



Campus: Asa Norte

Curso: Desenvolvimento Full Stack

Disciplina: Iniciando o caminho pelo Java

Matrícula: 2023.09.96862-2

Semestre Letivo: 3º Semestre

Integrantes: André Luis Soares de Oliveira

## Desenvolvimento e Modelagem de Banco de Dados com SQL Server Management Studio

### Objetivo da Prática

Desenvolver um sistema de gerenciamento de compra e venda de produtos, modelando as tabelas e implementando operações básicas de manipulação de dados em SQL Server. O objetivo principal é entender a estrutura de bancos de dados relacionais e as diferentes cardinalidades nos relacionamentos entre entidades.

### Códigos Solicitados

-- Criação do banco de dados

```
CREATE DATABASE SistemaComprasVendas;
```

```
GO
```

-- Selecionar o banco de dados para uso

```
USE SistemaComprasVendas;
```

```
GO
```

-- Criação da SEQUENCE para o identificador de pessoas

```
CREATE SEQUENCE seq_pessoa_id AS INT
```

```
START WITH 1
```

```
INCREMENT BY 1;
```

```
GO
```

-- Tabela Usuarios

```
CREATE TABLE Usuarios (
```

```
id_usuario INT PRIMARY KEY IDENTITY(1,1),
```

```
nome_usuario VARCHAR(100) NOT NULL,
```

```
senha VARCHAR(50) NOT NULL,
```

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

```
);
```

```
-- Tabela Pessoas
```

```
CREATE TABLE Pessoas (
```

```
    id_pessoa INT PRIMARY KEY DEFAULT NEXT VALUE FOR seq_pessoa_id,
```

```
    tipo_pessoa CHAR(2) CHECK (tipo_pessoa IN ('PF', 'PJ')),
```

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

```
    cpf VARCHAR(14) NULL,
```

```
    cnpj VARCHAR(18) NULL,
```

```
    endereco VARCHAR(200),
```

```
    telefone VARCHAR(15),
```

```
    email VARCHAR(100)
```

```
);
```

```
-- Tabela Produtos
```

```
CREATE TABLE Produtos (
```

```
    id_produto INT PRIMARY KEY IDENTITY(1,1),
```

```
    nome_produto VARCHAR(100) NOT NULL,
```

```
    quantidade INT NOT NULL,
```

```
    preco_venda DECIMAL(10, 2) NOT NULL
```

```
);
```

```
-- Tabela Compras
```

```
CREATE TABLE Compras (
```

```
    id_compra INT PRIMARY KEY IDENTITY(1,1),
```

```
    id_usuario INT NOT NULL,
```

```
id_produto INT NOT NULL,  
id_pessoa INT NOT NULL,  
quantidade INT NOT NULL,  
preco_unitario DECIMAL(10, 2) NOT NULL,  
data_compra DATE NOT NULL,  
FOREIGN KEY (id_usuario) REFERENCES Usuarios(id_usuario),  
FOREIGN KEY (id_produto) REFERENCES Produtos(id_produto),  
FOREIGN KEY (id_pessoa) REFERENCES Pessoas(id_pessoa)  
);
```

-- Tabela Vendas

```
CREATE TABLE Vendas (  
    id_venda INT PRIMARY KEY IDENTITY(1,1),  
    id_usuario INT NOT NULL,  
    id_produto INT NOT NULL,  
    id_pessoa INT NOT NULL,  
    quantidade INT NOT NULL,  
    preco_unitario DECIMAL(10, 2) NOT NULL,  
    data_venda DATE NOT NULL,  
    FOREIGN KEY (id_usuario) REFERENCES Usuarios(id_usuario),  
    FOREIGN KEY (id_produto) REFERENCES Produtos(id_produto),  
    FOREIGN KEY (id_pessoa) REFERENCES Pessoas(id_pessoa)  
);
```

-- Listar todos os usuários

```
SELECT * FROM Usuarios;
```

-- Listar todas as pessoas

```
SELECT * FROM Pessoas;
```

-- Listar todos os produtos

```
SELECT * FROM Produtos;
```

-- Listar todas as compras com detalhes do usuário e do fornecedor (Pessoa Jurídica)

```
SELECT c.id_compra, u.nome_usuario, p.nome AS fornecedor, c.quantidade, c.preco_unitario,  
c.data_compra
```

```
FROM Compras c
```

```
JOIN Usuarios u ON c.id_usuario = u.id_usuario
```

```
JOIN Pessoas p ON c.id_pessoa = p.id_pessoa
```

```
WHERE p.tipo_pessoa = 'PJ';
```

-- Listar todas as vendas com detalhes do usuário e do cliente (Pessoa Física)

```
SELECT v.id_venda, u.nome_usuario, p.nome AS cliente, v.quantidade, v.preco_unitario,  
v.data_venda
```

```
FROM Vendas v
```

```
JOIN Usuarios u ON v.id_usuario = u.id_usuario
```

```
JOIN Pessoas p ON v.id_pessoa = p.id_pessoa
```

```
WHERE p.tipo_pessoa = 'PF';
```

-- Selecionar compras onde o usuário existe

```
SELECT c.id_compra, c.id_usuario, u.nome_usuario
```

```
FROM Compras c
```

```
LEFT JOIN Usuarios u ON c.id_usuario = u.id_usuario
```

```
WHERE u.id_usuario IS NULL;
```

```
-- Esperado: sem resultados, o que significa que todas as compras têm um usuário válido.
```

```
-- Selecionar vendas onde o usuário existe
```

```
SELECT v.id_venda, v.id_usuario, u.nome_usuario
```

```
FROM Vendas v
```

```
LEFT JOIN Usuarios u ON v.id_usuario = u.id_usuario
```

```
WHERE u.id_usuario IS NULL;
```

```
-- Listar produtos com o número de vendas e compras para verificar relacionamentos 1xN
```

```
SELECT p.id_produto, p.nome_produto,
```

```
    (SELECT COUNT(*) FROM Compras c WHERE c.id_produto = p.id_produto) AS  
total_compras,
```

```
    (SELECT COUNT(*) FROM Vendas v WHERE v.id_produto = p.id_produto) AS total_vendas
```

```
FROM Produtos p;
```

```
-- Teste de integridade para compras com id_usuario inválido
```

```
INSERT INTO Compras (id_usuario, id_produto, id_pessoa, quantidade, preco_unitario,  
data_compra)
```

```
VALUES (9999, 1, 1, 10, 50.00, GETDATE());
```

```
-- Contar o número de pessoas físicas e jurídicas
```

```
SELECT tipo_pessoa, COUNT(*) AS total
```

```
FROM Pessoas
```

```
GROUP BY tipo_pessoa;
```

```
-- Inserindo usuários
```

```
INSERT INTO Usuarios (nome_usuario, senha, email)
```

```
VALUES
```

```
('op1', 'op1','eu@gmail.com'),
```

```
('op2', 'op2','ela@hotmail.com');
```

```
-- Inserindo produtos
```

```
INSERT INTO Produtos (nome_produto, quantidade, preco_venda)
```

```
VALUES
```

```
('Produto A', 100, 10.00),
```

```
('Produto B', 50, 20.00),
```

```
('Produto C', 75, 15.00);
```

```
-- Obter o próximo id de pessoa
```

```
DECLARE @id_pessoa INT = NEXT VALUE FOR seq_pessoa_id;
```

```
-- Inserir dados comuns em `Pessoas`
```

```
INSERT INTO Pessoas (id_pessoa, nome, endereco, telefone, tipo_pessoa, cpf, email)
```

```
VALUES (1, 'João Silva', 'Rua A, 123', '(11) 1234-5678', 'PF', '15935700011',  
'joao@tara.com');
```

```
INSERT INTO Pessoas (id_pessoa, nome, endereco, telefone, tipo_pessoa, cpf, email)
```

```
VALUES (2, 'Cleber Silva', 'Rua B, 555', '(21) 4321-0055', 'PF', '10022335511',  
'clebao@tara.com');
```

```
INSERT INTO Pessoas (id_pessoa, nome, endereco, telefone, tipo_pessoa, cnpj, email)
```

```
VALUES (3, 'Empresa XYZ Ltda.', 'Av. B, 456', '(11) 9876-5432', 'PJ', '11122233344455',  
'XYZrh@gmop.com');
```

-- Inserir uma compra (movimentação de entrada)

```
INSERT INTO Compras (id_usuario, id_produto, id_pessoa, quantidade, preco_unitario,
data_compra)
```

VALUES

```
(2, 1, 2, 20, 8.00, GETDATE()); -- Compra do produto A, fornecedor Pessoa Jurídica
```

-- Inserir uma venda (movimentação de saída)

```
INSERT INTO Vendas (id_usuario, id_produto, id_pessoa, quantidade, preco_unitario,
data_venda)
```

VALUES

```
(3, 1, 1, 5, 10.00, GETDATE()); -- Venda do produto A, para Pessoa Física
```

-- Dados Completos de Pessoas

```
SELECT p.id_pessoa, p.nome, p.endereco, p.telefone, cpf
```

```
FROM Pessoas p
```

-- Movimentações de Entrada (Compra)

```
SELECT c.id_compra, pr.nome_produto, p.nome AS fornecedor, c.quantidade,
c.preco_unitario,
```

```
(c.quantidade * c.preco_unitario) AS valor_total
```

```
FROM Compras c
```

```
JOIN Produtos pr ON c.id_produto = pr.id_produto
```

```
JOIN Pessoas p ON c.id_pessoa = p.id_pessoa;
```

--Movimentações de Saída (Venda)

```
SELECT v.id_venda, pr.nome_produto, p.nome AS comprador, v.quantidade, v.preco_unitario,
```



```

        (v.quantidade * v.preco_unitario) AS valor_total
FROM Vendas v
JOIN Produtos pr ON v.id_produto = pr.id_produto
JOIN Pessoas p ON v.id_pessoa = p.id_pessoa;

-- Valor Total das Entradas Agrupadas por Produto
SELECT pr.nome_produto, SUM(c.quantidade * c.preco_unitario) AS total_entrada
FROM Compras c
JOIN Produtos pr ON c.id_produto = pr.id_produto
GROUP BY pr.nome_produto;

--Valor Total das Saídas Agrupadas por Produto
SELECT pr.nome_produto, SUM(v.quantidade * v.preco_unitario) AS total_saida
FROM Vendas v
JOIN Produtos pr ON v.id_produto = pr.id_produto
GROUP BY pr.nome_produto;

--Operadores que Não Efetuaram Movimentações de Entrada
SELECT u.id_usuario, u.nome_usuario
FROM Usuarios u
LEFT JOIN Compras c ON u.id_usuario = c.id_usuario
WHERE c.id_usuario IS NULL;

--Valor Total de Entrada, Agrupado por Operador
SELECT u.nome_usuario, SUM(c.quantidade * c.preco_unitario) AS total_entrada
FROM Compras c

```

```
JOIN Usuarios u ON c.id_usuario = u.id_usuario
```

```
GROUP BY u.nome_usuario;
```

```
--Valor Total de Saída, Agrupado por Operador
```

```
SELECT u.nome_usuario, SUM(v.quantidade * v.preco_unitario) AS total_saida
```

```
FROM Vendas v
```

```
JOIN Usuarios u ON v.id_usuario = u.id_usuario
```

```
GROUP BY u.nome_usuario;
```

```
--Valor Médio de Venda por Produto (Média Ponderada)
```

```
SELECT pr.nome_produto,
```

```
       SUM(v.quantidade * v.preco_unitario) / NULLIF(SUM(v.quantidade), 0) AS  
valor_medio_ponderado
```

```
FROM Vendas v
```

```
JOIN Produtos pr ON v.id_produto = pr.id_produto
```

```
GROUP BY pr.nome_produto;
```

## Resultados da Execução

```
-- Tabela Vendas
CREATE TABLE Vendas (
    id_venda INT PRIMARY KEY IDENTITY(1,1),
    id_usuario INT NOT NULL,
    id_produto INT NOT NULL,
    id_pessoa INT NOT NULL,
    quantidade INT NOT NULL,
    preco_unitario DECIMAL(10, 2) NOT NULL,
    data_venda DATE NOT NULL,
    FOREIGN KEY (id_usuario) REFERENCES Usuarios(id_usuario),
    FOREIGN KEY (id_produto) REFERENCES Produtos(id_produto),
    FOREIGN KEY (id_pessoa) REFERENCES Pessoas(id_pessoa)
);

-- Listar todos os usuários
SELECT * FROM Usuarios;
```

0 %

Resultados Mensagens

id_venda	nome_usuario	cliente	quantidade	preco_unitario	data_venda
----------	--------------	---------	------------	----------------	------------

oja.sql - TILION.master (sa (54))\*

```
-- Tabela Vendas
CREATE TABLE Vendas (
    id_venda INT PRIMARY KEY IDENTITY(1,1),
    id_usuario INT NOT NULL,
    id_produto INT NOT NULL,
    id_pessoa INT NOT NULL,
    quantidade INT NOT NULL,
    preco_unitario DECIMAL(10, 2) NOT NULL,
    data_venda DATE NOT NULL,
    FOREIGN KEY (id_usuario) REFERENCES Usuarios(id_usuario),
    FOREIGN KEY (id_produto) REFERENCES Produtos(id_produto),
    FOREIGN KEY (id_pessoa) REFERENCES Pessoas(id_pessoa)
);

-- Listar todos os usuários
SELECT * FROM Usuarios;

-- Listar todas as pessoas
SELECT * FROM Pessoas;

-- Listar todos os produtos
SELECT * FROM Produtos;

-- Listar todas as compras com detalhes do usuário e do fornecedor (Pessoa Jurídica)
SELECT c.id_compra, u.nome_usuario, p.nome AS fornecedor, c.quantidade, c.preco_unitario, c.data_compra
FROM Compras c
```

00 %

Resultados Mensagens

	id_produto	nome_produto	quantidade	preco_venda
1	1	Produto A	100	10.00
2	2	Produto B	50	20.00
3	3	Produto C	75	15.00

Loja.sql - TILION.master (sa (54))\* -> X

```
-- Selecionar vendas onde o usuário existe
SELECT v.id_venda, v.id_usuario, u.nome_usuario
FROM Vendas v
LEFT JOIN Usuarios u ON v.id_usuario = u.id_usuario
WHERE u.id_usuario IS NULL;

-- Listar produtos com o número de vendas e compras para verificar relacionamentos 1xN
SELECT p.id_produto, p.nome_produto,
       (SELECT COUNT(*) FROM Compras c WHERE c.id_produto = p.id_produto) AS total_compras,
       (SELECT COUNT(*) FROM Vendas v WHERE v.id_produto = p.id_produto) AS total_vendas
FROM Produtos p;

-- Teste de integridade para compras com id_usuario inválido
INSERT INTO Compras (id_usuario, id_produto, id_pessoa, quantidade, preco_unitario, data_compra)
VALUES (9999, 1, 1, 10, 50.00, GETDATE());

-- Contar o número de pessoas físicas e jurídicas
```

100 %

Resultados Mensagens

	id_produto	nome_produto	total_compras	total_vendas
1	1	Produto A	1	1
2	2	Produto B	0	0
3	3	Produto C	0	0

## Análise e Conclusão

### Implementação de Diferentes Cardinalidades

- **Relacionamento 1x1:** Implementado usando uma chave estrangeira com uma restrição **UNIQUE** para assegurar a correspondência exclusiva entre as duas entidades.
- **Relacionamento 1xN:** Utilizado para as relações entre **Usuarios** e **Compras** ou **Vendas**, onde um usuário pode ter várias compras e vendas, mas cada compra e venda está associada a um único usuário.
- **Relacionamento NxN:** Realizado por meio de uma tabela intermediária. Em um cenário mais complexo, onde produtos e fornecedores, por exemplo, poderiam ter uma relação de muitos para muitos, uma tabela de associação entre **Produtos** e **Fornecedores** seria necessária.

### Relacionamento para Representar Herança

**Herança:** É representada pela tabela **Pessoas**, onde um campo **tipo\_pessoa** diferencia **Pessoa Física** e **Pessoa Jurídica**, permitindo o armazenamento de informações específicas (CPF para PF e CNPJ para PJ). Esse tipo de herança é conhecido como **Herança de Tabela Única**.

### SQL Server Management Studio e Produtividade

O SQL Server Management Studio melhora a produtividade ao oferecer uma interface gráfica que simplifica a criação de bancos de dados, tabelas e relacionamentos. A ferramenta de

**Intellisense** facilita a escrita de códigos, a geração de diagramas E-R, a execução de consultas e o monitoramento de desempenho.

### Diferenças no Uso de SEQUENCE e IDENTITY

- **SEQUENCE:** É um objeto separado da tabela que gera números em uma sequência, permitindo que o valor gerado seja usado em várias tabelas ou em contextos diferentes no banco. O valor é obtido com a função **NEXT VALUE FOR sequence\_name** e pode ser gerado e manipulado conforme necessário.
- **IDENTITY:** É uma propriedade aplicada diretamente a uma coluna específica de uma tabela, que incrementa automaticamente para cada nova linha inserida. Ela não pode ser usada fora da tabela onde foi definida e não permite reutilização em outras tabelas.

**Resumo:** **SEQUENCE** é mais flexível e reutilizável em vários contextos, enquanto **IDENTITY** é prático para chaves primárias autoincrementadas em uma tabela específica.

### Importância das Chaves Estrangeiras para a Consistência do Banco

Chaves estrangeiras garantem que o relacionamento entre tabelas seja consistente, pois elas **forçam a integridade referencial**. Ao definir uma chave estrangeira, o banco de dados impede que um registro faça referência a uma chave primária inexistente. Isso ajuda a evitar dados órfãos e inconsistências, pois os registros dependentes (em tabelas filhas) só podem existir quando há uma entrada correspondente na tabela de origem.

### Operadores do SQL na Álgebra Relacional e no Cálculo Relacional

- **Álgebra Relacional:** Operadores de álgebra relacional incluem operações que manipulam conjuntos de dados e executam operações como **Seleção (WHERE)**, **Projeção (SELECT)**, **Junção (JOIN)**, **União (UNION)**, **Intersecção (INTERSECT)** e **Diferença (EXCEPT)**.
- **Cálculo Relacional:** No cálculo relacional, as operações são baseadas em fórmulas lógicas. **Consultas com expressões lógicas (EXISTS, IN, ANY, ALL)** e condições aninhadas (**WHERE, HAVING**) representam o cálculo relacional, pois não manipulam diretamente os conjuntos, mas definem condições lógicas.

### Agrupamento em Consultas e Requisito Obrigatório

O agrupamento em SQL é feito com a cláusula **GROUP BY**, que permite agregar dados em conjuntos com base em valores de uma ou mais colunas.

- **Requisito obrigatório:** Todas as colunas no **SELECT** que não estão em uma função de agregação (como **SUM, COUNT, AVG**) devem aparecer no **GROUP BY**. Isso

assegura que as colunas agregadas e não agregadas sejam combinadas de forma consistente na consulta.

Esses princípios ajudam a manter dados organizados, consistentes e permitem extrair informações significativas a partir do banco.

### **Endereço do Repositório GIT**

<https://github.com/andreluissdo/Missao-Pratica-N2.git>