado com sucesso!' AS Resultado;
END $$

DELIMITER ;
```

Note o uso do `DELIMITER` para evitar que o ponto e vírgula interno da procedure finalize o comando precocemente.

3.3. Triggers.— São automações ”disparadas”por eventos específicos (INSERT, UPDATE ou DELETE) em uma tabela. Eles podem agir antes (BEFORE) ou depois (AFTER) do evento.

```sql
-- Criação da tabela para o exemplo
CREATE TABLE IF NOT EXISTS conta(
numero INT PRIMARY KEY,
total FLOAT
);

-- Trigger para calcular o produto acumulado antes de cada inserção
CREATE TRIGGER int_produto BEFORE INSERT
ON conta
FOR EACH ROW SET @produto = @produto * NEW.total;

-- Inicialização da variável de sessão (valor neutro para multiplicação)
SET @produto = 1;

-- Inserção de dados de teste para disparo do trigger
INSERT INTO conta() VALUES
(1, 20),
(2, 180),
(3, -50);

-- Verificação do resultado acumulado na variável
SELECT @produto AS 'Produto Total';
```

SEÇÃO 4

Objetos de Consulta

4.1. Views.— Uma View é uma ”tabela virtual”baseada no resultado de um SELECT. Ela não armazena dados próprios, mas funciona como um atalho para consultas complexas (como aquelas com vários JOINs).

```sql
-- Criando uma View para simplificar a relação de Alunos e Disciplinas
-- Esta View esconde a complexidade dos JOINs da tabela intermediária
CREATE OR REPLACE VIEW vw_lista_presenca AS
SELECT
A.nome AS Nome_Aluno,
D.nomeDisc AS Disciplina,
D.cargaHoraria AS Carga
FROM Aluno A
JOIN Aluno_Disciplina AD ON A.idAluno = AD.idAluno
JOIN Disciplina D ON AD.idDisc = D.idDisc;

-- Para o desenvolvedor Java/Python, basta chamar a View como uma tabela:
SELECT * FROM vw_lista_presenca
WHERE Disciplina = 'Banco de Dados';
```

Dica

No padrão **DAO**, usar Views facilita a manutenção: se o nome de uma coluna na tabela original mudar, você altera apenas a definição da View no banco, e não precisa atualizar todas as queries espalhadas no seu código Java ou Python.

4.2. CTE (Common Table Expression).— Sigla para *Common Table Expression*, a CTE é definida com a cláusula `WITH`. Diferente de uma View, ela existe apenas durante a execução da query e é descartada automaticamente ao fim.

```sql
-- CTE simples: calcula o total de vendas por mês e ano
WITH venda_por_mes (mes, ano, total_vendas) AS (
    SELECT MONTH(data_pedido), YEAR(data_pedido), SUM(preco_total)
    FROM pedidos
    GROUP BY MONTH(data_pedido), YEAR(data_pedido)
)
-- A CTE é usada como se fosse uma tabela comum
SELECT mes, ano, total_vendas
FROM venda_por_mes
ORDER BY ano, mes;
```

Dica

CTEs tornam consultas complexas mais legíveis ao dividir a lógica em etapas nomeadas. Como não são armazenadas no banco, não exigem permissão de criação de objetos — basta executar a query. Disponível no MySQL a partir da versão 8.0.

SEÇÃO 5

Integração

5.1. Arquitetura de Conexão.— A integração entre aplicação e banco segue uma estrutura em camadas:

Driver/Connector: Traduz as chamadas da linguagem para o protocolo do banco.

Connection Class: Gerencia a abertura e o fechamento do túnel de comunicação.

DAO (Data Access Object): Camada que contém exclusivamente os comandos SQL.

Model (Entidade): vide [1.6]

Tabela 9: Comparativo de Integração: Java vs. Python

Componente	Java (JDBC)	Python (PyMySQL)
Driver / Conector	MySQL Connector/J (.jar)	mysql-connector
Gerenciador de Conexão	`DriverManager`	`pymysql.connect()`
Objeto de Execução	`PreparedStatement`	`Cursor`
Objeto de Resultado	`ResultSet`	`Cursor.fetchall() / fetchone()`
Tratamento de Erros	`SQLException` (Checked)	`pymysql.Error` (Unchecked)
Passagem de Parâmetros	`ps.setX(index, valor)`	Parâmetros em tupla ou dict

Nota

Enquanto o Java exige o tratamento explícito de exceções (*Checked Exceptions*), o Python oferece uma sintaxe mais flexível, porém exige maior rigor do desenvolvedor na validação manual de dados antes da execução.

5.2. Integração com Java.— Em Java, a integração com o banco de dados é feita através da API JDBC (Java Database Connectivity). A ideia é simples: o código Java envia comandos SQL para o banco e recebe os resultados de volta, tudo através de uma conexão gerenciada pelo `DriverManager`.

Na prática, a estrutura se divide em três camadas: a **ConnectionDAO**, responsável por abrir e fechar a conexão; o **DAO** de cada entidade, onde ficam os comandos SQL; e o **Model**, que representa a tabela como um objeto Java (vide 1.6).

```java
// ConnectionDAO | abre o túnel com o banco
public class ConnectionDAO {
    public Connection getConnection() throws SQLException {
        String url  = "jdbc:mysql://localhost:3306/meu_banco";
        String user = "root";
        String pass = "senha";
        return DriverManager.getConnection(url, user, pass);
    }
}
```

```java
// ClienteDAO | executa os comandos SQL
public class ClienteDAO {
    public void insert(Cliente c) {
        String sql = "INSERT INTO Clientes (id, nome) VALUES (?, ?)";
        try (Connection con = new ConnectionDAO().getConnection();
             PreparedStatement ps = con.prepareStatement(sql)) {
            ps.setInt(1, c.getId());
            ps.setString(2, c.getNome());
            ps.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Atenção

O uso do `PreparedStatement` com `?` não é opcional — ele previne ataques de **SQL Injection**, onde um usuário mal-intencionado poderia manipular a query enviando valores inesperados pelo campo de entrada.

5.3. Integração com Python.— Em Python, a conexão é mais direta. A biblioteca `mysql-connector-python` já resolve tudo com poucos comandos, sem a necessidade de uma classe de conexão separada — embora seja recomendado isolar essa lógica para manter o código organizado.

```python
import mysql.connector

# Abrindo a conexão
con = mysql.connector.connect(
    host="localhost",
    user="root",
```

```
    password="senha",
    database="meu_banco"
)

cursor = con.cursor()

# Inserindo um cliente com parâmetros (também previne SQL Injection)
sql = "INSERT INTO Clientes (id, nome) VALUES (%s, %s)"
cursor.execute(sql, (106, "Maria Hello"))
con.commit()

# Buscando todos os clientes
cursor.execute("SELECT * FROM Clientes")
for linha in cursor.fetchall():
    print(linha)

con.close()
```

Java vs. Python

Em Java, o `try-catch` é obrigatório pelo compilador — se você não tratar a `SQLException`, o código nem compila. Em Python, o tratamento de erros é opcional, o que dá mais liberdade mas exige mais disciplina: um `con.commit()` esquecido, por exemplo, faz a inserção desaparecer silenciosamente.

Referências

[1] Edgar Frank Codd. A relational model of data for large shared data banks. *Association of Computer Machinery (ACM)*, 1970.

[2] DataCamp. Declaração mysql show: Uso e exemplos, s.d.

[3] Fábio dos Reis. Comandos dcl-sql e sua sintaxe, s.d.

[4] Ana Duarte. Cte: o que é e para que serve no sql, s.d.

[5] Edberto Ferneda. *Introdução à Linguagem SQL*. Universidade de São Paulo – FFCLRP, Ribeirão Preto, s.d.

[6] IBM. Relacionamentos do banco de dados, 2021.

[7] Renzo Paranaiba Mesquita. Linguagem sql: Slides de aula, s.d. Disponibilizado via material de disciplina.

[8] W3 Schools. Sql constraints, s.d.

[9] TiDB Team. primary key vs foreign key unlocking efficient data management in databases, 2024.

[10] Wikipedia. Acid, 2025.