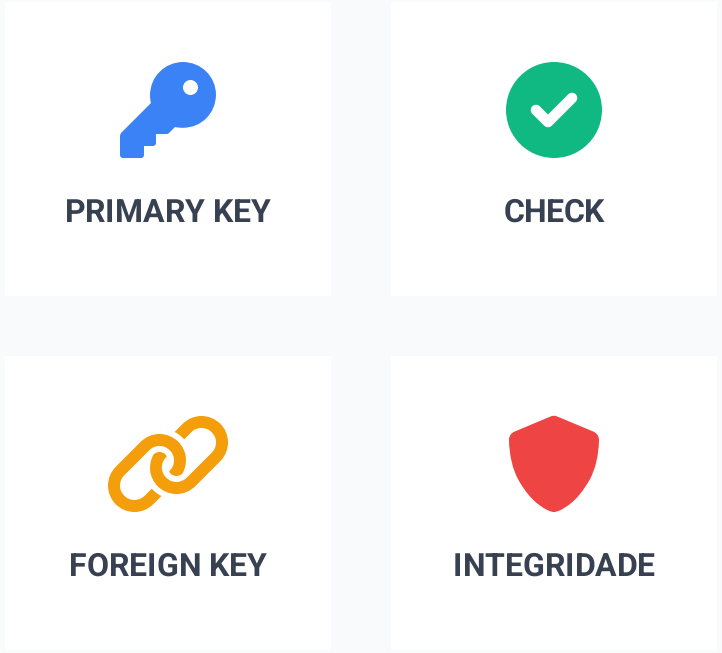
# Implementação de Restrições em SQL

## Data Definition Language (DDL)

**UC3:** Ciência de Dados

**Aula 04:** Restrições de Integridade

**Tema:** PRIMARY KEY, FOREIGN KEY, CHECK, UNIQUE, NOT NULL



# Fundamentos do DDL e Restrições de Integridade

> *"As tabelas são o esqueleto do banco de dados, e as restrições são as regras que mantêm esse esqueleto funcionando."*

## O que é DDL?

**DDL (Data Definition Language)** é o conjunto de comandos SQL usados para **criar e modificar a estrutura** do banco de dados.

**Comandos principais do DDL:**

- **CREATE TABLE:** Criar novas tabelas
- **ALTER TABLE:** Modificar tabelas existentes
- **DROP TABLE:** Excluir tabelas
- **CREATE INDEX:** Criar índices para otimização
- **CONSTRAINT:** Definir restrições de integridade

## O que são Restrições de Integridade?

**Restrições de Integridade** são **regras aplicadas no nível do banco de dados** para garantir a precisão e confiabilidade dos dados.

**Propósito das restrições:**

- Impedir dados **inválidos** (ex: preço negativo)
- Impedir dados **incompletos** (ex: nome vazio)
- Impedir dados **contraditórios** (ex: pedido sem cliente)
- Garantir **unicidade** (ex: CPF único)
- Manter **integridade referencial** entre tabelas

**As restrições criadas no nível do banco de dados são a última linha de defesa contra a inconsistência de dados, mesmo que a aplicação tenha bugs!**

# Restrições Essenciais em SQL

### NOT NULL

Garante que a coluna **NUNCA ficará vazia**. O valor deve ser sempre fornecido.

```
nome VARCHAR(100)
NOT NULL
```

### PRIMARY KEY

Identifica **UNICAMENTE** cada registro da tabela. É NOT NULL e UNIQUE por natureza.

```
id INT NOT NULL
PRIMARY KEY
```

### FOREIGN KEY

Garante **INTEGRIDADE REFERENCIAL**, ligando uma tabela a outra.

```
id_cliente INT,
FOREIGN KEY
(id_cliente)
REFERENCES
Cliente(id)
```

### UNIQUE

Garante que todos os valores em uma coluna serão **DIFERENTES**.

```
email VARCHAR(100)
NOT NULL UNIQUE
```

### CHECK

Define uma **CONDIÇÃO** que todos os dados inseridos devem satisfazer.

```
preco DECIMAL(10,2)
CHECK (preco > 0)
```

## Por que usar Restrições de Integridade?

**Proteção de Dados:** Impede que dados inválidos, incompletos ou contraditórios sejam inseridos no banco de dados.

**Automatização:** As restrições são aplicadas automaticamente pelo SGBD, sem necessidade de validação manual.

**Última Linha de Defesa:** Mesmo que a aplicação tenha bugs, o banco de dados protege a integridade dos dados.

**Consistência:** Garante que os dados permaneçam consistentes e confiáveis ao longo do tempo.

# PRIMARY KEY e NOT NULL

## PRIMARY KEY (PK)

**Propósito:** Identifica **UNICAMENTE** cada registro da tabela.

**Características:**

- É **NOT NULL** por natureza
- É **UNIQUE** por natureza
- Cada tabela tem **APENAS UMA** PK
- Pode ser simples ou composta
- Geralmente usa AUTO_INCREMENT

## NOT NULL

**Propósito:** Garante que a coluna **NUNCA ficará vazia** (NULL).

**Quando usar:**

- Campos **obrigatórios**
- Informações **essenciais**
- Dados que não podem ser **desconhecidos**
- Sempre em **chaves primárias**
- Campos críticos para o negócio

**EXEMPLO PRÁTICO: Criação da Tabela Cliente**

```sql
CREATE TABLE Cliente (
    -- Chave Primária (NOT NULL é implícito)
    id INT NOT NULL PRIMARY KEY,
    -- Campos obrigatórios
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
    cpf VARCHAR(11) NOT NULL UNIQUE,
    -- Campo opcional
    telefone VARCHAR(15),
    -- Campos com valor padrão
    data_cadastro DATE NOT NULL DEFAULT CURRENT_DATE,
    ativo BOOLEAN NOT NULL DEFAULT TRUE
);
```

## Discussão Pausada

**O que acontece se removermos o NOT NULL do campo id (chave primária)?**

**Resposta:** A chave primária poderia ser nula, o que é **contra a definição de PK** e causaria **inconsistência grave**. Se o id for NULL, não será possível identificar unicamente o registro, violando o princípio fundamental da chave primária. Por isso, **PRIMARY KEY sempre implica NOT NULL**, mesmo que não seja explicitamente declarado.

# UNIQUE - Garantindo Valores Únicos

A restrição **UNIQUE** garante que todos os valores em uma coluna (ou grupo de colunas) serão **DIFERENTES**. Diferente da PRIMARY KEY, uma tabela pode ter **VÁRIAS colunas UNIQUE**, mas apenas UMA PK.

## Sintaxe e Exemplo SQL

```
-- Criação da tabela Cliente
CREATE TABLE Cliente (
    id INT NOT NULL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    -- Campo UNIQUE: E-mail
    email VARCHAR(100) NOT NULL UNIQUE,
    -- Campo UNIQUE: CPF
    cpf VARCHAR(11) NOT NULL UNIQUE,
    -- Campo opcional
    telefone VARCHAR(15)
);

-- Inserção válida
INSERT INTO Cliente
(id, nome, email, cpf)
VALUES (1, 'Ana Silva',
'ana@email.com', '12345678901');
```

## Discussão Importante

### Por que o campo email precisa ser UNIQUE?

Para garantir que **não haja dois cadastros de clientes com o mesmo e-mail**, mantendo a integridade dos dados de contato e evitando problemas de autenticação.

### Testes de Violação:

**ERRO 1:** Inserir cliente com email já existente

```
INSERT INTO Cliente VALUES (2, 'Bruno', 'ana@email.com', '98765');
```

Resultado: **Duplicate entry 'ana@email.com'**

**ERRO 2:** Inserir cliente com CPF já existente

```
INSERT INTO Cliente VALUES (3, 'Carla', 'carla@email.com',
'12345678901');
```

Resultado: **Duplicate entry '12345678901'**

### Exemplos de Inserção: Casos Válidos vs Inválidos

| Comando SQL | Status | Explicação |
|---|---|---|
| INSERT INTO Cliente VALUES (1, 'Ana', 'ana@email.com', '12345678901'); | ✓ VÁLIDO | Primeira inserção, email e CPF únicos |
| INSERT INTO Cliente VALUES (2, 'Bruno', 'bruno@email.com', '98765432109'); | ✓ VÁLIDO | Email e CPF diferentes dos existentes |
| INSERT INTO Cliente VALUES (3, 'Carla', 'ana@email.com', '45678912345'); | ✗ ERRO | Email 'ana@email.com' já existe (violação UNIQUE) |
| INSERT INTO Cliente VALUES (4, 'Daniel', 'daniel@email.com', '12345678901'); | ✗ ERRO | CPF '12345678901' já existe (violação UNIQUE) |

# CHECK - Validando Regras de Negócio

## Conceito e Propósito da Restrição CHECK

A restrição **CHECK** define uma **condição lógica** que todos os dados inseridos ou atualizados devem satisfazer. É usada para **validar regras de negócio no nível do banco de dados**, como garantir que preços sejam positivos, idades sejam válidas, ou status estejam dentro de valores permitidos.

## </> Exemplo: Tabela Produto com CHECK

```sql
-- Criação da tabela Produto
CREATE TABLE Produto (
    id INT NOT NULL PRIMARY KEY,
    nome_produto VARCHAR(150) NOT NULL,
    descricao TEXT NOT NULL,

    -- CHECK: Preço deve ser maior que zero
    preco DECIMAL(10, 2) NOT NULL
        CHECK (preco > 0),

    -- CHECK: Estoque não pode ser negativo
    estoque INT NOT NULL
        CHECK (estoque >= 0),

    codigo_barras VARCHAR(13) UNIQUE,
    data_cadastro DATE NOT NULL
        DEFAULT CURRENT_DATE
);
```

## ⚠️ Testes de Violação

### ❌ ERRO 1: Preço negativo

```sql
INSERT INTO Produto VALUES (1, 'Mouse', 'Mouse USB', -50.00, 10);
```

⚠ ERRO: Check constraint 'preco' is violated

### ❌ ERRO 2: Preço zero

```sql
INSERT INTO Produto VALUES (2, 'Teclado', 'Teclado USB', 0.00, 5);
```

⚠ ERRO: Check constraint 'preco' is violated

### ❌ ERRO 3: Estoque negativo

```sql
INSERT INTO Produto VALUES (3, 'Webcam', 'Webcam HD', 99.90, -5);
```

⚠ ERRO: Check constraint 'estoque' is violated

### ✅ VÁLIDO: Estoque zero (permitido)

```sql
INSERT INTO Produto VALUES (4, 'Monitor', 'Monitor 24"', 899.90, 0);
```

✓ Inserção bem-sucedida (estoque >= 0)

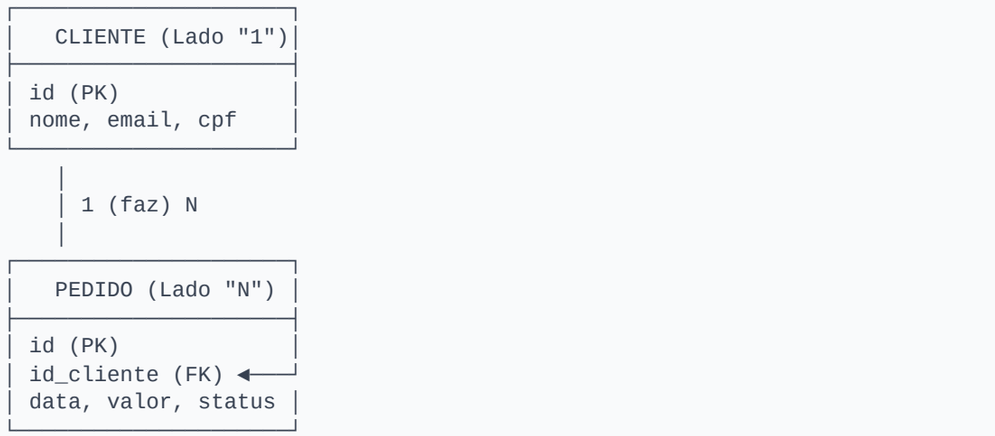
## 💬 Discussão: Por que usar CHECK para preço > 0 em vez de >= 0?

**Resposta:** Produtos com preço zero não fazem sentido comercialmente. O CHECK garante que todo produto tenha um **valor de venda válido**, protegendo a **integridade financeira** do sistema. Mesmo que a aplicação tenha bugs, o banco de dados rejeitará automaticamente qualquer tentativa de inserir preços inválidos, sendo a **última linha de defesa** contra dados incorretos.

# FOREIGN KEY - Integridade Referencial

**FOREIGN KEY (FK)** garante **INTEGRIDADE REFERENCIAL** entre tabelas. O valor da FK **DEVE existir** na tabela referenciada. Implementa relacionamentos **1:N** (um para muitos).

## Relacionamento: Cliente → Pedido (1:N)




**Interpretação:**
- Um cliente → MUITOS pedidos
- Cada pedido → UM cliente
- FK vai para o lado "N"

## Regras da FOREIGN KEY

**REGRA 1:** A FK deve referenciar uma PK existente na tabela pai

**REGRA 2:** Não é possível inserir um valor de FK que não existe na tabela pai

**REGRA 3:** Não é possível excluir um registro da tabela pai se houver registros filhos

**REGRA 4:** A ordem de criação importa: sempre criar a tabela pai primeiro

**DISCUSSÃO:** O que acontece se tentarmos inserir um pedido com id_cliente = 999 (inexistente)?

**RESPOSTA:** O banco impedirá a inserção, garantindo integridade referencial.

### 1. Criar Tabela Cliente (Tabela Pai)

```sql
-- Tabela referenciada (lado "1")
CREATE TABLE Cliente (
    id INT NOT NULL PRIMARY KEY,
    nome VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
    cpf VARCHAR(11) NOT NULL UNIQUE,
    telefone VARCHAR(15),
    data_cadastro DATE NOT NULL
        DEFAULT CURRENT_DATE
);
```

### 2. Criar Tabela Pedido (Tabela Filha com FK)

```sql
-- Tabela que referencia (lado "N")
CREATE TABLE Pedido (
    id INT NOT NULL PRIMARY KEY,
    -- FK: Referencia id da tabela Cliente
    id_cliente INT NOT NULL,
    data_pedido DATE NOT NULL
        DEFAULT CURRENT_DATE,
    valor_total DECIMAL(10,2) NOT NULL
        CHECK (valor_total > 0),
    status VARCHAR(20) NOT NULL
        DEFAULT 'Pendente',
    -- Definição da FOREIGN KEY
    FOREIGN KEY (id_cliente)
        REFERENCES Cliente(id)
);
```

# Atividade Prática 1 - Tabela Cliente

**Trabalho:** Em pares ou individualmente | **Ferramentas:** DBeaver, MySQL Workbench ou editor SQL online | **Tempo:** 20 minutos

## 1 Criação Básica da Tabela

Iniciar a criação da tabela Cliente com o comando CREATE TABLE.

```
CREATE TABLE Cliente (...);
```

## 2 Chave Primária e NOT NULL

Implementar o campo **id** como chave primária.

```
id INT NOT NULL PRIMARY KEY
```

💬 **Discussão:** O que acontece se removermos o NOT NULL do id?
**Resposta:** A PK pode ser nula, causando inconsistência.

## 3 Restrição de Unicidade (UNIQUE)

Implementar o campo **email** como único e obrigatório.

```
email VARCHAR(100) NOT NULL UNIQUE
```

💬 **Discussão:** Por que o campo email precisa ser UNIQUE?
**Resposta:** Para garantir que não haja dois cadastros com o mesmo e-mail.

## 4 Restrição de Opcionalidade

Implementar um campo opcional (ex: **telefone**).

```
telefone VARCHAR(15)
```

💬 **Discussão:** Por que telefone não tem NOT NULL?
**Resposta:** Nem todo cliente possui telefone ou deseja fornecê-lo.

### Script SQL Completo da Tabela Cliente

```sql
-- Criação da tabela Cliente com todas as restrições
CREATE TABLE Cliente (
    -- Passo 2: Chave Primária (NOT NULL é implícito)
    id INT NOT NULL PRIMARY KEY,
    -- Campo obrigatório
    nome VARCHAR(100) NOT NULL,
    -- Passo 3: Campo único e obrigatório
    email VARCHAR(100) NOT NULL UNIQUE,
    cpf VARCHAR(11) NOT NULL UNIQUE,
    -- Passo 4: Campo opcional (pode ser NULL)
    telefone VARCHAR(15),
    -- Campos com valor padrão
    data_cadastro DATE NOT NULL DEFAULT CURRENT_DATE,
    ativo BOOLEAN NOT NULL DEFAULT TRUE
);
```

# Atividade Prática 2 - Tabelas Produto e Pedido

## Tabela Produto (com CHECK)

**PASSO 1:** Criar tabela Produto com PK

```
id INT NOT NULL PRIMARY KEY
```

**PASSO 2:** Adicionar campo preço com CHECK

```
preco DECIMAL(10,2) NOT NULL CHECK (preco > 0)
```

💬 **Discussão:** Por que usar CHECK para preço > 0?
**Resposta:** Produtos com preço zero ou negativo não fazem sentido comercialmente.

**PASSO 3:** Adicionar campo estoque com CHECK

```
estoque INT NOT NULL CHECK (estoque >= 0)
```

💬 **Discussão:** Por que estoque >= 0 e não > 0?
**Resposta:** Estoque zero é válido (produto esgotado), mas estoque negativo não faz sentido.

## Tabela Pedido (com FK)

**PASSO 1:** Criar tabela Pedido com PK

```
id INT NOT NULL PRIMARY KEY
```

**PASSO 2:** Adicionar campo id_cliente (FK)

```
id_cliente INT NOT NULL, FOREIGN KEY (id_cliente) REFERENCES Cliente(id)
```

💬 **Discussão:** O que acontece se tentarmos inserir pedido com id_cliente = 999 (inexistente)?
**Resposta:** O banco impedirá a inserção (integridade referencial).

**PASSO 3:** Adicionar campo valor_total com CHECK

```
valor_total DECIMAL(10,2) NOT NULL CHECK (valor_total > 0)
```

💬 **Ordem de criação:** Cliente → Produto → Pedido (sempre criar a tabela referenciada primeiro!)

### Script SQL: Tabela Produto

```sql
-- Tabela Produto com CHECK
CREATE TABLE Produto (
    id INT NOT NULL PRIMARY KEY,
    nome_produto VARCHAR(150) NOT NULL,
    descricao TEXT NOT NULL,
    -- CHECK: Preço > 0
    preco DECIMAL(10,2) NOT NULL
        CHECK (preco > 0),
    -- CHECK: Estoque >= 0
    estoque INT NOT NULL
        CHECK (estoque >= 0),
    codigo_barras VARCHAR(13) UNIQUE,
    data_cadastro DATE NOT NULL
        DEFAULT CURRENT_DATE
);
```

### Script SQL: Tabela Pedido

```sql
-- Tabela Pedido com FK
CREATE TABLE Pedido (
    id INT NOT NULL PRIMARY KEY,
    -- FK: Referencia Cliente
    id_cliente INT NOT NULL,
    data_pedido DATE NOT NULL
        DEFAULT CURRENT_DATE,
    -- CHECK: Valor > 0
    valor_total DECIMAL(10,2) NOT NULL
        CHECK (valor_total > 0),
    status VARCHAR(20) NOT NULL
        DEFAULT 'Pendente',
    -- Definição da FK
    FOREIGN KEY (id_cliente)
        REFERENCES Cliente(id)
);
```

# Atividade Prática 3 - Teste e Validação

🎯 **Objetivo:** Testar as restrições criadas tentando inserir **dados que violem as regras**. As inserções devem **FALHAR**, confirmando que as restrições estão funcionando corretamente.

## ⚠️ Teste 1: Violação de UNIQUE

### ❌ Tentativa 1: Email duplicado

```
INSERT INTO Cliente (id, nome, email, cpf) VALUES (2, 'Bruno',
'ana@email.com', '98765432109');
```

⚠ ERRO ESPERADO: Duplicate entry 'ana@email.com'

### ❌ Tentativa 2: CPF duplicado

```
INSERT INTO Cliente (id, nome, email, cpf) VALUES (3, 'Carla',
'carla@email.com', '12345678901');
```

⚠ ERRO ESPERADO: Duplicate entry '12345678901'

## ⚠️ Teste 2: Violação de CHECK

### ❌ Tentativa 1: Preço negativo

```
INSERT INTO Produto (id, nome_produto, descricao, preco, estoque) VALUES
(2, 'Mouse', 'Mouse USB', -50.00, 10);
```

⚠ ERRO ESPERADO: Check constraint 'preco' violated

### ❌ Tentativa 2: Estoque negativo

```
INSERT INTO Produto (id, nome_produto, descricao, preco, estoque) VALUES
(3, 'Teclado', 'Teclado USB', 99.90, -5);
```

⚠ ERRO ESPERADO: Check constraint 'estoque' violated

## ⚠️ Teste 3: Violação de NOT NULL

### ❌ Tentativa 1: Nome vazio

```
INSERT INTO Cliente (id, nome, email, cpf) VALUES (4, NULL,
'daniel@email.com', '11122233344');
```

⚠ ERRO ESPERADO: Column 'nome' cannot be null

### ❌ Tentativa 2: Email vazio

```
INSERT INTO Cliente (id, nome, email, cpf) VALUES (5, 'Eduardo', NULL,
'55566677788');
```

⚠ ERRO ESPERADO: Column 'email' cannot be null

## ⚠️ Teste 4: Violação de FOREIGN KEY

### ❌ Tentativa 1: Cliente inexistente

```
INSERT INTO Pedido (id, id_cliente, data_pedido, valor_total, status)
VALUES (100, 999, CURRENT_DATE, 150.00, 'Pendente');
```

⚠ ERRO ESPERADO: Foreign key constraint fails

### ❌ Tentativa 2: id_cliente NULL

```
INSERT INTO Pedido (id, id_cliente, data_pedido, valor_total, status)
VALUES (101, NULL, CURRENT_DATE, 200.00, 'Pendente');
```

⚠ ERRO ESPERADO: Column 'id_cliente' cannot be null

## Checklist de Validação - Verifique se todas as restrições estão funcionando

- ☐ Todas as PKs foram definidas e testadas?
- ☐ Campos obrigatórios têm NOT NULL?
- ☐ Campos numéricos têm CHECK para valores válidos?
- ☐ O script DDL está documentado com comentários?
- ☐ Todas as FKs referenciam tabelas corretas?
- ☐ Campos únicos (CPF, email) têm UNIQUE?
- ☐ Tentativas de inserção inválida foram rejeitadas?
- ☐ A ordem de criação das tabelas está correta?

# Documentação e Reflexão Final

## Documentação do Script DDL

**Boas práticas:** Sempre documente seu script DDL com comentários explicativos. Isso facilita a manutenção futura e ajuda outros desenvolvedores a entenderem as decisões de design do banco de dados.

```
-- ================================================
-- SCRIPT DDL: Sistema de Gerenciamento
-- Autor: [Seu Nome] | Data: [Data Atual]
-- ================================================
CREATE TABLE Cliente (
    id INT NOT NULL PRIMARY KEY,  -- PK: ID único
    nome VARCHAR(100) NOT NULL,    -- Obrigatório
    email VARCHAR(100) NOT NULL UNIQUE,  -- Único
    ...
);
```

## Checklist de Validação

- Todas as **PRIMARY KEYS** foram definidas?
- Todas as **FOREIGN KEYS** referenciam tabelas corretas?
- Campos obrigatórios têm **NOT NULL**?
- Campos únicos (CPF, e-mail) têm **UNIQUE**?
- Campos numéricos têm **CHECK** para valores válidos?
- Campos com valores padrão têm **DEFAULT**?
- A **ordem de criação** das tabelas está correta?
- As restrições foram **testadas** com dados inválidos?

## Importância das Restrições

**Proteção Automática:** As restrições são aplicadas automaticamente pelo SGBD, sem validação manual.

**Última Linha de Defesa:** Mesmo que a aplicação tenha bugs, o banco protege a integridade dos dados.

**Consistência Garantida:** Garante que os dados permaneçam consistentes e confiáveis ao longo do tempo.

**Colaboração Facilitada:** Regras claras no banco facilitam o trabalho em equipe e manutenção.

**Qualidade dos Dados:** Dados de alta qualidade levam a melhores decisões de negócio.

> **"As restrições de integridade são a fundação sólida sobre a qual construímos sistemas de banco de dados confiáveis. Invista tempo em projetá-las corretamente!"**