Estácio

Campus: Polo Taguatinga Sul
Curso: **Desenvolvimento Full Stack**
Disciplina: Vamos Manter as Informações? (RPG0015)
Turma: 9001
Semestre: 3º Semestre
Integrantes: Yan Silva Sales

Criando Banco de dados

Objetivo da Pratica:

1. Identificar os requisitos de um sistema e transformá-los no modelo adequado.
2. Utilizar ferramentas de modelagem para bases de dados relacionais.
3. Explorar a sintaxe SQL na criação das estruturas do banco (DDL).
4. Explorar a sintaxe SQL na consulta e manipulação de dados (DML)
5. No final do exercício, o aluno terá vivenciado a experiência de modelar a base de dados para um sistema simples, além de implementá-la, através da sintaxe SQL, na plataforma do SQL Server.

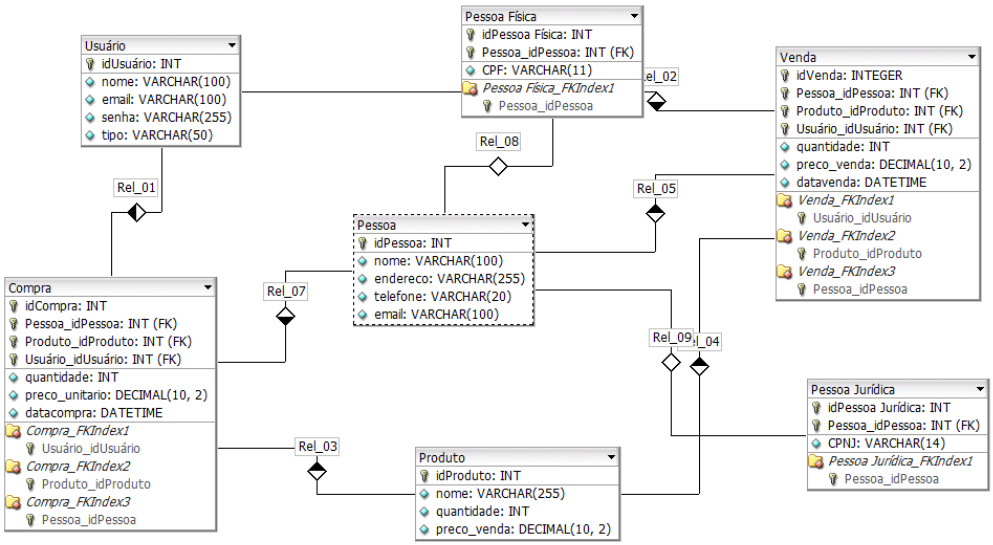
Códigos:

```sql
-- Criação do banco de dados

CREATE DATABASE SistemaComercial;

USE SistemaComercial;

-- Sequence para geração de IDs para Pessoa
CREATE SEQUENCE seq_pessoa
    AS INT
    START WITH 1
    INCREMENT BY 1;

-- Tabela de Usuários
CREATE TABLE Usuarios (
    ID INT IDENTITY(1,1) PRIMARY KEY,
    Nome VARCHAR(100),
    Email VARCHAR(100) UNIQUE,
    Senha VARCHAR(255), -- Deverá ser armazenada com hash
    Tipo VARCHAR(50)
);

-- Tabela de Pessoas
CREATE TABLE Pessoas (
    ID INT DEFAULT (NEXT VALUE FOR seq_pessoa) PRIMARY KEY,
    Nome VARCHAR(100),
    Endereço VARCHAR(255),
    Telefone VARCHAR(20),
    Email VARCHAR(100)
);

-- Tabela de Pessoa Física
CREATE TABLE Pessoa_Fisica (
    PessoaID INT PRIMARY KEY,
    CPF VARCHAR(11) UNIQUE,
    FOREIGN KEY (PessoaID) REFERENCES Pessoas(ID)
);
```

```sql
-- Tabela de Pessoa Jurídica
CREATE TABLE Pessoa_Juridica (
    PessoaID INT PRIMARY KEY,
    CNPJ VARCHAR(14) UNIQUE,
    FOREIGN KEY (PessoaID) REFERENCES Pessoas(ID)
);

-- Tabela de Produtos
CREATE TABLE Produtos (
    ID INT IDENTITY(1,1) PRIMARY KEY,
    Nome VARCHAR(100),
    Quantidade INT,
    PreçoVenda DECIMAL(10, 2)
);

-- Tabela de Compras
CREATE TABLE Compras (
    ID INT IDENTITY(1,1) PRIMARY KEY,
    ProdutoID INT,
    OperadorID INT,
    PessoaJuridicaID INT,
    Quantidade INT,
    PreçoUnitario DECIMAL(10, 2),
    DataCompra DATETIME,
    FOREIGN KEY (ProdutoID) REFERENCES Produtos(ID),
    FOREIGN KEY (OperadorID) REFERENCES Usuarios(ID),
    FOREIGN KEY (PessoaJuridicaID) REFERENCES Pessoa_Juridica(PessoaID)
);

-- Tabela de Vendas
CREATE TABLE Vendas (
    ID INT IDENTITY(1,1) PRIMARY KEY,
    ProdutoID INT,
    OperadorID INT,
    PessoaFisicaID INT,
    Quantidade INT,
    PreçoUnitario DECIMAL(10, 2),
    DataVenda DATETIME,
    FOREIGN KEY (ProdutoID) REFERENCES Produtos(ID),
    FOREIGN KEY (OperadorID) REFERENCES Usuarios(ID),
    FOREIGN KEY (PessoaFisicaID) REFERENCES Pessoa_Fisica(PessoaID)
);

USE SistemaComercial;

-- Inserção de usuários
INSERT INTO Usuarios (Nome, Email, Senha, Tipo)
VALUES ('Operador 1', 'op1@sistema.com', 'op1', 'Operador');

INSERT INTO Usuarios (Nome, Email, Senha, Tipo)
VALUES ('Operador 2', 'op2@sistema.com', 'op2', 'Operador');

-- Inserção de produtos

INSERT INTO Produtos (Nome, Quantidade, PreçoVenda)
VALUES ('Banana', 100, 5.00);

INSERT INTO Produtos (Nome, Quantidade, PreçoVenda)
VALUES ('Laranja', 500, 2.00);

INSERT INTO Produtos (Nome, Quantidade, PreçoVenda)
VALUES ('Manga', 800, 4.00);
```

```
-- Obter o próximo ID de pessoa a partir da sequence
DECLARE @ProximoIdPessoa INT;
SET @ProximoIdPessoa = NEXT VALUE FOR seq_pessoa;

-- Inserção de uma pessoa física
INSERT INTO Pessoas (ID, Nome, Endereço, Telefone, Email)
VALUES (@ProximoIdPessoa, 'João Silva', 'Rua das Flores, 123', '123456789',
'joao.silva@example.com');

INSERT INTO Pessoa_Fisica (PessoaID, CPF)
VALUES (@ProximoIdPessoa, '12345678901');

-- Obter o próximo ID de pessoa para pessoa jurídica
SET @ProximoIdPessoa = NEXT VALUE FOR seq_pessoa;

-- Inserção de uma pessoa jurídica
INSERT INTO Pessoas (ID, Nome, Endereço, Telefone, Email)
VALUES (@ProximoIdPessoa, 'Empresa XYZ', 'Av. das Nações, 456', '987654321',
'contato@xyz.com');

INSERT INTO Pessoa_Juridica (PessoaID, CNPJ)
VALUES (@ProximoIdPessoa, '12345678000199');

-- Inserir uma compra de produto "Banana" para "Empresa XYZ"
INSERT INTO Compras (ProdutoID, OperadorID, PessoaJuridicaID, Quantidade,
PreçoUnitario, DataCompra)
VALUES (1, 1, 2, 50, 4.50, GETDATE());

-- Inserir uma venda de produto "Banana" para "João Silva"
INSERT INTO Vendas (ProdutoID, OperadorID, PessoaFisicaID, Quantidade,
PreçoUnitario, DataVenda)
VALUES (1, 2, 1, 30, 5.00, GETDATE());
```

Resultados:



	name
1	Usuarios
2	Pessoas
3	Pessoa_Fisica
4	Pessoa_Juridica
5	Produtos
6	Compras
7	Vendas
8	sysdiagrams

	ID	Nome	Endereço	Telefone	Email	CPF
1	1	João Silva	Rua das Flores, 123	123456789	joao.silva@example.com	12345678901

	ID	Nome	Endereço	Telefone	Email	CNPJ
1	2	Empresa XYZ	Av. das Nações, 456	987654321	contato@xyz.com	12345678000199

	ProdutoID	Produto	Fornecedor	Quantidade	PreçoUnitario	ValorTotal
1	1	Banana	12345678000199	50	4.50	225.00

	ProdutoID	Produto	Comprador	Quantidade	PreçoUnitario	ValorTotal
1	1	Banana	12345678901	30	5.00	150.00

Conclusão

Com base nas atividades anteriores, foi possível analisar como as diferentes cardinalidades são implementadas em um banco de dados relacional, bem como a melhor forma de representar conceitos de herança. Também discutimos como o SQL Server Management Studio (SSMS) contribui para a melhoria da produtividade nas tarefas relacionadas ao gerenciamento de banco de dados.

Implementação de Cardinalidades

Em bancos de dados relacionais, a cardinalidade define como as tabelas estão relacionadas entre si. No caso do modelo desenvolvido, observamos os seguintes padrões:

- **Cardinalidade 1x1 (Um para Um)**: Essa configuração é usada quando para cada registro em uma tabela existe no máximo um registro correspondente em outra tabela. A implementação comum é por meio de chaves primárias compartilhadas ou chaves estrangeiras únicas. No modelo, vimos essa configuração entre "Pessoas" e "Pessoa_Física" ou "Pessoa_Jurídica".

- **Cardinalidade 1xN (Um para Muitos)**: Aqui, um registro em uma tabela pode estar associado a vários registros em outra tabela. Essa relação é implementada por meio de chaves estrangeiras. No exemplo, observamos essa cardinalidade entre "Produtos" e "Compras", e entre "Produtos" e "Vendas".
- **Cardinalidade NxN (Muitos para Muitos)**: Essa cardinalidade permite que vários registros de uma tabela estejam associados a vários registros de outra tabela. Isso geralmente é implementado com uma tabela de junção. Embora não tenhamos um exemplo explícito desta cardinalidade, descrevemos como ela seria usada para situações de relacionamento entre duas tabelas com muitos para muitos.

Representação de Herança

No contexto de bancos de dados relacionais, a herança é representada de forma diferente do que em linguagens de programação orientadas a objetos. No modelo, a herança é simulada com relacionamentos 1x1 entre uma tabela base e suas tabelas derivadas. O uso de uma chave primária compartilhada permite simular a herança, como visto entre "Pessoas" e suas subclasses "Pessoa_Física" e "Pessoa_Jurídica".

SQL Server Management Studio (SSMS)

O SQL Server Management Studio (SSMS) é uma ferramenta abrangente para gerenciamento de bancos de dados. Ele melhora a produtividade ao oferecer uma interface gráfica intuitiva, um editor SQL robusto e ferramentas para gerenciamento, otimização e monitoramento de banco de dados. Os recursos do SSMS, como o Object Explorer, editor SQL com destaque de sintaxe, ferramentas de análise de desempenho e backup/restauração, facilitam a gestão de banco de dados e a resolução de problemas.

Diferenças entre SEQUENCE e IDENTITY

Tanto **SEQUENCE** quanto **IDENTITY** são usados para gerar valores automáticos para chaves primárias ou outros campos em um banco de dados, mas eles têm diferenças importantes:

- **IDENTITY**:
    - É uma propriedade de coluna que automaticamente incrementa um valor cada vez que um novo registro é inserido na tabela. A definição ocorre no momento da criação da tabela, especificando um valor inicial e um incremento.
    - O valor gerado pelo **IDENTITY** é atrelado à tabela onde é definido, e geralmente é usado para campos de chave primária.
    - Não permite flexibilidade na obtenção do próximo valor antes da inserção ou para reutilização em outras tabelas.
- **SEQUENCE**:

- É um objeto separado da tabela, capaz de gerar valores automaticamente de forma sequencial. Pode ser criado e referenciado em diferentes contextos, permitindo que múltiplas tabelas compartilhem a mesma sequência.
- Oferece mais controle sobre o comportamento do incremento, permitindo reinicializações, ajustes de incremento e obtenção do próximo valor sem inserir registros.
- Ideal para situações em que múltiplas tabelas ou colunas precisam compartilhar a mesma sequência ou onde há necessidade de flexibilidade adicional.

Importância das Chaves Estrangeiras para a Consistência do Banco de Dados

Chaves estrangeiras são fundamentais para manter a consistência e integridade dos dados em um banco de dados relacional. Elas criam vínculos entre tabelas, estabelecendo relacionamentos que ajudam a garantir que:

- **Referências Consistentes**:
  - Uma chave estrangeira em uma tabela deve referenciar uma chave primária existente em outra tabela. Isso evita referências inválidas ou órfãs.
- **Integridade Referencial**:
  - As chaves estrangeiras mantêm a integridade referencial, garantindo que alterações ou exclusões em uma tabela afetem consistentemente as tabelas relacionadas.
- **Evitar Redundância de Dados**:
  - Chaves estrangeiras ajudam a evitar duplicação de dados, pois permitem referenciar informações em vez de replicá-las.

Operadores do SQL na Álgebra Relacional e no Cálculo Relacional

SQL é uma linguagem que se baseia em conceitos de álgebra relacional e cálculo relacional:

- **Álgebra Relacional**:
  - Os operadores que pertencem à álgebra relacional incluem:
    - **SELECT**: Para selecionar colunas específicas.
    - **PROJECT**: Para selecionar linhas que atendem a uma condição.
    - **JOIN**: Para combinar tabelas com base em chaves comuns.

        - **UNION**, **INTERSECT**, **MINUS**: Para operações em conjuntos de dados.

- **Cálculo Relacional**:
    - Os operadores relacionados ao cálculo relacional tendem a lidar com predicados e expressões lógicas para consultas mais complexas:
        - Operadores lógicos como **AND**, **OR**, **NOT**.
        - Subconsultas que utilizam **EXISTS**, **ALL**, **ANY**, entre outros.
        - Expressões que envolvem predicados para condições complexas.

Agrupamento em Consultas SQL

O agrupamento em consultas SQL é realizado com a cláusula **GROUP BY**, permitindo organizar registros em grupos com base em valores de uma ou mais colunas. Quando usado em conjunto com funções agregadas (**SUM**, **AVG**, **COUNT**, etc.), ele permite gerar resumos e estatísticas para esses grupos.

Um requisito obrigatório para o uso do **GROUP BY** é que qualquer coluna selecionada deve ser parte do agrupamento ou usada dentro de uma função agregada. Ou seja, quando se usa **GROUP BY**, todas as colunas no **SELECT** devem ser agregadas ou fazer parte do critério de agrupamento, garantindo consistência nos resultados.

Com essas conclusões, podemos entender melhor as diferenças entre **SEQUENCE** e **IDENTITY**, a importância das chaves estrangeiras para a consistência do banco de dados, a distinção entre operadores da álgebra relacional e do cálculo relacional, e como o agrupamento é feito em consultas SQL com seus requisitos obrigatórios. Esses conceitos são fundamentais para quem trabalha com bancos de dados relacionais e buscam otimizar e manter a integridade dos dados.

Considerações Finais

Este estudo demonstrou como diferentes cardinalidades são implementadas em bancos de dados relacionais e como a herança pode ser simulada com relacionamentos 1x1. Também exploramos a importância do SQL Server Management Studio para aumentar a produtividade no gerenciamento de bancos de dados. O modelo criado oferece um exemplo prático de como estruturar e manter um banco de dados relacional com essas características.