Aula 1

Aula Introdutória de SQL

1. Introdução à Análise de Dados

1.1 O que é análise de dados

Em seu livro *The Future of Data Analysis*, John Tukey definiu, pela primeira vez, a análise de dados como um conjunto de procedimentos e técnicas voltados para analisar, planejar e coletar dados, utilizando métodos e ferramentas estatísticas.

Atualmente, a análise de dados é mais comumente entendida como um processo que envolve a exploração, visualização, limpeza e modelagem dos dados, com o objetivo de extrair insights e apoiar a tomada de decisões — algo que dificilmente seria alcançado apenas com os dados brutos.

1.2 Importância da análise de dados

Atualmente, os dados são um dos ativos mais valiosos que uma empresa ou organização pode possuir. Quando utilizados de forma estratégica, permitem compreender melhor o público-alvo, embasar decisões com maior precisão e acompanhar a evolução do negócio ao longo do tempo.

Um exemplo comum é o uso de dados em campanhas de marketing digital, onde informações sobre o comportamento de compra e preferências dos clientes permitem segmentar anúncios, aumentar a taxa de conversão e otimizar o retorno sobre investimento (ROI).

Outro exemplo relevante está nos algoritmos de recomendação de redes sociais, que analisam padrões de navegação, curtidas e interações dos usuários para sugerir conteúdos personalizados e maximizar o engajamento.

Além disso, os dados são fundamentais para o desenvolvimento de modelos de inteligência artificial, que hoje fazem parte do nosso cotidiano. Grandes modelos, como o GPT ou o Gemini, foram treinados utilizando enormes volumes de informações — frequentemente na ordem de petabytes.

1.3 SQL no contexto da análise de dados

O SQL (*Structured Query Language*) é uma linguagem de programação padrão utilizada para criar, gerenciar e consultar bancos de dados relacionais. Ele permite que usuários e aplicações acessem, manipulem e analisem dados de forma estruturada e eficiente.

O SQL é, portanto, uma ferramenta essencial para a análise de dados, já que praticamente todas as aplicações atualmente utilizam algum tipo de banco de dados que pode ser acessado via SQL.

Esse destaque se reflete também no mercado de trabalho: de acordo com o artigo do *361 Data Scientist*, aproximadamente 60% das vagas em ciência de dados exigem conhecimento em SQL.

2. Estrutura Básica de Bancos de Dados Relacionais

2.1 O que é um banco de dados

Definição e exemplos reais

Um banco de dados, ou base de dados, é uma coleção organizada de informações – ou dados – estruturados, normalmente armazenados eletronicamente em um sistema de computador. Um banco de dados é geralmente controlado por um sistema de gerenciamento de banco de dados (SGBD). Juntos, os dados e o SGBD, juntamente com os aplicativos que estão associados a eles, são chamados de sistema de banco de dados, geralmente abreviado para apenas banco de dados.

O objetivo principal de um banco de dados é armazenar e gerenciar grandes volumes de dados de forma eficiente, segura e confiável, permitindo que sejam facilmente acessados, gerenciados e atualizados.

Exemplos reais de bancos de dados em nosso cotidiano incluem:

- **Redes Sociais:** Plataformas como Facebook, Instagram e Twitter utilizam bancos de dados massivos para armazenar informações de perfis de usuários, postagens, comentários, fotos e conexões entre os usuários.
- **Bancos e Instituições Financeiras:** Todas as transações bancárias, informações de contas de clientes, saldos, empréstimos e investimentos são meticulosamente registrados em bancos de dados seguros.
- **Comércio Eletrônico:** Lojas online como Amazon e Mercado Livre dependem de bancos de dados para gerenciar seus catálogos de produtos, informações de clientes, pedidos, pagamentos e controle de estoque.
- **Sistemas de Reserva de Voos e Hotéis:** Quando você reserva uma passagem aérea ou um quarto de hotel, suas informações são inseridas em um banco de dados que gerencia a disponibilidade, preços e detalhes da reserva.
- **Serviços de Streaming:** Plataformas como Netflix e Spotify utilizam bancos de dados para organizar seus vastos catálogos de filmes, séries e músicas, além de armazenar o histórico e as preferências de cada usuário para oferecer recomendações personalizadas.

2.2 Tabelas, linhas e colunas

Conceito de registros e atributos

A estrutura fundamental de um banco de dados relacional é a **tabela**. Uma tabela organiza os dados em um formato de grade, composto por linhas e colunas, semelhante a uma planilha.

- **Tabelas:** Representam uma entidade ou um conceito do mundo real sobre o qual desejamos armazenar informações. Por exemplo, em um banco de dados de uma escola, poderíamos ter tabelas como `Alunos`, `Professores` e `Disciplinas`.
- **Colunas (ou Atributos):** As colunas são as divisões verticais de uma tabela e definem os tipos de dados que serão armazenados em cada entrada. Cada coluna possui um nome único e um tipo de dado específico (por exemplo, texto, número, data). As colunas representam os **atributos** de uma entidade. Por exemplo, na tabela `Alunos`, as colunas poderiam ser `ID_Aluno`, `Nome`, `Data_Nascimento` e `Email`.
- **Linhas (ou Registros):** As linhas são as entradas horizontais em uma tabela e representam uma única ocorrência daquela entidade. Cada linha contém os valores para cada uma das colunas da tabela. Uma linha também é conhecida como um **registro**. Por exemplo, uma linha na tabela `Alunos` conteria as informações de um aluno específico, como o `ID_Aluno` 1, o `Nome` "João Silva", a `Data_Nascimento` "2005-03-15"e o `Email` "joao.silva@email.com".

Tabela 1: Exemplo de Tabela: Alunos

ID_Aluno	Nome	Data_Nascimento	Email
1	João Silva	2005-03-15	joao.silva@email.com
2	Maria Souza	2006-07-22	maria.souza@email.com
3	Pedro Costa	2005-11-10	pedro.costa@email.com

Responsável: Cesar

2.3 Relações entre tabelas

Chave primária (Primary Key)

Uma **chave primária** (*Primary Key*) é um campo (ou uma combinação de campos) em uma tabela que identifica unicamente cada linha. Ela possui duas regras principais:

1. **Não pode conter valores nulos:** Cada linha deve ter um valor para a chave primária.
2. **Deve ser única:** Não podem existir duas linhas com o mesmo valor de chave primária.

No exemplo da tabela `Alunos`, a coluna `ID_Aluno` seria uma excelente candidata a chave primária.

Chave estrangeira (Foreign Key)

Uma **chave estrangeira** (*Foreign Key*) é um campo (ou uma combinação de campos) em uma tabela que estabelece uma ligação com a chave primária de outra tabela. Ela é utilizada para garantir a integridade referencial dos dados, ou seja, para assegurar que a relação entre as tabelas seja válida.

Por exemplo, se tivéssemos uma tabela `Matriculas` para registrar em quais disciplinas cada aluno está matriculado, poderíamos ter a seguinte estrutura:

Tabela 2: Exemplo de Tabela com Chaves Estrangeiras: Matriculas

ID_Matricula	ID_Aluno (FK)	ID_Disciplina (FK)
101	1	5
102	1	7
103	2	5

Nesta tabela `Matriculas`, `ID_Aluno` é uma chave estrangeira que se refere à chave primária da tabela `Alunos`, e `ID_Disciplina` é uma chave estrangeira que se refere à chave primária de uma tabela `Disciplinas`.

2.4 Tipos de Dados

Ao criar uma tabela em um banco de dados, é fundamental definir o tipo de dado que cada coluna irá armazenar. Essa definição garante a integridade dos dados, otimiza o espaço de armazenamento e permite que o SGBD realize operações de forma mais eficiente.

Imagine uma coluna `Idade`. Não faria sentido armazenar um texto como "vinte"nessa coluna; o ideal é que ela aceite apenas números. Da mesma forma, uma coluna `Data_Nascimento` deve conter apenas datas válidas.

Cada sistema de gerenciamento de banco de dados (MySQL, PostgreSQL, etc.) possui seus próprios nomes para os tipos de dados, mas, de modo geral, eles se enquadram em algumas categorias principais:

Tipos de Dados Comuns em SQL:

- **Tipos de Texto (Strings):**
  - `CHAR(n)`: Armazena uma string de tamanho **fixo**. Se o texto inserido for menor que `n`, o restante será preenchido com espaços. É útil para dados com tamanho previsível, como siglas de estados (ex: `CHAR(2)` para "SP").
  - `VARCHAR(n)`: Armazena uma string de tamanho **variável**, até um limite máximo de `n` caracteres. É o tipo mais comum para textos, como nomes, e-mails e descrições, pois economiza espaço.
  - `TEXT`: Para textos longos, como postagens de blog ou descrições detalhadas de produtos, sem um limite de tamanho predefinido (na prática, o limite é bem alto).
- **Tipos Numéricos:**
  - `INTEGER` ou `INT`: Para armazenar números inteiros, positivos ou negativos. É ideal para colunas como `ID`, `Idade` ou `Quantidade`.
  - `DECIMAL(p, s)` ou `NUMERIC(p, s)`: Para números com casas decimais que exigem precisão exata, como valores monetários. `p` é a precisão (número total de dígitos) e `s` é a escala (número de dígitos após a vírgula). Por exemplo, `DECIMAL(10, 2)` pode armazenar um valor como `12345678.90`.

  - `FLOAT` e `REAL`: Para números de ponto flutuante (aproximados). São úteis em cálculos científicos onde a precisão absoluta não é a maior prioridade.

- **Tipos de Data e Hora:**
  - `DATE`: Armazena apenas a data (ano, mês e dia). Exemplo: `2023-10-26`.
  - `TIME`: Armazena apenas a hora (hora, minuto e segundo). Exemplo: `14:30:00`.
  - `DATETIME` ou `TIMESTAMP`: Armazena a data e a hora juntas. Exemplo: `2023-10-26 14:30:00`. `TIMESTAMP` é frequentemente utilizado para registrar o momento em que uma linha foi criada ou modificada.

- **Tipos Lógicos (Booleanos):**
  - `BOOLEAN`: Armazena valores de verdadeiro ou falso (`TRUE` ou `FALSE`). Alguns sistemas usam um tipo `BIT`, onde `1` representa verdadeiro e `0` representa falso.

A escolha correta do tipo de dado é um passo crucial no design de um banco de dados, impactando diretamente o desempenho e a confiabilidade da sua aplicação.

2.5 Bancos relacionais x não relacionais

Diferenças principais e quando usar cada um

Além do modelo relacional, existe outra categoria popular de bancos de dados conhecida como **não relacional** (ou **NoSQL**).

Tabela 3: Comparativo: Bancos Relacionais (SQL) vs. Não Relacionais (NoSQL)

Bancos de Dados Relacionais (SQL)	**Bancos de Dados Não Relacionais (NoSQL)**
Estrutura rígida e predefinida (esquema), baseada em tabelas, linhas e colunas.	Estrutura flexível e dinâmica (sem esquema ou esquema flexível). Os dados podem ser armazenados em documentos (JSON, XML), grafos, pares chave-valor, etc.
Geralmente escalam verticalmente (aumentando a capacidade de um único servidor).	Geralmente escalam horizontalmente (distribuindo os dados por múltiplos servidores).
Utilizam a Linguagem de Consulta Estruturada (SQL) como padrão.	Cada banco de dados NoSQL possui sua própria linguagem de consulta.
Priorizam a consistência e a integridade dos dados (propriedades ACID: Atomicidade, Consistência, Isolamento e Durabilidade).	Geralmente oferecem um modelo de consistência eventual, priorizando a disponibilidade e a escalabilidade.
Exemplos: MySQL, PostgreSQL, SQL Server, Oracle.	**Exemplos:** MongoDB, Cassandra, Redis, Neo4j.

Quando usar cada um:

- **Use um banco de dados relacional (SQL) quando:**
  - Seus dados são estruturados e você precisa de um esquema bem definido.
  - A integridade e a consistência dos dados são críticas (ex: sistemas financeiros).
  - Você precisa realizar consultas complexas que envolvem a junção de várias tabelas.
- **Use um banco de dados não relacional (NoSQL) quando:**
  - Você está lidando com grandes volumes de dados não estruturados ou semiestruturados.
  - Sua aplicação precisa de alta escalabilidade e disponibilidade.
  - O esquema dos seus dados está em constante evolução.
  - A velocidade de leitura e escrita é um fator primordial.

Responsável: Cesar

3. Primeiras Consultas SQL

Exemplo de Relação Alunos

ID	Nome	Idade	Curso
1	Alice Silva	21	Computação
2	Bob Souza	20	Matemática
3	Maria Oliveira	20	Engenharia
4	Gabriel Santos	19	Engenharia
5	Julia Almeida	23	Computação
6	Pedro Ferreira	22	Física
7	Luana Costa	21	Matemática
8	Gabriel Martins	24	Computação
9	Fernanda Rocha	20	Engenharia
10	Tiago Pereira	19	Estatística

3.1 SELECT e FROM

Uma consulta de dados relacionais apoia-se em duas operações: selecionar relações e, a partir dessas, selecionar atributos. Para isso, podemos utilizar os comandos FROM e SELECT, respectivamente. A estrutura básica deles é:

```
SELECT <Lista de Atributos>
FROM <Lista de Relações>;
```

Por estarmos lidando com listas, podemos selecionar diferentes relações e atributos simultaneamente. Para exemplificar o uso dessa estrutura, podemos realizar algumas consultas sobre a relação Alunos. Se quisermos selecionar o nome e o id dos alunos, podemos fazer

```
SELECT ID, Nome
FROM Alunos;
```

o que retornará uma nova relação somente com os atributos dados, como a que está a seguir.

ID	Nome
1	Alice Silva
2	Bob Souza
3	Maria Oliveira
4	Gabriel Santos
5	Julia Almeida
6	Pedro Ferreira
7	Luana Costa
8	Gabriel Martins
9	Fernanda Rocha
10	Tiago Pereira

Por outro lado, se quisermos selecionar todos os atributos da relação, além de podermos escrever todos os atributos, podemos utilizar o símbolo *:

```
SELECT *
FROM Alunos;
```

O uso de * torna a escrita mais simples e enxuta.

3.2 LIMIT

Ao realizar uma consulta como as mostradas anteriormente, retornamos todos os alunos da relação, cada qual com os atributos desejados. Contudo, em cenários onde a relação possui milhares ou bilhares de registros e é suficiente observar apenas uma parcela deles, pode-se utilizar o comando LIMIT, evitando que a query demore muito para ser executada e que o sistema seja sobrecarregado. A estrutura básica de uma consulta com LIMIT será:

```
SELECT <Lista de Atributos>
FROM <Lista de Relações>
LIMIT <Número de Registros>;
```

Se quisermos, por exemplo, obter 2 alunos de Alunos, basta fazer *LIMIT 2*. Vale observar, porém, que esse comando não garante que os registros retornados (e a ordem deles) em diferentes execuções serão iguais. Para obter isso, utilizaremos o comando ORDER BY (ordenação), que será visto na próxima aula.

3.3 WHERE (Filtros básicos)

O comando WHERE é semelhante ao comando *if* das linguagens de programação tradicionais: dada uma condição, realizamos uma ação; caso contrário, realizamos outra. No caso da consulta de dados, a ação é a seleção de registros conforme a condição. Com isso, temos a seguinte sintaxe do WHERE:

```
SELECT <Lista de Atributos>
FROM <Lista de Relações>
WHERE <Lista de Condições>;
```

Para construir a lista de condições, temos diversas possibilidades de operadores, como veremos a seguir.

As comparações tradicionais maior, menor, igual e suas composições podem ser utilizadas no WHERE. Por exemplo, na relação Alunos, podemos selecionar apenas aqueles que cursam computação:

```
SELECT ID, Nome
FROM Alunos
WHERE Curso = "Computação";
```

Podemos também utilizar mais de uma comparação e usar operadores lógicos entre elas. A consulta de alunos com idade igual ou superior a 20 anos e que não cursam computação poderia ser feita da seguinte forma:

```
SELECT *
FROM Alunos
WHERE Idade >= 20 AND Curso != "Computação";
```

3.4 WHERE (Filtros avançados)

Até agora, utilizamos operadores de comparação como =, >, <, etc., para encontrar correspondências exatas ou valores dentro de um intervalo. No entanto, muitas vezes precisamos buscar por informações que não conhecemos por completo, como encontrar todos os alunos cujo nome começa com "Jo"ou todos os produtos de uma fábrica ou comércio que contêm a palavra "madeira".

Para esses casos, utilizamos o operador LIKE, que permite a busca por padrões de caracteres dentro de uma coluna de texto. Para tanto, podemos usar caracteres especiais denominados curingas para manipular o funcionamento do LIKE.

O primeiro coringa que veremos será "%", que é responsável por representar zero ou mais caracteres. Por exemplo, se queremos determinar os alunos que têm "Gabriel"como primeiro nome, podemos fazer:

```
SELECT *
FROM Alunos
WHERE Nome LIKE "Gabriel%";
```

Com isso, nomes como "Gabriel", "Gabriel Santos"e "Gabriel Martins"serão retornados. A mesma lógica pode ser usada para procurar por produtos de um estoque que tem "madeira"como último nome ("%madeira") ou como nome intermediário ("%madeira%").

Além de podermos usar um coringa para completar uma string, podemos também utilizá-lo para completar um único caractere. Para isso, vamos usar "_". Por exemplo, se quisermos obter os alunos com nome inicial "João", podemos utilizar:

```
SELECT *
FROM Alunos
WHERE Nome LIKE "Jo_o%";
```

Com isso, garantimos que as variações "João", "Joao"e "JoAo", por exemplo, serão retornadas.

Por fim, uma relação pode conter valores inexistentes (*NULL*). Nesse caso, não é possível realizar operações como "= NULL"ou "!= NULL", pois o retorno será desconhecido. Para isso, podemos usar IS NULL e IS NOT NULL. Verificar se algum aluno não tem a idade na base pode ser feito pela consulta a seguir:

```
SELECT *
FROM Alunos
WHERE Idade IS NULL;
```

4. Manipulação de Tabelas e Dados (DDL e DML)

Até agora, focamos em como consultar dados que já existem em uma tabela. No entanto, antes de qualquer consulta, precisamos definir a estrutura onde esses dados serão armazenados. Os comandos que definem e gerenciam a estrutura do banco de dados, como a criação de tabelas, fazem parte da DDL (*Data Definition Language*, ou Linguagem de Definição de Dados).

4.1 CREATE TABLE: Criando a Estrutura dos Dados

O primeiro passo para armazenar qualquer informação é criar uma tabela. O comando `CREATE TABLE` é utilizado para definir essa estrutura, especificando o nome da tabela e, mais importante, o nome e o tipo de dado de cada coluna que ela conterá. A estrutura básica do comando é:

```
CREATE TABLE nome_da_tabela (
    nome_da_coluna1 tipo_de_dado,
    nome_da_coluna2 tipo_de_dado,
    nome_da_coluna3 tipo_de_dado,
    ...
);
```

Vamos usar este comando para criar a estrutura da tabela `Alunos` que utilizamos como exemplo ao longo desta aula. Com base nos dados que vimos, podemos definir a tabela da seguinte forma:

```
CREATE TABLE Alunos (
    ID INTEGER PRIMARY KEY,
    Nome VARCHAR(100),
    Idade INTEGER,
    Curso VARCHAR(50)
);
```

Vamos analisar cada parte deste comando:

- `CREATE TABLE Alunos (...);`: Inicia a criação de uma nova tabela chamada `Alunos`. Todo o conteúdo entre os parênteses define as colunas.
- `ID INTEGER PRIMARY KEY,`: Cria uma coluna chamada `ID` que armazenará números inteiros (`INTEGER`). A restrição `PRIMARY KEY` (Chave Primária) garante que cada valor nesta coluna seja único e não nulo, servindo como o identificador exclusivo de cada aluno.
- `Nome VARCHAR(100),`: Cria uma coluna chamada `Nome` para armazenar textos de tamanho variável (`VARCHAR`) com até 100 caracteres.
- `Idade INTEGER,`: Cria uma coluna chamada `Idade` que também armazenará números inteiros.
- `Curso VARCHAR(50)`: Cria a última coluna, chamada `Curso`, para textos de até 50 caracteres. Note que a última linha da definição não leva vírgula.

Adicionando Regras com Restrições (Constraints)

Além do tipo de dado, podemos adicionar regras extras às colunas para garantir a integridade e a qualidade dos dados. Essas regras são chamadas de **restrições** (*constraints*). Já vimos a `PRIMARY KEY`, mas existem outras muito úteis:

- `NOT NULL`: Garante que a coluna não pode ter valores nulos. É útil para campos obrigatórios, como o nome de um aluno.
- `UNIQUE`: Garante que todos os valores na coluna sejam diferentes uns dos outros. Ideal para campos como e-mail ou CPF.

Podemos aprimorar nossa tabela `Alunos` com essas restrições:

```
CREATE TABLE Alunos (
    ID INTEGER PRIMARY KEY,
    Nome VARCHAR(100) NOT NULL,
    Idade INTEGER,
    Curso VARCHAR(50) NOT NULL,
    Email VARCHAR(100) UNIQUE
);
```

Nesta versão melhorada, garantimos que todo aluno cadastrado tenha obrigatoriamente um nome e um curso, e que não existam dois alunos com o mesmo e-mail. Uma vez que a tabela é criada com `CREATE TABLE`, ela está pronta para receber dados através de comandos como o `INSERT`.