

# Unidade II

## 3 MODELO ENTIDADE–RELACIONAMENTO (MER)

O modelo entidade-relacionamento (MER) foi idealizado por Peter P. Chen em março de 1976 no trabalho intitulado *The entity-relationship model: toward the unified view of data*, no qual ele definiu uma possível abordagem para o processo de modelagem dos dados.

O MER possui uma técnica de diagramação e um conjunto de conceitos que devem ser bem entendidos e não devem ser desrespeitados quanto à simbologia. Nessa técnica de diagramação, a simplicidade é um dos preceitos básicos e serve como meio de representação dos próprios conceitos por ela manipulados. Utiliza-se um retângulo para representar as entidades, um losango para representar os relacionamentos e pequenos balões para indicar os atributos que cada entidade possui.

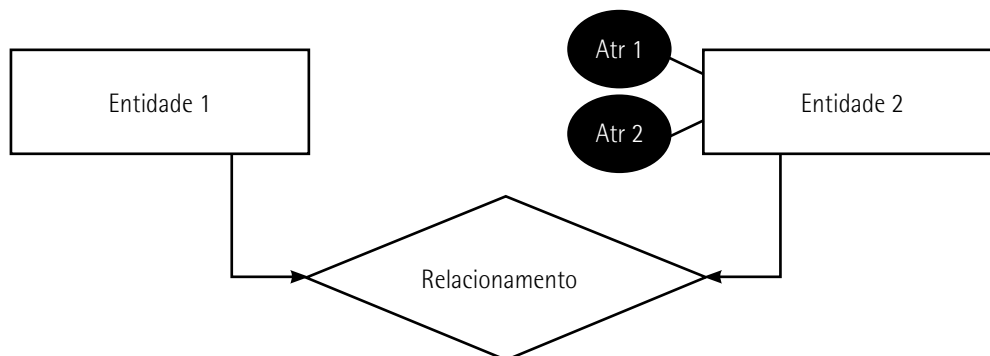


Figura 20 – Abordagem entidade-relacionamento

O desenvolvimento de um diagrama ER envolve diversas escolhas, incluindo as seguintes:

- Uma entidade ou um atributo deve possuir um conceito para que possa ser modelado?
- Uma entidade ou um relacionamento deve possuir um conceito para que possa ser modelado?
- Quais são os conjuntos de relacionamentos e seus conjuntos de entidades participantes?
- Devemos usar relacionamentos binários ou ternários?
- Devemos usar agregação?

Discutiremos agora os aspectos envolvidos ao fazer essas escolhas.

## Entidade *versus* atributo

Quando identificamos os atributos de um conjunto de entidades, algumas informações não ficam claras no início, entre elas: se uma propriedade pode ser modelada como um atributo ou como um conjunto de entidades. Como opção podemos partir do exemplo de entidade Funcionario; o atributo endereço pode ser um dado adequado para representar o endereço da entidade em questão, entretanto outros atributos podem e devem estar descritos nessa entidade.

### 3.1 Elementos do modelo de dados

**Entidade** é um objeto ou evento (real ou abstrato) que se torna um ponto de interesse dentro de uma determinada realidade e ao qual podem ser associados dados, relacionamentos etc. O retângulo representa o tipo Entidade. Exemplo: Cliente, Fornecedor, Produto, Aluno etc.

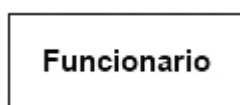


Figura 21 – Objeto Funcionario

**Atributos** são propriedades (características) que identificam as entidades. Uma entidade é representada por um conjunto de atributos. Os atributos podem ser simples, compostos ou multivalorados. Um atributo no DER é representado por um pequeno círculo ligado ao Tipo Entidade.

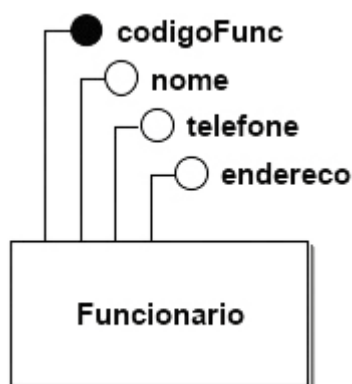


Figura 22 – Entidade Funcionario

**Atributo simples** não possui qualquer característica especial. A maioria dos atributos será simples. São chamados também de atributos atômicos, porque eles não são divisíveis.

**Atributo composto** tem seu conteúdo formado por vários itens menores. Exemplo: endereço. Seu conteúdo poderá ser dividido em vários outros atributos, como: rua, número, complemento, bairro, CEP e cidade. Esse tipo de atributo é chamado de atributo composto. Um atributo composto pode formar uma hierarquia.

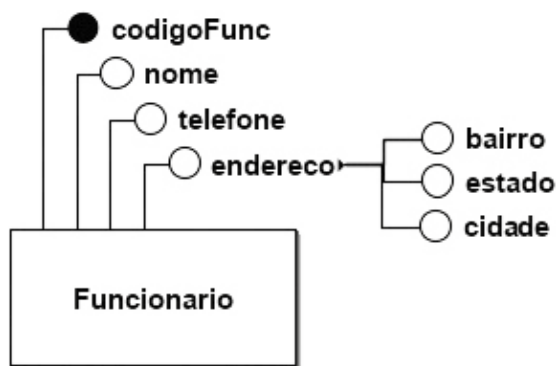


Figura 23 – Entidade Funcionario – composto

**Atributo multivalorado** tem seu conteúdo formado por mais de um valor. Exemplo: telefone. Uma pessoa poderá ter mais de um número de telefone. É indicado colocando-se um asterisco precedendo o nome do atributo. Observação: no DER, um atributo multivalorado é representado por um círculo com contorno em linha dupla.

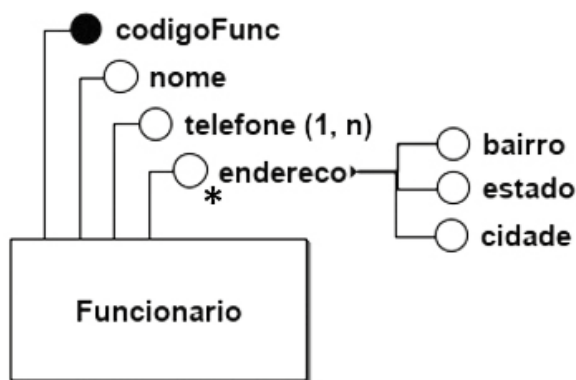


Figura 24 – Entidade Funcionario – multivalorado (telefone)

**Atributo derivado** tem um valor que pode ser derivado a partir de outro atributo (base).

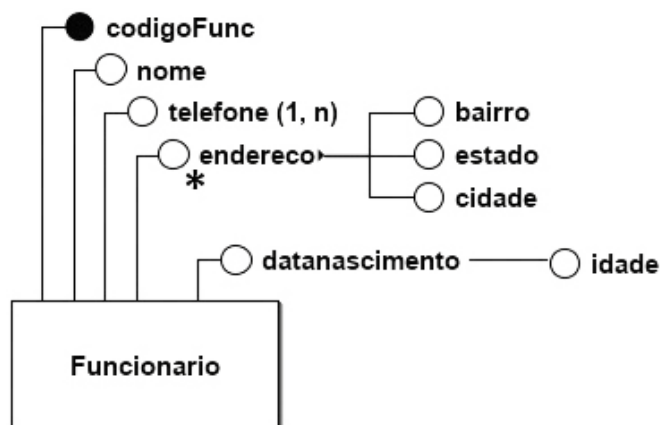


Figura 25 – Entidade Funcionario – derivado (datanascimento)

**Atributo identificador de entidade ou identificador único** é um conjunto de um ou mais atributos cujos valores servem para distinguir uma entidade em uma coleção de entidades. O identificador de uma entidade é denominado chave primária. Os atributos da entidade que formam seu identificador são representados graficamente através de círculos pretos ou um traço, conforme mostra a figura a seguir:

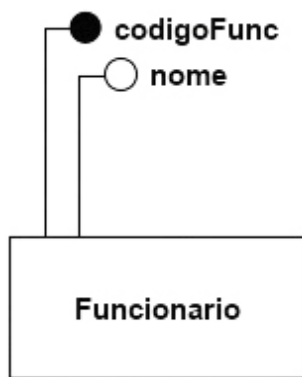


Figura 26 – Entidade Funcionario com atributos (codigoFunc e nome)

**Atributos de relacionamento** são utilizados com a finalidade de estabelecer um vínculo entre os dados de entidades distintas.

### 3.2 Generalização/especialização

Uma entidade pode ter propriedades particulares e específicas em um subconjunto de ocorrências de uma entidade genérica. Além de relacionamentos e atributos, propriedades podem ser atribuídas a entidades através do conceito de generalização/especialização. A partir desse conceito, é possível atribuir propriedades particulares a um subconjunto das ocorrências (especializadas) de uma entidade genérica. No DER, o símbolo para representar generalização/especialização é um triângulo isósceles, conforme mostra a figura a seguir. A generalização/especialização, representada nessa figura, expressa que a entidade CLIENTE é dividida em dois subconjuntos, as entidades PESSOA FÍSICA e PESSOA JURÍDICA, cada uma com propriedades próprias.

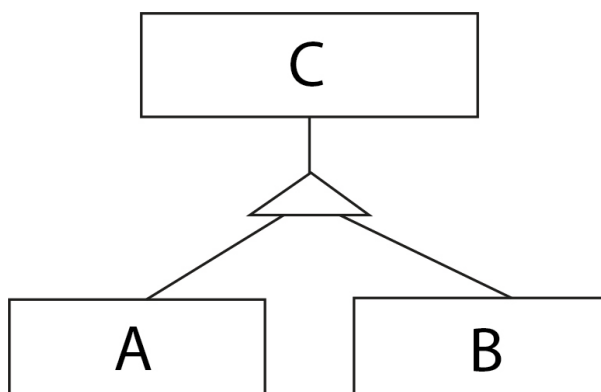


Figura 27 – Generalização/especialização

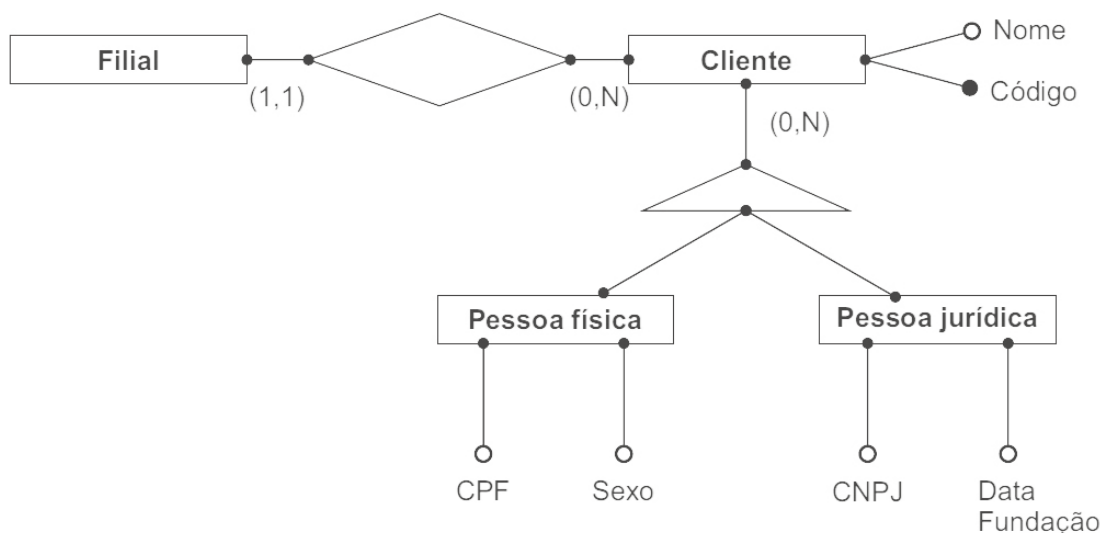


Figura 28 – Generalização/especialização entidade Cliente com pessoa física e pessoa jurídica



### Saiba mais

O conceito de generalização/especialização é amplamente utilizado na programação orientada a objeto e remete ao recurso POO de herança.

Para conhecer mais sobre esse recurso, pesquise sobre o assunto no livro a seguir:

THORWALD, G. A. G. *UML2: uma abordagem prática*. 2. ed. São Paulo: Novatec, 2011.

## 3.3 Relacionamento

Relacionamento é uma estrutura que indica uma associação entre duas ou mais entidades. Trata-se dos elementos de dados que representam associações entre entidades e que chamamos de relacionamentos. Alguns exemplos de relacionamentos típicos são "trabalha-em", "trabalha-para", "compra", "dirige" ou qualquer verbo que conecte entidades. Para cada relacionamento, os seguintes conceitos devem ser especificados: grau (binário, ternário etc.), conectividade (um-para-muitos etc.), participação opcional ou obrigatória; e quaisquer atributos associados ao relacionamento, e não às entidades. A seguir, vejamos algumas orientações para definir os tipos de relacionamentos mais difíceis.



### Observação

Todo relacionamento muitos-para-muitos pode ser entendido como uma entidade, no qual essas entidades se associam entre si, pois elas representam um fato.

Quadro 1 – Entidade e relacionamentos

Entidade	Relacionamento	Entidade
Empregado	Exerce	Tarefa
Tarefa	É exercida por	Empregado
Produto	É classificado por um	Tipo Produto
Tipo Produto	É uma classificação de	Produto
Pessoa	Faz	Reserva
Reserva	É feita por	Pessoa

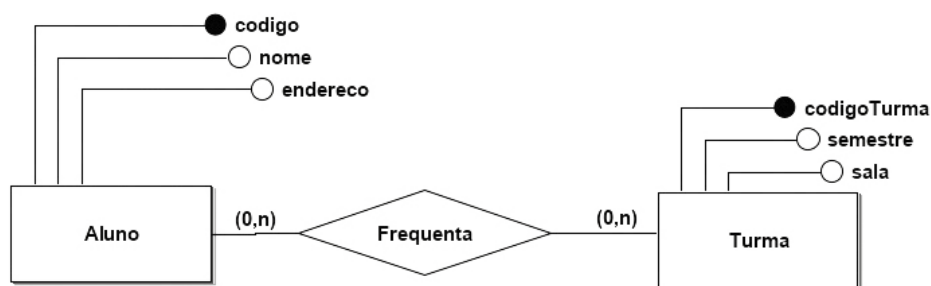


Figura 29 – Diagrama entidade-relacionamento entre Aluno e Turma

### 3.4 Cardinalidade do relacionamento

A cardinalidade expressa o número mínimo e máximo de ocorrências de cada uma das entidades que participam do mesmo relacionamento de uma dada entidade. No DER, a cardinalidade é indicada colocando-se os respectivos números dos lados das entidades, utilizando o formato (x, y). O primeiro valor representa o número mínimo de entidades associadas, enquanto o segundo representa o número máximo.

Ferramentas de modelagem de ER baseado na notação pé de galinha não apresentam a faixa de cardinalidade numérica no diagrama. Em vez disso, o usuário pode adicioná-la como texto. Na notação pé de galinha, a cardinalidade é denotada pela utilização dos símbolos, conforme vemos na próxima figura. A faixa de cardinalidade numérica pode ser adicionada utilizando-se a ferramenta de inserção de texto conforme vemos aqui.

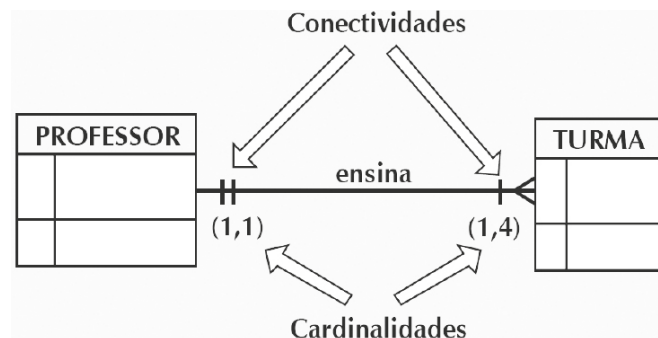


Figura 30 – Diagrama entidade-relacionamento entre Professor e Turma

As conectividades e cardinalidades são estabelecidas por afirmações muito concisas conhecidas como regras de negócio. Essas regras, resultantes da descrição precisa e detalhada do ambiente de dados de uma organização, também estabelecem as entidades, atributos, relacionamentos, conectividades, cardinalidades e restrições do ER. Como as regras de negócio definem os componentes do ER, assegurar que todas estejam identificadas é uma parte importante do trabalho de um projetista de banco de dados.

### 3.4.1 Força do relacionamento

O conceito de força de relacionamento baseia-se em como é definida a relação entre a chave primária de uma entidade relacionada. Para implementar um relacionamento, a chave primária de uma entidade aparece como chave estrangeira relacionada. Por exemplo, o relacionamento 1:M entre Escritor e Livro é implementado pela utilização de chave primária `idEscritor` de Livro como uma chave estrangeira de Escritor.



#### Lembrete

Relacionamento é uma representação das associações existentes entre entidades no mundo real. Muitos relacionamentos não têm existência física ou conceitual, outros dependem da associação de outras entidades.

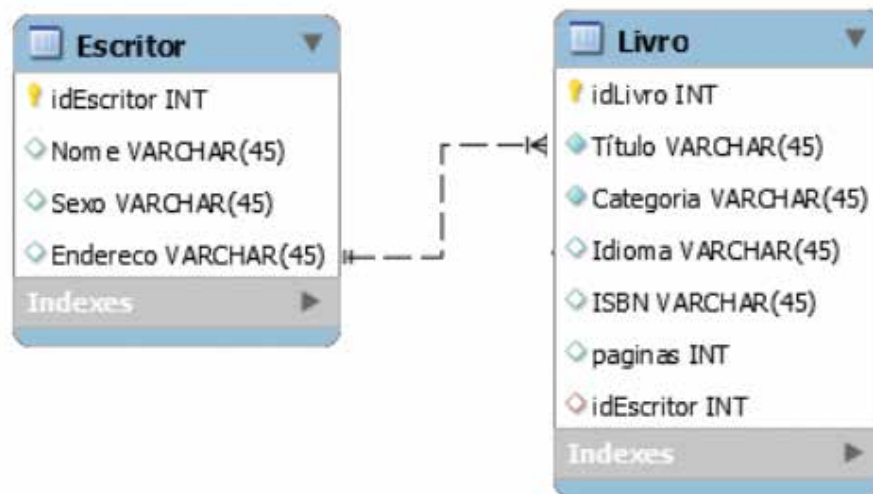


Figura 31 – Diagrama entidade-relacionamento entre Escritor e Livro

Quando trabalhamos com relacionamentos, a teoria do conjunto pode ser absorvida para o relacionamento binário R entre o conjunto de entidades A e B, e a cardinalidade do mapeamento precisa ser um dos seguintes tipos:

- **Um-para-um:** a entidade A está associada no máximo a uma entidade B e uma entidade B está associada no máximo à entidade de A.

- **Um-para-muitos:** a entidade A está associada a qualquer número de entidades de B. Uma entidade de B, entretanto, pode estar associada, no máximo, a uma entidade de A.
- **Muitos-para-um:** a entidade A está associada, no máximo, a uma entidade de B. Uma entidade de B, entretanto, pode estar associada a qualquer número de entidades de A.
- **Muitos-para-muitos:** a entidade A está associada a qualquer número de entidades de B e uma entidade de B está associada a qualquer número de entidades de A.

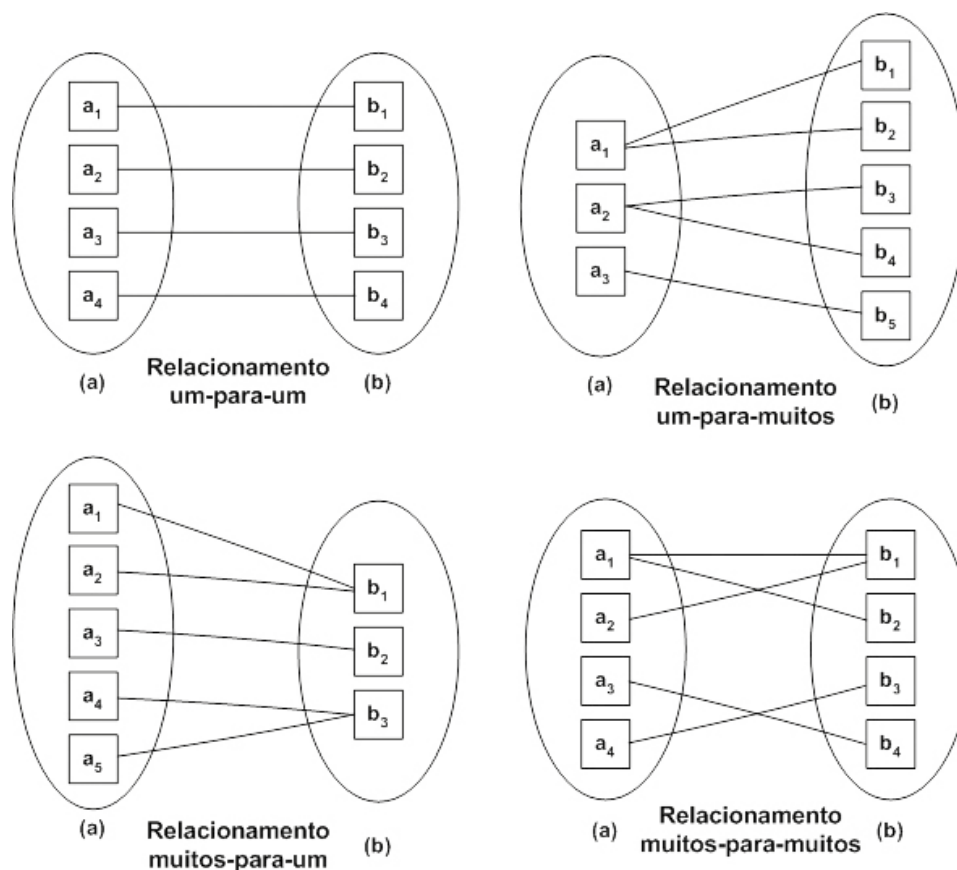


Figura 32 – Regras (baseadas nos tipos de relacionamento)

## 3.5 Regras de integridade

As questões de integridade de um modelo relacional estão ligadas aos conceitos de chave primária e chave estrangeira. Essas regras servem para garantir que as tabelas guardam informações compatíveis e são de extrema importância para a confiabilidade das informações contidas no banco de dados.

Os conceitos de chaves no projeto devem ser uma das questões centrais, pois as chaves de modelagem em um projeto eliminam o maior número de efeitos da redundância. Uma chave representa um conjunto de um ou mais atributos que, unidos, permite identificar unicamente uma entidade a partir de sua chave. Podemos encontrar mais de uma chave em uma entidade, que será chamada de chave candidata.



**Chave primária** pode ser definida como um conjunto de atributos que identificam unicamente uma entidade ou objeto. Imagine, por exemplo, uma entidade ANIMAL contendo um atributo Nro de controle animal. Esse atributo é uma chave primária (*primary key* – pk), pelo fato de não existir dois animais dentre os cadastrados com o mesmo Nro de controle animal. Em um banco de dados, é permitida apenas uma chave primária a cada entidade.

**Chave estrangeira** pode ser definida como conjunto de atributos de uma entidade cujos valores aparecem como chave primária em outra entidade. A presença de uma chave estrangeira (*foreign key* – fk) em uma entidade ocorre por força das regras de integridade referencial.

A regra de integridade de entidade diz que uma chave primária não pode conter um valor nulo (NULL). Esse NULL não é o valor 0 (zero) nem o caractere branco, é simplesmente a não existência de valor no campo. Em termos de DER, o atributo pk deve ser sempre obrigatório, nunca opcional.

A regra de integridade referencial determina que uma tabela A possui uma chave estrangeira, que é uma chave primária em outra tabela B. Para cada valor contido na chave estrangeira de A, deve haver um valor na chave primária da tabela B ou o valor da chave na tabela A deve ser nulo (NULL). Ou seja, não podem existir valores na chave estrangeira que não existam na tabela na qual ela é uma chave primária.

A integridade de domínio determina quais valores são permitidos em uma coluna. Normalmente, implementamos esse tipo de integridade ao definirmos os tipos de dados em uma coluna. Podemos citar uma coluna para armazenar datas. Alguns desenvolvedores definem o tipo de dados caractere para armazenamento das informações. Nesse caso, se o usuário digitar uma data inválida, não haverá bloqueio da inserção de dados.

### 3.6 Dicionário de dados

Presente em grandes implementações de banco de dados, um dicionário de dados representa um pequeno banco de dados, onde toda a descrição do DER está armazenada. O dicionário de dados deve ser integrado ao DER. A partir do DER, deve ser possível visualizar e editar as entradas do dicionário de dados correspondentes a elementos selecionados do DER.

Entre os documentos provenientes da análise de sistemas, temos o dicionário de dados. Este permite que os analistas obtenham informações sobre todos os objetos do modelo de forma textual, contendo explicações por vezes difíceis de incluir no diagrama. É válido lembrar que o objetivo do documento é ser claro e consistente.

Para construir o dicionário de dados, pode-se utilizar um processador de textos, uma planilha eletrônica ou um banco de dados (esta é a melhor opção). A escolha provavelmente irá recair sobre o programa que for mais conhecido e dominado na organização em questão.

Quando informações sobre uma tabela são necessárias, elas são obtidas a partir do catálogo do sistema. É claro que, no nível de implementação, sempre que um SGBD precisar descobrir o esquema de uma tabela de catálogo, o código que recupera essas informações deve ser manipulado de forma

especial (caso contrário, o código tem de recuperar essas informações das tabelas de catálogo sem, presumivelmente, conhecer o esquema das tabelas de catálogo).

**Tabela 2 – <Nome da Tabela>**

Nome	Descrição	Tipo	Tamanho	Domínio	Formato	Restrições
<campo1>	<descricao do campo1>	<tipo1>	<tamanho1>	<dominio1>	<formato1>	<regra1>
<campo2>	<descricao do campo2>	<tipo>	<tamanho2>	<dominio2>	<formato2>	<regra2>
<campoN>	<descricao do campoN>	<tipoN>	<tamanhoN>	<dominioN>	<formatoN>	<regraN>

Segue exemplo:

**Tabela 3 – Dados Entidade Funcionario**

Atributo	Descrição	Tipo	Tamanho	Domínio	Formato	Restrições
Código	Código que identifica os dados do funcionário no banco de dados	Numérico		-	-	Chave primária
Nome	Nome completo do funcionário	Texto	50	-	-	-
CPF	Número do Cadastro Pessoa Física do funcionário	Numérico	11	-	99999999-99	CPF deve ser válido
Sexo	Sexo do funcionário (Masc ou Fem)	Texto	12	M-Masculino F-Feminino	F - M	-
DataNasc	Data de nascimento do funcionário	Data	10	-	DD/MM/AAAA	O funcionário deve ser maior de idade
Email	Endereço de e-mail do funcionário	Texto	30	-	nome@dominio.com	Deve possuir função de validação
Telefone	Número de contato do funcionário	Numérico	20	-	(00)0000-0000	-
Celular	Número de contato celular do funcionário	Numérico	20	-	(00)90000-0000	-
Endereço	Endereço de correspondência do funcionário	Texto	60	-	-	-

Adaptada de: Rob e Coronel (2011).

Analisando a tabela 3, teremos:

- **Entidade:** é o nome da entidade que foi definida no MER. A entidade é uma pessoa, objeto ou lugar que será considerada como objeto pelo qual temos interesse em guardar informações a seu respeito.
- **Atributo:** os atributos são as características da entidade Funcionario que desejamos guardar.
- **Descrição:** é opcional e pode ser usada para descrever o que é aquele atributo ou dar informações adicionais que possam ser usadas futuramente pelo analista ou programador do sistema.

- **Tipo:** pode ser numérico, texto, data e booleano. Podemos chamar também de tipo do valor que o atributo irá receber. A definição desses tipos deve seguir um processo lógico, por exemplo: nome é texto, salário é numérico, data de nascimento é data, e assim por diante.
- **Tamanho:** define a quantidade de caracteres que serão necessários para armazenar o seu conteúdo. Geralmente, o tamanho é definido apenas para atributos de domínio texto.
- **Domínio:** determina os valores permitidos em uma coluna.
- **Formato:** especifica a formatação padrão para os dados armazenados.
- **Restrições:** informam restrições a serem implementadas em uma coluna de uma tabela.

Podemos também incluir no dicionário de dados a classe:

- **Classe:** as classes podem ser simples, composto, multivalorado e determinante (obrigatório).

Mais um exemplo de dicionário de dados:

**Tabela 4 – Dados Tb\_Aluno**

TABELA: Tb_Aluno				Cadastro de alunos		
Informações pertinentes ao aluno						
CAMPO LÓGICO	CAMPO FÍSICO	TIPO	PK	FK (Tabela/Campo)	RESTRIÇÕES	OBSERVAÇÕES
Código	Alu_codigo	SMALLINT	PK		NÃO NULO E MAIOR QUE ZERO	Campo autoincremento
Nome	Alu_nome	VARCHAR(100)			NÃO NULO	Informar se usuário incluir somente uma palavra
Data de Nascimento	ento	DATE			Data mínima < HOJE	
Código da turma	Tur_codigo	SMALLINT		Tb_Turma/Tur_codigo	ZERO	
Sexo	Alu_sexo	CHAR(1)			Somente "M" ou "F"	M = Masculino / F = Feminino
Nome do pai	Alu_pai	VARCHAR(100)				
Nome da mãe	Alu_mae	VARCHAR(100)				
Nota Media	Not_codigo	SMALLINT		Tb_Acompanhamento Aluno/aco_codigo		
Registro do Aluno	Alu_RA	SMALLINT			NÃO NULO E MAIOR QUE ZERO	Campo autoincremento
Informações complementares	Alu_Complemento	BLOB				
Telefone	Alu_telefone	INTEGER				Telefone para contato
Celular	Alu_celular	INTEGER				Celular para contato
Endereco	Alu_endereco	VARCHAR(150)				Endereco do aluno

Adaptada de: Rob e Coronel (2011).

### 3.7 Normalização

Ao estudarmos modelagem e o papel de um banco de dados, independentemente da estrutura de *hardware* e *software* onde ele será implementado, é necessário que os dados estejam normalizados. Vimos que existem entidades que são compostas por um conjunto de atributos e estas se relacionam entre si através de relacionamentos especificados, como chave primária e estrangeira, para que haja uma ligação entre as entidades. Entretanto, percebemos que, no cenário atual, as dúvidas começam a

surgir em cascata. Quantas entidades o meu modelo deverá conter? Que atributos devo colocar? Em quais entidades coloco quais atributos? Quais relacionamentos vão existir? Qual tipo de cardinalidade será aplicado a cada relacionamento?

Os atributos de uma entidade não podem ser colocados ao acaso, pois dessa forma o mais provável será perder informações ou, então, iremos nos deparar com os cenários em que há duplicação de informações. Para evitar esses problemas existem mecanismos bem definidos que nos ajudam a "ajustar" o nosso modelo, passo a passo. Esse processo chama-se **normalização**.

O processo de normalização consiste em submeter o esquema das relações a uma bateria de testes com o objetivo de determinar qual o estado (forma normal) em que se encontra.

A normalização consiste em uma técnica geralmente usada como guia no projeto de banco de dados relacionais. Esse modelo, idealizado por Codd em 1970, propôs inicialmente três formas normais, tendo esse conjunto sido posteriormente estendido para outras formas normais.

Normalmente, ao normalizar aumenta-se o número de entidades e o número total de atributos, embora o número de atributos em cada entidade tenha tendência a diminuir, pois se pretende que um determinado dado ou informação seja armazenado no banco de dados uma única vez. No entanto, duas ameaças pairam sobre todos os bancos de dados, limitando ou reduzindo a sua eficiência:

- Redundância de dados.
- Existência de valores NULL.

Assim, para que o banco de dados se mantenha otimizado, é necessário limitar e minimizar a influência desses dois fatores negativos. Para isso, é preciso que o projeto do banco de dados seja bem estruturado.

A normalização é um processo de otimização estrutural que converte entidades com grande nível de redundância e valores nulos em entidades mais saudáveis, com menor redundância e com escassez de valores nulos.

O termo redundância é definido nos dicionários de língua portuguesa como "algo em excesso e em desperdício". Logo, ao trabalharmos com informações pretendemos que estas sejam precisas e que estejam replicadas em diferentes locais do banco de dados.

Através do processo de normalização, vão-se eliminando "anomalias" no esquema que se pretende implementar para suportar o banco de dados. Dessa forma, evita-se:

- Existência de grupos repetitivos.
- Existência de atributos multivalorados.

- Redundância de dados.
- Perda de informações.
- Falhas na sincronização dos mesmos dados existentes.
- Existência de atributos que dependem apenas de parte da chave primária.
- Existência de dependências transitivas entre atributos.

A maior parte do trabalho realizado no processo de projetos de um banco de dados consiste na sua normalização até um determinado nível (forma normal).

A teoria da normalização é baseada no conceito de **forma normal**. Existem regras às quais a estrutura da informação deve obedecer para que se possa afirmar qual o nível de normalização da estrutura. A essas regras dá-se, habitualmente, o nome de "formas normais":

- Primeira forma normal (1FN).
- Segunda forma normal (2FN).
- Terceira forma normal (3FN).
- Forma normal de Boyce-Codd (FNBC).
- Quarta forma normal (4FN).
- Quinta forma normal (5FN).

O objetivo final da normalização é permitir criar um conjunto de tabelas em um banco de dados livre de informações redundantes e que possa ser modificado de forma correta e consistente.

Objetivos da normalização:

- Minimizar ou eliminar a redundância de informações.
- Melhorar a performance do sistema, ao eliminar informações redundantes.
- Permitir a integridade referencial entre entidades.

O processo de normalização permite obter um esquema de banco de dados relacional capaz de suportar, de forma adequada, os dados relevantes de um determinado universo, evitando a redundância de informações e a existência de valores NULL propagada ao longo das tabelas e não permitindo, assim, decomposições ruins ou decomposições com perda de informações.

A existência de redundância de dados em uma determinada implementação acarreta problemas muitos sérios, dos quais destacamos:

- **Armazenamento:** a existência de redundância implica a ocupação de espaço físico adicional.
- **Manutenção:** o processamento dos dados (inserção, alteração, remoção) pode implicar o acesso a múltiplas tabelas devido à redundância existente. Tal situação é difícil de gerenciar, pois dificulta manter a consistência dos dados.
- **Desempenho:** quanto maior a redundância, maior será o conjunto de tabelas a serem processadas, o que implicará maior número de acessos a disco e correspondente aumento nos tempos de resposta.

A normalização pode ser vista como um processo através do qual as relações não satisfatórias são decompostas em um conjunto de outras relações por meio da operação de projeção. Essa decomposição terá de ser realizada de tal modo que não haja perda de informação, garantindo que o processo contrário continue a ser possível.

Existem seis formas normais e, em geral, se considera que um esquema de banco de dados que se encontre na terceira forma normal está em um bom nível e é considerado suficiente para a maioria das aplicações.

A terceira forma normal (3FN), habitualmente, é o objetivo da aplicação da normalização. As duas primeiras formas normais (1FN e 2FN) correspondem a passos intermediários para obter a 3FN.

A primeira, a segunda e a terceira formas normais e a forma normal de Boyce-Codd baseiam-se no conceito de dependência funcional entre atributos de uma relação.

A quarta forma normal baseia-se em dependência multivalorada e a quinta forma normal baseia-se em dependência de junção.



### Lembrete

Para realizar a normalização de dados, é primordial que seja definido para a tabela o objeto da normalização, um campo de chave primária para sua estrutura de dados, o qual permite identificar os demais campos da estrutura.

#### 3.7.1 Dependências funcionais

O conceito de dependência funcional é a base das primeiras três formas normais e da forma normal de Boyce-Codd, e está ligado à possibilidade de existir, em uma relação, um subconjunto de atributos com uma relação de dependência de outro subconjunto de atributos nessa mesma relação.

Se R for uma relação de X e Y subconjuntos arbitrários do conjunto de atributos de R, então se diz que Y é funcionalmente dependente de X ( $X \rightarrow Y$ ) se e somente se para todos os possíveis valores de R cada valor de X tiver associado um valor bem preciso de Y.

Dizer que Y é funcionalmente dependente de X é o mesmo que dizer que os valores do atributo ou conjunto de atributos X identificam o atributo ou conjunto de atributos Y, isto é, que existe pelo menos um valor no atributo Y para cada valor do atributo X.

Se o valor do atributo X determina o valor do atributo Y, isto é, se  $X \rightarrow Y$ , então se diz que o atributo X é determinante e que o atributo é funcionalmente dependente de X. Um exemplo simples pode ser observado em uma fatura:

**Tabela 5 – Tabela Fatura**

Produto	Preco	Quant	Total
Carrinho de feira	180,00	8	1.080,00
Bolacha água e sal	3,50	10	35,00
Refrigerante	4,99	5	24,95
Boneca Ref#004	89,90	10	890,00

Adaptada de: Rob e Coronel (2011).

Como a coluna Total de cada linha é sempre igual à soma da coluna Preco com a coluna Quant, podemos dizer que a coluna Total é funcionalmente dependente das colunas Preco e Quant, isto é:

$\{\text{Preco, Quant}\} \rightarrow \text{Total}$

Se X for chave primária, então todos os demais atributos são funcionalmente dependentes de X.

Lembre-se de que uma chave é um campo ou conjunto de campos que identifica univocamente cada linha de uma tabela. Portanto, podemos afirmar que, usando o conceito de dependências funcionais:

Chave  $\rightarrow$  Atributos não chave

Se descobirmos que a relação R satisfaz a dependência funcional  $A \rightarrow B$  e A não é chave, então R terá algum grau de redundância.

Estamos diante de uma dependência funcional completa quando a chave primária é composta por mais do que um atributo e qualquer dos outros atributos tem de ser identificado pela totalidade da chave, e não apenas por parte dela.

### 3.7.2 Dependências triviais e não triviais

Um bom ponto de partida para a remoção do conjunto de dependências funcionais consiste na eliminação das denominadas dependências triviais.

Uma dependência funcional existe somente se a parte direita da sua expressão for um subconjunto da sua parte esquerda. Exemplo:

{Aluno, Curso, Disciplina}  $\rightarrow$  Aluno

Para o processo de normalização, interessa particularmente o estudo das dependências funcionais não triviais.

### 3.7.3 Regras de inferência e axiomas de Armstrong

Segundo Elmasri e Navathe (2011), em um esquema de relação R, é possível definir o conjunto F das dependências funcionais imediatas ou óbvias. No entanto, existem outras dependências funcionais implícitas em F.

Chama-se fecho de F o conjunto de todas as dependências funcionais inferidas de F (denominada  $F^+$ ).

Para determinar  $F^+$ , usam-se os axiomas de Armstrong e as regras de inferência que eles dão origem.

#### Quadro 2 – Axiomas de Armstrong

Axiomas de Armstrong	
Reflexividade	Se B é subconjunto de A, então $A \rightarrow B$
Aumento2	Se $A \rightarrow B$ , então $AC \rightarrow BC$
Transitividade	Se $A \rightarrow B$ e $B \rightarrow C$ , então $A \rightarrow C$

Adaptado de: Ramakrishnan e Gehrke (2011).

#### Quadro 3 – Regras de inferência

Regras de inferência (derivada dos axiomas)	
Decomposição	Se $A \rightarrow BC$ , então $A \rightarrow B$ e $A \rightarrow C$
União	Se $A \rightarrow B$ e $A \rightarrow C$ , então $A \rightarrow BC$
Pseudotransitividade	Se $A \rightarrow B$ e $B \rightarrow K$ , então $AK \rightarrow C$
Composição	Se $A \rightarrow B$ e $C \rightarrow D$ , então $AB \rightarrow CD$

Adaptado de: Ramakrishnan e Gehrke (2011).

## 3.8 Formas normais

Normalização é uma técnica idealizada para aumentar a confiabilidade na extração e atualização de dados num banco de dados. Idealizado em 1972 por Edgar F. Codd, é um processo onde é criado uma



relação de séries de testes para certificar-se de que os dados manipulados de certa maneira satisfaçam uma certa forma normal. Para entendermos melhor a aplicação das formas normais, vamos considerar os seguintes exemplos:

Uma fatura é emitida a um cliente com todos os produtos adquiridos, incluindo a quantidade e o valor total dos produtos adquiridos em um determinado estabelecimento.

Fatura = n. Fatura, CodCliente, NomeCliente, EnderecoCliente, EmailCli, Produtos, TotalFatura, DataCompra, FormaPagamento.

Diante das informações do objeto Fatura, vamos abordar quatro diretrizes informais que podem ser usadas como medidas para determinar a qualidade de projeto do esquema da relação:

- Garantir que a semântica dos atributos seja clara no esquema.
- Reduzir a informação redundante nas tuplas.
- Reduzir os valores NULL nas tuplas.
- Reprovar a possibilidade de gerar tuplas falsas.

Sempre que agrupamos atributos para formar um esquema de relação, consideramos que aqueles atributos pertencentes a uma relação têm certo significado no mundo real e uma interpretação apropriada associada a eles. A semântica de uma relação refere-se a seu significado resultante da interpretação dos valores de atributo em uma tupla. Se o projeto conceitual for feito cuidadosamente e o procedimento de mapeamento for seguido de maneira sistemática, o projeto do esquema relacional deverá ter um significado claro.

Em geral, quanto mais fácil for explicar a semântica da relação, melhor será o projeto do esquema de relação. Para ilustrar isso, considere o quadro a seguir, que mostra uma versão simplificada do esquema de banco de dados relacional EMPRESA e apresenta um exemplo de estados de relação preenchidos desse esquema. O significado do esquema de relação FUNCIONARIO é muito simples: cada tupla representa um funcionário, com valores para o nome do funcionário (Fnome), número do Cadastro de Pessoa Física (FCPF), data de nascimento (FDataNasc), endereço (FEndereco) e o número do departamento para o qual o funcionário trabalha (CodDepto). O atributo CodDepto é uma chave estrangeira que representa um relacionamento implícito entre FUNCIONARIO e DEPARTAMENTO. A semântica dos esquemas DEPARTAMENTO e PROJETO é muito simples: cada tupla DEPARTAMENTO representa uma entidade de departamento e cada tupla PROJETO representa uma entidade de projeto. O atributo CPFGerente de DEPARTAMENTO relaciona um departamento ao funcionário que é seu gerente, enquanto PNumero de PROJETO relaciona um projeto a seu departamento de controle; ambos são atributos de chave estrangeira. A facilidade com que o significado dos atributos de uma relação pode ser explicado é uma medida informal de quão bem a relação está projetada.

Quadro 4 – Esquema de banco de dados relacional EMPRESA simplificado

FUNCIONARIO					
FNome	FCPF	FDataNasc		FEndereco	CodDepto
DEPARTAMENTO					
CodDepto		DNome		CPFGerente	
DEP_LOCALIZACAO					
DNumero			Local		
PROJETO					
PNome		PNumero	PLocal	CodDepto	
TRABALHA_EM					
FCPF		PNumero		Horas	

Adaptado de: Elmasri e Navathe (2011).

Projete um esquema de relação de modo que seja fácil explicar o seu significado. Não combine atributos de vários tipos de entidade e de relacionamento em uma única relação. Intuitivamente, se um esquema de relação corresponde a um tipo de entidade ou um tipo de relacionamento, é simples interpretar e explicar o seu significado. Caso contrário, se a relação corresponder a uma mistura de várias entidades e relacionamentos, haverá ambiguidades semânticas e a relação não poderá ser explicada com facilidade.

### Exemplos de violação

Os esquemas de relação das figuras a seguir têm semântica clara. Uma tupla no esquema de relação FUNC\_DEP na figura 34 (a) representa um único funcionário, mas inclui informações adicionais — a saber, o nome (DNome) do departamento para o qual o funcionário trabalha e o número do Cadastro de Pessoa Física (CPFGerente) do gerente de departamento. Para a relação FUNC\_PROJ da figura 34 (b), cada tupla relaciona um funcionário a um projeto, mas também inclui o nome do funcionário (FNome), nome do projeto (Projnome) e local do projeto (Projlocal). Embora não haja nada logicamente errado com essas duas relações, elas violam a Diretriz 1 ao misturar atributos de entidades distintas do mundo real: FUNC\_DEP mistura atributos dos funcionários e departamentos, e FUNC\_PROJ mistura atributos de funcionários, projetos e o relacionamento TRABALHA\_EM. Logo, elas se saem mal contra a medida de qualidade de projeto citado. Elas podem ser usadas como visões, mas causam problemas quando utilizadas como relações da base.

### FUNCIONARIO

FNome	FCPF	FDataNasc	FEndereco	CodDepto
Ana Flor de Carvalho	12345678910	10-09-1998	Rua das Dores, 174, São Paulo	1
Felipe Almeida Neto	78910121314	08-04-2000	Rua Joia Norte, 32, Osasco	3
Alex Moura Santos	234567812310	18-09-1978	Av. Brasil, 900, Bahia	4
Raquel Souza Brito	88844487698	11-12-1970	Rua Paulista, 10, Brás	4
Carla Lima Toledo	45678990712	02-09-1989	Rua Treze, 90, Santos	5

## DEPARTAMENTO

DNome	CodDepto	CPFGerente
Administrativo	1	33344455578
Contabilidade	2	34565434512
Financeiro	5	89876543211

## LOCALIZACAO\_DEP

DNumero	Local
1	São Paulo
2	Barueri
3	Santos
4	Brasília
5	Itu

## TRABALHA\_EM

FCPF	PNUMERO	HORAS
12345678910	1	32.5
78910121314	2	7.8
88844487698	3	40.0
33344555587	1	40.0

## PROJETO

PNome	PNumero	PLocal	CodDepto
ProdutoX	1	Santo André	5
ProdutoY	2	Itu	5
ProdutoZ	3	São Paulo	5
Informatização	10	Mauá	4
Reorganização	20	São Paulo	1

Figura 33 – Exemplo de banco de dados com esquema relacional

### (a) FUNC\_DEP

FNome	FCPF	FDataNasc	FEndereco	DNome	CPFGerente

## (b) FUNC\_PROJ

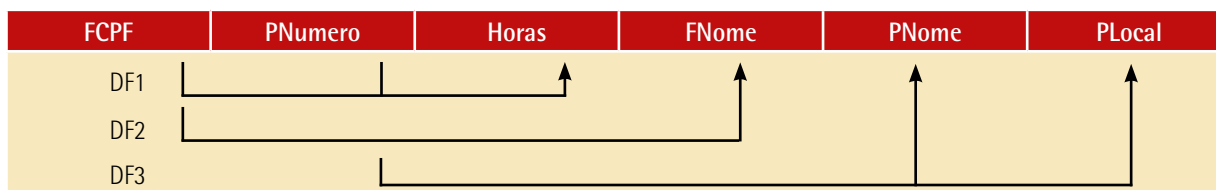


Figura 34 – Dois esquemas de relação de anomalias de atualização (a) FUNC\_DEP e (b) FUNC\_PROJ

O armazenamento de junções naturais de relações da base leva a um problema adicional conhecido como **anomalias de atualização**. Elas podem ser classificadas em anomalias de inserção, anomalias de exclusão e anomalias de modificação.

**Anomalias de inserção** podem ser diferenciadas em dois tipos, ilustrados pelos seguintes exemplos baseados na relação FUNC\_DEP:

Para inserir uma nova tupla de funcionário em FUNC\_DEP, temos de incluir ou os valores de atributo do departamento para o qual o funcionário trabalha ou NULL (se o funcionário ainda não trabalha para nenhum departamento).

Por exemplo, para inserir uma nova tupla para um funcionário que trabalha no departamento 5, temos de inserir todos os valores de atributo do departamento 5 corretamente, de modo que eles sejam coerentes com os valores correspondentes para o departamento 5 em outras tuplas de FUNC\_DEP. No projeto da figura 34 não temos de nos preocupar com esse problema de coerência, pois entramos apenas com o número do departamento na tupla do funcionário. Todos os outros valores de atributo do departamento 5 são registrados apenas uma vez no banco de dados, como uma única tupla na relação DEPARTAMENTO.

É difícil inserir um novo departamento que ainda não tenha funcionários na relação FUNC\_DEP. A única maneira de fazer isso é colocar valores NULL nos atributos para funcionário. Isso viola a integridade de entidade para FUNC\_DEP, porque CPF é sua chave primária. Além do mais, quando o primeiro funcionário é atribuído a esse departamento, não precisamos mais dessa tupla com valores NULL. Esse problema não ocorre no projeto da figura 33, visto que um departamento é inserido na relação DEPARTAMENTO independentemente de haver ou não funcionários trabalhando para ele, e sempre que um funcionário é atribuído a esse departamento, uma tupla correspondente é inserida em FUNCIONARIO.

Agora, vamos apresentar as três primeiras formas normais: 1FN, 2FN e 3FN. Elas foram propostas por Codd como uma sequência para conseguir o estado desejável de relações 3FN ao prosseguir pelos estados intermediários de 1FN e 2FN, se necessário. Conforme veremos, 2FN e 3FN atacam diferentes problemas. Contudo, por motivos históricos, é comum segui-los nessa sequência. Logo, por definição, uma relação 3FN já satisfaz a 2FN.

## 3.8.1 Primeira forma normal

A **primeira forma normal (1FN)** agora é considerada parte da definição formal de uma relação no modelo relacional básico (plano). Historicamente, ela foi definida para reprovar atributos multivalorados, atributos compostos e suas combinações. Ela afirma que o domínio de um atributo deve incluir apenas valores atômicos (simples, indivisíveis) e que o valor de qualquer atributo em uma tupla deve ser um único valor do domínio desse atributo. Logo, 1FN reprova ter um conjunto de valores, uma tupla de valores ou uma combinação de ambos como um valor de atributo para uma única tupla. Em outras palavras, a 1FN reprova relações dentro de relações ou relações como valores de atributo dentro de tuplas. Os únicos valores de atributo permitidos pela 1FN são os **valores atômicos** (ou indivisíveis).

Considere o esquema de relação DEPARTAMENTO da figura a seguir, cuja chave primária é CodDeppto, e suponha que a estendamos ao incluir o atributo Dlocal, conforme mostra a figura 35 (a). Vamos supor também que cada departamento pode ter certo número de locais. O esquema DEPARTAMENTO e um exemplo de estado de relação são mostrados na figura 36. Como podemos ver, esta não está em 1FN, porque Dlocal não é um atributo atômico, conforme ilustrado pela primeira tupla na figura 35 (b). Existem duas maneiras possíveis para examinar o atributo Dlocal:

- O domínio de Dlocal contém valores atômicos, mas algumas tuplas podem ter um conjunto desses valores. Nesse caso, Dlocal não é funcionalmente dependente da chave primária CodDeppto.

### (a) DEPARTAMENTO



### (b) DEPARTAMENTO

DNome	CodDeppto	CPFGerente	Dlocal
Pesquisa	5	3333444455588	Santo André, Itu, São Paulo
Administração	4	22233344401	Mauá
Matriz	1	88899945676	São Paulo

### (c) DEPARTAMENTO

DNome	CodDeppto	CPFGerente	Dlocal
Pesquisa	5	3333444455588	São Paulo
Pesquisa	5	3333444455588	Itu
Pesquisa	5	3333444455588	Santo André
Administração	4	98765432168	Mauá
Matriz	1	88866555576	São Paulo

Figura 35 – Normalização na 1FN: (a) um esquema de relação que não está em 1FN; (b) exemplo de estado da relação DEPARTAMENTO; (c) versão 1FN da mesma relação com redundância

O domínio de Dlocal contém conjuntos de valores e, portanto, não é atômico. Nesse caso, CodDepto  $\rightarrow$  Dlocal, pois cada conjunto é considerado um único membro do domínio de atributo.

De qualquer forma, a relação DEPARTAMENTO da figura anterior não está na 1FN; de fato, ela nem sequer se qualifica como uma relação. Existem três técnicas principais para conseguir a primeira forma normal para tal relação:

1) Remover o atributo Dlocal que viola a 1FN e colocá-lo em uma relação separada LOCALIZACAO\_DEP, junto com a chave primária CodDepto de DEPARTAMENTO. A chave primária dessa relação é a combinação {CodDepto, Dlocal}, como mostra a figura 36. Existe uma tupla distinta em LOCALIZACAO\_DEP para cada local de um departamento. Isso decompõe a relação não 1FN em duas relações 1FN.

2) Expandir a chave de modo que haverá uma tupla separada na relação original DEPARTAMENTO para cada local de um DEPARTAMENTO, como mostra a figura 36 (c). Nesse caso, a chave primária torna-se a combinação {CodDepto, Dlocal}. Essa solução tem a desvantagem de introduzir a redundância na relação.

3) Se o número máximo de valores for conhecido para o atributo — por exemplo, se for conhecido que no máximo três locais poderão existir para um departamento —, substituir o atributo Dlocal pelos três atributos atômicos: Dlocal1, Dlocal2 e Dlocal3. Essa solução tem a desvantagem de introduzir valores NULL se a maioria dos departamentos tiver menos de três locais. Ela ainda introduz uma falsa semântica sobre a ordenação entre os valores de local, que não era intencionada originalmente. A consulta sobre esse atributo torna-se mais difícil. Por exemplo, considere como você escreveria a consulta: Listar os departamentos que têm "Santo André" como um de seus locais nesse projeto.

Das três soluções anteriores, a primeira geralmente é considerada a melhor, pois não sofre de redundância e é completamente genérica, não tendo limite imposto sobre o número máximo de valores. De fato, se escolhermos a segunda solução, ela será decomposta ainda mais durante as etapas de normalização subsequentes para a primeira solução.

A primeira forma normal também desaprova atributos multivalorados que por si sós sejam compostos. Estes são chamados de **relações aninhadas**, pois cada tupla pode ter uma relação dentro dela. A figura a seguir mostra como a relação FUNC\_PROJ poderia aparecer se o aninhamento for permitido. Cada tupla representa uma entidade de funcionário, e a relação PROJS (PNumero, Horas) dentro de cada tupla representa os projetos do funcionário e o número de horas por semana que ele trabalha em cada projeto. O esquema dessa relação FUNC\_PROJ pode ser representado da seguinte forma:

FUNC\_PROJ(FCPF, FNome, {PROJS(PNumero,Horas)})

### (a) FUNC\_PROJ

FCPF	FNome	PNumero	Horas
------	-------	---------	-------

### (b) FUNC\_PROJ

FCPF	FNome	PNumero	Horas
12345678966	João Bispo Santos	1	32.5
		2	7.5
6688444476	Ronaldo Kennedy	3	40
33344555587	Fernando Tavares Junior	2	10
		3	10
		10	10
		20	10
99988777767	Zelaya, Alice J.	30	30
		10	10
98798798733	Pereira, André V.	4	35

### (c) FUNC\_PROJ1

FCPF	FNome
------	-------

### (d) FUNC\_PROJ2

FCPF	PNumero	Horas
------	---------	-------

Figura 36 – Normalizando relações aninhadas para a 1FN: (a) esquema da relação FUNC\_PROJ com um atributo de relação aninhada PROJS; (b) exemplo de extensão da relação FUNC\_PROJ mostrando relações aninhadas dentro de cada tupla; (c) decomposição de FUNC\_PROJ nas relações FUNC\_PROJ1 e FUNC\_PROJ2 pela propagação da chave primária

O conjunto de chaves { } identifica o atributo PROJS como multivalorado, enquanto os atributos componentes que formam o PROJS são listados entre parênteses ( ). O interessante é que as tendências recentes para dar suporte a objetos complexos e dados XML tentam permitir e formalizar as relações aninhadas nos sistemas de bancos de dados relacionais, que eram reprovadas inicialmente pela 1FN.

Observe que FCPF é a chave primária da relação FUNC\_PROJ nos elementos (a) e (b) da figura anterior, enquanto PNumero é a chave **parcial** da relação aninhada; ou seja, dentro de cada tupla, a relação aninhada precisa ter valores únicos de PNumero. Para normalizar isso para a 1FN, removemos os atributos da relação aninhada para uma nova relação e propagamos a chave primária para ela. A chave primária da nova relação combinará a parcial com a chave primária da relação original. A decomposição e a propagação da chave primária resultam nos esquemas FUNC\_PROJ1 e FUNC\_PROJ2, como vemos no elemento (c) da figura mostrada anteriormente.

Esse procedimento pode ser aplicado recursivamente a uma relação com aninhamento em nível múltiplo para **desaninhar** a relação para um conjunto de relações 1FN. Isso é útil na conversão de um esquema de relação não normalizado com muitos níveis de aninhamento em relações 1FN. A existência de mais de um atributo multivalorado em uma relação deve ser tratada com cuidado. Como um exemplo, considere a seguinte relação não 1FN:

PESSOA (CPF, {Placa}, {Telefone})

Essa relação representa o fato de uma pessoa ter vários carros e vários telefones. Se a estratégia 2, citada anteriormente, for seguida, ela resulta em uma relação com todas as chaves:

PESSOA\_NA\_1FN (CPF, Placa, Telefone)

Para evitar a introdução de qualquer relacionamento estranho entre Placa e Telefone, todas as combinações de valores possíveis são representadas para cada FCPF, fazendo surgir a redundância. Isso leva aos problemas tratados pelas dependências multivaloradas e 4FN. O modo certo de lidar com os dois atributos multivalorados em PESSOA mostrados anteriormente é decompô-los em duas relações separadas, usando a estratégia 1 P1(CPF, Placa) e P2(CPF, Telefone).

### 3.8.2 Segunda forma normal

A **segunda forma normal (2FN)** é baseada no conceito de dependência funcional total. Uma dependência funcional  $X \rightarrow Y$  é uma dependência funcional total se a remoção de qualquer atributo A de X significar que a dependência não se mantém mais; ou seja, para qualquer atributo  $A \in X$ ,  $(X - \{A\})$  não determina Y funcionalmente. Uma dependência funcional  $X \rightarrow Y$  é uma dependência parcial se algum atributo  $A \in X$  puder ser removido de X e a dependência ainda se mantiver; ou seja, para algum  $A \in X$ ,  $(X - \{A\}) \rightarrow Y$ . Na figura 37,  $\{FCPF, PNumero\} \rightarrow Horas$  é uma dependência total (nem FCPF  $\rightarrow$  Horas nem PNumero  $\rightarrow$  Horas se mantêm). Contudo, a dependência  $\{FCPF, PNumero\} \rightarrow FNome$  é parcial porque FCPF  $\rightarrow$  FNome se mantém.

**Definição:** um esquema de relação R está em 2FN se cada atributo não principal A em R for total e funcionalmente dependente da chave primária de R.

O teste para a 2FN envolve testar as dependências funcionais cujos atributos do lado esquerdo fazem parte da chave primária. Se a chave primária tiver um único atributo, o teste não precisa ser aplicado. A relação FUNC\_PROJ na figura 37 está na 1FN, mas não está na 2FN. O atributo não principal FNome viola a 2FN por causa da DF2, assim como os atributos não principais PNome e PLocal, por causa da DF3.

As dependências funcionais DF2 e DF3 tornam FNome, PNome, PLocal parcialmente dependentes da chave primária  $\{FCPF, PNumero\}$  de FUNC\_PROJ, violando, assim, o teste da 2FN.

Se um esquema de relação não estiver na 2FN, ele pode ser segundo normalizado ou normalizado pela 2FN para uma série de relações 2FN em que os atributos não principais são associados com a parte da chave primária em que eles são total e funcionalmente dependentes. Portanto, as dependências



funcionais DF1, DF2 e DF3 da figura 37 levam à decomposição de FUNC\_PROJ nos três esquemas de relação FP1, FP2 e FP3 mostrados na figura 37 cada qual estando na 2FN.

### 3.8.3 Terceira forma normal

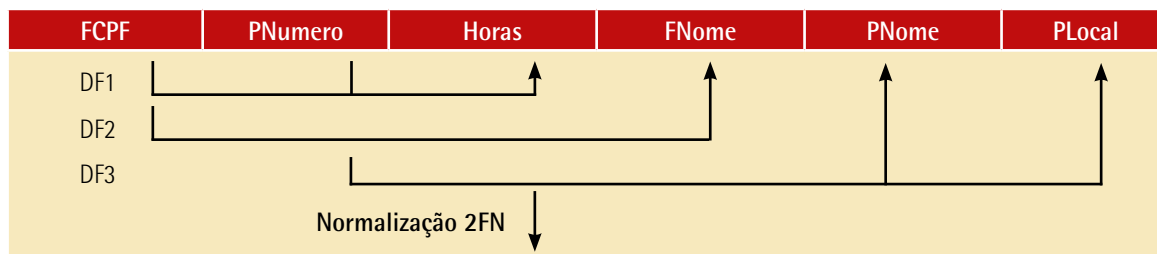
A **terceira forma normal (3FN)** é baseada no conceito de dependência transitiva. Uma dependência funcional  $X \rightarrow Y$  em um esquema de relação R é uma **dependência transitiva** se houver um conjunto de atributos Z em R que nem sejam uma chave candidata nem um subconjunto de qualquer chave de R, e tanto  $X \rightarrow Z$  quanto  $Z \rightarrow Y$  se mantiverem. A dependência  $FCPF \rightarrow CPFGerente$  é transitiva por meio de DNumero em FUNC\_DEP na figura 37, pois ambas as dependências  $FCPF \rightarrow DNumero$  e  $DNumero \rightarrow CPFGerente$  se mantêm e DNumero não é nem uma chave por si só nem um subconjunto da chave de FUNC\_DEP. Intuitivamente, podemos ver que a dependência de CPFGerente sobre DNumero é indesejável em FUNC\_DEP, pois DNumero não é uma chave de FUNC\_DEP.

**Definição:** de acordo com a definição original de Codd, um esquema de relação R está na 3FN se ele satisfizer a 2FN e nenhum atributo não principal de R for transitivamente dependente da chave primária.

O esquema de relação FUNC\_DEP da figura 37 (a) está na 2FN, pois não existe dependência parcial sobre uma chave. Porém, FUNC\_DEP não está na 3FN devido à dependência transitiva de CPFGerente (e também DNome) em CPF por meio de DNumero. Podemos normalizar FUNC\_DEP decompondo-o nos dois esquemas de relação 3FN DF1 e DF2 mostrados na figura 37 (b). Intuitivamente, vemos que DF1 e DF2 representam fatos de entidades independentes sobre funcionários e departamentos. Uma operação JUNÇÃO NATURAL sobre DF1 e DF2 recuperará a relação original FUNC\_DEP sem gerar tuplas falsas.

De maneira intuitiva, podemos ver que qualquer dependência funcional de que o lado esquerdo faz parte (é um subconjunto apropriado) da chave primária, ou qualquer dependência funcional de que o lado esquerdo é um atributo não chave, é uma DF problemática. A normalização 2FN e 3FN remove essas DFs problemáticas ao decompor a relação original em novas relações. Em relação ao processo de normalização, não é necessário remover as dependências parciais antes das dependências transitivas, porém, historicamente, a 3FN tem sido definida com a suposição de que uma relação é testada primeiro pela 2FN, antes de ser testada pela 3FN. A figura a seguir resume informalmente as três formas normais com base nas chaves primárias, os testes usados em cada uma e a solução ou normalização realizada para alcançar a forma normal.

## (a) FUNC\_PROJ



## (b) FUNC\_DEP

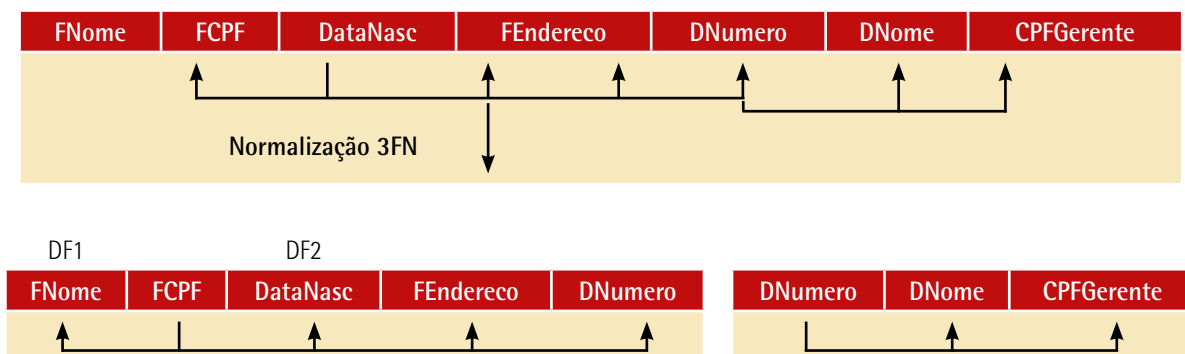


Figura 37 – Normalizando para 2FN e 3FN: (a) normalizando FUNC\_PROJ em relações 2FN; (b) normalizando FUNC\_DEP em relações 3FN

### Quadro 5 – Resumo das formas normais baseadas em chaves primárias e a normalização correspondente

Forma normal	Teste	Solução (normalização)
Primeira (1FN)	Relação não deve ter atributos multivalorados ou relações aninhadas.	Formar novas relações para cada atributo multivalorado ou relação aninhada.
Segunda (2FN)	Para relações em que a chave primária contém múltiplos atributos, nenhum atributo não chave deverá ser funcionalmente dependente de uma parte da chave primária.	Decompor e montar uma nova relação para cada chave parcial com seu(s) atributo(s) dependente(s). Certificar-se de manter uma relação com a chave primária original e quaisquer atributos que sejam total e funcionalmente dependentes dela.
Terceira (3FN)	A relação não deve ter um atributo não chave determinado funcionalmente por outro atributo não chave (ou por um conjunto de atributos não chave). Ou seja, não deve haver dependência transitiva de um atributo não chave sobre a chave primária.	Decompor e montar uma relação que inclua o(s) atributo(s) não chave que determina(m) funcionalmente outro(s) atributo(s) não chave.

Fonte: Elmasri e Navathe (2011, p. 353).

### Definição de ferramentas CASE (*computer-aided software engineering*)

Todo SGBD necessita utilizar ferramentas CASE e no mercado existem diversas. Elas têm como propósito auxiliar na modelagem de banco de dados relacionais, o que permite criar dados de forma gráfica (desenhos de tabelas e relacionamentos) e salvar os modelos criados em arquivos binários que, posteriormente, poderiam ser usados no SGBD, em uma parte da fase de análise e projeto para a criação de um *software* de aplicação de negócios eficiente (ver figura a seguir), mas normalmente é a parte mais desafiadora e mais crítica para se realizar.

A exploração de ferramentas clássicas é muito comum para criar projetos de banco de dados eficientes e eficazes, incluindo a modelagem ER e a transformação dos modelos ER em construtores por meio de regras de transformação. Permite ainda verificar a normalização, formas normais e desnormalização, além das topologias específicas usadas no *warehousing*, como o *star schema*.

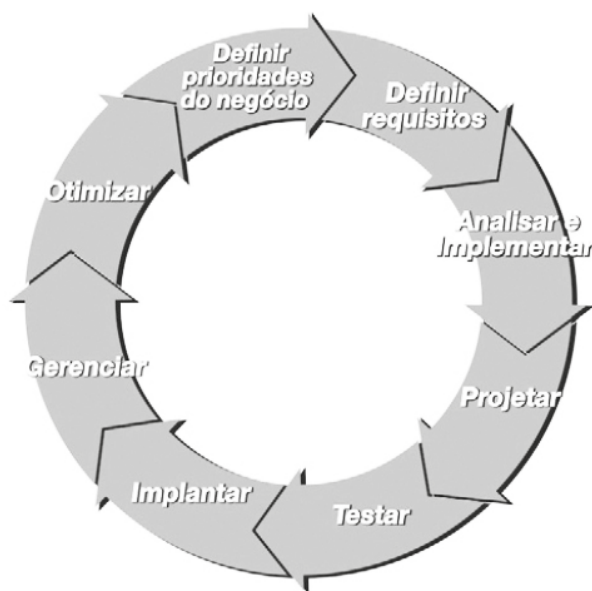


Figura 38 – Ciclo de vida do sistema de negócios

Existem ferramentas disponíveis comercialmente para simplificar esses processos de projeto. Essas ferramentas de engenharia de sistemas auxiliadas por computador, ou CASE (*computer-aided system engineering*), oferecem funções que auxiliam no projeto do sistema. As ferramentas CASE são bastante usadas em diversos setores e domínios, como projeto de circuitos, manufatura e arquitetura.

O projeto lógico de banco de dados é outra área na qual as ferramentas CASE têm provado a sua eficácia. Empresas como MySQL, Microsoft, Oracle têm fornecido tecnologias e recursos avançados para o desenvolvimento de projetos lógicos de banco de dados e sua evolução para o projeto físico.

Vale ressaltar que, caso você escolha usar uma ferramenta diferente desse exemplo, algumas construções podem não ter o mesmo resultado final. Uma questão deve ficar clara: é impossível registrar informações sobre produtos de *software* sem emitir alguma visão e opinião pessoal sobre a

ferramenta, quais são as capacidades desses produtos com o mínimo de parcialidade ou crítica. Além disso, é impossível descrever os recursos desses produtos em grandes níveis de detalhes em um capítulo.

### 4 INTRODUÇÃO ÀS FERRAMENTAS CASE

Alguns dos produtos mais populares e poderosos disponíveis no mercado para ajudar no projeto lógico do banco de dados são: Rational Data Architect, da IBM, AllFusion ERwin Data Modeler, da Computer Associates, MySQL Workbench, da Oracle, o qual será utilizado como ferramenta para criação de exemplo do modelo lógico de banco de dados, conforme podemos ver na figura a seguir.

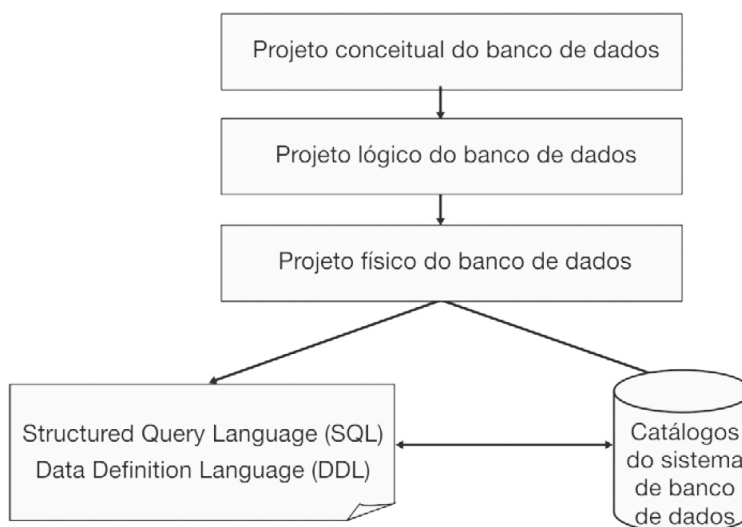


Figura 39 – Processo de projeto de banco de dados

O desenvolvimento de um projeto lógico de banco de dados necessita de um projeto de relacionamentos da ER, o qual possui vários tipos de definições de entidade e modelagem de relacionamentos (dados não relacionados, ou com algumas relações um-para-muitos e muitos-para-muitos), para que esses relacionamentos combinem e para que haja normalização em padrões de esquemas conhecidos como formas normais (por exemplo, 3FN). Um projeto eficaz necessita de definições claras de chaves como as chaves primárias, estrangeiras e candidatas dentro dos relacionamentos. O projeto lógico deve ser criado de forma eficaz e terá um impacto profundo sobre o desempenho do sistema, o qual, além de facilitar no uso do sistema de banco de dados, irá ajudar a executar as manutenções necessárias ao negócio.

O MySQL Workbench pode ser usado para executar consultas SQL, administrar o sistema e modelar, além de criar e manter a base de dados através de um ambiente integrado. Ele será usado como ferramenta de interface de modelagem de base de dados, para definir as entidades da base de dados, seus atributos e relacionamentos.

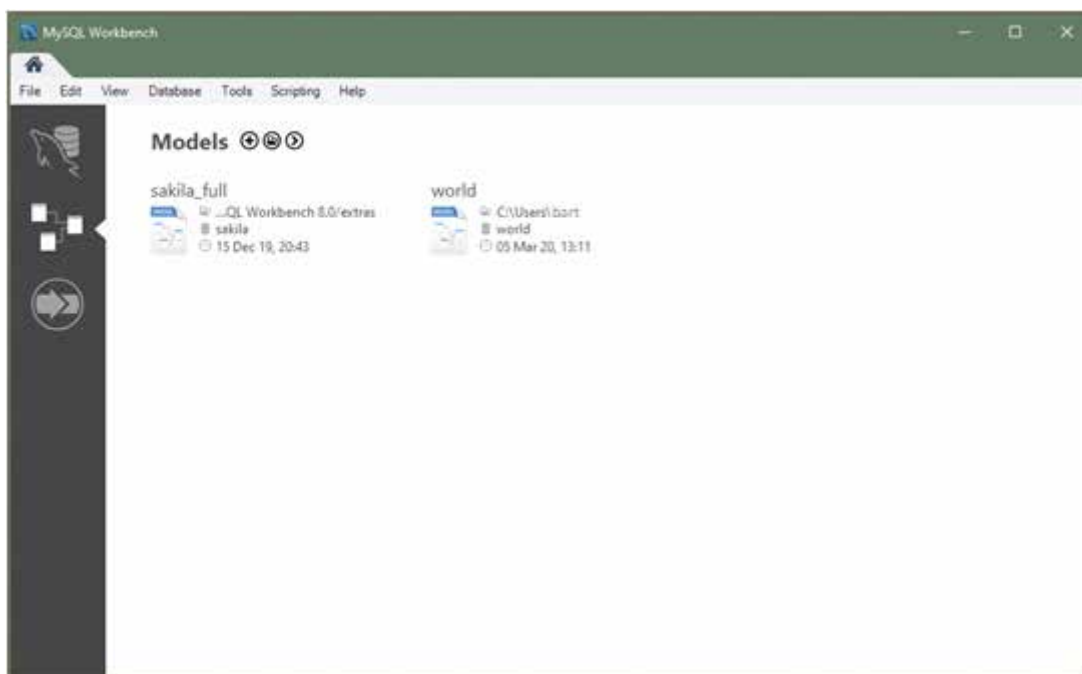


Figura 40 – Modelos de dados na tela inicial

### 4.1 Criação de modelos lógicos

Antes de examinar a sintaxe de SQL para criação e definição de tabelas e outros elementos, devemos primeiro criar o modelo de banco de dados e as tabelas que formarão a base de muitos exemplos de SQL que serão abordados. No exemplo a seguir, um banco de dados simples é composto das seguintes tabelas: Categoria, Venda, Funcionario, Produto, Fornecedor, Itens\_Vendas, que serão usadas para ilustrar os comandos SQL.



#### Saiba mais

AllFusion ERwin Data Modeler é uma ferramenta de modelagem usada para encontrar, visualizar, projetar e implantar dados de um SGBD. Permite documentar com consistência, clareza e reutilização de artefatos em integração de dados em grande escala, gerenciamento de dados mestre, gerenciamento de metadados, Big Data, *business intelligence* e iniciativas de análise – tudo ao mesmo tempo, apoiando os esforços de governança e inteligência de dados.

Acesse através do *link* a seguir:

Disponível em: <https://bit.ly/3dXa6iw>. Acesso em: 12 abr. 2021.

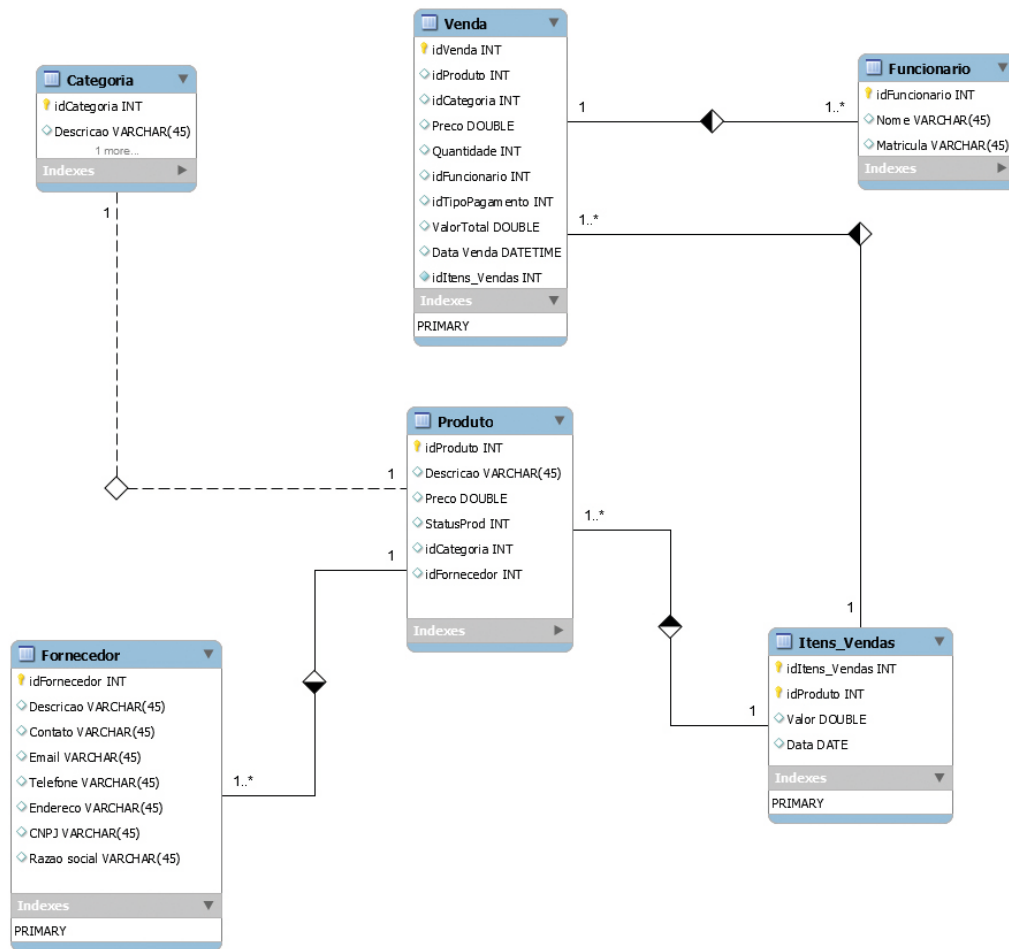


Figura 41 – Modelos de dados referentes às vendas de produtos

- Uma Venda contém Itens\_Venda. Cada Venda está associada a uma única venda que poderá possuir vários produtos de uma categoria. Um produto poderá ter vários fornecedores.
- Um fornecedor pode fornecer vários produtos. Alguns fornecedores podem ainda não fornecer produtos. Por exemplo, quando se tem uma lista que inclui potenciais fornecedores.
- Se um produto é fornecido por um fornecedor, esse produto é suprido por um único fornecedor.
- Os funcionários serão responsáveis por efetuar as vendas e cada venda deverá estar associada a apenas um vendedor.

É importante ressaltar que com exceção da criação de banco de dados, a maioria dos SGBDs utiliza uma SQL que desvia pouco do SQL padrão ANSI. Por exemplo, grande parte desses sistemas exige que cada comando de SQL termine com ponto e vírgula. No entanto, algumas implementações não utilizam esse sinal. As diferenças importantes de sintaxe entre implementação serão destacadas nas caixas de notas.

A linguagem SQL padrão frequentemente é utilizada para representar as mais diversas funções relacionadas a banco de dados e programação. Ela suporta uma série de funções específicas que serão exploradas mais à frente.

**Linguagem de manipulação de dados** (DML – *data manipulation language*) é um subconjunto da SQL que permite que os usuários formulem consultas e insiram, excluam e modifiquem tuplas. Os comandos SQL desse subconjunto são:

- **INSERT:** utilizado para inserir registros (tuplas) em uma tabela.

Exemplo: `INSERT into CLIENTE(ID, NOME) values(1,'Rafaela');`

- **UPDATE:** utilizado para alterar valores de uma ou mais linhas (tuplas) de uma tabela.

Exemplo: `UPDATE CLIENTE set NOME = 'Roberta' WHERE ID = 1;`

- **DELETE:** utilizado para excluir um ou mais registros (tupla) de uma tabela.

Exemplo: `DELETE FROM CLIENTE WHERE ID = 1;`

- **SELECT:** principal comando da SQL, o comando SELECT é utilizado para efetuar consultas no banco de dados.

Exemplo: `SELECT ID, NOME FROM CLIENTE;`

**Linguagem de definição de dados** (DDL – *data definition language*) é um subconjunto da SQL que suporta a criação, exclusão e modificação das definições das tabelas e visões. As restrições de integridade podem ser definidas nas tabelas, tanto quando a tabela é criada como posteriormente. Fornece ainda a dinâmica para criação e exclusão de índices. Os comandos SQL desse subconjunto são:

- **CREATE:** utilizado para criar objetos no banco de dados.

Exemplo (criar uma tabela): `CREATE TABLE CLIENTE (ID INT PRIMARY KEY, NOME VARCHAR(50));`

- **ALTER:** utilizado para alterar a estrutura de um objeto.

Exemplo (adicionar uma coluna em uma tabela existente): `ALTER TABLE CLIENTE ADD SEXO CHAR (1);`

- **DROP:** utilizado para remover um objeto do banco de dados.

Exemplo (remover uma tabela): `DROP TABLE CLIENTE;`

**Linguagem de controle de dados** (DCL – *data control language*) é um subconjunto da SQL que suporta controlar o acesso aos dados, basicamente, com dois comandos que permitem ou bloqueiam o acesso de usuários a dados. Vejamos os comandos:

- **GRANT:** autoriza um usuário a executar alguma operação.

Exemplo (dar permissão de consulta na tabela cliente para o usuário diogo): GRANT SELECT ON cliente TO diogo;

- **REVOKE:** restringe ou remove a permissão de um usuário executar alguma operação.

Exemplo (não permitir que o usuário diogo crie tabelas no banco de dados): REVOKE CREATE TABLE FROM diogo;

**Linguagem de controle de transações** (DTL – *data transaction language*) é um subconjunto da SQL que fornece mecanismos para controlar transações no banco de dados. São três comandos: iniciar uma transação (BEGIN TRANSACTION), efetivar as alterações no banco de dados (COMMIT TRANSACTION) e cancelar as alterações (ROLLBACK TRANSACTION).

- **Gatilhos e restrições de integridade avançadas:** o novo padrão SQL:1999 inclui suporte a gatilhos (*triggers*), que são ações executadas pelo SGBD sempre que alterações no banco de dados satisfazem condições especificadas no gatilho.
- **SQL embutida e dinâmica:** os recursos de SQL embutida permitem que o código SQL seja chamado por meio de uma linguagem hospedeira como C ou COBOL. Os recursos de SQL dinâmica permitem que uma consulta seja construída (e executada) em tempo de execução.
- **Execução cliente-servidor e acesso a banco de dados remoto:** esses comandos controlam como um programa de aplicativo cliente pode se conectar a um servidor de banco de dados SQL, ou acessar dados de um banco de dados através de uma rede.
- **Gerenciamento de transação:** diversos comandos permitem que um usuário controle explicitamente os aspectos da execução de uma transação.
- **Segurança:** a SQL provê mecanismos para controlar o acesso dos usuários aos objetos de dados, tais como tabelas e visões.

### 4.2 Geração de scripts

Quando se utiliza um SGBD empresarial, antes de começar a criar tabelas, é necessário ser autenticado por esse sistema. A autenticação é o processo por meio do qual um SGBD garante que somente usuários registrados possam acessar o banco de dados. Para ser autenticado, deve-se fazer *logon* no SGBD utilizando uma ID de usuário e uma senha criada pelo administrador do banco de dados. Em um SGBDR empresarial, toda ID de usuário está associada a um esquema de banco de dados. Nesse



contexto, usando o Transact-SQL, daremos sequência ao ambiente de programação para inicializar todos os procedimentos. Nesse primeiro contato com a ferramenta, vamos inicializar o mecanismo de banco de dados. Veja o passo a passo a seguir:

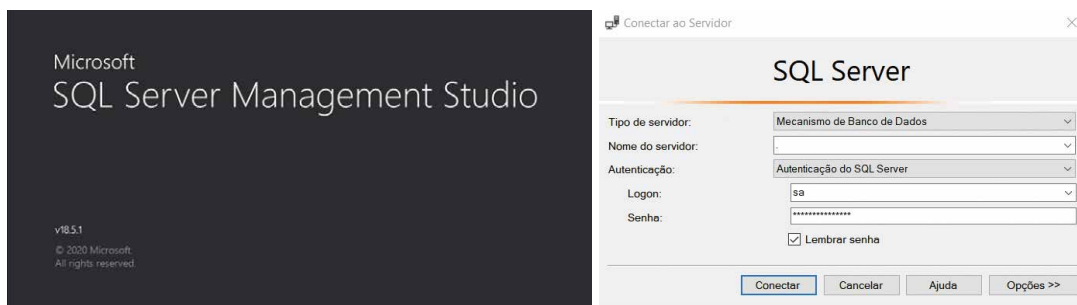


Figura 42 – Inicializando o mecanismo de banco de dados para, em seguida, adicionar as credenciais solicitadas

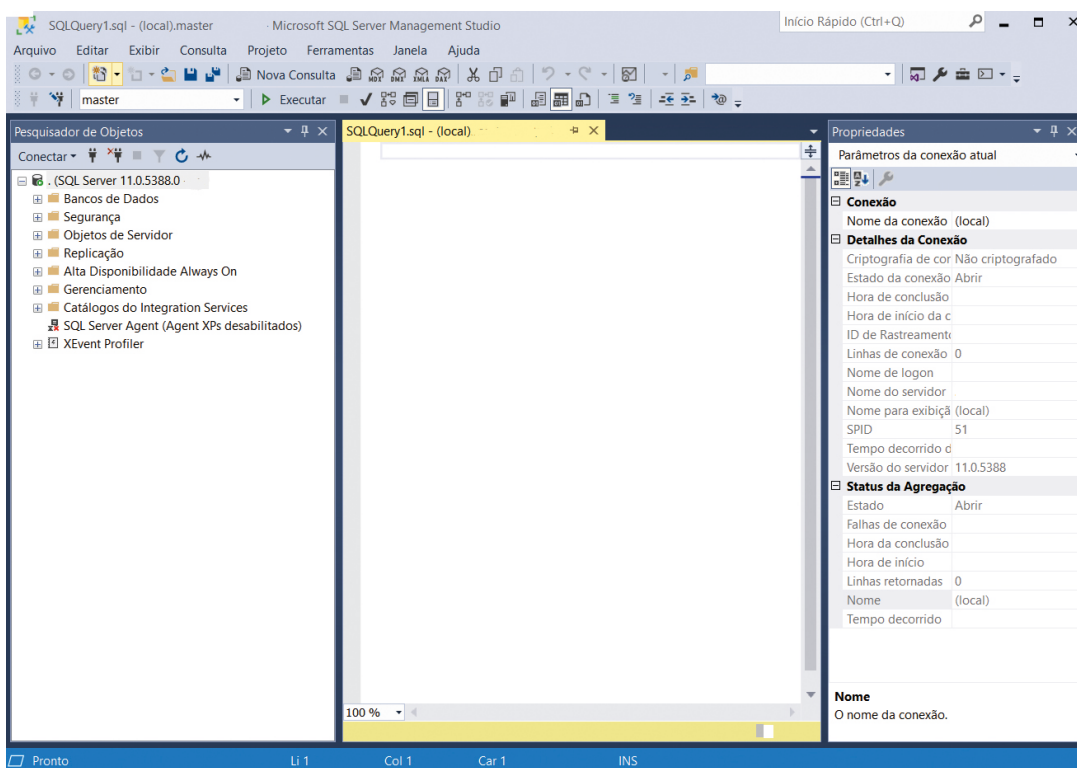


Figura 43 – Tela inicial

### 4.2.1 Criando um banco de dados

Na área SQLQuery1.sql, copie e cole o exemplo a seguir na janela de consulta e clique em **executar**. Esse exemplo cria o banco de dados chamado MODELOBD. Como a palavra-chave PRIMARY não é usada, o primeiro arquivo (MODELOBD\_dat) torna-se o arquivo primário. Como nem MB nem KB é especificado no parâmetro SIZE do arquivo MODELOBD\_dat, ele usa MB e é alocado em *megabytes*. O *backup* do banco de dados MODELOBD\_log é alocado em *megabytes*, porque o sufixo MB é explicitamente declarado no parâmetro SIZE.

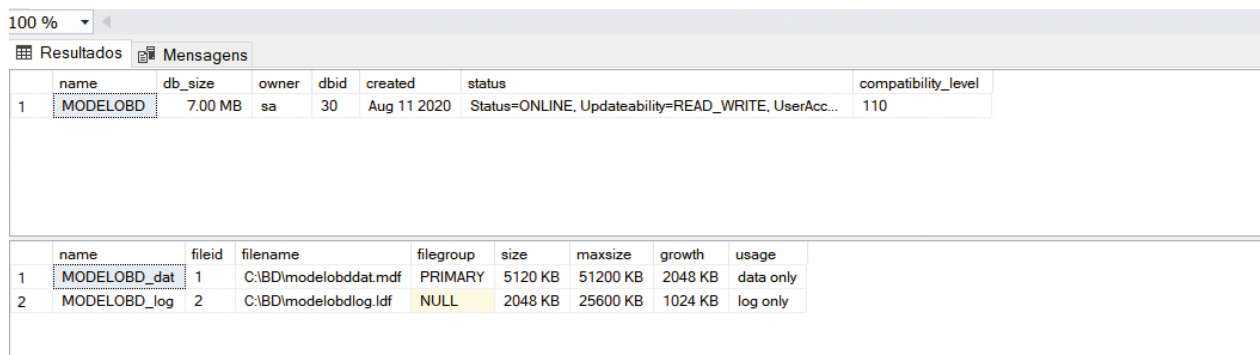
-----Criando o Banco de dados MODELOBD-----

```
USE master;
GO
CREATE DATABASE MODELOBD
ON
( NAME = modelobd_dat,
FILENAME = 'C:\BD\modelobddat.mdf',
SIZE = 5,MAXSIZE = 50, FILEGROWTH = 2 )
LOG ON
( NAME = modelobd_log,
FILENAME = 'C:\BD\modelobdlog.ldf',
SIZE = 2MB, MAXSIZE = 25MB, FILEGROWTH = 1MB );
GO
```

Para visualizarmos as informações de um banco de dados, podemos utilizar o procedimento armazenado SP\_HELPDB, conforme segue:

Sintaxe básica: SP\_HELPDB <nome do banco de dados>

Exemplo: SP\_HELPDB MODELOBD



	name	db_size	owner	dbid	created	status	compatibility_level
1	MODELOBD	7.00 MB	sa	30	Aug 11 2020	Status=ONLINE, Updateability=READ_WRITE, UserAcc...	110

	name	fileid	filename	filegroup	size	maxsize	growth	usage
1	MODELOBD_dat	1	C:\BD\modelobddat.mdf	PRIMARY	5120 KB	51200 KB	2048 KB	data only
2	MODELOBD_log	2	C:\BD\modelobdlog.ldf	NULL	2048 KB	25600 KB	1024 KB	log only

Figura 44 – Janela gerada após a execução do HELPDB

Executar o procedimento ALTER DATABASE modifica os dados de um banco de dados ou os arquivos e grupos de arquivos associados ao banco de dados, adiciona ou remove arquivos e grupos de arquivos de um banco de dados, altera os atributos de um banco de dados ou seus arquivos e grupos de arquivos, altera o agrupamento de banco de dados e define opções de banco de dados. Instantâneos de banco de dados não podem ser modificados.

A seguir, algumas opções de alteração.

## Alterando o nome do banco de dados

O exemplo a seguir altera o nome do banco de dados MODELOBD para MODELOBDNOVO.

```
-----Alterando o nome do Banco MODELOBD-----
-----
USE master;
GO
ALTER DATABASE MODELOBD
Modify Name = MODELOBDNOVO;
GO
```

## Alterando o agrupamento de um banco de dados

O exemplo a seguir cria um banco de dados denominado BDTESTE com o agrupamento SQL\_Latin1\_General\_CP1\_CI\_AS e, em seguida, altera o agrupamento do banco de dados BDTESTE para COLLATE French\_CI\_AI.

```
CREATE DATABASE BDTESTE
COLLATE SQL_Latin1_General_CP1_CI_AS ;
GO
ALTER DATABASE BDTESTE
COLLATE French_CI_AI ;
```

## Configurando o banco de dados como READ\_ONLY

Alterar o estado de um banco de dados ou grupo de arquivos para READ\_ONLY ou READ\_WRITE requer acesso exclusivo ao banco de dados. O exemplo a seguir define o banco de dados como o modo SINGLE\_USER para obter acesso exclusivo. Em seguida, o exemplo define o estado do banco de dados BDTESTE como READ\_ONLY e retorna o acesso ao banco de dados para todos os usuários.

```
-----Configurando o banco de dados para READ_ONLY-----
-----
ALTER DATABASE BDTESTE
SET SINGLE_USER
WITH ROLLBACK IMMEDIATE;
GO
ALTER DATABASE BDTESTE
SET READ_ONLY
GO
ALTER DATABASE BDTESTE
SET MULTI_USER;
GO
```

100 % ▾

Resultados Mensagens

	name	db_size	owner	dbid	created	status	compatibility_level
1	BDTESTE	5.38 MB	sa	31	Aug 11 2020	Status=ONLINE, Updateability=READ_ONLY, UserAcce...	110

	name	fileid	filename	filegroup	size	maxsize	growth	usage
1	BDTESTE	1	C:\Program Files\Microsoft SQL Server\MSSQL11.MSS...	PRIMARY	4160 KB	Unlimited	1024 KB	data only
2	BDTESTE_log	2	C:\Program Files\Microsoft SQL Server\MSSQL11.MSS...	NULL	1344 KB	2147483648 KB	10%	log only

Figura 45 – Configurando o modelo de recuperação do banco de dados

O modelo de recuperação é uma propriedade de banco de dados que controla como as transações são registradas, se o *log* de transações exige (e permite) *backup* e que tipos de operações de restauração estão disponíveis. Existem três modelos de recuperação: simples, completo e *bulk-logged*. Geralmente, um banco de dados usa o modelo de recuperação completo ou o modelo de recuperação simples. É possível alternar para outro modelo de recuperação do banco de dados a qualquer momento. Os bancos de dados modelos definem o modelo de recuperação padrão de novos bancos de dados.

Veja em seguida o comando para alterar o modelo de recuperação de um banco de dados:

```
--Criando o modo de recuperação BD MODELOBDNOVO--
```

```
ALTER DATABASE MODELOBDNOVO SET RECOVERY FULL;
```

100 % ▾

Resultados Mensagens

	name	db_size	owner	dbid	created	status	compatibility_level
1	MODELOBDNOVO	7.00 MB	sa	30	Aug 11 2020	Status=ONLINE, Updateability=READ_WRITE, UserAcc...	110

	name	fileid	filename	filegroup	size	maxsize	growth	usage
1	MODELOBD_dat	1	C:\BD\modelobddat.mdf	PRIMARY	5120 KB	51200 KB	2048 KB	data only
2	MODELOBD_log	2	C:\BD\modelobdlog.ldf	NULL	2048 KB	25600 KB	1024 KB	log only

Figura 46 – Alterando modelo de recuperação do banco de dados

## 4.2.2 Removendo um banco de dados criado

Um banco de dados pode ser removido independentemente de seu estado: *off-line*, somente leitura, suspeito, e assim por diante. Para exibir o estado atual de um banco de dados, use o `sys.databases` exibição do catálogo.

Um banco de dados cancelado poderá ser recriado somente por meio da restauração de um *backup*. Não é possível efetuar *backup* de instantâneos do banco de dados, portanto eles não podem ser restaurados.

```
SELECT name, user_access_desc, is_read_only, state_desc, recovery_model_desc
FROM sys.databases;
```

100 %

Resultados

Mensagens

	name	user_access_desc	is_read_only	state_desc	recovery_model_desc
1	master	MULTI_USER	0	ONLINE	SIMPLE
2	tempdb	MULTI_USER	0	ONLINE	SIMPLE
3	model	MULTI_USER	0	ONLINE	FULL
4	msdb	MULTI_USER	0	ONLINE	SIMPLE
5	ReportServer	MULTI_USER	0	ONLINE	FULL
6	ReportServerTempDB	MULTI_USER	0	ONLINE	SIMPLE
7	MODELOBDNOVO	MULTI_USER	0	ONLINE	FULL

Figura 47 – Consultando a exibição do sys.databases

A exclusão de um banco de dados e sua instância do SQL Server exclui os arquivos de discos físicos usados pelo banco de dados. Se o banco de dados ou qualquer um de seus arquivos estiverem *off-line* quando forem cancelados, os arquivos em disco não serão excluídos. Esses arquivos podem ser excluídos manualmente usando o Windows Explorer. Para remover um banco de dados do servidor atual sem excluir os arquivos do sistema de arquivos, use `sp_detach_db`. Exemplo de seleção de informações do sistema:

```
-----
---- EXCLUINDO O BANCO DE DADOS  MODELOBDNOVO-----
-----
```

```
DROP DATABASE MODELOBDNOVO
```

Na prática, uma linguagem de banco de dados permite a criação de bancos e estruturas de tabelas para executar tarefas básicas de gerenciamento de dados (adicionar, excluir e modificar) e consultas complexas, transformando dados brutos em informações úteis. Ainda deve executar essas funções básicas exigindo menor esforço possível do usuário, com uma estrutura e sintaxe de comandos fáceis de aprender. Deve ser portátil, ou seja, adequar-se a um padrão básico para que os usuários se adaptem à utilização do SGBD. A linguagem SQL possui duas categorias que atendem todas essas demandas, são elas: DDL (linguagem de definição de dados) e DML (linguagem de manipulação de dados). Entretanto, é necessário esclarecer outros fatores relevantes para a criação e manipulação desses dados.

## 4.2.3 Tipos de dados

Na plataforma SQL Server, dados de coluna, de variável local, expressão e parâmetro devem possuir um tipo de dado relacionado. Esse tipo de dado é um atributo que especifica os tipos de dados que o objeto pode manter: dados inteiros, dados de caractere, dados monetários, data e hora, cadeias x de caracteres binárias etc. O SQL Server oferece um conjunto de tipos de dados do sistema que define todos os tipos de dados que podem ser usados com o SQL Server. Você também pode definir seus próprios tipos de dados em Transact-SQL ou Microsoft .NET Framework. Após a criação do esquema de banco de dados, pode-se agora definir a estrutura das tabelas.

Quadro 6 – Dados baseados em caracteres

Tipo	Descrição	Tamanho
Char(n)	Trata-se de um <i>datatype</i> que aceita como valor qualquer dígito, sendo que o espaço ocupado no disco é de 1 dígito por caractere.	Permite usar até 8 mil dígitos.
Varchar(n)	Também aceita como valor qualquer dígito e o espaço ocupado em disco é de 1 dígito por caractere. A diferença para o Char é que o Varchar geralmente é usado quando não se sabe o tamanho fixo de um campo.	Permite usar até 8 mil dígitos.
Text	Qualquer dígito pode ser usado neste <i>datatype</i> , sendo ocupado 1 byte por caractere.	Permite usar até 2.147.483.647 bytes.

Quadro 7 – Dados baseados em caracteres Unicode

Tipo	Descrição	Tamanho
Nchar(n)	Nesse <i>datatype</i> , pode-se usar qualquer dígito, sendo ocupados 2 bytes a cada caractere.	É possível usar até 8 mil bytes.
Nvarchar(n)	Igual ao tipo anterior, com a única diferença que se usa esse tipo quando não se sabe o tamanho fixo de um campo. 2 bytes são ocupados a cada caractere.	É possível usar até 8 mil bytes.
Ntext	Também aceita qualquer dígito, 2 bytes são ocupados a cada caractere.	É possível usar até 1.073.741.823 bytes.

Quadro 8 – Dados baseados em números inteiros

Tipo	Descrição	Tamanho
Bigint	Aceita valores entre $-2^{63}$ (-9.223.372.036.854.775.808) a $2^{63}-1$ (9.223.372.036.854.775.807).	É possível usar até 8 bytes.
Int	Os valores aceitos aqui variam entre $-2^{31}$ (-2.147.483.648) a $2^{31}-1$ (2.147.483.647).	É possível usar até 4 bytes.
Smallint	Aceita valores entre -32768 até 32767.	É possível usar até 4 bytes.
Tinyint	Os valores aceitos aqui variam entre 0 e 255.	É possível usar até 1 byte.
Bit	É um tipo de dado inteiro (conhecido também como booleano), cujo valor pode corresponder a NULL, 0 ou 1. Podemos converter valores de string TRUE e FALSE em valores de bit, sendo que TRUE corresponde a 1 e FALSE a 0.	

Quadro 9 – Dados baseados em números exatos

Tipo	Descrição	Tamanho
Decimal	Os valores aceitos variam entre $-10^{38}-1$ e $10^{38}-1$ , sendo que o espaço ocupado varia de acordo com a precisão.	Os valores aceitos variam entre $-10^{38}-1$ e $10^{38}-1$ , sendo que o espaço ocupado varia de acordo com a precisão. Se a precisão for de 1 a 9, o espaço ocupado é de 5 bytes. Se a precisão é de 10 a 19, o espaço ocupado é de 9 bytes, já se a precisão for de 20 a 28, o espaço ocupado é de 13 bytes, e se a precisão for de 29 a 38, o espaço ocupado é de 17 bytes.
Numérico	Considerado um sinônimo do <i>datatype</i> decimal, o numérico também permite valores entre $-10^{38}-1$ e $10^{38}-1$ .	O espaço ocupado é o mesmo do anterior.

**Quadro 10 – Dados baseados em números aproximados**

Tipo	Descrição	Tamanho
Float[(n)]	O mesmo que <i>double precision</i> quando o valor de n é 53, este <i>datatype</i> aceita valores entre $-1.79E + 308$ e $1.79E + 308$ .	O espaço ocupado varia de acordo com o valor de n. Se esse valor estiver entre 1 e 24, a precisão será de 7 dígitos, sendo que o espaço ocupado será de 4 bytes. Se o valor de n estiver entre 25 e 53, sua precisão será de 15 dígitos, assim sendo o espaço ocupado será de 8 bytes.
Real	Este <i>datatype</i> é similar ao float(n) quando o valor de n é 24.	Os valores aceitos variam entre $-3.40E + 38$ e $3.40E + 38$ . Esse <i>datatype</i> ocupa 4 bytes.

**Quadro 11 – Dados baseados em números monetários**

Tipo	Descrição	Tamanho
Money	Este <i>datatype</i> aceita valores entre $-2^{63}$ e $2^{63}-1 = -922.337.203.685.477,5808$ para $922.337.203.685.477,5807$ ( $-922.337.203.685.477,58$ para $922.337.203.685.477,58$ para Informática).	Informática só dá suporte a dois decimais e não quatro, sendo que 8 bytes são ocupados.
Smallmoney	É possível usar valores entre $-2^{31}$ e $2^{31}-1$ ( $-214.748,3648$ a $214.748,3647$ ).	Suporta até 4 bytes.

**Quadro 12 – Dados baseados em data e hora**

Tipo	Descrição	Tamanho
Datetime	Permite o uso de valores entre 1/1/1753 e 31/12/9999.	Ocupa até 8 bytes e sua precisão atinge 3,33 milissegundos.
Smalldatetime	Aceita o uso de valores entre 1/1/1900 e 06/06/2079.	Sua precisão é de 1 minuto e ocupa 4 bytes em disco.
Date	Aceita o uso de valores entre 01/01/0001 e 31/12/9999.	Sua precisão é de 1 dia e ocupa 3 bytes em disco.
Time	Aceita o uso de valores entre 00:00:00.0000000 a 23:59:59.9999999.	Sua precisão é de 100 nanossegundos e ocupa de 3 a 5 bytes em disco.

**Quadro 13 – Dados baseados em binários**

Tipo	Descrição	Tamanho
Binary[(n)]	Este <i>datatype</i> representa os dados que serão usados no formato binário.	Ocupa de n+4 bytes, sendo que n pode variar entre 1 e 8000 bytes.
Varbinary[(n)]	Aqui também é usado o formato binário.	O espaço ocupado e a variação de n são iguais ao anterior.
Image	O formato binário também é usado aqui.	O espaço ocupado é de $2^{31}-1$ bytes ou 2.147.483.647.

Quadro 14 – Dados baseados em tipos de dados

Tipo	Descrição	Tamanho
Uniqueidentifier	O formato hexadecimal é usado para o armazenamento de dados binários.	Ocupa até 16 bytes.
Timestamp	Um valor binário é gerado pelo SQL Server.	Ocupa até 8 bytes.
Bit	Este <i>datatype</i> pode apresentar 0, 1 ou NULL.	Ocupado até 1 byte.
Sql_Variant	Os valores aqui podem ser de qualquer <i>datatype</i> .	Com esse tipo é possível armazenar até 8016 bytes.

Quadro 15 – Cursor – *datatype* usado somente quando trabalhamos com variáveis

Tipo	Descrição	Tamanho
Table	<i>Datatype</i> usado somente quando trabalhamos com variáveis de memória.	
Xml	Por este <i>datatype</i> , podemos armazenar fragmentos de documentos XML em um banco de dados SQL. Esses fragmentos correspondem às instâncias XML que não possuem um determinado elemento de nível superior. Essas instâncias são armazenadas quando criamos variáveis e colunas com <i>datatype</i> XML.	O espaço máximo ocupado deve ser de 2 GB.

Fonte: Microsoft (2020).

### 4.3 Conexão com o SQL Server para geração/atualização do banco de dados

Para realizar a criação de tabelas, é necessário utilizar o comando CREATE TABLE. O primeiro passo é descrever a coluna a ser definida, em seguida o tipo de dado que deve ser compatível com a informação armazenada e ainda quais são as restrições que esse campo recebe. Por exemplo, se um campo pode ser vazio ou não (*not null* ou *null*).

Para compreender melhor a leitura do código SQL, na programação utilizamos linhas de comando para definir campo (atributo). Além disso, são utilizados espaços para alinhar as características e restrições dos atributos. Por fim, tanto os nomes das tabelas como de atributos aparecem totalmente em letras maiúsculas. Essas convenções são utilizadas nos seguintes exemplos que criam as tabelas FORNECEDOR e PRODUTO.

#### Exemplo: criar a tabela FORNECEDOR

```

-----
----- CRIANDO TABELA FORNECEDOR -----
-----
CREATE TABLE FORNECEDOR
(
F_COD                INTEGER          NOT NULL          UNIQUE,
F_NOME               VARCHAR (35)     NOT NULL,
F_CONTATO             VARCHAR (15)    NOT NULL,

```



```
F_CODAREA          CHAR      (3)      NOT NULL,
F_FONE             CHAR      (9)      NOT NULL,
F_ESTADO           CHAR      (2)      NOT NULL,
F_ORDER            CHAR      (1)      NOT NULL,
PRIMARY KEY (F_COD)
```

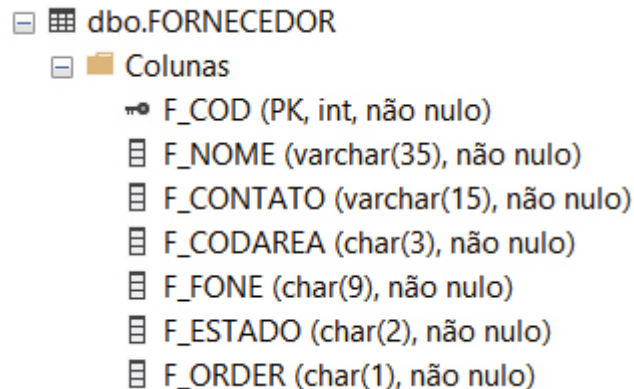


Figura 48 – dbo.FORNECEDOR

### Exemplo: criar a tabela PRODUTO

```
----- CRIANDO TABELA PRODUTO -----
CREATE TABLE PRODUTO
(
P_COD              VARCHAR (10)          NOT NULL          UNIQUE,
P_DESCRICAO        VARCHAR (35)          NOT NULL,
P_DATA             DATE                  NOT NULL,
P_UNIDISP          SMALLINT              NOT NULL,
P_MIN              SMALLINT              NOT NULL,
P_PRECO            NUMERIC (8,2)          NOT NULL,
P_TAXADESC         NUMERIC (5,2)          NOT NULL,
F_COD              INTEGER,
PRIMARY KEY (P_COD),
FOREIGN KEY (F_COD) REFERENCES FORNECEDOR ON UPDATE CASCADE)
```

Entendendo melhor a sequência de comandos SQL utilizada para a criação das tabelas FORNECEDOR e PRODUTO:

- As especificações NOT NULL dos atributos garantem que seja feita uma entrada de dados. Quando for indispensável haver dados disponíveis, a especificação NOT NULL não permitirá que o usuário deixe o atributo vazio (sem nenhuma entrada de dados). Como essa especificação é feita no nível da tabela e armazenada no dicionário de dados, os aplicativos podem utilizá-la para criar a validação do dicionário automaticamente.
- A especificação UNIQUE cria um índice exclusivo no respectivo atributo. Utilize-a para evitar a duplicação de valores em uma coluna.

- Os atributos de chave primária contêm tanto uma especificação NOT NULL como uma UNIQUE. Elas garantem as exigências de integridade de entidades. Se essas especificações não forem suportadas, utilize PRIMARY KEY sem elas.
- A definição de tabela inteira fica entre parênteses. Utiliza-se uma vírgula para separar cada definição de elemento da tabela (atributos, chave primária e chave estrangeira).
- A especificação ON UPDATE CASCADE garante que uma mudança F\_COD de qualquer FORNECEDOR se aplique automaticamente a todas as referências de chave estrangeira no sistema (cascata), garantindo a manutenção da integridade referencial.
- Um SGBDR aplicará automaticamente integridade referencial às chaves estrangeiras. Ou seja, não é possível haver uma entrada inválida na coluna de chave estrangeira. Ao mesmo tempo, não se pode excluir uma linha de fornecedor enquanto uma linha de produtos referenciar esse fornecedor.

### Restrições SQL

Existem algumas regras de integridade de entidade, e sua referência é fundamental em um ambiente de banco de dados relacional. A maioria das implementações de SQL dá suporte a ambas as regras. A integridade de entidades é aplicada automaticamente quando se especifica a chave primária na sequência de comandos CREATE TABLE. Por exemplo, é possível criar a estrutura de tabela FORNECEDOR e preparar a aplicação das regras de integridade de entidades utilizando:

```
PRIMARY KEY (V_CODE)
```

Na sequência CREATE TABLE da tabela PRODUTO, observe que a integridade referencial foi aplicada especificando o seguinte na tabela PRODUTO:

```
FOREIGN KEY (F_COD) REFERENCES FORNECEDOR ON UPDATE CASCADE
```

A definição de restrição de chave estrangeira garante que:

- Não seja possível excluir um fornecedor da respectiva tabela se pelo menos uma linha de produto for referenciada a esse fornecedor. Esse é o padrão de comportamento no tratamento de chaves estrangeiras.
- Por outro lado, se uma alteração for feita em F\_COD existente na tabela PRODUTO (ON UPDATE CASCADE), essa restrição torna impossível existir um valor F\_COD na tabela PRODUTO que aponte para um valor F\_COD inexistente na tabela FORNECEDOR. Em outras palavras, a especificação ON UPDATE CASCADE garante a preservação da integridade referencial.

Além de PRIMARY KEY e FOREIGN KEY, o padrão SQL ANSI também define as seguintes restrições:

- **NOT NULL:** garante que uma coluna não aceite nulos.

- **UNIQUE:** garante que todos os valores de uma coluna sejam exclusivos.
- **DEFAULT:** atribui um valor a um atributo quando uma nova linha é adicionada à tabela. O usuário final pode, evidentemente, inserir um valor diferente desse padrão.
- **CHECK:** é utilizado para validar dados quando é inserido um valor de atributo. A restrição CHECK não é exatamente o que seu nome sugere – ela verifica se uma condição especificada ocorre. Alguns exemplos dessa restrição são:
  - O valor mínimo de pedido deve ser 10.
  - A data deve ser posterior a 1 de janeiro de 2008.

Se a restrição CHECK for atendida em determinado atributo (ou seja, se a condição for verdadeira), os dados são aceitos. Se a condição encontrada for falsa, gera-se uma mensagem de erro e os dados não são aceitos. Observe que o comando CREATE TABLE permite definir restrições em dois locais diferentes:

- Quando se cria a definição de coluna (conhecida como restrição de coluna).
- Quando se utiliza a palavra-chave CONSTRAINT (restrição em diferentes locais).

Uma restrição de coluna aplica-se a apenas uma coluna. Uma restrição de tabela pode se aplicar a várias colunas. Essas restrições são suportadas em diversos níveis de conformidades pelos SGBDRs.

No seguinte comando SQL para a criação da tabela FATURA, a restrição DEFAULT atribui uma data padrão a nova fatura e a restrição CHECK valida se sua data é posterior a 1 de janeiro de 2008.

```
-----  
----- CRIANDO TABELA FATURA -----  
-----  
CREATE TABLE FATURA (  
    FAT_NO NUMERIC PRIMARY KEY,  
    FAT_COD NUMERIC NOT NULL REFERENCES CLIENTE (CLI_COD),  
    FAT_DATA DATETIME DEFAULT SYSDATETIME() NOT NULL,  
    CONSTRAINT FAT_CK1 CHECK (FAT_DATA > ('01-01-2008')));
```

Nesse caso, observe o seguinte:

- A definição de atributo FAT\_COD contém REFERENCES FATURA (FAT\_COD) para indicar que CLI\_COD é uma chave estrangeira. Esse é outro modo de definir a chave estrangeira.
- A restrição DEFAULT utiliza a função especial SYSDATETIME ( ). Essa função sempre retorna a data de hoje.

- O atributo de data da FATURA (FAT\_DATA) recebe automaticamente a data de hoje (retornada pelo SYSDATETIME ( )) quando uma nova linha é adicionada e nenhum valor é inserido no atributo.
- Utiliza-se uma restrição CHECK para validar se a data da FATURA é posterior a 1 de janeiro de 2008 ('01-01-2008').

A última sequência de comandos SQL cria a tabela LINHA. Essa tabela possui uma chave primária (FAT\_NO, LINHA\_NO) e utiliza uma restrição UNIQUE em LINHA\_NO e P\_COD para garantir que o mesmo produto não seja pedido duas vezes na mesma fatura.

```
-----  
----- CRIANDO TABELA LINHA-----  
-----  
CREATE TABLE LINHA  
  
(  
    FAT_NO          NUMERIC          NOT NULL,  
    LINHA_NO        NUMERIC (2,0)    NOT NULL,  
    P_COD           VARCHAR (10)     NOT NULL,  
    LINHA_UNIT      NUMERIC (9,2)    DEFAULT 00          NOT NULL,  
    LINHA_PRECO     NUMERIC (9,2)    DEFAULT 0.00        NOT NULL,  
    PRIMARY KEY     (FAT_NO, LINHA_NO),  
    FOREIGN KEY     (FAT_NO) REFERENCES FATURA ON DELETE CASCADE,  
    FOREIGN KEY (P_COD) REFERENCES PRODUTO (P_COD),  
    CONSTRAINT LINHA_U1 UNIQUE (FAT_NO, P_COD)  
)
```

Na criação da tabela LINHA, observe que é adicionada uma restrição UNIQUE para evitar a duplicação de linha da FATURA. Essa restrição é aplicada por meio da criação de um índice exclusivo. Observe também que a ação ON DELETE CASCADE é recomendada a entidades fracas para garantir que a exclusão de uma linha na entidade forte acione automaticamente a exclusão das linhas correspondentes na entidade dependente. Nesse caso, a exclusão de uma linha da FATURA excluirá automaticamente todas as linhas da tabela LINHA relacionada a FATURA.

### 4.4 Índices em SQL

Os índices podem aprimorar a eficiência de buscas de dados e evitar a duplicação de valores de colunas. Na verdade, quando se cria uma chave primária automaticamente se cria um índice exclusivo. Mesmo com esse recurso, é comum precisarmos de índices adicionais. A capacidade de criar índices de modo rápido e eficiente é importante. Utilizando o comando CREATE INDEX, é possível criar índices com base em qualquer atributo selecionado. A sintaxe é:

```
CREATE [UNIQUE] INDEX nome do índice ON nome da tabela (coluna1, coluna2, colunaN)
```

Por exemplo, com base no atributo P\_DATA, armazenado na tabela PRODUTO, o comando a seguir cria um índice chamado P\_DATA\_X.

```
CREATE UNIQUE INDEX P_CODX ON PRODUTO (P_COD);
```

Agora, se o usuário tentar inserir um valor de P\_COD duplicado, a SQL produz uma mensagem de erro "duplicate value in index". Muitos SGBDRs criam automaticamente um índice exclusivo para os atributos de PK quando de sua declaração.

Uma prática comum é criar um índice para qualquer campo que seja utilizado como chave de busca, em operações de comparação de uma expressão condicional ou quando se deseja listar linhas em uma ordem específica. Por exemplo, caso queira criar um relatório de todos os produtos por fornecedores seria útil criar um índice para o atributo F\_COD na tabela PRODUTO. Lembre-se de que um fornecedor pode suprir vários produtos. Portanto, não se deve criar um índice UNIQUE nesse caso. Para tornar a busca o mais eficiente possível, recomenda-se, ainda, um índice composto.

Os índices compostos costumam ser utilizados para evitar duplicações de dados. Considere, por exemplo, o caso ilustrado na tabela a seguir, no qual são armazenadas pontuações de funcionários em testes exigidos. (Um funcionário pode fazer um teste apenas uma vez em determinada data.) Dada a estrutura da tabela, a PK é FUNC\_NUM + TEST\_NUM. A terceira entrada de testes do funcionário 111 atende às exigências de integridade de entidades – a combinação 111, 3 é exclusiva – ainda que a entrada do teste WEA esteja claramente duplicada.

**Tabela 6 – Índices compostos**

FUNC_NUM	TEST_NUM	TEST_COD	TEST_DATA	TEST_PONT
00110	1	WEA	10/01/2012	93
00110	2	WEA	11/01/2012	87
00111	1	HAZ	17/03/2009	91
00111	2	WEA	27/07/2009	95
00111	3	WEA	02/07/2009	95
00112	1	CHEM	27/11/2015	91

Essa duplicação poderia ter sido evitada utilizando um índice composto exclusivo com os atributos FUNC\_NUM, TEST\_COD e TEST\_DATA:

```
CREATE UNIQUE INDEX FUNC_TESTEDEX ON TEST
```

```
(FUNC_NUM, TEST_COD, TEST_DATA);
```

Por padrão, todos os índices produzem resultados listados em ordem crescente, mas é possível criar um índice que apresente um produto em ordem decrescente. Por exemplo, caso imprima com frequência um relatório que liste todos os produtos em ordem decrescente de preço, é possível criar um índice chamado PROD\_PRECOX, digitando-se:

```
CREATE INDEX PROD_PRECOX ON PRODUTO (P_PRECO DESC)
```

Para excluir um índice, utilize o comando DROP INDEX:

DROP INDEX nome do índice

Por exemplo, caso queira eliminar o índice PROD\_PRECOX, digite:

DROP INDEX PROD\_PRECOX;

Após a criação das tabelas e de alguns índices, já estaremos prontos para começar a inserir dados. A seguir, serão demonstrados comandos de manipulação de dados.

### 4.5 Comandos de manipulação de dados

Os comandos básicos de manipulação de dados em SQL são: INSERT, SELECT, COMMIT, UPDATE, ROLLBACK e DELETE.

#### Inserção de dados em linhas de tabelas

A SQL exige a utilização do comando INSERT para inserir dados em uma tabela. A sintaxe básica desse comando será abordada a seguir:

INSERT INTO nome da tabela VALUES (valor1, valor2,...,valorN)

Como a tabela PRODUTO utiliza F\_COD para referenciar o mesmo atributo da tabela FORNECEDOR pode ocorrer uma violação de integridade se não existirem valores F\_COD nesta última tabela. Portanto, é necessário inserir as linhas de FORNECEDOR antes das de PRODUTOS. Dada a estrutura de tabela de FORNECEDOR, deve-se inserir as duas primeiras linhas de dados da seguinte maneira:

```
-----  
-----  INSERÇÃO  DADOS TABELA FORNECEDOR  -----  
-----  
  
INSERT INTO FORNECEDOR VALUES  
(10011, 'Royal, LTDA', 'Antonio', '011', '987879087', 'SP', 'S')  
INSERT INTO FORNECEDOR VALUES  
(10012, 'Places, LTDA', 'Manoela Lima', '021', '986540874', 'RJ', 'S')
```

E assim por diante, até que todos os registros da tabela FORNECEDOR tenham sido inseridos.

As linhas da tabela PRODUTO devem ser inseridas do mesmo modo, utilizando os dados. Por exemplo, as duas primeiras linhas de dados seriam inseridas da seguinte maneira, pressionando-se a tecla Enter ao término de cada linha:

### ----- INSERÇÃO DADOS TABELA PRODUTO -----

```
INSERT INTO PRODUTO VALUES  
( '200Q/1', 'Notebook 2 em 1 Dell Inspiron', '10-06-2020', 24, 5, 4.999, 0, 10011)
```

```
INSERT INTO PRODUTO VALUES  
( 'C11CG', 'Multifuncional Epson Tanque Tinta', '10-08-2020', 24, 5, 1.439, 0, 10011)
```

Nas linhas de entrada de dados citadas, observe que:

- O conteúdo da linha é inserido entre parênteses. O primeiro caractere após VALUES é um parêntese, assim como o último caractere da sequência de comando.
- Os valores de caracteres (STRING) e datas devem ser inseridos entre apóstrofos ( ' ' ).
- As entradas numéricas não são cercadas por apóstrofos.
- As entradas de atributos são separadas por vírgulas.
- É necessário um valor para cada coluna da tabela.

Essa versão dos comandos INSERT adiciona uma única linha de tabela por vez.

### Inserção de linhas com atributos nulos

Na maioria das vezes, inserimos linhas com todos os valores dos atributos especificados. Mas o que fazer se um produto não tiver um fornecedor ou se ainda não souber o código do fornecedor? Nesses casos, deve-se deixar o código de fornecedor como nulo. Para inserir um nulo, utilize a seguinte sintaxe:

```
INSERT INTO PRODUTO VALUES  
( 'C11GU31', 'SSD External Solid', '01-08-2020', 24, 5, 739, 0, NULL)
```

Nesse caso, a inserção da entrada NULL é aceita apenas porque o atributo F\_COD é opcional - a declaração NOT NULL não foi utilizada no comando CREATE TABLE desse atributo.

### Inserção de linhas com atributos opcionais

Em situações em que mais de um atributo é opcional, em vez de declarar cada atributo como NULL no comando INSERT, é possível indicar apenas os atributos que tenham valores obrigatórios. Isso pode ser feito listando os nomes de atributos entre parênteses após o nome da tabela. Para esse exemplo, assumo que os únicos atributos necessários da tabela PRODUTO sejam P\_COD e P\_DESCRIÇÃO.

```
INSERT INTO PRODUTO (P_COD, P_DESCRIÇÃO, P_DATA, P_UNIDISP, P_MIN, P_PREÇO,  
P_TAXADESC)  
VALUES ( 'M1352', 'Mouse Sem Fio Ergonomic', '10-03-2019', 24, 5, 319.00, 0);
```

## SELECT (projeção em listas)

O comando SELECT é utilizado para listar o conteúdo de uma tabela. A sintaxe desse comando é a seguinte:

SELECT lista de colunas FROM nome da tabela

A lista de colunas representa um ou mais atributos separados por vírgulas. Pode-se utilizar \* (asterisco) como caractere coringa para listar os atributos. O caractere coringa é um símbolo que pode ser utilizado como substituto geral para outros caracteres ou comandos. Por exemplo, para listar todos os atributos e todas as linhas da tabela PRODUTO, utilize:

SELECT\*FROM PRODUTO

	P_COD	P_DESCRIÇÃO	P_DATA	P_UNIDISP	P_MIN	P_PREÇO	P_TAXADESC	F_COD
1	20001	Notebook 2 em 1 Dell Inspiron	2020-06-01	24	5	4999.00	2.00	10011
2	200Q/1	Notebook 2 em 1 Dell Inspiron	2020-10-06	24	5	5.00	0.00	10011
3	C11CG	Multifuncional Epson Tanque Tinta	2020-10-08	24	5	1.44	0.00	10011
4	M1352	Mouse Sem Fio Ergonomic	2019-10-03	24	5	319.00	0.00	NULL

Figura 49 – Conteúdo da tabela PRODUTO

A figura mostrada apresenta a saída gerada por esse comando. Ela apresenta todas as linhas da tabela PRODUTO que serão utilizadas para os próximos exemplos. Caso insira apenas dois registros dessa tabela, como resultado do comando SELECT apresentará as novas linhas inseridas.

Os comandos SQL podem ser agrupados em uma única linha, sequências complexas são mais bem apresentadas em linhas separadas, com espaço entre os comandos e seus componentes. Por exemplo, observe o seguinte comando:

```
SELECT P_COD, P_DESCRIÇÃO, P_DATA, P_UNIDISP, P_MIN, P_PREÇO, P_TAXADESC, F_COD
FROM PRODUTO
```

Quando o comando SELECT é executado em uma tabela, o SGBDR retorna um conjunto de uma ou mais linhas que tenham as mesmas características de uma tabela relacional. O SELECT lista todas as linhas da tabela específica na cláusula FROM. Essa prática é muito importante nos comandos SQL. Por padrão, a maioria dos comandos de manipulação de dados opera em uma tabela (ou relação) interna. Por isso, se diz que os comandos de SQL são orientados a conjuntos. Esse tipo de comando atua sobre um conjunto de linhas. Tal conjunto pode incluir uma ou mais colunas e zero ou mais linhas de uma ou mais tabelas.

## UPDATE para atualização de linhas da tabela

Utilize o comando UPDATE para modificar os dados de uma tabela. A sintaxe desse comando é:



UPDATE    nome da tabela

SET        nome da tabela = expressão [, nome da coluna=expressão]

[WHERE    lista de condições];

Por exemplo, caso deseje alterar P\_DATA de 10-03-2019 para 10-06-2019 na quarta linha da tabela PRODUTO, utilize a chave primária (M1352) para localizar a linha correta (quarta). Portanto, digite:

```
UPDATE PRODUTO
SET P_DATA = '06-06-2020'
WHERE P_COD = '200Q/1' ;
```

Se mais de um atributo deve ser atualizado na linha, separe as correções com vírgula:

```
UPDATE PRODUTO
SET P_DATA = '10-03-2019 ', P_PRECO = 359.99, P_MIN = 10
WHERE P_COD = 'M1352' ;
```

O que teria acontecido se o comando anterior UPDATE não tivesse incluído a condição WHERE? Os valores P\_DATA, P\_PRECO, P\_MIN teriam sido alterados em todas as linhas da tabela PRODUTO. Lembre-se de que o comando UPDATE aplicará as alterações a todas as linhas da tabela especificada.

Confirme as correções utilizando o seguinte comando SELECT para verificar a listagem da tabela PRODUTO:

```
SELECT * FROM PRODUTO
```

### Restaurando conteúdo das tabelas

Podemos realizar a restauração de conteúdos em um banco de dados, caso o comando COMMIT ainda não tenha sido executado para armazenar permanentemente as alterações no banco de dados. É possível restaurar o banco à sua condição anterior usando o comando ROLLBACK. Esse comando desfaz quaisquer alterações desde o último comando COMMIT e retorna os dados para os valores existentes antes de as alterações serem feitas.

Para restaurar os dados à sua condição "pré-alteração", digite ROLLBACK; em seguida, pressione a tecla Enter. Utilize o comando SELECT novamente para ver que ROLLBACK, de fato, restaurou os dados aos valores originais.

COMMIT e ROLLBACK funcionam apenas com comandos de manipulação de dados utilizados para adicionar, modificar e excluir linhas de tabelas. Por exemplo, suponha a execução das seguintes ações:

- 1) CREATE uma tabela chamada VENDAS.
- 2) INSERT 10 linhas na tabela VENDAS.
- 3) UPDATE 2 linhas da tabela VENDAS.
- 4) Comando ROLLBACK é executado.

A tabela VENDAS será removida pelo comando ROLLBACK? Não, ROLLBACK desfaz apenas os resultados dos comandos INSERT e UPDATE. Todos os comandos de definição de dados (CREATE TABLE) são automaticamente gravados (COMMIT) no dicionário de dados e não podem ser desfeitos.

### DELETE: excluindo linhas da tabela

Um dos comandos mais fáceis de executar é o de exclusão de dados de uma tabela utilizando o comando DELETE. A sintaxe é:

```
DELETE FROM      nome da tabela

[WHERE           lista de condições]
```

Caso queira excluir da tabela PRODUTO um dos produtos adicionados anteriormente, podemos fazer isso utilizando a referência do campo código através do campo P\_COD 'M1352'.

```
DELETE FROM PRODUTO
WHERE P_COD = 'M1352'
```

## 4.6 Expressões e strings no comando SELECT

A linguagem SQL possui variações mais genéricas da lista-seleção do que apenas uma lista de colunas. Os itens de uma seleção podem ser usados com a expressão **AS** nome-coluna, onde expressão é qualquer expressão aritmética ou de string envolvendo os nomes de colunas (possivelmente prefixadas por variáveis de intervalo) e constantes, e nome-coluna é um novo nome para essa coluna na saída da consulta. A seleção também pode conter funções agregadas, tais como SUM e COUNT. O padrão SQL também inclui expressões envolvendo valores de data e hora, os quais serão abordados logo mais. Embora não seja parte do padrão SQL, muitas implementações também suportam o uso de funções embutidas como sqrt, sin e mod.

Pode-se comparar strings usando os operadores de comparação. Se necessitarmos ordenar strings em uma ordem que não seja alfabética (por exemplo, ordenar strings que denotam os nomes dos meses na ordem do calendário: janeiro, fevereiro, março etc.), a SQL suporta um conceito genérico de **collation**, ou ordem de classificação, para um conjunto de caracteres. Uma collation permite que o usuário especifique quais caracteres são "menores do que" outros e provê grande flexibilidade na manipulação de strings.

### Operadores de comparação

A qualificação da cláusula FROM é uma combinação booleana (isto é, uma expressão usando os conectivos lógicos AND, OR e NOT) de condições no formato expressão op expressão, onde op é um dos operadores de comparação { <, <=> = < >, >=, > }. Uma expressão é um nome de coluna, uma constante ou uma expressão (aritmética ou de string).

**Tabela 7 – Operadores relacionais**

SÍMBOLO	SIGNIFICADO
=	IGUAL A
<	MENOR QUE
<=	MENOR OU IGUAL A
>	MAIOR QUE
>=	MAIOR OU IGUAL A
<> OU !=	Diferente de

Fonte: Rob e Coronel (2011, p. 267).

```
SELECT P_DESCRICAO, P_DATA, P_PRECO, P_COD
FROM PRODUTO
WHERE P_COD <> '12345';
```

	P_DESCRICAO	P_DATA	P_PRECO	P_COD
1	Notebook 2 em 1 Dell Inspiron	2020-06-01	4999.00	20001
2	Notebook 2 em 1 Dell Inspiron	2020-06-06	5000.00	200Q/1
3	Multifuncional Epson Tanque Tinta	2020-10-08	349.00	C11CG
4	Fone de ouvido bluetooth	2020-08-22	69.00	F345673
5	Fone de ouvido sem fio	2020-08-29	49.00	F345677
6	Mouse Microsoft	2020-08-22	199.00	M2345
7	Teclado sem fio	2020-08-22	299.00	T121345

Figura 50 – Resultado de dados com a cláusula diferente de 12345

Outro exemplo:

Liste os códigos, nomes, preços e código do fornecedor dos PRODUTOS que possuem preço menor que 50 e data maior que 01-01-2020 ou que possuam o código do fornecedor = 100111.

```
SELECT P_COD, P_DESCRICAO, P_PRECO, F_COD
FROM PRODUTO
WHERE (P_PRECO < 50 AND P_DATA > '01-01-2020')
OR F_COD = 10011;
```

	P_COD	P_DESCRICAO	P_PRECO	F_COD
1	20001	Notebook 2 em 1 Dell Inspiron	4999.00	10011
2	200Q/1	Notebook 2 em 1 Dell Inspiron	5000.00	10011
3	C11CG	Multifuncional Epson Tanque Tinta	349.00	10011
4	F345673	Fone de ouvido bluetooth	69.00	10011
5	F345677	Fone de ouvido sem fio	49.00	10011
6	M2345	Mouse Microsoft	199.00	10011
7	T121345	Teclado sem fio	299.00	10011

Figura 51 – Resultado de dados com a cláusula &lt; e &gt; ou OR

### Utilização de operadores de comparação em datas

Os procedimentos de datas costumam ser mais específicos em *software* do que os outros procedimentos de SQL. Por exemplo, a consulta para listar todas as linhas em que a data de estoque em que estoque ocorre em ou após 01-01-2017:

```
SELECT P_DESCRICAO, P_UNIDISP, P_MIN, P_DATA
FROM PRODUTO
WHERE P_DATA >= '01-01-2017'
```

	P_DESCRICAO	P_UNIDISP	P_MIN	P_DATA
1	Notebook 2 em 1 Dell Inspiron	24	5	2020-06-01
2	Notebook 2 em 1 Dell Inspiron	24	5	2020-06-06
3	Multifuncional Epson Tanque Tinta	24	5	2020-10-08
4	Fone de ouvido bluetooth	24	5	2020-08-22
5	Fone de ouvido sem fio	24	5	2020-08-29
6	Mouse Microsoft	24	5	2020-08-22
7	Teclado sem fio	24	5	2020-08-22

Figura 52 – Resultado de dados com a cláusula &gt;=

### Utilizando colunas computadas e alias de colunas

Suponha que se queira determinar o valor total de cada produto atualmente mantido em estoque. É lógico que essa determinação exige multiplicação de quantidade de cada produto disponível por seu preço atual. É possível realizar essa tarefa com o seguinte comando:

```
SELECT P_DESCRICAO, P_UNIDISP, P_PRECO,
       P_UNIDISP * P_PRECO AS RESULT
FROM PRODUTO
```

	P_DESCRIÇÃO	P_UNIDISP	P_PREÇO	RESULT
1	Notebook 2 em 1 Dell Inspiron	24	4999.00	119976.00
2	Notebook 2 em 1 Dell Inspiron	24	5000.00	120000.00
3	Multifuncional Epson Tanque Tinta	24	349.00	8376.00
4	Fone de ouvido bluetooth	24	69.00	1656.00
5	Fone de ouvido sem fio	24	49.00	1176.00
6	Mouse Microsoft	24	199.00	4776.00
7	Teclado sem fio	24	299.00	7176.00

Figura 53 – Resultado comando SELECT com uma coluna computada em um alias

Também é possível utilizar uma coluna computada, um alias e a aritmética de datas em uma única consulta. Por exemplo, suponha que se queira obter uma lista de produtos fora da garantia que tenham sido armazenados há mais de 90 dias. Nesse caso, P\_DATA é pelo menos 90 dias menor que a data atual do sistema, como apresentado a seguir:

```
SELECT P_COD, P_DATA, GETDATE() - 90 AS CUTDATE
FROM PRODUTO
WHERE P_DATA <= GETDATE() - 90;
```

### Operadores aritméticos

É possível utilizar operadores aritméticos com atributos de tabela em uma lista de colunas ou em uma expressão condicional. Na verdade, os comandos de SQL costumam ser utilizados em conjunto com os operadores aritméticos apresentados na tabela.

**Tabela 8 – Operadores aritméticos**

Operador aritmético	Descrição
+	Somar
-	Subtrair
*	Multiplicar
/	Dividir
^	Elevar a potência de (algumas aplicações utilizam ** em vez de ^)

Fonte: Rob e Coronel (2011, p. 270).

Não confunda o símbolo de multiplicação (\*) com o símbolo coringa utilizado por algumas implementações do SQL.

Ao executar operações matemáticas em atributos, lembre-se das regras de precedência. Como o nome sugere, as regras de precedência estabelecem a ordem em que os cálculos são feitos. Por exemplo, observe a ordem da sequência computacional a seguir:

- 1) Efetuar operações entre parênteses.
- 2) Efetuar operações de potenciação.
- 3) Efetuar multiplicações e divisões.
- 4) Efetuar somas e subtrações.

A aplicação das regras de precedência diz que  $6+4*5=20+6=26$ , mas  $(6+4)*5=10*5=50$ . De modo similar,  $5+4^2*6=5+8*6=53$ , mas  $(5+4)^2*6=81*6=489$ , ao passo que a operação expressa por  $(5+4^2)*6$  produz a resposta  $(5+8)*6=13*6=78$ .

### Operadores lógicos: AND, OR e NOT

A busca de dados normalmente envolve diversas condições. Por exemplo, quando se está comprando uma nova casa, busca-se certa área, determinado número de quartos, banheiros, andares etc. Do mesmo modo, a SQL permite a utilização de várias condições em uma consulta por meio de operadores lógicos. Os operadores lógicos são AND, OR e NOT. Por exemplo, caso deseje uma lista do conteúdo da tabela para  $F\_COD = 10011$  ou  $F\_COD = 10012$ , pode-se utilizar o operador OR, como na seguinte sequência de comandos:

```
SELECT P_COD, P_DESCRICAO, P_DATA, P_PRECO, F_COD
FROM PRODUTO
WHERE F_COD = 10011 OR F_COD = 10012 ;
```

	P_COD	P_DESCRICAO	P_DATA	P_PRECO	F_COD
1	20001	Notebook 2 em 1 Dell Inspiron	2020-06-01	4999.00	10011
2	200Q/1	Notebook 2 em 1 Dell Inspiron	2020-06-06	5000.00	10011
3	C11CG	Multifuncional Epson Tanque Tinta	2020-10-08	349.00	10011
4	C1234	Caneta digital	2020-08-09	199.00	10012
5	F345673	Fone de ouvido bluetooth	2020-08-22	69.00	10011
6	F345677	Fone de ouvido sem fio	2020-08-29	49.00	10011
7	M2345	Mouse Microsoft	2020-08-22	199.00	10011
8	T121345	Teclado sem fio	2020-08-22	299.00	10011

Figura 54 – Comando SELECT com seleção de colunas usando operador lógico OR

A utilização dos operadores lógicos OR e AND pode ser muito complexa quando se colocam várias restrições a uma consulta. Na verdade, há um campo de especialidade em matemática, conhecido como álgebra booleana.

O operador lógico NOT é utilizado para negar o resultado de uma expressão condicional. Ou seja, em SQL, todas as expressões condicionais são avaliadas como verdadeiras ou falsas. Se uma expressão for

verdadeira, a linha é selecionada; se for falsa, a linha não é selecionada. O operador lógico NOT costuma ser utilizado para encontrar linhas que não atendam a determinada condição. Por exemplo, caso se queira listar todas as linhas cujo código de fornecedor não seja 10013, deve-se utilizar a sequência de comandos:

```
SELECT *  
FROM FORNECEDOR  
WHERE NOT (F_COD= 10013);
```

### Operadores especiais: BETWEEN, IS NULL, LIKE, IN e EXISTS

A linguagem SQL do padrão ANSI permite a utilização de operadores especiais com a cláusula WHERE. Esses operadores incluem:

- **BETWEEN**: utilizado para verificar se o valor de um atributo está dentro de uma faixa.
- **IS NULL**: utilizado para verificar se o valor de um atributo é nulo.
- **LIKE**: utilizado para verificar se o valor de um atributo coincide com um determinado padrão de caractere.
- **IN**: utilizado para verificar se o valor de um atributo coincide com qualquer valor de uma lista.
- **EXISTS**: utilizado para verificar se uma subconsulta retorna alguma linha.

### Operador especial BETWEEN

O operador BETWEEN, geralmente, é utilizado para verificar se o valor de um atributo está dentro de uma faixa de valores. Por exemplo, para listar todos os produtos que possuem valores entre R\$ 250,00 e R\$ 900,00, deve-se utilizar a sequência de comandos a seguir:

```
SELECT *  
FROM PRODUTO  
WHERE P_PRECO BETWEEN 250.00 AND 900.00
```

### Operador especial IS NULL

A utilização de IS NULL permite verificar valores de atributos nulos. Por exemplo, caso queira listar todos os produtos que não tenham um fornecedor que possua o F\_COD é nulo.

```
SELECT P_COD, P_DESCRICAO, F_COD  
FROM PRODUTO  
WHERE F_COD IS NULL ;
```



## Lembrete

Null não é um "valor" como o número 0 (zero) ou um espaço em branco. Na verdade, trata-se de uma propriedade especial de um atributo que representa precisamente a ausência de qualquer valor.

## Operador especial LIKE

Outro operador suportado pela SQL é o operador LIKE, o qual encontra padrões em atributos strings e juntamente com o uso dos símbolos coringa % (que quer dizer zero ou mais caracteres arbitrários) e \_ (que quer dizer exatamente um caractere arbitrário). Assim, '\_AB%' denota um padrão correspondente a toda string que contém no mínimo três caracteres, com o segundo e o terceiro caracteres sendo A e B, respectivamente. Observe que, diferentemente dos demais operadores de comparação, os brancos podem ser significativos para o operador LIKE (dependendo da collation (ordem de classificação) do conjunto de caracteres subjacente). Assim, 'Antonio' = 'Antonio' é verdadeiro enquanto 'Antonio' LIKE 'Antonio' é falso. Um exemplo do uso de LIKE em uma consulta é dado a seguir.

Liste a matrícula e o *e-mail* dos funcionários cujo nome comece com R e tenha como terceira letra B.

```
SELECT  FUNC_MAT, FUNC_EMAIL, FUNC_DATADM
FROM    FUNCIONARIO
WHERE   FUNC_NOME LIKE 'R_B%';
```

	FUNC_MAT	FUNC_EMAIL
1	003	roberto@gmail.com

Figura 55 – Resultado do comando LIKE 'R\_B%'

## Operador especial IN

Muitas consultas que utilizam o operador lógico OR podem ser tratadas mais facilmente com a ajuda do operador especial IN. Veja o exemplo:

```
SELECT  *
FROM    PRODUTO
WHERE   F_COD='10011'
OR F_COD='10012';
```

De forma mais simples, o exemplo a seguir:

```
SELECT  *
FROM    PRODUTO
WHERE   F_COD IN ('10011','10012');
```



O operador IN é especialmente útil quando utilizado com subconsulta. No exemplo, desejamos listar F\_COD e F\_NOME apenas dos fornecedores que suprem os produtos. Para isso, utilizamos uma subconsulta no operador IN para gerar automaticamente a lista de valores, como vemos a seguir:

```
SELECT  F_COD, F_NOME
FROM    FORNECEDOR
WHERE   F_COD IN (SELECT F_COD FROM PRODUTO);
```

### Operador especial EXISTS

O operador especial EXISTS pode ser utilizado com frequência quando se deseja executar um comando com base no resultado de outra consulta. Ou seja, se uma subconsulta retorna quaisquer linhas, deve-se executar a consulta principal; do contrário, não. Veja o exemplo, a seguinte consulta listará todos os fornecedores, mas apenas se houver produtos a encomendar:

```
SELECT  *
FROM    FORNECEDOR
WHERE   EXISTS (SELECT * FROM PRODUTO WHERE P_UNIDISP <= P_MIN);
```

Esse operador especial é utilizado, no exemplo a seguir, para listar todos os fornecedores, mas apenas se houver produtos com quantidade disponível menor do que o dobro da quantidade mínima:

```
SELECT  *
FROM    FORNECEDOR
WHERE   EXISTS (SELECT * FROM PRODUTO WHERE P_UNIDISP <= P_MIN*2);
```

## 4.7 Comandos avançados de definição de dados

Alterações na estrutura das tabelas são feitas utilizando o comando ALTER TABLE, seguido por uma palavra-chave que produz a alteração desejada. Há três opções disponíveis: ADD, MODIFY e DROP. Utiliza-se ADD para adicionar uma coluna, MODIFY para alterar as características de uma coluna e DROP para excluir uma coluna de uma tabela. A maioria dos SGBDs não permite a exclusão de uma coluna (a menos que ela não contenha nenhum valor), pois essa ação pode excluir dados fundamentais utilizados por outras tabelas. A sintaxe básica para adicionar ou modificar é:

```
ALTER TABLE NOME DA TABELA
{ADD | MODIFY} NOME DA COLUNA TIPO DE DADOS [{ADD | MODIFY} NOME DA COLUNA TIPO
DE DADOS];
```

O comando ALTER TABLE também pode ser utilizado para adicionar restrições de tabelas. Nesses casos, a sintaxe seria:

```
ALTER TABLE PRODUTO
ADD RESTRIÇÃO [ ADD RESTRIÇÃO];
```

Podemos utilizar o comando ALTER TABLE para remover uma restrição de coluna ou tabela. A sintaxe correta é:

```
ALTER TABLE NOME DA TABELA  
DROP {PRIMARY KEY | COLUMN NOME DA COLUNA | CONSTRAINT NOME DA RESTRIÇÃO};
```



### Lembrete

As restrições PRIMARY KEY e FOREIGN KEY podem ser acrescentadas ou excluídas. Para a exclusão de uma FOREIGN KEY, é necessário a cláusula ON DELETE CASCADE, onde serão apagadas todas as linhas que contêm vínculo com essa chave estrangeira.

Removendo uma restrição, é necessário especificar o seu nome. Para isso, deve-se sempre nomear as restrições nos comandos CREATE TABLE e ALTER TABLE.

### Alteração do tipo dos dados da coluna

Utilizando a sintaxe de ALTER, o F\_COD (inteiro) da tabela PRODUTO pode ser alterado para F\_COD de caracteres utilizando-se:

```
ALTER TABLE PRODUTO  
MODIFY F_COD CHAR (5);
```

### Adicionando uma coluna

Outra alteração de uma tabela pode ser a inserção de uma ou mais colunas. Exemplo: adicionando a coluna chamada P\_CODPROM à tabela PRODUTO:

```
ALTER TABLE PRODUTO  
ADD P_CODPROM CHAR (1);
```

### Excluindo uma coluna

Ocasionalmente, pode-se excluir uma coluna. Suponha que se deseja excluir o atributo F\_ORDER da tabela FORNECEDOR. Para isso, deve-se utilizar o seguinte comando:

```
ALTER TABLE FORNECEDOR  
DROP COLUMN F_ORDER
```

## 4.8 Consultas avançadas

O SQL permite realizar consultas complexas de forma livre. Os operadores lógicos funcionam de modo similar ao do ambiente de consulta. Além disso, a SQL fornece funções úteis que contam, encontram

valores mínimos e máximos, calculam média etc. Além disso, podemos limitar consultas de usuário apenas a dados que não apareçam duplicados.

### ORDER BY

Essa cláusula é especialmente útil quando a ordem de listagem é importante. A sintaxe é:

```
SELECT      lista de colunas
FROM        lista de tabelas
WHERE       lista de condicoes
ORDER BY    lista de colunas [ASC DESC];
```

Embora haja a opção de declarar o tipo de ordem – crescente (ASC) ou decrescente (DESC) –, a ordem padrão é crescente. Por exemplo, caso deseje listar o conteúdo da tabela PRODUTO por P\_PRECO em ordem crescente, utilize:

```
SELECT P_COD, P_DESCRICAO, P_DATA, P_PRECO
FROM    PRODUTO
ORDER BY P_PRECO
```

	P_COD	P_DESCRICAO	P_DATA	P_PRECO
1	F345677	Fone de ouvido sem fio	2020-08-29	49.00
2	F345673	Fone de ouvido bluetooth	2020-08-22	69.00
3	M2345	Mouse Microsoft	2020-08-22	199.00
4	C1234	Caneta digital	2020-08-09	199.00
5	T121345	Teclado sem fio	2020-08-22	299.00
6	C11CG	Multifuncional Epson Tanque Tinta	2020-10-08	349.00
7	20001	Notebook 2 em 1 Dell Inspiron	2020-06-01	4999.00
8	200Q/1	Notebook 2 em 1 Dell Inspiron	2020-06-06	5000.00

Figura 56 – Resultado de dados por ordenação crescente de P\_PRECO

As listagens ordenadas são utilizadas com frequência. Suponha, por exemplo, que se queira criar um diretório de CEP. Seria útil poder produzir uma sequência ordenada (sobrenome, nome, inicial) em três estágios:

- 1) ORDER BY sobrenome.
- 2) No interior dos sobrenomes ORDER BY nome.
- 3) No interior dos nomes e sobrenomes ORDER BY inicial do meio.

Essa sequência ordenada em vários níveis é conhecida como sequência de ordem em cascata e pode ser criada facilmente, listando diversos atributos separados por vírgula após a cláusula ORDER BY.



## Lembrete

Se a coluna que está sendo ordenada tiver nulos, eles são listados no início ou no fim, dependendo do SGBDR. A cláusula ORDER BY sempre deve ser listada por último na sequência de comandos SELECT.

## Listando valores únicos

Quanto fornecedores diferentes são representados atualmente pela tabela PRODUTO? Uma listagem simples (SELECT) não é muito útil se a tabela contiver milhares de linhas, sendo necessário selecionar manualmente os códigos de fornecedor. Felizmente, a cláusula de SQL DISTINCT produz uma linha que contém apenas os valores diferentes uns dos outros. Exemplo:

```
SELECT DISTINCT F_COD
FROM   PRODUTO
```

## Funções de agregação

A SQL pode executar vários resumos matemáticos, como contagem do número de linhas que apresentem uma condição específica, obtenção dos valores máximo e mínimo de um determinado atributo, soma e média dos valores de uma coluna escolhida.

Para ilustrar outros formatos – padrão de comando SQL, a maioria das sequências de entrada e saída será apresentada nos próximos comandos.

**Quadro 16 – Funções de agregação básica**

FUNÇÃO	RESULTADO
COUNT	Número de linhas que contem valores não nulos
MIN	Valor mínimo do atributo encontrado em determinada coluna
MAX	Valor máximo do atributo encontrado em determinada coluna
SUM	Soma de todos os valores de determinada coluna
AVG	Média aritmética de uma coluna

Fonte: Rob e Coronel (2011, p. 286).

## COUNT

A função **COUNT** é utilizada para contar os números de valores não nulos de um atributo. Ela pode ser utilizada como a cláusula DISTINCT. Utilize-a caso queira obter quantos fornecedores diferentes há na tabela PRODUTO. A resposta gerada pelo primeiro conjunto de códigos de SQL é apresentada a seguir:

```
SELECT COUNT (DISTINCT F_COD)
FROM      PRODUTO
```

-----

2 RESULTADOS

```
SELECT COUNT (DISTINCT F_COD)
FROM      PRODUTO
WHERE     P_PRECO <=5000;
```

-----

2 RESULTADOS

```
SELECT COUNT (*)
FROM      PRODUTO
WHERE     P_PRECO <=5000;
```

-----

8 RESULTADOS

É possível combinar as funções agregadas com os comandos SQL. Por exemplo, o segundo conjunto de comandos pode dar resposta à seguinte pergunta: "Quantos fornecedores referenciados em uma tabela PRODUTO supriram produtos com preços menores ou iguais a R\$ 5.000,00?". A resposta é 2, o que indica que 2 fornecedores referenciados na tabela PRODUTO supriram produtos que atendam às especificações de preço.

A função agregada COUNT utiliza um parâmetro entre parentêses, geralmente um nome de coluna, como COUNT (F\_COD) ou (P\_COD). O parâmetro também pode ser uma expressão do tipo COUNT (DISTINCT F\_COD) ou COUNT (P\_PRECO+10). Utilizando essa sintaxe, COUNT sempre retorna um número de valores não nulos na coluna determinada. (É diferente que os valores da coluna sejam computados ou apresentem valores de linhas armazenados.) Por outro lado, a sintaxe COUNT (\*) retorna o valor de linhas totais encontradas pela consulta, incluindo as com nulos. No exemplo, SELECT COUNT(P\_COD) FROM PRODUTO e SELECT COUNT (\*) FROM PRODUTO reproduzirá a mesma resposta, pois não há valores nulos na coluna chave primária P\_COD.

### MAX e MIN

As funções MAX e MIN ajudam a encontrar respostas para problemas como:

- O maior (máximo) preço da tabela PRODUTO.
- O menor (mínimo) preço da tabela PRODUTO.

O maior preço, R\$ 5.000,00, é fornecido pelo primeiro conjunto de comandos de SQL. O segundo conjunto resulta no preço mínimo.

O terceiro mostra como as funções numéricas podem ser utilizadas em conjunto com consultas mais complexas. Lembrando que as funções numéricas produzem apenas um valor com base em todos os

valores encontrados na tabela: o único valor máximo, o único valor mínimo, uma única contagem ou um único valor médio ponto. É comum negligenciar essa advertência. Por exemplo, examine a questão: "Qual o produto possui maior valor?".

Embora sua consulta pareça simples, a sequência de comandos SQL não produz os resultados esperados. Isso ocorre por causa da utilização de MAX (P\_PRECO) ao lado direito de um operador de comparação incorreto, produzindo assim uma mensagem de erro. A função de agregação MAX (nome da coluna) pode ser utilizada apenas na lista de colunas de um comando SELECT. Além disso, em uma comparação que utilize o símbolo de igualdade, é possível utilizar apenas um valor do lado direito do sinal de igual.

Portanto, para responder à pergunta, deve-se computar primeiro o preço máximo e, em seguida, compará-lo a cada preço retornado pela consulta de volta na mesma linha. Para fazer isso, é necessária uma consulta integrada, conforme segue:

- Consulta interna, executada primeiro.
- Consulta externa, executada por último.

```
SELECT MAX (P_PRECO)
FROM PRODUTO
```

-----

1 RESULTADOS

```
SELECT MIN (P_PRECO)
FROM PRODUTO
```

-----

1 RESULTADOS

```
SELECT P_COD, P_DESCRICAO, P_PRECO
FROM PRODUTO
WHERE P_PRECO = (SELECT MAX (P_PRECO) FROM PRODUTO)
```

-----

1 RESULTADOS

Usando essa sequência de comandos, como no exemplo, a consulta interna encontra primeiro o valor máximo, que é armazenado em uma memória. Como a consulta externa agora dispõe de um valor para comparar cada P\_PRECO, a consulta é executada imediatamente e adequadamente.

## SUM

A função SUM calcula a soma do total de um atributo utilizando quaisquer condições impostas. Por exemplo: para calcular a quantidade total devida pelos clientes pode-se usar o comando a seguir:

```
SELECT SUM (CLI_SALDO) AS TOTSALDO
FROM CLIENTE
```

Também é possível calcular a soma total de uma expressão. Por exemplo, para encontrar o valor de todos os itens mantidos em estoque, utiliza-se o comando a seguir:

```
SELECT SUM (P_UNIDISP* P_PRECO) AS TOTSALDO
FROM PRODUTO
```

### AVG

A função AVG é similar ao MIN e MAX e está sujeita às mesmas restrições de operação. O primeiro conjunto de comandos SQL mostra como o simples valor médio P\_PRECO pode ser gerado para obter o valor médio calculado. O segundo conjunto produz um resultado de 5 linhas, que descreve produtos cujos preços excedem o preço médio. A segunda sequência de linhas utiliza comandos de SQL integrada à cláusula ORDER BY.

```
SELECT AVG (P_PRECO)
FROM PRODUTO
```

-----  
1 RESULTADOS

```
SELECT P_COD, P_PRECO, P_DESCRICAO, P_UNIDISP, F_COD
FROM PRODUTO
WHERE P_PRECO > (SELECT AVG (P_PRECO) FROM PRODUTO)
ORDER BY P_PRECO DESC;
```

-----  
2 RESULTADOS

### Agrupando dados

As distribuições de frequência podem ser criadas de modo rápido e fácil utilizando a cláusula **GROUP BY** no comando SQL. A sintaxe é esta:

SELECT <campo1>, <campo2>, <campo n>, <função agregada (campo)>

FROM <tabela> [WHERE conditions]

GROUP BY <campo1>, <campo2>, <campo n>;

A cláusula **GROUP BY** é válida apenas quando utilizada em conjunto com uma das funções agregadas de SQL, como **COUNT**, **MAX**, **MIN**, **AVG** e **SUM**. Veja o exemplo a seguir:

```
SELECT F_COD, P_COD, P_DESCRICAO, P_PRECO
FROM PRODUTO
GROUP BY F_COD
```

Será gerado um erro "not a GROUP BY expression" (não é uma expressão GROUP BY). No entanto, escrevendo a sequência de comandos SQL precedente com a função agregada, a cláusula GROUP BY funciona adequadamente.

```
SELECT      F_COD, COUNT ( DISTINCT (P_COD) )
FROM        PRODUTO
GROUP BY    F_COD
```

	F_COD	(Nenhum nome de coluna)
1	10011	7
2	10012	1

Figura 57 – Resultado de dados com a cláusula COUNT

### Cláusula HAVING do recurso GROUP BY

Uma extensão especialmente útil do recurso GROUP BY é a cláusula HAVING. Ela opera de modo muito parecido com a cláusula WHERE. No entanto, WHERE aplica-se à coluna expressa nas linhas individuais, enquanto HAVING é aplicada ao resultado de uma operação GROUP BY. Pode ser utilizada, por exemplo, se a intenção for gerar uma listagem de número de produtos em estoque entregues pelo fornecedor, mas, desta vez, limitando a listagem de produtos cujo preço tem média inferior a R\$ 100,00. A primeira parte exigida pode ser satisfeita com a ajuda de uma cláusula GROUP BY. Observe que a cláusula HAVING é utilizada com a cláusula GROUP BY gerando o resultado desejado.

Se for utilizada a cláusula WHERE em vez de HAVING, o segundo conjunto de comando produzirá um erro.

Também é possível combinar várias cláusulas e funções agregadas. Considere os seguintes comandos:

```
SELECT      F_COD, SUM ( P_UNIDISP *P_PRECO) AS TOTALCOTACAO
FROM        PRODUTO
GROUP BY    F_COD
HAVING      (SUM ( P_UNIDISP *P_PRECO)>500)
ORDER BY    SUM ( P_UNIDISP *P_PRECO) DESC;
```

	F_COD	TOTALCOTACAO
1	10011	263136.00
2	10012	597.00

Figura 58 – Resultado de dados com a cláusula SUM



Esse comando fará o seguinte:

- Agregar o custo total de produtos agrupados por F\_COD.
- Selecionar apenas as linhas que possuam totais superiores a R\$ 100,00.
- Listar os resultados em ordem decrescente por custo total.

Na sintaxe utilizada nas cláusulas HAVING e ORDER BY, deve-se especificar a expressão (fórmula) da coluna utilizada na lista do comando SELECT, e não o alias da coluna (TOTALCOTACAO).

### 4.9 Junções: consultas agregadas de tabelas

As junções (*joins*) permitem a seleção de dados de duas ou mais tabelas em uma única instrução. Nesse tipo de operação, as tabelas devem possuir relacionamentos entre si. Isso só será possível se a chave estrangeira de uma tabela tiver como referência a chave primária da tabela precedente.

Sua criação traz grandes dificuldades aos iniciantes na linguagem SQL, e sua construção de consultas só será possível com junções entre tabelas, um recurso fundamental para visualizar os dados de um banco relacional.

Existem vários tipos de *joins* e cada um tem um determinado propósito.

#### INNER JOIN

Também chamado de junção de igualdade, este é um dos métodos de junção mais conhecidos. Utilizado sempre que houver necessidade de unirmos tabelas por um campo em comum, sendo que o valor do campo deve existir em todas as tabelas envolvidas no *join*. Retorna os registros que são comuns às duas tabelas.

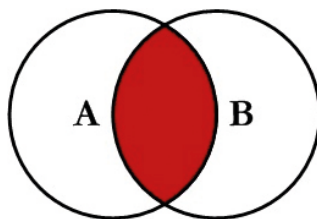


Figura 59 – INNER JOIN

Entenda a sintaxe básica desse comando:

```
SELECT <lista de campos> from [nome da tabela à esquerda do join]
```

```
INNER JOIN [nome da tabela à direita do join]
```

ON [nome da tabela à esquerda do *join*].[coluna de relacionamento] =

[nome da tabela à direita do *join*].[coluna de relacionamento]

Analise a estrutura das tabelas a seguir:

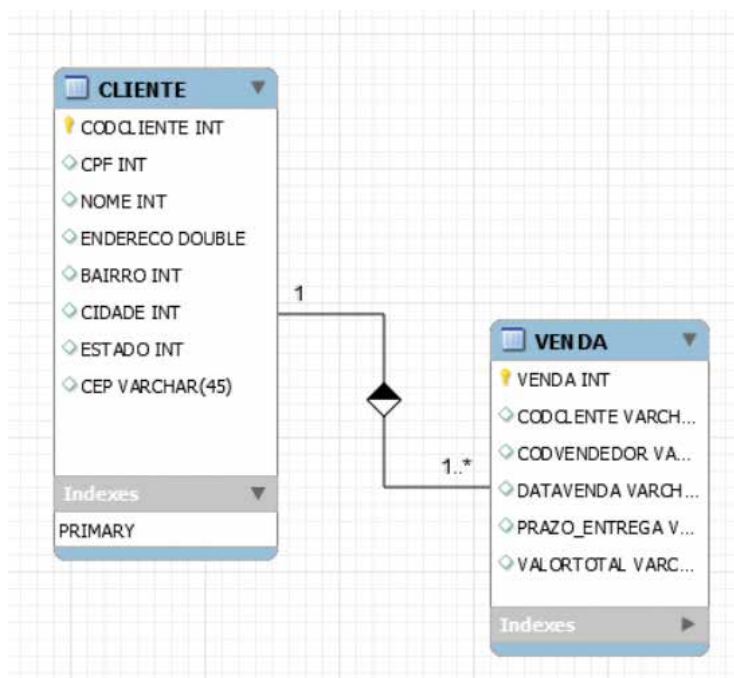


Figura 60 – INNER JOIN entre as tabelas CLIENTE e VENDA

Visualizando as tabelas, percebe-se que possuem uma coluna de relacionamento: CODCLIENTE. Temos 5 clientes cadastrados na tabela e nem todos tiveram vendas realizadas.

```
SELECT * FROM CLIENTE;
SELECT * FROM VENDA;
```

Vamos construir uma consulta com comandos SQL que retorna o CODVENDA, VALORTOTAL e o NOME do cliente. Observe que as informações estão em tabelas separadas.

```
SELECT V.CODVENDA, V.CODCLIENTE, C.NOME, V.VALORTOTAL FROM VENDA V
INNER JOIN CLIENTE C ON V.CODCLIENTE = C.CODCLIENTE;
```

Verifique que apenas os clientes que tiveram vendas realizadas apareceram no resultado apresentado.

Execute a instrução a seguir:

```
SELECT V.CODVENDA, V.CODCLIENTE, C.NOME, V.VALORTOTAL, V.PRAZO_ENTREGA, C.ESTADO
FROM VENDA V
INNER JOIN CLIENTE C ON V.CODCLIENTE = C.CODCLIENTE
WHERE V.PRAZO_ENTREGA >= 5 AND V.PRAZO_ENTREGA <= 10 AND C.ESTADO = 'SP';
```

A instrução é muito semelhante à que foi executada anteriormente, porém utiliza alguns filtros com campos constantes nas duas tabelas. Verifique que somente as vendas com prazo de entrega entre 5 e 10 dos clientes do estado de SP aparecem.

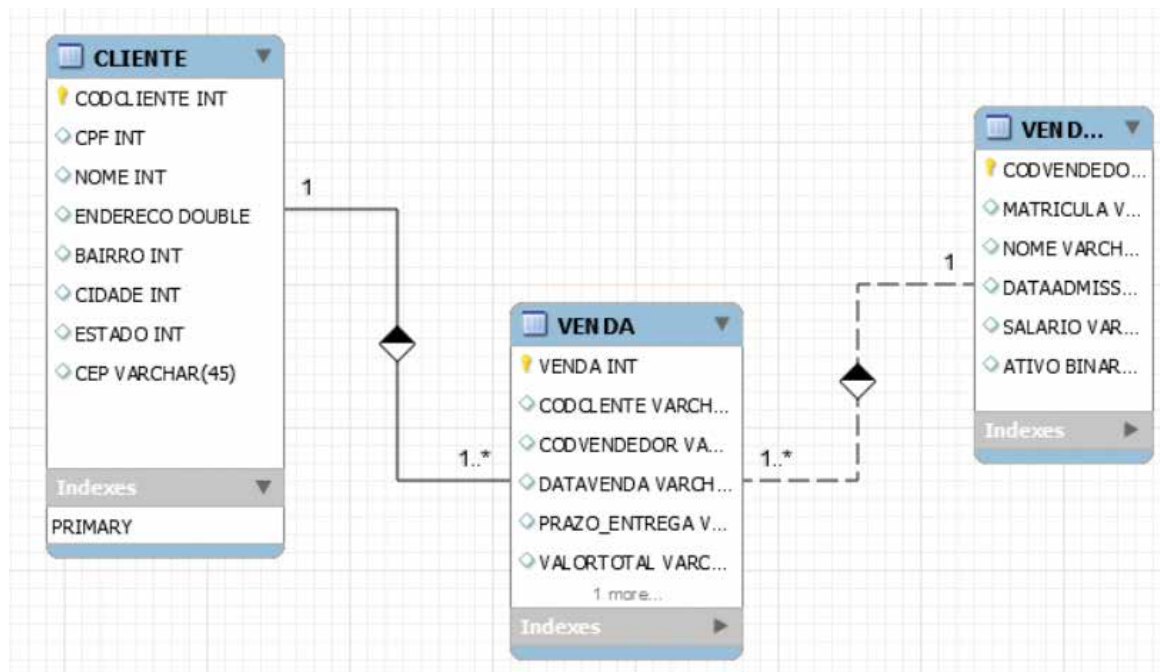


Figura 61 – INNER JOIN entre as tabelas CLIENTE e VENDA e VENDEDORES

Analise o modelo anterior e verifique que a tabela VENDA possui relacionamentos com as tabelas CLIENTE e VENDEDORES. Vamos construir agora uma consulta com comandos SQL que apresentem os seguintes dados: código da venda, valor total, nome do cliente e nome do vendedor.

```
SELECT V.CODVENDA, V.CODCLIENTE, C.NOME AS NOMECLIENTE, V.VALORTOTAL, VE.NOME AS NOMEVENDEDOR
FROM VENDA V INNER JOIN CLIENTE C ON V.CODCLIENTE = C.CODIGOCLIENTE
INNER JOIN VENDEDOR VE ON V.CODVENDEDOR = VE.CODVENDEDOR;
```

### LEFT OUTER JOIN

É um tipo de *join* que permite o retorno de todos os registros da tabela relacionada à esquerda da junção, mesmo que não tenha registro correspondente na tabela da direita.

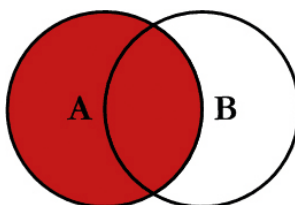


Figura 62 – LEFT OUTER JOIN

Sintaxe básica:

SELECT <lista de campos> from [nome da tabela à esquerda do *join*]

LEFT JOIN [nome da tabela à direita do *join*]

ON [nome da tabela à esquerda do *join*].[coluna de relacionamento] =  
[nome da tabela à direita do *join*].[coluna de relacionamento]

Observe as tabelas do modelo a seguir:

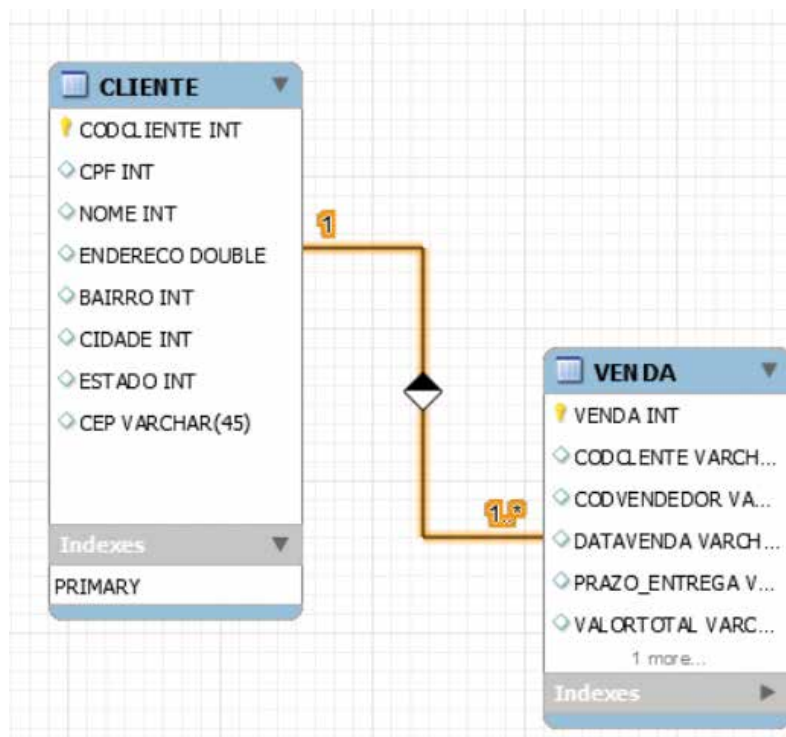


Figura 63 – LEFT OUTER JOIN entre as tabelas CLIENTE e VENDA

Verifique a seguir que temos 10 clientes cadastrados:

```
SELECT * FROM CLIENTE;
```

Dos 10 clientes, apenas 6 estão presentes nas vendas.

```
SELECT * FROM VENDA;
```

Quando executamos o INNER JOIN, foram exibidas apenas as linhas que possuem relacionamentos entre as tabelas. Verifique agora que vamos exibir os dados de todos os clientes, mesmo que não tenham vendas relacionadas. Isso só é possível se utilizarmos o LEFT JOIN. No exemplo a seguir, a tabela CLIENTE é a tabela à esquerda do *join* e devido a isso visualizamos os dados de todos os clientes.

Tabela à esquerda		Tabela à direita
CLIENTE C	LEFT JOIN	VENDA V

```
SELECT V.CODVENDA, V.CODCLIENTE, C.NOME, V.VALORTOTAL, V.PRAZO_ENTREGA, C.ESTADO  
FROM CLIENTE C  
LEFT JOIN VENDA V ON C.CODCLIENTE = V.CODCLIENTE;
```

Vamos agora exibir os dados dos vendedores e, caso tenham vendas relacionadas, visualizaremos os dados da venda:

```
SELECT V.CODVENDA, V.CODVENDEDOR, VE.NOME AS NOMEVENDEDOR, V.VALORTOTAL,  
V.PRAZO_ENTREGA FROM VENDEDOR VE  
LEFT JOIN VENDA V ON VE.CODVENDEDOR = V.CODVENDEDOR;
```

### RIGHT OUTER JOIN

É um tipo de *join* que permite o retorno de todos os registros da tabela relacionada à direita da junção, mesmo que não tenha registro correspondente na tabela da esquerda.

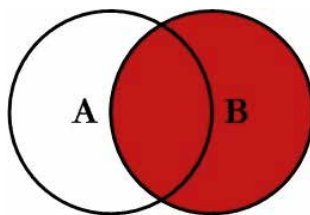


Figura 64 – RIGHT OUTER JOIN

Sintaxe básica:

SELECT <lista de campos> from [nome da tabela à esquerda do *join*]

RIGHT JOIN [nome da tabela à direita do *join*]

ON [nome da tabela à esquerda do *join*].[coluna de relacionamento] =

[nome da tabela à direita do *join*].[coluna de relacionamento]

Observe o modelo a seguir:

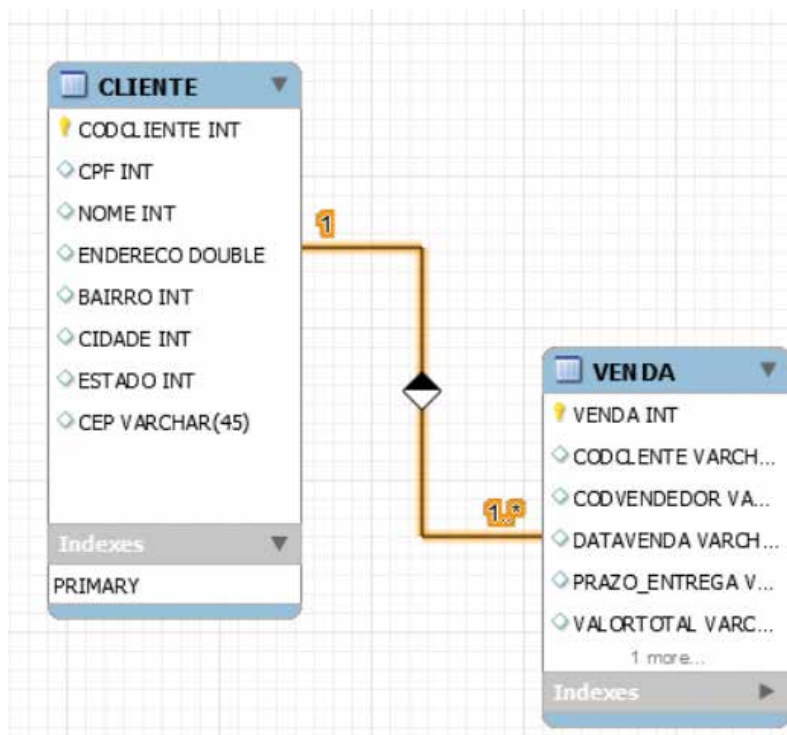


Figura 65 – RIGHT OUTER JOIN entre tabela CLIENTE e VENDA

O modelo apresentado é semelhante ao modelo utilizado no exemplo de LEFT JOIN. Porém, a tabela CLIENTE será a tabela que fica à direita do *join*.

A instrução a seguir seleciona os dados da tabela CLIENTE, mesmo que estes não possuam vendas relacionadas.

```
SELECT V.CODVENDA, V.CODCLIENTE, C.NOME, V.VALORTOTAL, V.PRAZO_ENTREGA, C.ESTADO
FROM VENDA V
RIGHT JOIN CLIENTE C ON V.CODCLIENTE = C.CODCLIENTE;
```

Agora, vamos selecionar os dados do fornecedor e, se ele possuir compras relacionadas, informaremos o código da compra e o valor total.

```
SELECT FO.CODFORNECEDOR, FO.NOME AS NOMEFORNECEDOR, FO.CIDADE, CO.CODCOMPRA,
CO.DATACOMPRA, CO.VALORTOTAL
FROM COMPRA CO RIGHT JOIN FORNECEDOR FO ON CO.CODFORNECEDOR = FO.CODFORNECEDOR;
```

Verifique que os fornecedores com código 6, 7, 9 e 10 não possuem compras relacionadas.

## FULL OUTER JOIN

O FULL OUTER JOIN (também conhecido por *outer join* ou *full join*), conforme vemos a seguir, tem como resultado todos os registros que estão na tabela A e todos os registros da tabela B.

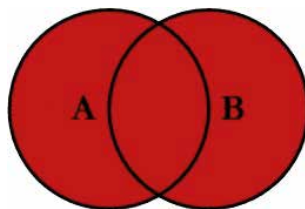


Figura 66 – FULL OUTER JOIN

Sintaxe básica:

SELECT <lista de campos> from [nome da tabela à esquerda do *join*]

FULL OUTER JOIN [nome da tabela à direita do *join*]

ON [nome da tabela à esquerda do *join*].[coluna de relacionamento] =

[nome da tabela à direita do *join*].[coluna de relacionamento]

```
SELECT FO.CODFORNECEDOR, FO.NOME AS NOMEFORNECEDOR, FO.CIDADE, CO.CODCOMPRA,  
CO.DATACOMPRA, CO.VALORTOTAL  
FROM COMPRA CO FULL OUTER JOIN FORNECEDOR FO ON CO.CODFORNECEDOR =  
FO.CODFORNECEDOR;
```

Verifique, a seguir, que foram apresentados os dados das duas tabelas envolvidas no JOIN.

### LEFT EXCLUDING JOIN e RIGHT EXCLUDING JOIN

Podemos exibir também apenas os registros que estão na tabela A e que não estejam na tabela B ou que estejam na tabela B e não estejam na tabela A.

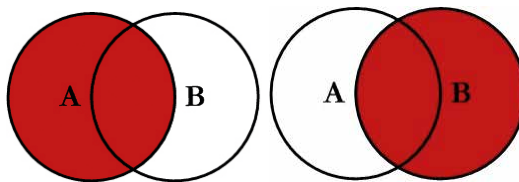


Figura 67 – LEFT EXCLUDING JOIN e RIGHT EXCLUDING JOIN

Neste caso, basta acrescentar a instrução à cláusula WHERE e pedir para filtrar os registros nulos.

```
SELECT V.CODVENDA, V.CODCLIENTE, C.NOME, V.VALORTOTAL, V.PRAZO_ENTREGA, C.ESTADO  
FROM CLIENTE C  
LEFT JOIN VENDA V ON C.CODCLIENTE = V.CODCLIENTE WHERE V.CODVENDA IS NULL;
```

## OUTER EXCLUDING JOIN

Usando o OUTER EXCLUDING JOIN, conforme vemos a seguir, teremos como resultado todos os registros que estão na tabela B, mas que não estejam na tabela A, e todos os registros que estão na tabela A, mas que não estejam na tabela B.

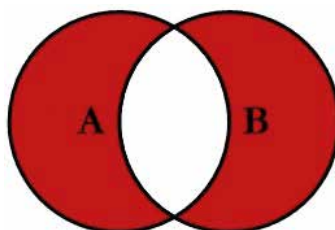


Figura 68 – OUTER EXCLUDING JOIN

Execute as instruções a seguir:

```
USE MODELOBDNOVO;  
GO  
UPDATE PEDIDOS SET CodFuncionario = NULL WHERE NumeroDoPedido = 12064;
```

Na instrução anterior, estamos alterando o funcionário do pedido 12064 colocando no lugar valor nulo.

Veja o exemplo a seguir:

```
SELECT F.CODFUNCIONARIO, F.NOME, P.NUMERODOPEDIDO, P.FRETE  
FROM Pedidos P  
FULL OUTER JOIN Funcionarios F ON F.CodFuncionario = P.CodFuncionario  
WHERE F.CodFuncionario is null or P.CodFuncionario is null;
```

Nesse exemplo, estamos visualizando todos os funcionários que não possuem pedidos e todos os pedidos que não possuem um funcionário atribuído a ele.

## Visões (Views)

Views são objetos do banco de dados que retornam dados através de Query's (SELECT), gerando assim tabelas virtuais compostas de linhas e colunas. Essas tabelas são geradas, dinamicamente, no momento em que fazemos referência a elas.

As views podem conter uma ou mais tabelas ou até mesmo outras views. Como o próprio nome diz, são uma visão de dados e não contêm dados.

Podemos utilizar views para alterar dados, porém com algumas restrições. As views não ocupam espaço no banco de dados, pois trabalham com os dados das tabelas já existentes. Ao criarmos uma



*view*, podemos filtrar o conteúdo a ser exibido, filtrar tabelas, agrupá-las, proteger certas colunas e simplificar o código de programação.

São muito utilizadas quando necessitamos de comandos SELECT complexos. Ficam permanentes em nosso banco de dados e podem ser lidas simultaneamente por vários usuários.

As cláusulas SELECT em uma definição de exibição não podem incluir o seguinte:

- Uma cláusula ORDER BY, a não ser que exista também uma cláusula TOP na lista de seleção da instrução SELECT.
- A palavra-chave INTO.
- A cláusula OPTION.
- Uma referência para uma tabela temporária ou uma variável de tabela.

### 4.10 Criação de visões

Sintaxe básica:

```
CREATE VIEW <nome da view>
```

```
AS
```

```
<instruções SELECT>
```

Exemplo:

Acesse o banco de dados MODELOBDNOVO. Em seguida visualize os dados da tabela PRODUTOS.

```
USE MODELOBDNOVO;  
GO  
SELECT * FROM PRODUTOS;
```

Todos os produtos possuem a quantidade de unidades existentes em estoque. Vamos criar uma *view* que permite visualizar apenas os produtos com estoque igual a zero.

```
CREATE VIEW VW_ESTOQUE  
AS  
SELECT * FROM PRODUTOS WHERE UnidadesEmEstoque = 0;
```

Para visualizar os dados da *view*, basta executar a instrução SELECT fazendo referência a ela:

```
SELECT * FROM VW_ESTOQUE;
```

No exemplo a seguir, selecionaremos dados das tabelas CLIENTE e VENDA. Nesse caso, vamos utilizar os comandos de *join* dentro da *view*.

```
USE MODELOBDNOVO;
GO
CREATE VIEW VW_VENDAS
AS
SELECT C.CODCLIENTE, C.NOME, V.VALORTOTAL FROM CLIENTE C
INNER JOIN VENDA V ON C.CODCLIENTE = V.CODCLIENTE
WHERE V.VALORTOTAL > 1000;
```

Verifique os dados exibidos pela *view*:

```
SELECT * FROM VW_VENDAS;
```

Agora, vamos criar uma visão que retorna os dados de todos os clientes que nunca compraram, isto é, que não constam em nenhuma venda.

```
USE MODELOBDNOVO;
GO

CREATE VIEW VW_VENDASNULAS
AS
SELECT V.CODVENDA, C.CODCLIENTE, C.NOME, V.VALORTOTAL, V.PRAZO_ENTREGA, C.ESTADO
FROM CLIENTE C
LEFT JOIN VENDA V ON C.CODCLIENTE = V.CODCLIENTE WHERE V.CODVENDA IS NULL;
SELECT * FROM VW_VENDASNULAS
```

### Atualização de dados utilizando visões

Existem algumas limitações no uso de *views* para atualização das tabelas subjacentes de uma *view*:

- Apenas uma única tabela base em uma *view* pode ser atualizada.
- Colunas sendo atualizadas devem ser diretamente referenciadas na *view*, sem qualquer cálculo sobre elas.
- Colunas sendo modificadas não devem ser afetadas por um GROUP BY, DISTINCT, ou cláusula HAVING.
- Sua *view* não contém uma cláusula TOP, quando o CHECK OPTION é utilizado.

No exemplo anterior, criamos a visão VW\_VENDASNULAS. Podemos alterar dados de uma *view* através do comando UPDATE:

```
UPDATE VW_VENDASNULAS SET NOME = 'ANTONIO CESAR' WHERE CODCLIENTE = 6;
```

Verifique que os dados foram alterados na *view* e na tabela CLIENTE:

```
SELECT * FROM VW_VENDASNULAS  
SELECT * FROM CLIENTE;
```

O mesmo não acontece se tentarmos efetuar a exclusão do cliente com código igual a 6:

```
DELETE VW_VENDASNULAS WHERE CODIGOCLIENTE = 6;
```

Verifique a seguir a mensagem exibida pelo sistema:

Mensagem 4405, Nível 16, Estado 1, Linha 43

A exibição ou a função 'VW\_VENDASNULAS' não é atualizável porque a modificação afeta várias tabelas base.

As alterações em *views* podem ser afetadas quando utilizamos a cláusula WITH CHECK OPTION. Utilizando essa opção, as atualizações ou inserções em *views* só serão permitidas se os dados alterados são visualizados através da *view*.

### Alteração de visões

Sintaxe básica:

```
ALTER VIEW <nome da view>
```

```
AS
```

```
<instruções SELECT>
```

Veja o exemplo a seguir, nele estamos alterando a *view* criada anteriormente:

```
USE MODELOBDNOVO;  
GO  
ALTER VIEW VW_ESTOQUE  
AS  
SELECT * FROM PRODUTOS WHERE UnidadesEmEstoque = 0 AND CODDACATEGORIA = 6;
```

### Exclusão de visões

Sintaxe básica:

```
DROP VIEW <nome da view>
```

Exemplo:

A instrução a seguir apaga a view VW\_ESTOQUE:

```
DROP VIEW VW_ESTOQUE;
```

### 4.11 Introdução a funções: funções definidas pelo usuário

Assim como as funções em linguagens de programação, as funções do SQL Server definidas pelo usuário são rotinas que aceitam parâmetros, executam uma ação, como um cálculo complexo, e retornam o resultado dessa ação como um valor. O valor de retorno pode ser um único valor escalar ou um conjunto de resultados.

Funções definidas pelo usuário:

- **Permitem programação modular:** você pode criar a função uma vez, armazená-la no banco de dados e chamá-la quantas vezes quiser em seu programa. Funções definidas pelo usuário podem ser modificadas independentemente do código-fonte do programa.
- **Permitem execução mais rápida:** semelhantemente aos procedimentos armazenados, as funções definidas pelo usuário reduzem o custo de compilação do código Transact-SQL colocando os planos em cache e reusando-os para execuções repetidas. Isso significa que a função definida pelo usuário não precisa ser reanalisada e reotimizada em cada utilização, resultando em tempos de execução mais rápidos.
- **Podem reduzir o tráfego de rede:** uma operação que filtra dados com base em alguma restrição complexa que não pode ser expressa em uma única expressão escalar pode ser expressa como uma função. Em seguida, a função pode ser invocada na cláusula WHERE para reduzir o número ou linhas enviadas ao cliente.

#### 4.11.1 Tipos de funções

##### Função escalar

As funções escalares definidas pelo usuário retornam um valor único de dados do tipo definido na cláusula RETURNS. Para uma função escalar embutida, não há um corpo de função; o valor escalar é o resultado de uma única instrução. Para uma função escalar de várias instruções, o corpo da função, definido em um bloco BEGIN...END, contém uma série de instruções Transact-SQL, que retornam o valor único. O tipo de retorno pode ser qualquer tipo de dados, exceto **text**, **ntext**, **image**, **cursor** e **timestamp**.

### Funções com valor de tabela

As funções com valor de tabela definidas pelo usuário retornam um tipo de dados **table**. Para uma função com valor de tabela embutida, não há um corpo de função; a tabela é o conjunto de resultados de uma única instrução SELECT.

### Funções com valor de tabela (multitabelas)

Funções com valor de tabela (multitabelas) definem explicitamente a estrutura da tabela que será retornada, com seus nomes de colunas e seus tipos de dados. Essas funções são utilizadas quando não podemos gerar o retorno a partir de um único SELECT, sendo necessário executar 2 ou mais SELECTs em sequência para gerar os dados que serão retornados.

### Funções do sistema

O SQL Server fornece muitas funções de sistema que você pode usar para executar uma variedade de operações. Elas não podem ser modificadas.

### Instruções válidas em uma função

Os tipos de instruções que são válidos em uma função incluem:

- As instruções DECLARE podem ser usadas para definir variáveis de dados e cursores que são locais à função.
- A atribuição de valores a objetos locais à função, como o uso de SET para atribuir valores para escalar e para as variáveis locais à tabela.
- As operações de cursor que referenciam cursores locais são declaradas, abertas, fechadas e desalocadas na função. As instruções FETCH que retornam os dados aos clientes não são permitidas. Somente instruções FETCH que atribuem valores a variáveis locais usando a cláusula INTO são permitidas. [...]
- Instruções SELECT com listas de seleção com expressões que atribuem valores às variáveis que são locais à função.
- Instruções UPDATE, INSERT e DELETE que modificam variáveis de tabela locais à função.
- Instruções EXECUTE que chamam um procedimento armazenado estendido (MICROSOFT, 2016).

No exemplo a seguir, vamos criar uma função que retorna valores de uma tabela. Na execução da função, devemos informar o valor que será utilizado como parâmetro:

```
USE MODELOBDNOVO;

--Criando função que retorna dados de tabelas

Create Function F_Cidade (@NomeCidade VarChar(100))
Returns Table
As
Return(Select * from pedidos Where CidadeDeDestino = @NomeCidade)
```

Para a execução da função, devemos informar o nome da cidade como parâmetro:

```
Select * from F_Cidade('Sao Paulo')
```

Agora, vamos criar a função onde informamos o número do pedido e o sistema retorna o imposto (10%) relacionado ao frete do pedido.

```
CREATE FUNCTION F_CalculaImposto(@pedido decimal(10,2))
RETURNS TABLE
AS
RETURN
    SELECT NumeroDoPedido, frete, frete * 0.1 as imposto from Pedidos
    WHERE NumeroDoPedido = @pedido

Select * from F_CalculaImposto(10250)
```

Dando continuidade ao aprendizado, vamos criar uma função escalar:

```
CREATE FUNCTION IDADE (@NASCIMENTO DATETIME,@DATA DATETIME)
RETURNS INT
AS
BEGIN
    DECLARE @IDADE INT
    SET @IDADE = YEAR(@DATA) - YEAR(@NASCIMENTO) - 1
    IF (MONTH(@DATA) - MONTH(@NASCIMENTO) > 0)
        BEGIN
            SET @IDADE = @IDADE + 1
        END
    ELSE
        BEGIN
            IF (MONTH(@DATA) - MONTH(@NASCIMENTO) = 0)
                IF (DAY(@DATA) - DAY(@NASCIMENTO) >= 0)
                    BEGIN
                        SET @IDADE = @IDADE + 1
                    END
                ELSE
                    BEGIN
                        SET @IDADE = @IDADE
                    END
            ELSE
                SET @IDADE = @IDADE
        END
    ELSE
        SET @IDADE = @IDADE
END
```

```

                                BEGIN
                                IF (MONTH (@DATA) - MONTH (@
NASCIMENTO) < 0)
                                BEGIN
                                    SET @IDADE = @IDADE
                                END
                                END
END
RETURN @IDADE
END

SELECT DBO.IDADE ('11/08/2000', GETDATE ());
```

### 4.12 Introdução a *stored procedures*: procedimentos armazenados

*Stored procedures* são objetos do banco de dados que contêm uma série de comandos SQL Padrão, a fim de facilitar e agilizar o trabalho com o banco. Podem ser de sistema ou criadas pelo usuário. Por exemplo, poderemos ter uma *stored procedure* (sp) para atualizar dados, outra para retornar valores, outra para deletar um determinado conjunto de dados etc. Os procedimentos armazenados em uma sp são pré-compilados, de maneira que sua execução, em comparação com a execução de comandos que realizem a mesma tarefa, é mais rápida.

São usadas tanto para obter dados como para modificá-los, mas não ambos na mesma sp. Sua sintaxe é verificada na primeira vez que são executadas, quando são compiladas e armazenadas em cache. Portanto, chamadas subsequentes a uma mesma sp serão ainda mais rápidas que a primeira. Podem ser utilizadas em mecanismos de segurança: uma pessoa poderá possuir direitos de execução de uma sp, mesmo não possuindo permissões sobre as tabelas e *views* que ela referencia.

Assim, por exemplo, poderíamos liberar o acesso a uma sp que calcula o total de salários de um determinado setor, pesquisando para isso todos os salários individuais desse setor, mas a pessoa que tivesse acesso à execução desta sp não teria acesso à tabela de salários. Como resultado, nosso usuário hipotético poderia conhecer o total de salários de cada departamento sem jamais ter contato com salários individuais.

A sp armazena tarefas repetitivas e aceita parâmetros de entrada para que a tarefa seja efetuada de acordo com a necessidade individual. São muito úteis para o dia a dia de DBAs e desenvolvedores. Uma sp pode reduzir o tráfego na rede, melhorar a performance de um banco de dados, criar tarefas agendadas, diminuir riscos, criar rotinas de processamento etc.

Entre os principais tipos de *procedures*, podemos citar:

- **Procedimentos locais:** são criados a partir de um banco de dados do próprio usuário.
- **Procedimentos temporários:** existem dois tipos de procedimentos temporários – locais, que devem começar com #, e globais, que devem começar com ##.

- **Procedimentos de sistema:** armazenados no banco de dados padrão do SQL Server (Master), podemos identificá-los com a sigla *sp*, que se origina de *stored procedure*. Tais *procedures* executam as tarefas administrativas as quais podem ser executadas a partir de qualquer banco de dados.

### Permissões, procedimentos e parâmetros

Para a criação e manipulação de *stored procedures* em um Servidor SQL Server, é necessária a permissão `CREATE PROCEDURE` no banco de dados e a permissão `ALTER` no esquema no qual o procedimento está sendo criado.

Sintaxe básica:

```
CREATE PROCEDURE ProcedureName  
(  
  Parâmetros  
)  
AS  
... INSTRUÇÕES SQL
```



### Observação

**Schema\_name** é o nome do esquema ao qual o procedimento pertence. Os procedimentos são associados a esquemas. Se não for especificado um nome de esquema quando o procedimento é criado, será atribuído automaticamente o esquema padrão do usuário que estiver criando o procedimento.

**Procedure\_name** é o nome do procedimento. Os nomes de procedimento devem estar de acordo com as regras para identificadores e devem ser exclusivos no esquema.

Evite o uso do prefixo `sp_` ao nomear procedimentos. Esse prefixo é usado pelo SQL Server para designar procedimentos de sistema. O uso do prefixo poderá causar a quebra do código do aplicativo se houver um procedimento de sistema com o mesmo nome.

Os procedimentos temporários locais ou globais podem ser criados com uma tecla de cerquilha (`#`) antes de *procedure\_name* (`#procedure_name`) para procedimentos temporários locais e duas teclas de cerquilha para procedimentos temporários globais (`##procedure_name`). Um procedimento temporário local é visível somente à conexão que o criou e é descartado quando essa conexão é fechada. Um procedimento temporário global fica disponível para todas as conexões e é descartado ao término da última sessão que usa o procedimento.



O nome completo de um procedimento ou um procedimento temporário global, incluindo ##, não pode exceder 128 caracteres. O nome completo de um procedimento temporário local, incluindo #, não pode exceder 116 caracteres.

**Parâmetros:** @ parameter é um parâmetro declarado no procedimento. Especifique um nome de parâmetro usando o sinal (@) como o primeiro caractere. Os parâmetros são locais para o procedimento; os mesmos nomes de parâmetro podem ser usados em outros procedimentos.

Podem ser declarados um ou mais parâmetros de entrada ou saída, sendo que o número máximo é 2.100. O valor de cada parâmetro declarado deve ser fornecido pelo usuário quando o procedimento é chamado, a menos que um valor padrão para o parâmetro seja especificado ou o valor seja definido como igual a outro parâmetro. Se o procedimento contiver parâmetros com valor de tabela e faltar um parâmetro na chamada, um padrão de tabela vazia será transmitido. Os parâmetros podem assumir apenas o lugar de expressões constantes. Eles não podem ser usados no lugar de nomes de tabela, nomes de coluna ou nomes de outros objetos de banco de dados.

A seguir, alguns exemplos de *stored procedures*. Para esses exemplos, vamos utilizar o BD criado anteriormente.

```
USE MODELOBDNOVO;
```

Verifique os dados da tabela Transportadoras:

```
SELECT * FROM Transportadoras;
```

Em seguida, criaremos uma *stored procedure* que ao ser executada retorna todas as fornecedoras ordenando os dados retornados pelo nome da empresa:

### **Stored procedure sem passagem de parâmetro – pesquisa de registros**

```
Create PROCEDURE pesq_ fornecedoras
AS
BEGIN
    select * from fornecedoras order by NomeDaEmpresa;
END
GO
```

Para executar a *stored procedure* criada, utilize o comando EXECUTE/EXEC, conforme segue:

```
EXECUTE pesq_ fornecedoras;
```

ou

```
EXEC pesq_ fornecedoras;
```

## Stored procedure com passagem de parâmetro – inserindo registros

Agora, vamos criar uma *stored procedure* que permite incluir dados na tabela fornecedoras:

```
CREATE PROCEDURE insere_ fornecedoras --- Declarando o nome da procedure
-- Declarando parâmetro/variável (utilizamos o @ antes do nome da variável)
@codigo          INT,
@nome            VARCHAR(60), -- cada parâmetro possui um tipo de dados
relacionado
@fone            VARCHAR(10)
AS
BEGIN -- início do bloco de comandos

    -- comandos transact sql
    INSERT INTO fornecedoras (codigoda fornecedora, nomeda empresa, telefone)
        VALUES      (@codigo, @nome, @fone)

END -- fim do bloco de comandos
GO
```

Verifique que a *stored procedure* criada possui três parâmetros que são necessários para a sua execução.

Execute a *procedure* insere\_ fornecedora, informando os parâmetros solicitados:

```
EXEC insere_ fornecedora 4, 'Sol do luar', '0012321212121';
```

Verifique os dados da tabela transportadora:

```
select * from fornecedoras;
```

## Stored procedure com passagem de parâmetro – alterando registros

Em seguida, vamos criar uma *stored procedure* que permite alterar dados na tabela fornecedoras:

```
create PROCEDURE altera_ fornecedoras
@codigo          INT,
@nome            VARCHAR(60),
@fone            VARCHAR(10)
AS
BEGIN
    UPDATE fornecedoras SET nomeda empresa = @NOME, telefone = @fone
        WHERE codigoda fornecedora = @codigo

END
GO
```

Execute a *procedure* altera\_ fornecedora, informando os parâmetros solicitados:

```
EXEC altera_ fornecedora 1, 'Xpress 1', '11989898989';
```

Verifique os dados da tabela transportadora e observe que foram alterados:

```
select * from fornecedoras;
```

No próximo exemplo, vamos criar uma *stored procedure* que permite excluir um registro da tabela fornecedoras, informando como parâmetro o código das fornecedoras.

### Stored procedure com passagem de parâmetro – excluindo registros

```
create PROCEDURE exclui_ fornecedora
@codigo          INT
AS
BEGIN

    DELETE Fornecedoras WHERE codigodaforneecedora = @codigo;

END
GO

EXEC exclui_ fornecedora 1
```

Verifique que a fornecedora com código igual a 1 foi excluída da tabela.

Vimos que a implementação de *stored procedure* busca efetuar pesquisas, inclusão, alteração e exclusão de registros da tabela fornecedora. Podemos simplificar o processo construindo apenas uma *stored procedure* que possibilita executar todas as instruções conforme o parâmetro informado.

Para isso, adicionaremos um novo parâmetro, informando a operação a ser realizada.

```
CREATE PROCEDURE crud_ fornecedora
@codigo          INT,
@nome            VARCHAR(60),
@fone            VARCHAR(10),
@operacao        CHAR(1)
AS
BEGIN
IF @operacao = 'p'
    begin
        select * from Fornecedoras where codigodaforneecedora = @codigo
        order by NomeDaEmpresa;
    end
ELSE
    IF @operacao = 'i'
        begin
            INSERT INTO Fornecedor (codigodaforneecedora , nomedaempresa,
            telefone)
                VALUES (@codigo, @nome, @fone);
        end
    ELSE
        IF @operacao = 'a'
            begin
```

```
UPDATE Fornecedora SET nomedaempresa = @NOME,
telefone = @fone
WHERE codigodafofnecedora = @codigo;
end
ELSE
IF @operacao = 'e'
begin
DELETE Fornecedora WHERE codigodafofnecedora = @
codigo;
end
END;
```

Em seguida, executamos a *stored procedure crud*, realizando as operações de pesquisa (p), inclusão (i), alteração (a) e exclusão (e) de registro. Para a execução, todos os parâmetros são necessários.

A cada operação realizada a seguir, verifique a alteração da tabela com a instrução SELECT.

```
exec crud_fornecedora 2, '', '', 'p'
exec crud_fornecedora 1, 'XPRESS1', '123456789', 'I'
exec crud_fornecedora 4, 'XPRESS4', '545456789', 'I'
exec crud_fornecedora 4, 'XPRESS4 NEW', '545456789', 'A'
exec crud_fornecedora 4, '', '', 'E'
```

### 4.13 Gatilhos (TRIGGERS)

Um TRIGGER é um tipo especial de procedimento armazenado, que é executado sempre que há uma tentativa de modificar os dados de uma tabela que é protegida por ele.

Associados a uma tabela, os TRIGGERS são definidos em uma tabela específica, que é denominada tabela de TRIGGERS, e são executados automaticamente quando há uma tentativa de inserir, atualizar ou excluir os dados em uma tabela. Esses disparadores não podem ser chamados diretamente, ao contrário dos procedimentos armazenados do sistema, e não passam nem aceitam parâmetros.

O TRIGGER e a instrução que o aciona são tratados como uma única transação, que poderá ser revertida em qualquer ponto do procedimento, utilizando o ROLLBACK

Orientações básicas quando estiver usando TRIGGER:

- As definições de TRIGGERS podem conter uma instrução ROLLBACK TRANSACTION, mesmo que não exista uma instrução explícita de BEGIN TRANSACTION.
- Se uma instrução ROLLBACK TRANSACTION for encontrada, então toda a transação (o TRIGGER e a instrução que o disparou) será revertida ou desfeita. Se uma instrução no script do TRIGGER seguir uma instrução ROLLBACK TRANSACTION, a instrução será executada, então, isso nos obriga a ter uma condição IF contendo uma cláusula RETURN para impedir o processamento de outras instruções.

- Não é uma boa prática utilizar ROLLBACK TRANSACTION dentro de seus TRIGGERS, pois isso gerará um retrabalho, afetando muito no desempenho de seu banco de dados, pois toda a consistência deverá ser feita quando uma transação falhar, lembrando que tanto a instrução quanto o TRIGGER formam uma única transação. O mais indicado é validar as informações fora das transações com TRIGGER para então efetuar, evitando que a transação seja desfeita.
- Para que um TRIGGER seja disparado, o usuário que entrou com as instruções deverá ter permissão de acessar tanto a entidade e, conseqüentemente, o TRIGGER.

### Usos e aplicabilidade dos gatilhos

Os gatilhos são usados para realizar tarefas relacionadas com validações, restrições de acesso, rotinas de segurança e consistência de dados. Dessa forma, esses controles deixam de ser executados pela aplicação e passam a ser executados pelos TRIGGERS em determinadas situações:

- Mecanismos de validação envolvendo múltiplas tabelas.
- Criação de conteúdo de uma coluna derivada de outras colunas da tabela.
- Realização de análise e atualizações em outras tabelas com base em alterações e/ou inclusões da tabela atual.

A criação de um TRIGGER envolve duas etapas:

- 1) Um comando SQL que vai disparar o TRIGGER (INSERT, DELETE, UPDATE).
- 2) A ação que o TRIGGER vai executar (geralmente, um bloco de códigos SQL).

Quando incluimos, excluimos ou alteramos algum registro em um banco de dados, são criadas tabelas temporárias que passam a conter os registros excluídos, inseridos e também o antes e o depois de uma atualização.

Ao excluir um determinado registro de uma tabela, na verdade, será apagada a referência desse registro, que ficará, após o DELETE, em uma tabela temporária de nome DELETED. Um TRIGGER implementado com uma instrução SELECT poderá lhe trazer todos ou um número de registro que foi excluído.

Assim como acontece com DELETE, também ocorrerá com inserções em tabelas, podendo obter os dados ou o número de linhas afetadas buscando na tabela INSERTED.

Já no UPDATE ou atualização de registros em uma tabela, temos uma variação e uma concatenação para verificar o antes e o depois.

De fato, quando executamos uma instrução UPDATE, a "engrenagem" de qualquer banco de dados tem um trabalho semelhante, primeiro exclui os dados da tupla e, posteriormente, faz a inserção do novo

registro que ocupará aquela posição na tabela, ou seja, um DELETE seguido por um INSERT. Quando, então, há uma atualização, podemos buscar o antes e o depois, pois o antes estará na tabela DELETED e o depois estará na tabela INSERTED.

Nada nos impede de retornar dados das tabelas temporárias (INSERTED, DELETED) de volta às tabelas de nosso banco de dados, mas perceba que os dados manipulados são temporários, assim como as tabelas, e só estarão disponíveis nessa conexão. Após fechar, as tabelas e os dados não serão mais acessíveis.

### Sintaxe básica para a criação de TRIGGERS

```
CREATE TRIGGER [NOME DO TRIGGER]
```

```
ON [NOME DA TABELA]
```

```
[FOR/AFTER/INSTEAD OF] [INSERT/UPDATE/DELETE]
```

```
AS
```

```
--CORPO DO TRIGGER
```

Os parâmetros são:

**Nome do TRIGGER:** nome que identificará o gatilho como objeto do banco de dados. Deve seguir as regras básicas de nomenclatura de objetos.

**Nome da tabela:** tabela à qual o gatilho estará ligado, para ser disparado mediante ações de INSERT, UPDATE ou DELETE.

**FOR/AFTER/INSTEAD OF:** uma dessas opções deve ser escolhida para definir o momento em que o TRIGGER será disparado. FOR é o valor padrão e faz com que o gatilho seja disparado junto da ação. AFTER faz com que o disparo se dê somente após a ação que o gerou ser concluída. INSTEAD OF faz com que o TRIGGER seja executado no lugar da ação que o gerou.

**INSERT/UPDATE/DELETE:** uma ou várias dessas opções (separadas por vírgula) devem ser indicadas para informar ao banco qual é a ação que disparará o gatilho. Por exemplo, se o TRIGGER deve ser disparado após toda inserção, deve-se utilizar AFTER INSERT.

Agora, vamos criar um TRIGGER que armazenará alguns dados excluídos da tabela PEDIDOS em outra tabela. Acesse o banco de dados AULA e verifique os dados armazenados na tabela PEDIDOS:

```
USE MODELOBDNOVO;  
GO  
SELECT * FROM PEDIDOS;
```

Em seguida, vamos criar a tabela PEDIDOS\_EXCLUIDOS que armazenará os dados dos pedidos excluídos da tabela PEDIDOS:

```
CREATE TABLE PEDIDOS_EXCLUIDOS
(
    NUMERODOPEDIDO INT NOT NULL,
    CODIGODOCLIENTE VARCHAR(10) NOT NULL,
    DATADOPEDIDO DATETIME NOT NULL,
    FRETE SMALLMONEY NOT NULL
);
GO
```

A instrução a seguir cria um TRIGGER que é executado toda vez que excluimos um registro da tabela PEDIDO:

```
CREATE TRIGGER TB_REGISTRO          -- Nome do Trigger
    -- Tabela onde o trigger está vinculado e a operação disparadora da
trigger
ON PEDIDOS FOR DELETE
AS
    -- início do bloco de comandos
BEGIN
    -- declarando as variáveis para armazenamento das informações
    DECLARE @NUM INT, @CODCLI VARCHAR(10), @DT DATETIME, @FRETE
SMALLMONEY
    -- atribuindo valores as variáveis
    SELECT @NUM = NUMERODOPEDIDO, @CODCLI = CODIGODOCLIENTE, @DT = DATADOPEDIDO, @
FRETE = FRETE FROM DELETED
    -- inserindo os dados excluídos na tabela
    INSERT INTO TB_PEDIDOS_EXCLUIDOS VALUES (@NUM, @CODCLI, @DT, @FRETE)
    -- fim do bloco de comandos
END;
```

Execute a instrução a seguir para excluir um pedido:

```
DELETE FROM Pedidos WHERE NumeroDoPedido = 10250;
```

Verifique as duas tabelas. O registro já foi excluído e alguns dados foram salvos na tabela PEDIDOS\_EXCLUIDOS.

```
SELECT * FROM PEDIDOS;
GO
SELECT * FROM TB_PEDIDOS_EXCLUIDOS;
```

### 4.14 Cursores de dados

Uma das principais características dos bancos de dados relacionais diz respeito às operações em conjunto de linhas, que são geradas através da instrução SELECT. Esse conjunto completo de linhas retornado pela instrução é conhecido como conjunto de resultados. Muitas das aplicações desenvolvidas hoje não podem e nem precisam trabalhar efetivamente com todo o conjunto de resultados. Às vezes, precisamos trabalhar com uma única linha ou um pequeno bloco de linha de cada vez.

Nesse caso, podemos utilizar cursores de dados que ajudam a lidar com algumas dessas situações, pois é um objeto que aponta para uma determinada linha dentro de um conjunto, podendo, assim, executar operações como atualizar, excluir ou mover dados.

Um cursor nada mais é do que uma área de memória que contém um conjunto de linhas que podem ser acessadas individualmente, uma a uma. Esse mecanismo permite que se possa tratar cada linha individualmente, de forma procedural, como é feito pela maioria das linguagens de programação quando realizam o processamento dos registros de um arquivo de dados.

### **Características dos cursores**

Com relação às características dos cursores, podemos dividi-los em três grupos:

- Capacidade de refletir alterações feitas nos dados retornados pelo cursor.
- Capacidade de rolar pelo conjunto de linhas do cursor para frente e para trás.
- Capacidade de atualizar o conjunto de linhas do cursor (cursor de atualização).

### **Refletindo alterações**

Quando criamos um cursor, um conjunto de dados é retornado, mas o que acontece se um outro usuário atualizar as tabelas que foram usadas pelo cursor, alterando as linhas nas tabelas? O cursor será atualizado para refletir as alterações ou o cursor manterá os dados originais lidos da tabela antes de ter sido atualizada?

Existem dois tipos de reflexos que podem ser determinados separadamente quando você criar o seu cursor:

- Alterações nas quais as linhas estejam incluídas no conjunto.
- Alterações nos valores das linhas subjacentes.

Ao buscar e atualizar dados por meio de SQL Server cursores, um aplicativo SQL Server será consumidor do provedor OLE DB do Native Client e estará ligado pelas mesmas considerações e restrições que se aplicam a qualquer outro aplicativo cliente.

Apenas as linhas em cursores do SQL Server participam do controle de acesso a dados simultâneos.

### **Rolagem (leitura para frente e para trás)**

Outra questão é se você poderá utilizar o cursor para rolar para frente e para trás ou somente para frente. Aqui, encontraremos o velho dilema velocidade x flexibilidade. Os cursores que vão apenas para frente são significativamente mais rápidos, mas menos flexíveis.



### Atualização

Por último, questionamos se as linhas podem ser atualizadas pelo cursor. Mais uma vez, os cursores de somente leitura, geralmente, são mais eficientes, porém menos flexíveis.

### Tipos de cursor

Cada tipo de cursor armazena os dados de maneira diferente e suporta diferentes tipos de combinações. Veja as características de cada tipo de cursor a seguir:

- **Estático:** um cursor estático faz uma espécie de "foto" dos dados especificados pela sua instrução SELECT e a armazena no banco de dados tempdb. Ele é apenas de leitura. Ele pode ser *forward-only* (apenas para frente) ou *scrollable* (rolável).

Os cursores estáticos detectam poucas ou nenhuma alteração, porém consomem relativamente poucos recursos durante a rolagem.

Um cursor estático não exibe novas linhas inseridas no banco de dados após o cursor ter sido aberto, mesmo se elas corresponderem aos critérios de pesquisa da instrução SELECT do cursor.

Se as filas que constituem o conjunto de resultados forem atualizadas por outros usuários, os novos valores de dados não serão exibidos no cursor estático.

O cursor estático exibe linhas excluídas do banco de dados após o cursor ter sido aberto. Nenhuma operação UPDATE, INSERT ou DELETE é refletida em um cursor estático (a menos que o cursor esteja fechado e seja reaberto), nem mesmo alterações feitas usando a mesma conexão que abriu o cursor.

- **Keyset:** um cursor *keyset* copia para o tempdb apenas as colunas necessárias para identificar exclusivamente cada linha. O cursor *keyset* pode ser *updatable* (atualizável) ou *read-only* (somente leitura) e também *scrollable* (rolável) ou *forward-only* (apenas para frente). Linhas adicionadas nas tabelas subjacentes não serão adicionadas ao cursor. Alterações feitas por outro usuário não se refletirão no cursor.

Para declarar um cursor desse tipo, cada tabela envolvida na instrução SELECT de definição deverá ter um índice exclusivo que defina o conjunto de chaves a ser copiado.

Um cursor *keyset* é fixado quando você declara o cursor. Se uma linha que satisfaça as condições selecionadas for adicionada enquanto o cursor estiver aberto, ela não será adicionada ao conjunto de dados.

Embora a associação na definição do cursor seja fixada quando você abre o cursor, as alterações aos valores de dados nas tabelas subjacentes geralmente são refletidas. Por exemplo, se o valor de

uma linha fosse alterado pelo cursor, este retornaria, então, o valor alterado. Porém, se a alteração fosse feita por outro usuário, esse cursor continuará retornando o valor anterior.

- **Dynamic:** teoricamente, os cursores *dynamic* comportam-se como se uma instrução SELECT fosse lançada novamente sempre que uma linha fosse referenciada. Os cursores *dynamic* refletem as alterações de valor tanto da associação quanto dos dados subjacentes, quer essas alterações tenham sido feitas pelo cursor ou por qualquer outro usuário.

Porém, há uma restrição para o cursor *dynamic* – a instrução SELECT usada para defini-lo só pode conter uma cláusula ORDER BY, se existir um índice contendo as colunas na cláusula ORDER BY. Se você declarar um cursor *dynamic* com uma cláusula ORDER BY que não esteja apoiada por um índice, o SQL converterá o cursor para o cursor *keyset*.

- **Cursor de somente avanço (firehose):** esta é uma forma especializada de cursor do tipo somente leitura fixa, suportada pelo SQL Server. Esse tipo de cursor é declarado usando FAST\_FORWARD, mas é mais conhecido como um cursor *firehose*.

Esses cursores são muito eficientes, mas existem algumas restrições importantes quanto ao seu uso:

- Se a instrução SELECT que você usou para definir o cursor referenciar colunas text, ntext ou image e contiver a cláusula TOP, o SQL Server converterá o cursor *firehose* em um cursor *keyset*.
- Se a instrução SELECT que você usou para definir o cursor combinar tabelas que contenham gatilhos com tabelas que não tenham gatilhos, o cursor será convertido para um cursor *static*.
- Não dá suporte à rolagem. Ele suporta apenas a busca das linhas em série do início ao término do cursor. As linhas não são recuperadas do banco de dados até que sejam buscadas.

## Criação de cursores

A declaração de um cursor de dados é feita com o uso de um comando DECLARE CURSOR. As linhas são mantidas em uma área de memória reservada ao cursor e através do cursor podemos acessar as linhas individualmente para processamento.

```
Transact-SQL Extended Syntax
DECLARE cursor_name CURSOR [ LOCAL | GLOBAL ]           (1)
    [ FORWARD_ONLY | SCROLL ]                           (2)
    [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]        (3)
    [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]          (4)
    [ TYPE_WARNING ]                                     (5)
FOR select_statement                                     (6)
    [ FOR UPDATE [ OF column_name [ ,...n ] ] ]         (7)
[;]
```

Veja a seguir a explicação para cada linha da instrução dada:

(1) É usado para definir o escopo do cursor assim como funciona em tabelas temporárias (@local ou @@global).

(2) Indica a rolagem a ser definida para o cursor e aceita as palavras-chaves: FORWARD\_ONLY e SCROLL.

(3) Usado para definir o tipo do cursor a ser criado: STATIC, KEYSET, DYNAMIC e FAST\_FORWARD.

(4) Indica o tipo de bloqueio, se as linhas poderão ser atualizadas pelo cursor e, se assim for, se outros usuários também poderão atualizá-los.

(5) Esse parâmetro instrui o SQL Server para enviar uma mensagem de aviso para o cliente se um cursor for convertido do tipo especificado em outro tipo.

(6) Especifica as linhas a serem incluídas no conjunto do cursor.

(7) Este parâmetro é opcional. Por padrão, os cursores são atualizáveis a não ser que o parâmetro de bloqueio seja READ\_ONLY. Neste parâmetro podem-se especificar as linhas que permitem a atualização. Se forem omitidas, todas as colunas na instrução serão atualizáveis.

### Variáveis de cursor

O T-SQL permite declarar variáveis do tipo CURSOR. A sintaxe DECLARE padrão não cria o cursor. Para isso, use a cláusula SET para identificar a variável. Esse tipo de sintaxe é útil quando desejamos criar variáveis que possam ser atribuídas a diferentes cursores, o que poderá fazer se criar um procedimento genérico que opere em vários conjuntos de resultados.

### Abrir um cursor

A declaração de um cursor cria um objeto cursor, mas não cria o conjunto de linhas que serão manipuladas pelo cursor. O conjunto do cursor não será criado até que se abra o cursor.

```
OPEN [GLOBAL] cursor_ou_variável
```

### Fechar um cursor

Após ter terminado de usar um cursor, devemos fechá-lo. A instrução CLOSE libera os recursos usados para manter o conjunto do cursor e também liberta quaisquer bloqueios que tenham sido colocados nas linhas.

```
CLOSE [GLOBAL] cursor_ou_variável
```

## Desalocar um cursor

Na sequência de criação de um cursor, o DEALLOCATE é o último comando. Sua sintaxe é parecida com os comandos anteriores:

```
DEALLOCATE [GLOBAL] cursor_ou_variável
```

Esse comando remove o identificador do cursor e não o cursor ou variável. O cursor não será removido até que os identificadores sejam desalocados ou fiquem fora do escopo.

## Manipulando linhas com um cursor

O T-SQL suporta três comandos diferentes para trabalhar com cursores: FETCH, UPDATE e DELETE.

O comando FETCH recupera uma linha específica do conjunto do cursor. Em sua forma mais simples, o comando FETCH possui a seguinte sintaxe:

```
FETCH cursor_ou_variável
```

A seguir, veremos um cursor criado para retornar o primeiro registro da tabela CIDADE.

```
DECLARE crCidade CURSOR LOCAL FOR SELECT nome_cidade FROM TB_CIDADE
OPEN crCidade
FETCH crCidade
CLOSE crCidade
DEALLOCATE crCidade
```

Podemos também utilizar o FETCH para armazenar o resultado em uma variável utilizando o FETCH cursor INTO variável. Veja o exemplo a seguir:

```
DECLARE crCidade CURSOR LOCAL FOR SELECT nome_cidade FROM TB_CIDADE
DECLARE @cidade CHAR(20)
OPEN crCidade
FETCH crCidade INTO @cidade
CLOSE crCidade
DEALLOCATE crCidade
```

Além disso, podemos utilizar o FETCH e uma combinação de palavras chaves, por exemplo:

- FETCH FIRST – retorna a primeira linha da variável.
- FETCH NEXT – retorna a linha seguinte.
- FETCH PRIOR – retorna a linha anterior.

- FETCH RELATIVE n – retorna a linha n.
- FETCH ABSOLUNT n – pode especificar linhas antes da linha atual.

### Atualizando linhas com um cursor

Desde que o cursor seja atualizável, alterar os valores subjacentes em um conjunto do cursor é bastante simples. Vejamos o próximo exemplo:

```
DECLARE crCidade CURSOR LOCAL KEYSET FOR SELECT codd_cidade, nome_cidade FROM
tb_cidade
FOR UPDATE
OPEN crCidade
FETCH crCidade
UPDATE tb_cidade SET nome_cidade = 'Cidade Teste' WHERE CURRENT OF crCidade
CLOSE crCidade
DEALLOCATE crCidade
```

### Estrutura de um programa usando cursor

A estrutura de um programa que faz uso de cursor segue o modelo a seguir, onde podemos ver comandos para:

- Declaração de variáveis do programa.
- Declaração do cursor.
- Abertura do cursor.
- Leitura da linha do cursor.

Estrutura de repetição com *while* para processamento dos dados retornados do cursor:

- Leitura da próxima linha do cursor.
- Fechamento do cursor.
- Liberação do cursor da memória.

A utilização de cursores facilita o desenvolvimento e o trabalho em conjunto de dados, ou seja, dados que não poderiam ser manipulados somente utilizando cláusulas básicas do SQL.



### Resumo

Na unidade II, foram apresentadas as características e elementos constitutivos do modelo entidade-relacionamento (MER), idealizado por Peter Chen em meados dos anos 1970. Vimos que o MER se baseia na percepção do mundo real constituído por um conjunto de objetos chamados entidades e relacionamentos. Para isso, define que a técnica de diagramação para modelos de dados deve possuir características suficientes para modelar qualquer realidade, por meio de uma forma de trabalho muito simples, que usa diagramas para representar, de forma geral, os dados que serão gerenciados.

Vimos que a normalização é um processo que tem como objetivo evitar problemas que possam provocar falhas no projeto do banco de dados, além de eliminar as redundâncias desnecessárias e as falhas na sincronização de dados, dentre outras anomalias. Esse processo aplica uma série de regras sobre as tabelas de um banco de dados e, assim, verifica se estão corretamente projetadas. Embora exista um conjunto de cinco formas normais, apresentamos apenas três formas normais propostas por Edgar F. Codd.

Mostramos, também, como utilizar as ferramentas CASE, que auxiliam na modelagem do banco de dados relacionais e permitem criar dados de forma gráfica. Para a compreensão do desenvolvimento de um projeto lógico de banco de dados, observamos como acontece a criação de modelos lógicos, quais linguagens podem ser utilizadas e como as manipulamos.



## Exercícios

**Questão 1.** Considere o projeto de um sistema que vai utilizar um banco de dados relacional. Uma das tarefas do projetista envolve a criação de um diagrama de entidade-relacionamento, também chamado de DER. Considere a situação em que o projetista quer modelar uma entidade chamada de Cliente e que apresenta, entre os seus atributos, o telefone. Contudo, foi informado ao projetista que um cliente pode ter um ou mais telefones. Além disso, um cliente tem um nome, que pode ser dividido em um primeiro nome e em um ou mais sobrenomes. O cliente também tem um endereço, que pode ser dividido em rua, bairro, estado e cidade.

Com base nessas informações, o projetista começou a trabalhar no diagrama de entidade-relacionamento apresentado a seguir.

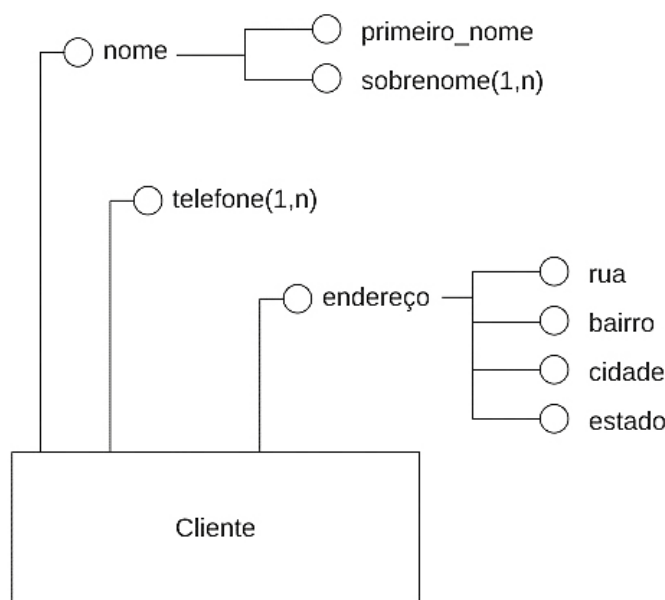


Figura – Início da criação de um diagrama de entidade-relacionamento

Com base no enunciado, na figura e nos seus conhecimentos, avalie as afirmativas.

- I – O diagrama de entidade-relacionamento da figura contém apenas uma entidade chamada de Cliente.
- II – O diagrama de entidade-relacionamento da figura contém várias entidades, entre elas, Cliente, nome, telefone e endereço.
- III – A entidade Cliente exibida no diagrama de entidade-relacionamento da figura contém um atributo chamado de endereço, que é composto.
- IV – A entidade Cliente exibida no diagrama de entidade-relacionamento da figura contém um atributo chamado de telefone, que é multivalorado.

É correto o que se afirma em:

- A) I, apenas.
- B) II, apenas.
- C) III, apenas.
- D) II e III, apenas.
- E) I, III e IV, apenas.

Resposta correta: alternativa E.

### Análise das afirmativas

I – Afirmativa correta.

Justificativa: o diagrama apresenta apenas a entidade Cliente, representada por um retângulo.

II – Afirmativa incorreta.

Justificativa: na figura do enunciado, nome, telefone e endereço são atributos da entidade Cliente, não sendo representado como entidades.

III – Afirmativa correta.

Justificativa: na figura do enunciado, o atributo endereço (da entidade Cliente) é composto, uma vez que ele é dividido em rua, bairro, estado e cidade.

IV – Afirmativa correta.

Justificativa: na figura do enunciado, o atributo telefone (da entidade Cliente) é multivalorado, uma vez que um cliente pode ter vários telefones e está representado como telefone (1,n).

**Questão 2.** Uma empresa está construindo um sistema de cadastro de funcionários. Esse sistema vai utilizar um banco de dados relacional e o projetista está trabalhando no modelo de entidade-relacionamento (MER). Nesse modelo, ele decide criar um conjunto de entidades chamado de "Funcionário". As entidades associadas têm atributos como "nome", "CPF", "número de matrícula" (chamado apenas de "matrícula"), número de telefone (chamado apenas de "telefone") e "e-mail". Entre as diversas regras de negócio identificadas, sabe-se que o número de matrícula do funcionário é único e deve estar associado a todos os funcionários, independentemente do seu cargo. Além disso, mesmo se um funcionário sair da empresa, seu número de matrícula se mantém associado a ele e não é reutilizado. Contudo, sabe-se que existe a possibilidade de mais de um funcionário ter o mesmo nome e o mesmo



sobrenome. Ainda que a maioria dos funcionários tenha telefone e e-mail, isso não é obrigatório, e existe a possibilidade de que um funcionário não tenha nem um nem outro. Também foi identificada a possibilidade de que mais de um funcionário compartilhe o mesmo telefone e/ou o mesmo e-mail. Com base nesse cenário, o projetista busca identificar as chaves candidatas para o conjunto de entidade "Funcionário". O projeto ainda não terminou, e o projetista ainda não decidiu qual será a chave primária desse conjunto de entidades.

Nesse contexto, considere as afirmativas a seguir:

I – Podemos utilizar apenas o atributo "nome" do conjunto de entidades "Funcionário" como chave primária desse conjunto de entidades.

II – Não podemos utilizar apenas o atributo "telefone" do conjunto de entidades "Funcionário" como chave primária desse conjunto de entidades.

III – O atributo "e-mail" do conjunto de entidades "Funcionário" pode ser considerado uma chave candidata desse conjunto de entidades.

IV – O atributo "matrícula" do conjunto de entidades "Funcionário" pode ser considerado uma chave candidata desse conjunto de entidades.

É correto o que se afirma em:

A) I, apenas.

B) II, apenas.

C) III, apenas.

D) II e IV, apenas.

E) I, III e IV, apenas.

Resposta correta: alternativa D.

### **Análise das afirmativas**

I – Afirmativa incorreta.

Justificativa: como dois funcionários podem ter o mesmo nome e o mesmo sobrenome, esse atributo não pode ser utilizado para identificar unicamente uma entidade, e, portanto, não pode ser uma chave primária.

