



## Banco de Dados

### Elaini Simoni Angelotti

Mestre em Informática Aplicada pela PUCPR.

Graduada em Ciência da Computação pela UEM.

Professora da Disciplina de Banco de Dados para cursos de graduação (Tecnólogo em Processamento de Dados e Bacharelado em Sistemas de Informação) desde 2001.

Professora da disciplina de Banco de Dados para os Cursos: Técnico em Informática Integrado ao Ensino Médio e Técnico em Jogos Digitais Integrado ao Ensino Médio do Instituto Federal do Paraná (IFPR) desde 2009.

<b>Direção Geral</b>	Jean Franco Sagrillo
<b>Direção Editorial</b>	Jeanine Grivot
<b>Edição</b>	Leonel Francisco Martins Filho
<b>Assistente Editorial</b>	Melissa Harumi Inoue Pieczarka
<b>Gerência de Produção e Arte</b>	Marcia Tomeleri
<b>Revisão</b>	Alessandra Domingues e Jeferson Turbay Braga
<b>Revisão Comparativa</b>	Renee Cleyton Faletti
<b>Projeto Gráfico</b>	Adriana de Oliveira

Angelotti, Elaini Simoni.  
 Banco de dados / Elaini Simoni Angelotti. – Curitiba: Editora do Livro Técnico, 2010.

120 p.  
 ISBN: 978-85-63687-02-9

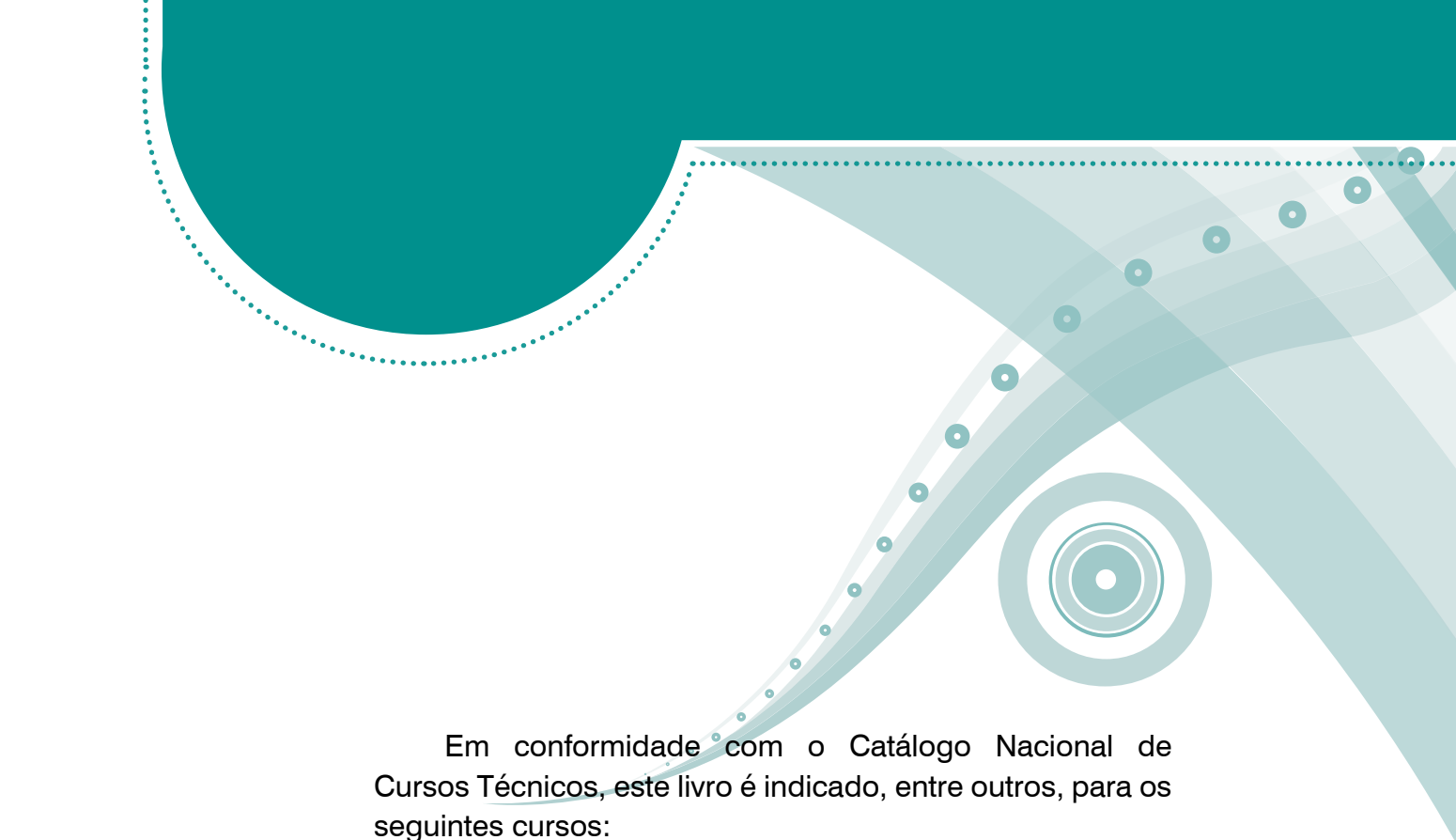
1. Informática 2. Programação (Computadores). 3. Tecnologia. I. Título.

CDD 005.74

Eutália Cristina do Nascimento Moreto CRB-9/947

2010

Todos os direitos reservados pela Editora do Livro Técnico  
 Edifício Comercial Sobral Pinto  
 Avenida Cândido de Abreu 469, 2º andar, conj. n.ºs 203-205  
 Centro Cívico – CEP: 80530-000  
 Tel.: 41 3027-5952 / Fax: 41 3076-8783  
[www.editoralt.com.br](http://www.editoralt.com.br)  
 Curitiba – PR



Em conformidade com o Catálogo Nacional de Cursos Técnicos, este livro é indicado, entre outros, para os seguintes cursos:

**Eixo tecnológico: Informação e Comunicação**

- Técnico em Informática
- Técnico em Informática para Internet
- Técnico em Programação de Jogos Digitais



# Apresentação

Este livro é resultado da experiência que tenho em ministrar aulas da disciplina de Banco de Dados nos cursos técnicos de Ensino Médio, e nas dificuldades dos alunos em compreender os conteúdos dessa disciplina.

Um primeiro problema, observado em anos de experiência, é que os alunos têm grande dificuldade para entender a importância da modelagem de uma base de dados e já querem ir direto à parte de implementação (Linguagem SQL). No entanto, é importante que eles compreendam que uma base de dados bem modelada poupa muitos problemas futuros. Por esse motivo, metade desse livro (capítulos 1, 2, 3, 4 e 5) é dedicado à modelagem.

Em relação à modelagem de base de dados, a maior dificuldade que os alunos têm está em compreender os conceitos apresentados e aplicá-los num problema real. Foi pensando nessa dificuldade que todos os capítulos de modelagem consideram uma situação real, e o capítulo 6, em especial, apresenta um exemplo prático, desde a concepção do modelo conceitual (modelo de entidade e relacionamento) até o desenvolvimento do modelo lógico (modelo relacional) e toda a sua documentação.

Outra dificuldade em relação à modelagem encontra-se na transformação do modelo de entidade e relacionamento para o modelo relacional. Embora essa transformação seja apenas a aplicação de regras, a maioria dos livros de banco de dados apresenta essas regras de forma bastante complicada, o que dificulta o entendimento dos alunos. Nesse livro, as regras são apresentadas em linguagem simples e introduzidas de forma gradual, sempre acompanhadas de exemplo. O mesmo acontece para as regras de normalização do modelo relacional.

Nos capítulos 7, 8, 9 e 10, o aluno irá aprender a implementar os modelos construídos nos capítulos anteriores, usando a linguagem SQL. Também irá aprender a utilizar a linguagem SQL para manipular e recuperar as informações armazenadas nas bases de dados. Nesses capítulos, também são utilizados exemplos, para maior compreensão.

É importante destacar que todos os capítulos contam com um conjunto de exercícios. O grau de dificuldade desses exercícios vai aumentando gradualmente. Assim, o aluno começa resolvendo exercícios mais simples até chegar àqueles mais complexos. A ideia é que o aluno se sinta estimulado a resolver todos os exercícios, uma vez que ele vai gradualmente aplicando os conceitos aprendidos.

Por fim, este livro tem por objetivo dar o embasamento necessário para o desenvolvimento e a criação de uma base de dados. Ou seja, pode ser utilizado na disciplina de Fundamentos de Banco de Dados. Espera-se que, assim, o aluno possa estar preparado e tenha adquirido conhecimentos necessários para estudar conteúdos de Banco de Dados mais avançados.

Este livro é recomendado para alunos de Cursos Técnicos em Informática e Cursos Técnicos de Informática Integrados ao Ensino Médio, entre outros.

# Sumário

<b>CAPÍTULO 1 – Uma Breve Introdução a Banco de Dados</b>	<b>9</b>
O Que São Dados? O Que É Informação? .....	9
O Que É uma Base de Dados? .....	10
O Que É um Sistema de Banco de Dados? .....	10
Quem Usa um Banco de Dados? .....	13
Fases no Desenvolvimento de um Projeto de Banco de Dados .....	14
Modelo de Dados .....	15
Atividades .....	17
<b>CAPÍTULO 2 – Introdução ao Modelo de Entidade e Relacionamento</b>	<b>18</b>
Entidades e Atributos .....	19
Chave Primária .....	22
Entidade Fraca .....	23
Atividades .....	24
<b>CAPÍTULO 3 – ER: Relacionamento, Especialização e Agregação</b>	<b>26</b>
Relacionamento .....	26
Especialização .....	31
Agregação .....	34
Atividades .....	35
<b>CAPÍTULO 4 – Introdução ao Modelo Relacional</b>	<b>37</b>
Chave Estrangeira e Integridade Referencial .....	37
Conversão entre o Modelo de ER e o Modelo Relacional .....	40
Atividades .....	48
<b>CAPÍTULO 5 – Modelo Relacional: Tópicos Avançados</b>	<b>50</b>
Especialização .....	50
Diagrama do Modelo Relacional .....	52
Dicionário de Dados da Base de Dados .....	53
Normalização .....	55
Atividades .....	59
<b>CAPÍTULO 6 – Um Exemplo Prático</b>	<b>61</b>
O Problema .....	61
O Diagrama de Entidade e Relacionamento .....	62
A Descrição do Modelo Relacional .....	63
O Diagrama do Modelo Relacional .....	64

Verificação se o Modelo Está Normalizado.....	65
O Dicionário de Dados.....	65
Atividades.....	69
<b>CAPÍTULO 7 – Implementação do Modelo Lógico: Linguagem SQL</b>	<b>74</b>
Comandos DDL.....	75
Índices.....	82
Atividades.....	84
<b>CAPÍTULO 8 – Linguagem SQL – DML</b>	<b>86</b>
Comando INSERT.....	86
Comando DELETE.....	88
Comando UPDATE.....	90
Comando SELECT.....	91
Comando WHERE.....	93
Comando FROM.....	94
Atividades.....	97
<b>CAPÍTULO 9 – Outros Comandos SQL – DML</b>	<b>99</b>
Aliases.....	99
Comando LIKE.....	100
Comando ORDER BY.....	101
Funções Agregadas.....	101
Comando GROUP BY.....	103
Comando HAVING.....	104
Valores Nulos.....	105
Tabelas Derivadas.....	106
Atividades.....	107
<b>CAPÍTULO 10 – SQL – DML: Subconsulta e Tipos de Junção</b>	<b>109</b>
Operador IN e NOT IN.....	109
Operador EXISTS.....	111
Operadores ALL e SOME.....	114
Tipos de Junção.....	115
Atividades.....	118
<b>Referências Bibliográficas</b>	<b>120</b>



# Uma Breve Introdução a Banco de Dados

A área de banco de dados é de grande importância no mundo da informática, uma vez que a informação é um bem precioso e deve ser armazenada de forma coerente e adequada.

Quando você utiliza o Orkut, por exemplo, armazena em um sistema de banco de dados suas informações pessoais, recados de amigos, depoimentos, etc. Se você sair da sua página no Orkut e entrar nela novamente, as informações ainda estarão lá. Isso ocorre porque essas informações foram armazenadas em um banco de dados, e, portanto, podem ser recuperadas no momento oportuno, quando o usuário solicitar.

Atualmente, por menor e mais simples que seja um Sistema de Informação, ele precisará ter a capacidade de armazenar e recuperar dados rapidamente. Por exemplo, se você desenvolver um Sistema de Informação para um salão de beleza do seu bairro, este sistema terá que armazenar dados de clientes, produtos, valores, funcionários, serviços, etc. É para armazenar essas informações e recuperá-las rapidamente que utilizamos um sistema de banco de dados.

No entanto, para que possamos entender o que está compreendido na área de banco de dados, ou seja, quais os modelos, quais as ferramentas disponíveis, como modelar, implementar uma base de dados e como recuperar dados, é necessário compreender alguns conceitos básicos sobre o assunto.

## O Que São Dados? O Que É Informação?

**Dados** são tudo que podemos inferir ou coletar sobre uma situação específica. Os dados podem ser úteis ou não. Por exemplo, em uma sala de aula a quantidade de carteiras, a cor da parede, o tipo do assoalho, as dimensões da sala, etc., fornecem-nos dados sobre o ambiente. No entanto, no desenvolvimento de uma aplicação, esses dados podem ser úteis ou não, dependendo do objetivo do projeto. Se, por exemplo, uma arquiteta tem a intenção de desenvolver um novo leiaute para aquela sala de aula, os dados acima mencionados serão úteis.

Os dados úteis é o que chamamos de **informação**. E esses dados são o que armazenamos em uma base de dados.

Por convenção, na área de banco de dados, os termos “informação” e “dado” significam a mesma coisa. Isso ocorre porque devemos armazenar apenas aquilo que é útil para a nossa aplicação. Sendo assim, daqui para frente, os dois termos serão usados como sinônimos.

## O Que É uma Base de Dados?

Uma base de dados é um local, ou espaço, onde informações estão armazenadas e de onde elas são recuperadas. Uma base de dados terá um nome, e este nome deverá representar o que aquela base armazena. Por exemplo, se a aplicação for uma agenda de contatos pessoal e profissional, o nome da base poderá ser `bd_Agenda`. Essa base de dados deverá armazenar todas as informações sobre os contatos pessoais e profissionais como, por exemplo, nome, endereço, tipo do endereço (residencial, comercial, de referência, etc.), telefone residencial, telefone celular, telefone comercial, nome da empresa em que trabalha, grau de parentesco, e-mail, etc.

Uma base de dados permite que os dados fiquem centralizados e que se relacionem de forma coerente.

## O Que É um Sistema de Banco de Dados?

Computacionalmente, um Sistema de Banco de Dados é uma ferramenta que será utilizada para armazenar informações. Essa ferramenta possui três principais características:

- Armazenar os dados.
- Relacionar os dados armazenados.
- Recuperar os dados rapidamente.

**Armazenar os dados** significa que a ferramenta possui um repositório onde as informações são gravadas. Esse repositório permite centralizar os dados, evitando que eles fiquem espalhados em vários arquivos.

**Relacionar os dados armazenados** é muito importante. Imagine uma base de dados em uma escola que contenha informações sobre os alunos, os professores, as disciplinas, as turmas e os cursos. Se não for possível relacionar essas informações, como se saberá que um determinado aluno faz o curso de Técnico em Informática e tem aula da disciplina de Banco de Dados com determinado professor? Portanto, é importante que um sistema de Banco de Dados permita relacionar as informações armazenadas de forma coerente.

### Lembre-se!

Armazenar dados e não relacioná-los não é nada interessante para quem desenvolve um Sistema de Informação.

**Recuperar os dados** por meio de consultas ao Sistema de Banco de Dados. Nos bancos de dados relacionais, as consultas são feitas utilizando-se a Linguagem **SQL**, cuja apresentação será feita nos capítulos 7, 8, 9 e 10.

**SQL (Strutured Query Language):**  
*Linguagem de consulta estruturada.*

Cabe dizer que existem outros tipos de Sistemas de Banco de Dados além do relacional, embora esse seja o mais utilizado no mercado. Procure se informar sobre quais são os outros tipos de banco de dados.

## O Que É um Sistema Gerenciador de Banco de Dados (SGBD)?

Atualmente, os Sistemas de Banco de Dados evoluíram para **Sistemas Gerenciadores de Banco de Dados (SGBD)**.

Um SGBD é uma ferramenta muito mais completa que um Sistema de Banco de Dados. Um SGBD disponibiliza uma série de funcionalidades que permitem controlar e acompanhar melhor os dados armazenados.

As principais características de um SGBD são:

- Permitir o acesso concorrente às bases de dados;
- Realizar o gerenciamento de transações;
- Permitir criar e aplicar regras de segurança às bases de dados;
- Permitir criar regras que garantam a integridade da base de dados.

Exemplos de SGBD utilizados atualmente:

- Oracle Database (da Oracle)
- SQL Server (da Microsoft)
- PostgreSQL (código aberto)
- DB2 (da IBM)
- MySQL (código aberto, atualmente da Oracle)
- MariaDB (código aberto)

## Acesso Concorrente

O acesso concorrente à base de dados significa que o SGBD permite que duas ou mais pessoas acessem uma mesma base de dados ao mesmo tempo e o sistema controla para que um acesso não interfira no outro.

Por exemplo, em um sistema de compras na Web, várias pessoas podem realizar uma compra ao mesmo tempo, e o próprio SGBD controla para que os dados de todas as compras sejam gravados corretamente na base de dados.

## Gerenciamento de Transações

Uma transação em banco de dados consiste em um conjunto de operações que é tratado como uma **unidade lógica indivisível**. Isso significa que quando começa a execução de uma transação, esta deve ter executadas **todas** as operações dentro dela.

Se acontecer qualquer falha durante a execução da transação (por exemplo: falta de energia, alguém desligar o servidor, cancelamento da transação pelo usuário, etc.) as operações pendentes devem ser canceladas, e aquelas que foram executadas deverão ser desfeitas. Isso acontece para garantir a integridade dos dados dentro da base.

Por exemplo, imagine que você está fazendo a compra de um jogo de computador pela Internet. Suponha que você já escolheu o jogo, colocou no carrinho de compras, preencheu o cadastro e entrou com os dados do seu cartão de crédito. Agora só falta você confirmar a compra.

No entanto, antes de você confirmar a compra ocorre um erro (que pode ser na sua máquina ou no servidor que tem a aplicação). Nessa situação, como a compra não foi confirmada, ela é cancelada, e todas as operações que haviam sido executadas anteriormente são desfeitas. Isso significa que o jogo que você estava tentando obter se tornará disponível para compra novamente, e poderá ser adquirido por outra pessoa.

Ainda sobre as transações: imagine que você queira trocar os pontos do seu cartão fidelidade por algum produto do seu interesse. Durante a troca (que é feita via sistema pela Internet ou na loja) ocorre uma falha. Você já havia escolhido o produto e o sistema já havia verificado que seus pontos eram suficientes para obtê-lo, e também já havia lhe dito quantos pontos ainda sobrariam no seu cartão. Como a falha ocorreu antes da troca ser confirmada, os pontos do seu cartão não poderão ser debitados e a troca será cancelada.



Um questionamento interessante:

E se a falha ocorresse após a confirmação da troca, mas antes da emissão do comprovante?

Nesse caso, mesmo que o problema seja no servidor, se a transação foi concluída antes da falha, o SGBD tem um mecanismo que verifica que essa transação foi concluída com sucesso e quando o servidor voltar a funcionar o SGBD fará as atualizações necessárias na base de dados. Assim, os pontos serão debitados e você receberá o seu produto em troca dos pontos.

## Regras de Segurança

Outro aspecto importante com o qual o SGBD se preocupa é com a segurança das informações armazenadas. Sendo assim, um SGBD tem mecanismos para criação de regras de segurança.

As regras de segurança vão desde a definição de *login* e senha para os usuários, até a permissão de acesso ao SGBD e acesso aos dados armazenados.

Em relação às regras de permissão de acesso ao SGBD, é possível definir o que o usuário pode fazer no SGBD, ou seja, definir o papel do usuário no SGBD. Por exemplo: posso ter um usuário do SGBD que só tem permissão para leitura de dados; ou outro usuário que tenha permissão para criar base de dados e manipulá-la, mas não pode criar novos usuários ou realizar *backup*.

Quanto ao acesso aos dados, pode-se definir em uma base de dados qual usuário tem acesso a qual informação. Por exemplo: pode-se definir que os alunos tenham acesso aos dados de disciplinas e turmas, mas que não possam acessar dados do professor.

## Regras de Integridade

Outra característica importante de um SGBD a se destacar é a possibilidade de criação de regras que garantam a integridade da base de dados.

As regras de integridade são interessantes porque ficam definidas para uma base de dados, e todas as aplicações que acessaram aquela base poderão utilizar a regra de integridade.

Imagine a necessidade de desenvolver uma aplicação onde o usuário deve preencher o estado em que mora. Suponha que os valores permitidos são apenas “PR”, “SC” e “RS”. Pode-se escrever uma regra na base de dados que verifique se o estado é válido ou não. O próprio SGBD vai fazer a verificação todas as vezes que o estado for inserido na base de dados.

Outro exemplo: Supondo que em um sistema de compras pela Internet o usuário só possa parcelar suas compras se o valor da compra for superior a R\$ 150,00. Pode-se criar essa regra no banco, e assim o próprio SGBD poderá validar isso para o usuário.

Além das quatro características citadas e explicadas acima, os SGBD possuem várias outras funcionalidades como, por exemplo, algoritmos para extração de conhecimento em base de dados (*data mining*), técnicas OLAP e de definição de *data warehouse*, etc.

Atualmente, os SGBD funcionam em uma arquitetura cliente-servidor. Isso significa que você pode instalar o SGBD em um servidor e instalar o cliente em várias máquinas para que vários usuários tenham acesso simultâneo ao SGBD. É importante ressaltar que nessa arquitetura todas as bases de dados são criadas, alteradas e excluídas no servidor e não na máquina do cliente.

## Quem Usa um Banco de Dados?

Os usuários de um banco de dados podem ser divididos em 3 categorias:

1. **Administrador do banco de dados (DBA):** é o responsável por monitorar e gerenciar todas as bases de dados criadas no SGBD. Também é quem controla as permissões dos usuários, garante que os usuários tenham acesso aos dados, realiza *backups*, recupera os dados em caso de falhas, garante o melhor desempenho para o banco de dados, monitora serviços (*Jobs*) de usuários no banco de dados, etc. Normalmente, um DBA é responsável por um SGBD específico, e deve ter estudado e feito cursos para aquele SGBD (por exemplo: DBA Oracle ou DBA SQL Server).
2. **Analistas de sistemas e programadores de aplicações:** são responsáveis por modelar a base de dados e implementá-la no SGBD escolhido. Também são responsáveis por desenvolver a aplicação (programa escrito em uma linguagem de programação como: Java, PHP, C++, C#, etc.) e conectar essa aplicação à base de dados do sistema. Esse usuário precisa conhecer a área de banco de dados, saber modelar uma base de dados e também conhecer a linguagem SQL.
3. **Usuários finais:** os usuários finais são aquelas pessoas que vão trabalhar diariamente com as aplicações desenvolvidas. São eles os responsáveis pela entrada de dados no banco de dados e pelas alterações nos dados armazenados. Esses usuários não precisam ter nenhum conhecimento sobre banco de dados ou saber qual o SGBD utilizado. Para eles, o banco de dados é transparente, e só interessa que as informações estejam sendo salvas e possam ser recuperadas.

O esquema a seguir ilustra os papéis de cada usuário em uma hierarquia em camadas.

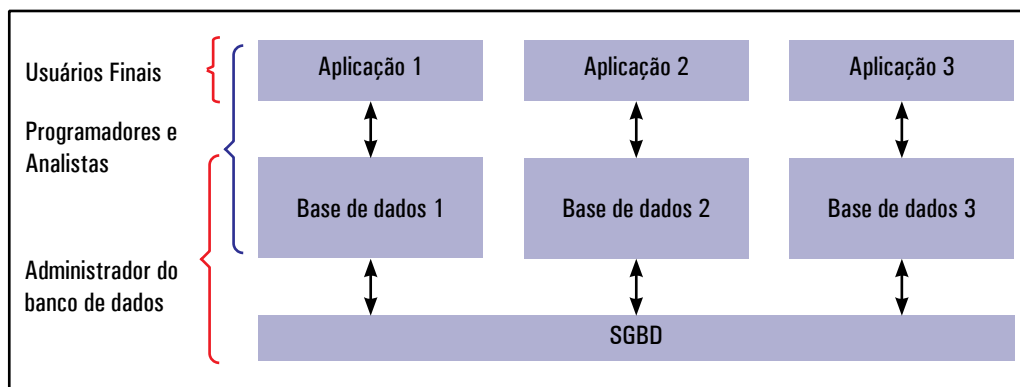


Figura 1.1 – Acesso dos usuários ao banco de dados

Os administradores de banco de dados devem manter o SGBD e monitorar as bases criadas naquele SGBD. Os analistas e programadores são responsáveis tanto pelo desenvolvimento das aplicações quanto pela modelagem e implementação da base de dados. Além disso, eles devem conectar a aplicação à base de dados, para que seja possível armazenar e recuperar os dados. Os usuários finais têm acesso apenas à aplicação e toda inserção, alteração, exclusão e consultas aos dados são feitas via aplicação.

## Fases no Desenvolvimento de um Projeto de Banco de Dados

Um projeto de banco de dados compreende as seguintes fases:

- **Modelagem Conceitual:** A modelagem conceitual refere-se ao desenvolvimento de um modelo inicial da base de dados que reflita as necessidades do usuário. Essa modelagem preocupa-se em descrever **quais dados** serão armazenados na base de dados e **quais dados** se relacionam. Para fazer o modelo conceitual, é necessário entender o que o usuário final espera que o sistema armazene e que informações este usuário espera que o sistema disponibilize (como por exemplo, relatórios). Para obter as informações necessárias para desenvolver a modelagem conceitual do sistema, deve-se realizar entrevistas com o usuário para entender os objetivos do sistema e as expectativas que o usuário tem em relação a ele. Um dos principais modelos desta etapa é o *Modelo de Entidade e Relacionamento*.
- **Modelagem Lógica:** A modelagem lógica compreende o processo de descrever **como** os dados serão armazenados no sistema e **como** irão se relacionar. Isso significa transformar o modelo conceitual obtido na primeira fase num modelo mais próximo da implementação.

Para banco de dados relacionais, o modelo utilizado nessa fase é o modelo relacional. Também é necessário descrever o dicionário de dados da base de dados nessa etapa. Antes da fase de implementação, é necessário, ainda verificar se o modelo está normalizado e em caso negativo, deve-se normalizar o modelo.

- **Implementação do Modelo Lógico:** Uma vez que toda a etapa de modelagem esteja concluída, será necessário implementar ou criar a base de dados no SGBD escolhido. Essa fase requer que o desenvolvedor conheça a Linguagem SQL e conheça o SGBD selecionado.

A importância de uma boa modelagem se deve ao fato de que as aplicações que estarão acessando a base de dados devem estar em consonância com o modelo desenvolvido. É muito desanimador e trabalhoso perceber a necessidade de alterar o modelo de dados, depois da base de dados programada e da aplicação do usuário desenvolvida. A verdade é que isso gera um retrabalho, uma vez que não só a implementação da base terá que ser refeita, mas também os diagramas e a aplicação deverão ser atualizadas na maioria dos casos.

A fase de modelagem é a principal etapa no desenvolvimento de uma base dados. Por isso é muito importante que se dedique tempo e esforço no desenvolvimento de uma boa modelagem da base de dados.

## Modelo de Dados

O modelo da base de dados é também conhecido como *Esquema da base de dados*. Um esquema não deve mudar com frequência, porque uma vez alterado, tudo o que estiver envolvido com esse esquema (aplicações de usuários que acessem a base de dados, diagramas, etc.) terá de ser revisado e, muitas vezes, alterado.

O ideal é que haja *independência dos dados* em relação às aplicações que acessam aqueles dados. No entanto, essa independência é bastante difícil de alcançar, mas uma vez que alterada a estrutura (ou esquema) de uma base de dados, há a necessidade de se alterar também as aplicações que acessam aquela base. Por exemplo, imagine que depois que a base de dados foi implementada e a aplicação do usuário foi desenvolvida, é percebido que uma das tabelas da base de dados precise ser dividida em duas. A aplicação que acessava apenas a tabela X, agora tem que acessar as tabelas X e Y.

Consequentemente, a aplicação terá que ser alterada para refletir as mudanças que foram feitas no esquema da base de dados. Assim, a independência de dados lógica (alteração no esquema lógico de uma base de dados sem necessidade de alterar outros diagramas ou a aplicação) é muito difícil de ser alcançada e, portanto, deve ser evitada.



É claro que existem situações em que a alteração na base de dados pode ser feita sem que ocorram maiores transtornos. Por exemplo, precisa-se criar um índice na base de dados para aumentar o desempenho da mesma ou precisa-se alterar o tipo de dado de um atributo (por exemplo, alterar de *smallint* para *int*) isso não exigirá alterações nem nos diagramas, nem na aplicação. A esse tipo de independência denomina-se de **independência física dos dados**, onde é possível alterar o esquema físico da base de dados sem a necessidade de reescrever aplicações dos usuários ou alterar os modelos.

## Cuidado!

Às vezes, mesmo mudanças simples, como a do exemplo acima, provocam grandes mudanças na aplicação. Por exemplo, o *bug* do milênio foi causado pelos tipos de dados utilizados. Em alguns casos mesmo só mudando os tipos de dados será necessário rever toda a aplicação.

Agora, que se explicou a importância de uma boa modelagem, vamos entender o que é um modelo.

Um modelo de dados compreende a descrição dos dados que devem ser armazenados pelo sistema e como esses dados devem se relacionar. Para que seja possível fazer essa descrição, é utilizada uma linguagem de modelagem, que pode ser textual ou gráfica. Um modelo de dados também deve explicitar os tipos dos dados armazenados e as restrições que esses dados possuem.

Os modelos de dados podem ser basicamente de dois tipos:

- **Modelo de dados conceitual**, que fornece uma visão mais próxima do modo como os usuários visualizam os dados realmente;
- **Modelo de dados lógico**, que fornece uma visão mais detalhada do modo como os dados estão realmente armazenados no computador.



É importante destacar que algumas literaturas tratam o modelo resultante do Modelo Lógico, que é dependente de um SGBD específico, de Modelo Físico.

Um ponto importante a se destacar aqui é que para ser possível criar o modelo da base de dados é necessário grande interação com o usuário ou o responsável pela análise de requisitos do sistema.

Essa interação se faz necessária uma vez que para o projetista da base de dados poder desenvolvê-la, ele deve ter uma clara compreensão do que o usuário espera do sistema, que tipos de relatórios o usuário espera que este disponibilize, bem como saber quais são os objetivos do sistema.



É importante destacar ainda que o primeiro passo antes de modelar e desenvolver um sistema é identificar as necessidades e definir os objetivos do projeto. Ou seja, deve estar claro para o desenvolvedor o que o cliente quer e o que ele espera do *software* que será desenvolvido.



## Atividades

- 1) Dê três exemplos de situações, diferentes das citadas no capítulo 1, nas quais seria necessário utilizar uma base de dados.
- 2) Para cada uma das situações que você descreveu no exercício 1, descreva os dados que seriam úteis armazenar na base de dados.
- 3) O que significa dizer que as informações que vamos armazenar em uma base de dados devem ser úteis? Para quem elas devem ser úteis?
- 4) Defina o que caracteriza um Sistema de Banco de Dados computacional.
- 5) Qual a diferença entre uma base de dados e um Sistema de Banco de Dados?
- 6) Cite e explique as principais características de um SGBD.
- 7) O Access da Microsoft pode ser considerado um SGBD? Justifique sua resposta.
- 8) Com base no que foi trabalhado neste primeiro capítulo, explique qual a importância de uma boa modelagem de dados?
- 9) Faça uma síntese explicando o que é um modelo de dados e para que ele serve.
- 10) Vimos que é importante a participação do usuário na modelagem de uma base de dados. Apresente três argumentos a favor dessa participação.
- 11) Por que não é necessário que o usuário final tenha conhecimento de área de banco de dados?

# Introdução ao Modelo de Entidade e Relacionamento

O modelo de Entidade e Relacionamento (ER) é um modelo conceitual e deve estar o mais próximo possível da visão que o usuário tem dos dados, não se preocupando em representar como estes dados estarão realmente armazenados.

Este modelo tem por objetivo descrever quais dados devem ser armazenados pela aplicação e quais desses dados se relacionam.

Suponha que uma escola precise armazenar informações sobre seus alunos, professores, e disciplinas. O modelo da base de dados deve informar quais dados sobre alunos, professores e disciplinas são importantes para serem armazenados. Assim, sobre os alunos, pode-se armazenar informações como nome, data de nascimento e matrícula. Sobre os professores, armazenam-se informações como nome, telefone e matrícula. A respeito das disciplinas, pode-se armazenar informações como código da disciplina e o nome da disciplina. Além dessas informações, é necessário saber qual professor é responsável por qual disciplina e quais alunos fazem a disciplina. O modelo para esse pequeno exemplo ficaria como mostra a figura 2.1.

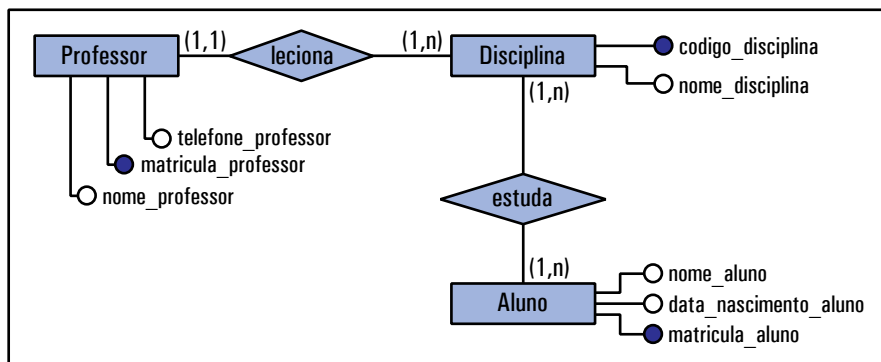


Figura 2.1 – Exemplo de modelo conceitual

Note que não estamos preocupados aqui em saber como os dados serão armazenados ou como eles devem ser implementados. A nossa preocupação é conseguir entender o que precisa ser armazenado e quais informações devem se relacionar.



A ferramenta utilizada para criação deste diagrama foi a **brModelo**. Esta é uma ferramenta para modelagem de base de dados gratuita e foi desenvolvida por Carlos Henrique Cândido, sob a orientação do Prof. Dr. Ronaldo dos Santos Mello.

Observe que o modelo da figura 2.1 informa que a base de dados contém informações sobre professores, alunos e disciplinas, mas não se preocupa em descrever o valor que esses dados armazenam.

O Modelo de Entidade e Relacionamento, apresentado na figura 2.1, utiliza uma representação gráfica chamada de Diagrama de Entidade e Relacionamento (DER).

É importante que se compreenda que o Modelo de Entidade e Relacionamento define os conceitos que serão aplicados no desenvolvimento de um Diagrama de Entidade e Relacionamento (DER). Assim, o DER será utilizado para representar graficamente o conjunto de objetos do Modelo de Entidade e Relacionamento, tais como entidades, atributos, atributos-chave, relacionamentos, restrições estruturais, etc.

Um modelo de Entidade e Relacionamento consiste em um conjunto de objetos básicos chamados **entidades** e de **relacionamentos** entre as entidades.

## Entidades e Atributos

Você desenvolve um Sistema de Informação para resolver um problema do mundo real. Assim, toda a informação referente ao problema que se quer solucionar representa parte deste mundo.

Uma **entidade** representa um conjunto de objetos do mesmo tipo do mundo real e sobre os quais se pretende armazenar dados. Por exemplo, ao desenvolver um Sistema de Informação para uma escola, as possíveis entidades desse sistema serão: professores, alunos, disciplinas, turmas e cursos.

Uma **entidade** é representada graficamente por um retângulo com o nome da entidade dentro do retângulo. Por exemplo:

Professor

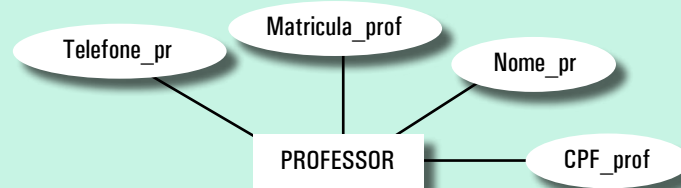
Cada uma dessas entidades armazenará um conjunto de objetos do mesmo tipo. Ou seja, ter uma entidade denominada professor significa que vários professores serão cadastrados nessa entidade e cada professor representa, portanto, um objeto da entidade.

Além de uma entidade representar objetos do mundo real, ela também deve possuir um conjunto de propriedades que a caracterize e a descreva, bem como aos seus objetos. A esse conjunto de propriedades dá-se o nome de **atributos**. Por exemplo, para a entidade Professor, é necessário armazenar dados como: CPF, nome, telefone, endereço, grau de escolaridade, número da matrícula, etc. Esses dados são os atributos da entidade Professor e são eles que identificam e caracterizam um objeto do tipo professor.

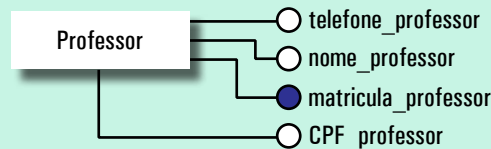
Outro exemplo, para uma entidade chamada Cadeira, os possíveis atributos dessa entidade serão: número de pernas, cor, tamanho, peso, altura, tecido, etc.

O nome dos atributos deve representar o que aquele atributo armazena.

Um atributo pode ser representado graficamente por uma elipse com o nome do atributo dentro da elipse. A elipse é ligada à entidade por uma linha, conforme exemplo:



Outra forma de representação utilizada por algumas ferramentas é representar o atributo como uma bolinha ligada à entidade e com o nome do atributo ao lado, conforme exemplo:



Uma entidade deve ter ao menos dois atributos. Uma entidade que possui apenas um atributo não é entidade e esse único atributo deveria estar em alguma outra entidade do modelo.

Todo atributo possui um tipo de dado que representa os valores permitidos para aquele atributo. A esse tipo de dados dá-se o nome de **domínio do atributo**. Por exemplo: o atributo “número de pernas” da entidade “Cadeira” é do tipo inteiro, ou seja, só permite que sejam armazenados valores inteiros para esse atributo.

Os tipos de dados dependem do SGBD que o desenvolvedor está utilizando. De forma geral, todos os SGBD disponibilizam tipos de dados como: *inteiro*, *caracter*, *real (ou float)*, *data* e *hora*.

Quando se define o tipo de um atributo, pode-se definir inclusive o tamanho máximo que o atributo vai permitir armazenar. Por exemplo, o atributo “nome” é do tipo caracter (500), ou seja, armazena no máximo 500 caracteres.

Os atributos podem ainda ser divididos em 6 categorias: simples, compostos, monovalorado, multivalorado, derivado e nulo. É importante ressaltar que os atributos podem pertencer a mais de uma categoria ao mesmo tempo. Isso significa que é comum um único atributo ser simples, monovalorado e derivado ao mesmo tempo. A seguir, serão explicadas e exemplificadas cada uma das categorias.

- **Atributo simples:** é o atributo indivisível, que não pode ou não deve ser decomposto. Por exemplo: “CPF”, “numero da matrícula”, “RG”, “preço do produto”, etc.

- **Atributo composto:** é o atributo que pode ser decomposto em outros atributos simples. Por exemplo, o atributo “endereço” pode ser decomposto em “nome da rua”, “número” e “complemento”.
- **Atributo monovalorado:** é o atributo que permite apenas o armazenamento de um valor por vez. Por exemplo, o atributo “CPF” é monovalorado porque uma pessoa possui apenas um número de CPF. Caso o CPF seja alterado ele é substituído pelo novo valor. Assim, uma pessoa **nunca** terá cadastrado mais de um CPF no mesmo campo.
- **Atributo multivalorado:** é o atributo que permite armazenar mais de um valor ao mesmo tempo no mesmo campo. Por exemplo, o atributo e-mail pode ser multivalorado uma vez que uma pessoa possui, normalmente, mais de um endereço de e-mail.

É interessante que atributos compostos sejam decompostos ainda no 1º modelo, que é o Modelo de Entidade e Relacionamento, uma vez que isso vai ter que ocorrer obrigatoriamente no modelo relacional.

Alguns atributos como, por exemplo, “nome do aluno” pode ser classificado como simples ou composto dependendo da aplicação. Se na aplicação forem realizadas consultas pelo sobrenome do aluno, é interessante que este atributo seja decomposto em dois atributos simples: “primeiro nome” e “sobrenome”. Isso ocorre por questão de desempenho.

O atributo multivalorado deve ser evitado sempre que possível. No entanto, em situações em que não é possível evitá-lo, ele deve ser representado no diagrama como multivalorado (ver como representar na figura 2.2). Quando formos passar o DER para o Modelo Relacional (capítulo 4), vamos entender o que acontece com esse atributo. Outro ponto importante: quem determina se o atributo é multivalorado ou não, muitas vezes, é o próprio usuário do sistema. No caso do exemplo do atributo “e-mail”, o usuário pode determinar que somente é necessário armazenar um e-mail e sendo assim o atributo deixa de ser multivalorado e passa a ser monovalorado.

- **Atributo nulo:** é o atributo que permite que seja inserido um valor nulo para ele. Valor nulo representa a inexistência de um valor, ou seja, significa que o usuário não precisa cadastrar um valor para o atributo e pode deixá-lo vazio. Em algumas situações, é inevitável que permitamos **valores nulos** para os atributos. Vamos usar novamente o atributo “e-mail” como exemplo. Como nem todas as pessoas possuem e-mail, esse atributo deve permitir valores nulos, porque se ele não permitir algumas pessoas não poderão se cadastrar ou terão que criar um e-mail para poder efetivar o cadastro. Novamente é o usuário quem, muitas vezes, vai definir se um atributo é obrigatório ou não.

#### **Valor nulo:**

*Valor nulo é diferente de valor zero!!! O valor nulo (representado por null em banco de dados) significa que aquele campo está vazio.*

O valor nulo na base de dados pode levar o banco a ficar inconsistente e ter baixo desempenho. Mesmo que o atributo não seja obrigatório, é interessante que ele receba um valor padrão (*default*) via aplicação ou via SGBD para evitar os valores nulos.

- **Atributo derivado:** é o atributo cujo valor para ele deriva de outro(s) atributo(s). Por exemplo, suponha que a sua entidade se chame compra e que ela tenha os seguintes atributos: “número da compra”, “data da compra”, “valor da compra”, “percentual de desconto” e “valor da compra com o desconto”. O valor para este último atributo é calculado considerando-se o “valor da compra” e o “percentual de desconto”. Assim, esse atributo é derivado porque seu valor deriva dos valores de outros atributos e é calculado automaticamente pela aplicação ou pelo SGBD.

A figura 2.2 apresenta formas de representação dos tipos dos atributos descritos anteriormente.

Representações de Atributos:

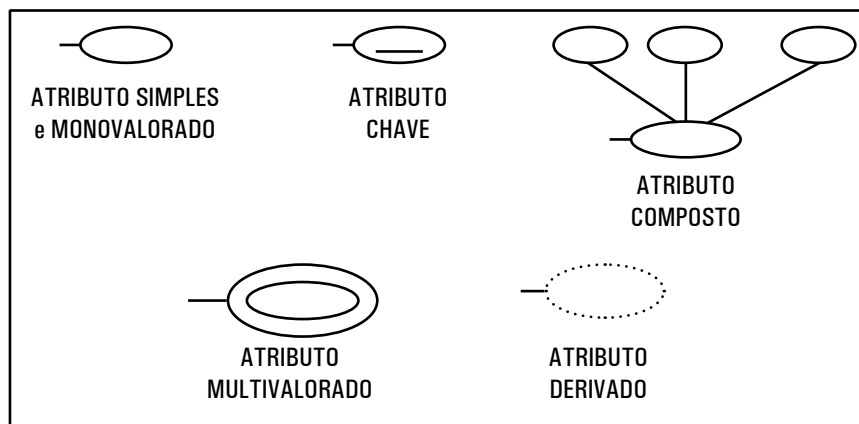


Figura 2.2 – Representação gráfica dos atributos

## Chave Primária

Um conceito importante no Modelo de Entidade e Relacionamento é o conceito de chave primária (ou *Primary Key* ou ainda **PK**). Uma **chave primária** é um atributo da entidade que identifica apenas um objeto dessa entidade. Portanto, o valor dentro de uma chave primária não poderá se repetir e também não poderá receber um valor nulo.

Por exemplo, na entidade “Professor”, tanto o atributo “CPF” quanto o atributo “matrícula” não se repetem, uma vez que esses atributos são únicos para cada indivíduo. Nesse caso, qualquer um dos dois atributos poderia ser definido como uma chave primária.

A pergunta é: Qual deles eu devo definir como uma chave primária? Bem, se mais de um atributo for único para a entidade, a escolha da chave primária vai depender do escopo do problema e de como serão realizadas as consultas. Considerando-se que a maioria dos SGBD vincula um índice à chave primária, é interessante que essa chave primária seja o atributo mais utilizado nas consultas. Para o exemplo da entidade “Professor”, a maioria das consultas seriam feitas considerando-se a matrícula do professor na instituição e não o CPF. A opção por consultas que tenham como condição a matrícula do professor se deve por causa do escopo do problema, uma vez que a matrícula do professor é um atributo muito mais específico para a instituição de ensino. Sendo assim, o atributo `matricula_professor` seria a melhor opção para ser a chave primária.

Outro ponto importante a considerar durante a decisão de qual atributo deverá ser a chave primária é que se deve dar preferência a atributos numéricos (inteiros) em vez de atributos do tipo caractere, data ou hora. Como “CPF” é um atributo do tipo caractere, ele poderia ser descartado como chave primária por esse motivo também.

## Tipos de Chave Primária

Uma chave primária pode ser simples ou composta. Uma chave primária simples é aquela que será formada por apenas um atributo. Por exemplo: `matricula_professor`.

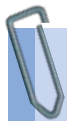
Uma chave primária composta é formada por dois ou mais atributos. Por exemplo, imagine que tenhamos uma entidade chamada “Localização” e esta entidade tem os seguintes atributos: “nome da cidade”, “nome do estado”, “nome do país”. Cada um desses atributos sozinhos não pode ser chave primária porque eles se repetem, como mostra a figura 2.3.

nome_cidade	nome_estado	nome_país
Curitiba	Paraná	Brasil
Maringá	Paraná	Brasil
Campo Grande	Mato Grosso do Sul	Brasil
Campo Grande	Rio de Janeiro	Brasil

Figura 2.3 – Exemplos de dados armazenados na entidade “Localização”

Como os atributos individualmente podem se repetir, vamos tentar encontrar uma chave primária composta. Sabemos que no Brasil um estado não tem duas cidades com o mesmo nome. Sendo assim, poderíamos criar uma chave primária composta do “nome da cidade” mais o “nome do estado”, porque o valor para esses dois atributos juntos nunca vai se repetir.

Nesse caso, o atributo “nome da cidade” não é uma chave primária, e sim **faz parte** da chave primária.



É importante destacar que não existe mais de uma chave primária por entidade. Essa chave primária poderá ser simples ou composta. Mesmo composta, é uma **única** chave primária composta de mais de um atributo.

A figura 2.4, na página 24, identifica graficamente uma chave primária, pintando de azul o atributo que corresponde a chave primária. Outra forma de identificação de uma chave primária no diagrama é grifar o nome do atributo que é chave ou que compõe a chave, como mostra a figura 2.5.

## Entidade Fraca

Entidade fraca é um tipo de entidade que não possui atributo chave primária por si só. Isso significa que não é possível definir uma chave primária, nem simples e nem composta, para a entidade.

Além disso, uma entidade fraca é dependente de uma outra entidade, e o relacionamento entre a entidade fraca e a outra entidade é normalmente **1:N**, e o N fica junto à entidade fraca.

**1:N**

Notação que indica cardinalidade. Lê-se 1 para N.

A entidade “Contato” da figura 2.4 é uma entidade fraca porque não possui um atributo (ou conjunto de atributos) que identifique um único objeto. Além disso, ela é dependente da entidade “Aluno”, porque só existe um contato se houver o aluno para aquele contato. Finalmente, o relacionamento entre as duas entidades (Aluno e Contato) é 1:N, ou seja, um aluno pode ter vários contatos, mas um contato pertence a apenas um aluno, este conceito de 1:N será aprofundado no capítulo 3.

Na figura 2.4, a entidade fraca é representada por uma linha mais grossa (da entidade ao relacionamento).

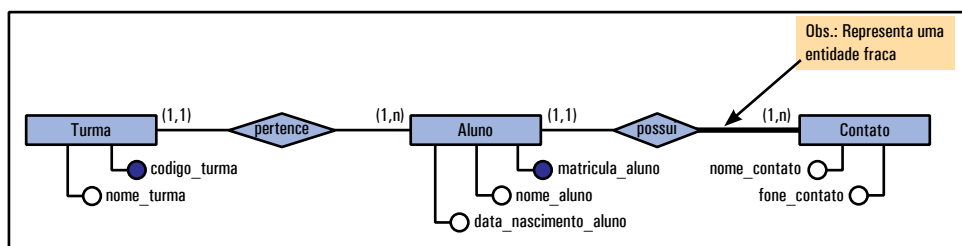


Figura 2.4 – Exemplo de uma entidade fraca

Alguns livros de banco de dados representam a entidade fraca por um retângulo duplo e o relacionamento entre entidade fraca e outra entidade por um losango duplo, como mostra a figura 2.5.

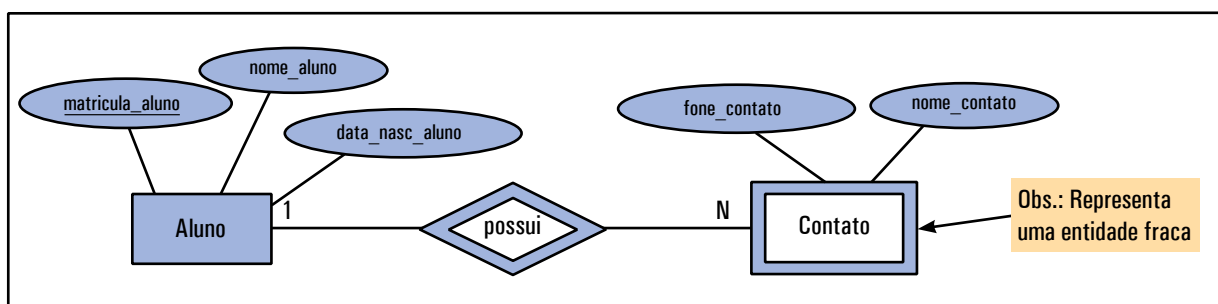


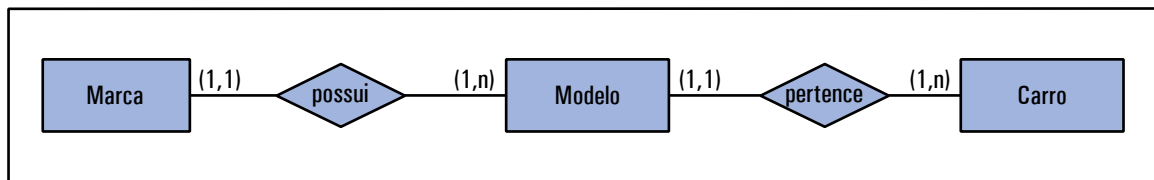
Figura 2.5 – Exemplo de uma entidade fraca usando outra notação

## Atividades

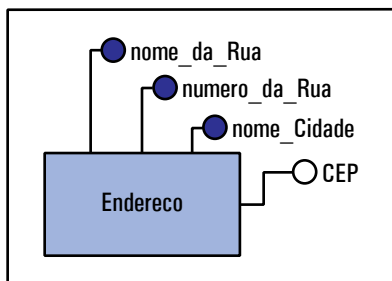
- 1) Para que serve o Diagrama de Entidade e Relacionamento?
- 2) Quando um diagrama de ER deve ser construído? Quem é responsável pela sua construção?



- 3) Um DER pode mudar com frequência? Explique.
- 4) Defina o que é uma entidade e dê pelo menos três exemplos de entidades (diferentes dos apresentados no capítulo 2).
- 5) Para cada entidade que você apresentou no exercício 4, cite 4 atributos e diga qual o domínio de cada atributo.
- 6) Explique quais os tipos de atributos que podemos ter. Para cada tipo de atributo, cite 3 exemplos.
- 7) Apresente 2 situações, diferentes das apresentadas no capítulo 2, em que se justifique o uso de uma entidade fraca.
- 8) Explique o que é uma chave primária e para que ela serve. Apresente 3 exemplos de atributos que poderiam ser chave primária e explique o porquê.
- 9) Uma chave primária pode assumir o valor nulo? Justifique sua resposta.
- 10) Dado o diagrama de ER abaixo, coloque os atributos para cada entidade e marque as chaves primárias para cada entidade.



- 11) Dado o diagrama abaixo, pode-se afirmar que a entidade “Endereço” possui três chaves primárias? Justifique sua resposta.



# ER: Relacionamento, Especialização e Agregação

## Relacionamento

Um relacionamento é uma associação entre as entidades. Como vimos no capítulo 1, os dados devem ser armazenados e estarem relacionados na base de dados para que possamos utilizá-los eficientemente.

Esse relacionamento entre os dados é o que nos permite descobrir, dada duas entidades como “Aluno” e “Turma”, a qual turma um aluno pertence, conforme mostra a figura 3.1.

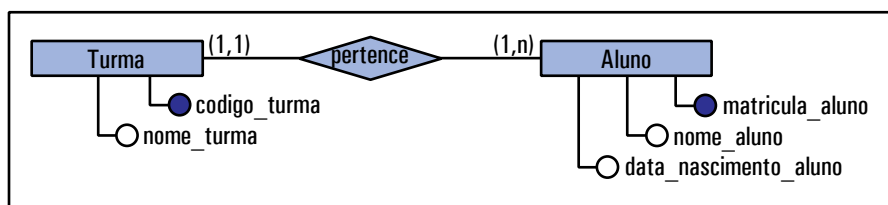


Figura 3.1 – Exemplo de relacionamento

Um relacionamento é representado por um losango com o nome do relacionamento no centro. O nome do relacionamento representa a relação que existe entre os objetos das entidades. O nome do relacionamento pode ser um verbo como, por exemplo: pertence, leciona, estuda, possui, etc.; ou também pode ser uma composição dos nomes das entidades como por exemplo “Aluno\_Turma” em vez de “pertence”.

Um relacionamento pode ter atributos. Esses atributos são denominados de **atributos descritivos**. O exemplo da figura 3.2 apresenta uma situação na qual é necessário um atributo descritivo. Imagine que seja necessário armazenar a data que um professor lecionou

determinada disciplina. O atributo “data” não pertence nem à entidade “Professor” e nem à entidade “Disciplina”. Esse atributo pertence ao relacionamento “leciona”, ou seja, é um atributo do relacionamento. E ele só deve ser preenchido com um valor, quando for feita a relação entre professor e disciplina.

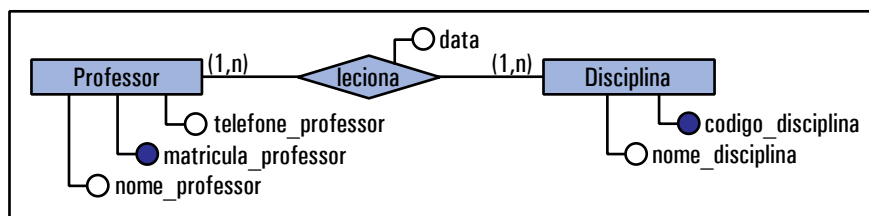


Figura 3.2 – Exemplo de um atributo descritivo



O relacionamento entre uma entidade fraca e outra entidade é chamado **relacionamento identificador**, e este relacionamento não possui atributos descritivos.

## Restrições de Mapeamento ou Cardinalidade

As restrições de mapeamento ou cardinalidade expressam o número de objetos de uma entidade ao qual outra entidade pode ser associada, via um relacionamento.

Para descobrir a cardinalidade de um relacionamento, a pergunta que deve ser feita é: “Se eu pegar um único objeto da minha entidade X, a quantos objetos da entidade Y ele pode se associar?”

Isto é, se eu pegar o objeto “Elaini” da entidade “Professor”, esse objeto “Elaini” poderá lecionar quantas disciplinas da entidade “Disciplina”? E se eu pegar o objeto “Banco de Dados” da entidade “Disciplina”, a quantos professores ele estará relacionado?

Esse número de associações entre objetos pode ser 0, 1 ou vários (representado por N). A figura 3.1 apresenta-nos um diagrama com cardinalidade 1 para N. Isso significa que uma turma pode ter um ou vários alunos e que um aluno pode pertencer a uma e apenas uma turma. Assim, não teremos o mesmo aluno em duas turmas diferentes.

Na figura 3.2, a cardinalidade do modelo é N para N, permitindo que um professor leccione várias disciplinas e que uma disciplina seja ministrada por mais de um professor. Por exemplo, a disciplina de Banco de Dados poderia ser ministrada pela prof.<sup>a</sup> Elaini e pelo prof. João. Além de Banco de Dados, a prof.<sup>a</sup> Elaini poderia leccionar também, a disciplina de Análise de Projetos.

A cardinalidade dos relacionamentos podem ser de 3 tipos: Um para Um; Um para Muitos; e Muitos para Muitos.

### Cardinalidade Um para Um (1:1)

A cardinalidade Um para Um (1:1) ocorre quando um objeto de uma entidade pode se relacionar a apenas um objeto de outra entidade e vice-versa. Imagine que você esteja desenvolvendo um sistema para uma rede de escolas. Cada escola terá um diretor (que é um professor da escola), e este diretor só poderá dirigir uma escola. Sendo assim, o relacionamento “dirige” entre as entidades “Escola” e “Professor” será 1:1, como mostra a figura 3.3.

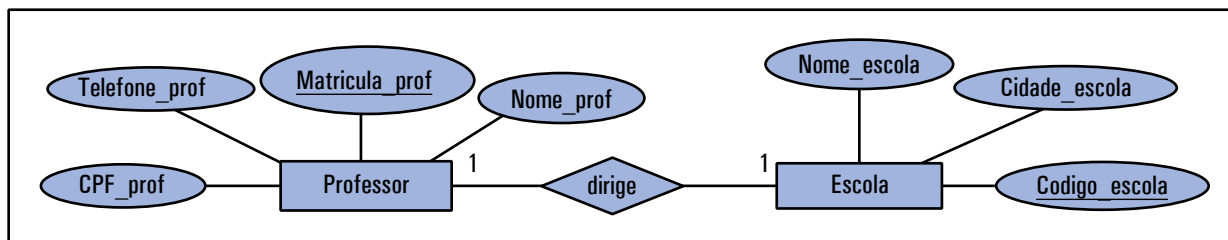


Figura 3.3 – Exemplo de um relacionamento Um para Um

O relacionamento no diagrama da figura 3.3 pode ser lido da seguinte forma: Um professor dirige uma escola e uma escola é dirigida por um professor.

A notação utilizada na figura 3.3 é a que normalmente aparece nos livros de banco de dados. No entanto, algumas ferramentas de modelagem utilizam uma notação um pouco diferente.

Utilizando a **ferramenta de modelagem brModelo**, o diagrama da figura 3.3 ficaria como mostra a figura 3.4.

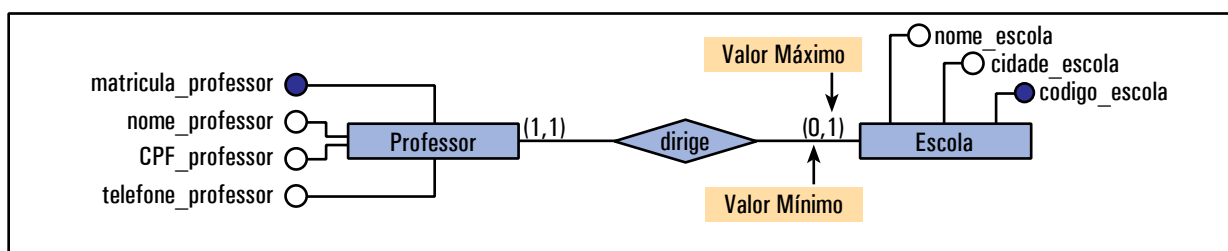


Figura 3.4 – Exemplo de um relacionamento Um para Um no brModelo

Observe que a cardinalidade no diagrama da figura 3.4 possui dois valores entre parênteses. O primeiro valor corresponde ao valor mínimo, e o segundo ao valor máximo. Assim, um professor dirige no mínimo nenhuma (zero), e no máximo 1 escola, e uma escola é dirigida por apenas um professor.

Daqui para frente, os exemplos serão feitos utilizando-se a ferramenta de modelagem de dados brModelo.

## Cardinalidade Um para Muitos (1:N)

A cardinalidade Um para Muitos ocorre quando um objeto de uma entidade pode se relacionar a vários objetos da outra entidade, mas o contrário não é verdadeiro. Imagine que um professor possa trabalhar em apenas uma escola da rede municipal e que uma escola possa ter vários professores. Nessa situação, teremos um relacionamento 1:N, conforme mostra a figura 3.5.

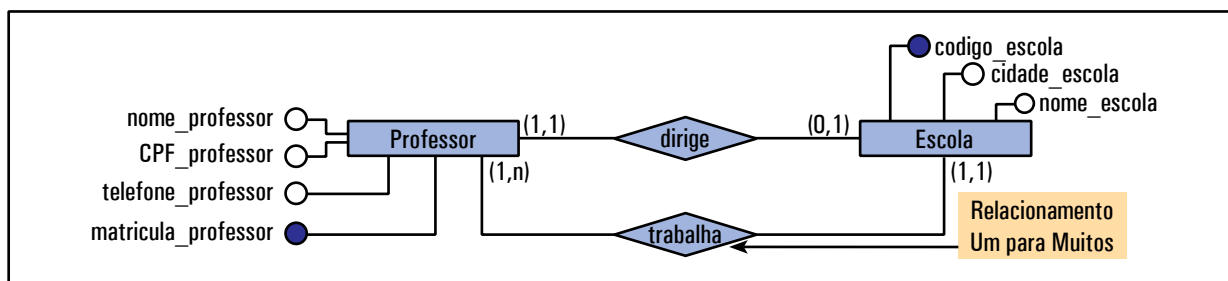


Figura 3.5 – Exemplo de um relacionamento Um para Muitos

A leitura do diagrama da figura 3.5 para o relacionamento “trabalha” (Um para Muitos) é feita da seguinte forma: Um professor pode trabalhar em 1 e apenas 1 escola, mas uma escola pode ter 1 ou vários professores trabalhando nela.

Observe que, neste diagrama, existem 2 relacionamentos (“dirige” e “trabalha”) entre as mesmas entidades. Isso é perfeitamente normal e ocorre porque esses relacionamentos representam situações bem diferentes e, portanto, devem ser separados.

## Cardinalidade Muitos para Muitos (N:N)

A cardinalidade Muitos para Muitos (N:N) ocorre quando um objeto de uma entidade pode se relacionar a vários objetos da outra entidade e vice-versa. Imagine que um professor possa ministrar aulas em várias disciplinas e que uma disciplina possa ser ministrada por vários professores. Nesse caso, temos um relacionamento N:N, como mostra a figura 3.6.

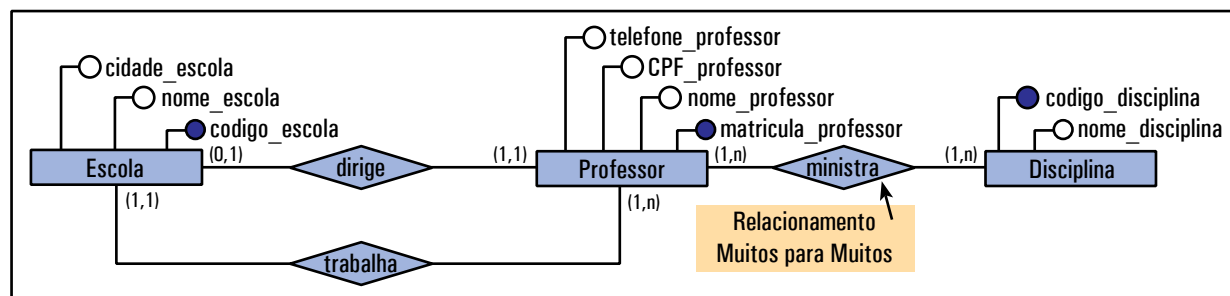


Figura 3.6 – Exemplo de um relacionamento Muitos para Muitos

A leitura do diagrama para o relacionamento “ministra” (Muitos para Muitos) é feita da seguinte forma: Um professor ministra uma ou várias disciplinas e uma disciplina pode ser ministrada por um ou vários professores.

## Relacionamento Ternário ou Maiores

Até agora, todos os exemplos apresentados são de relacionamentos binários, ou seja, entre duas entidades. No entanto, um relacionamento pode ocorrer também entre três ou mais entidades.

Considere o exemplo em que um professor pode ministrar disciplinas para diferentes turmas. Nesta situação, um professor poderia dar aula de uma ou mais disciplinas, e poderiam existir um ou mais professores que dessem a mesma disciplina em diferentes turmas. Com a mudança do ano, o professor que dava aula da disciplina de Banco de Dados na turma A poderia pegar a mesma disciplina para a turma B, e deixar a turma A para outro professor.

Para que as informações possam ser armazenadas e recuperadas de forma completa, teremos que criar um relacionamento ternário entre as entidades, conforme mostra a figura 3.7.

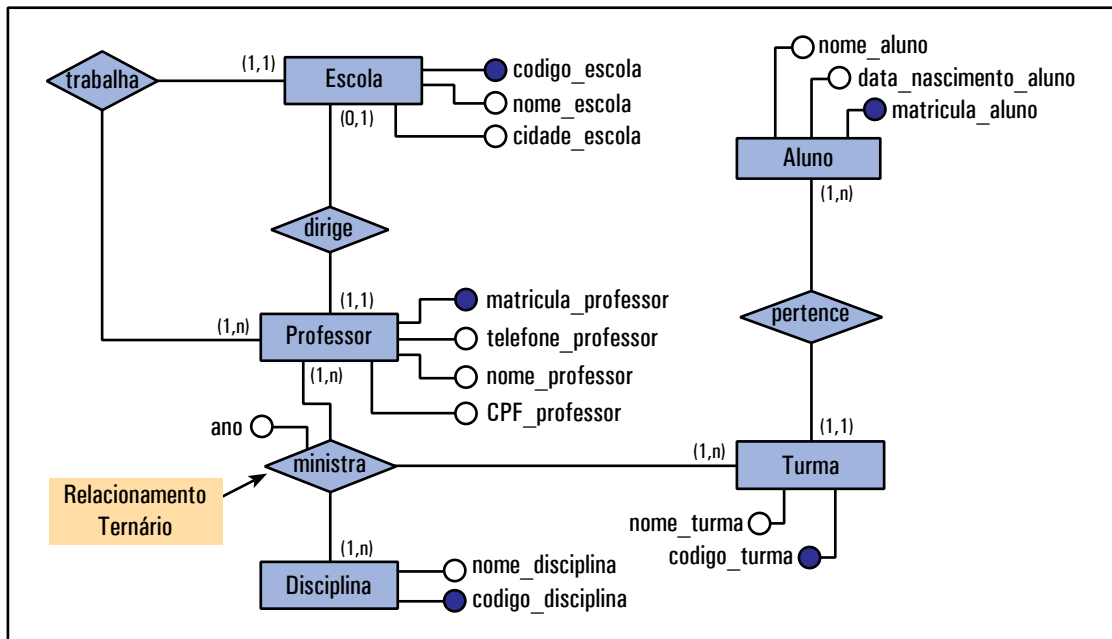


Figura 3.7 – Exemplo de um relacionamento ternário

Poderíamos tentar criar três relacionamentos binários para o problema descrito acima, no entanto, se fizermos isso, não teremos as informações de forma completa como a temos no relacionamento ternário.

O relacionamento “ministra” possui ainda o atributo descritivo “ano” que permite distinguir qual foi o professor que ministrou determinada disciplina para uma turma em um determinado ano.

Relacionamentos maiores que ternários devem ser evitados (se possível) porque são difíceis de serem compreendidos e de serem implementados, tornando a relação bastante complexa.

## Relacionamento Recursivo

Um tipo especial de relacionamento é aquele que relaciona objetos de uma mesma entidade. Esse tipo de relacionamento é denominado de **relacionamento recursivo** ou **autorrelacionamento**.

Imagine que existam alguns poucos alunos que representem grupos de outros alunos em reuniões e assuntos estudantis. Nesse caso, um objeto aluno representa vários outros objetos dentro da mesma entidade. A figura 3.8 apresenta o relacionamento “representante” como um relacionamento recursivo.

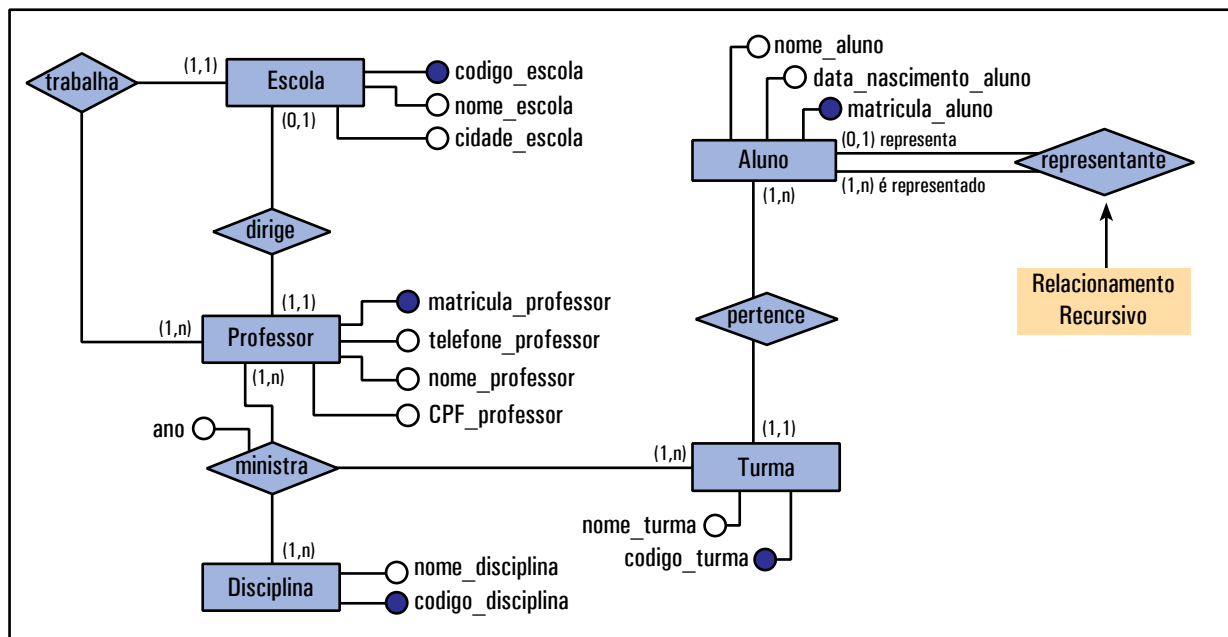


Figura 3.8 – Exemplo de um relacionamento recursivo

Observe que existe um texto escrito sobre as linhas do relacionamento recursivo. Esse texto representa o papel que o objeto de uma entidade desempenha no relacionamento. Neste caso, o papel distingue a participação dos alunos no relacionamento, que pode ser a participação de “representar alunos” ou “ser representado por alunos”. A descrição dos papéis não é obrigatória em um relacionamento recursivo. A cardinalidade também não é obrigatória, mas algumas ferramentas de modelagem exigem que você as coloque.

## Especialização

Especialização consiste na subdivisão de uma entidade mais genérica (ou entidade pai) em um conjunto de entidades especializadas (ou entidades filhas).

Isso ocorre quando um conjunto de entidades pode conter subgrupos de entidades com atributos específicos a cada subgrupo. Esse processo tem por finalidade refinar o esquema da base de dados, tornando-o mais específico.

A figura 3.9 apresenta um exemplo de especialização entre as entidades “Pessoa”, “Professor” e “Aluno”.

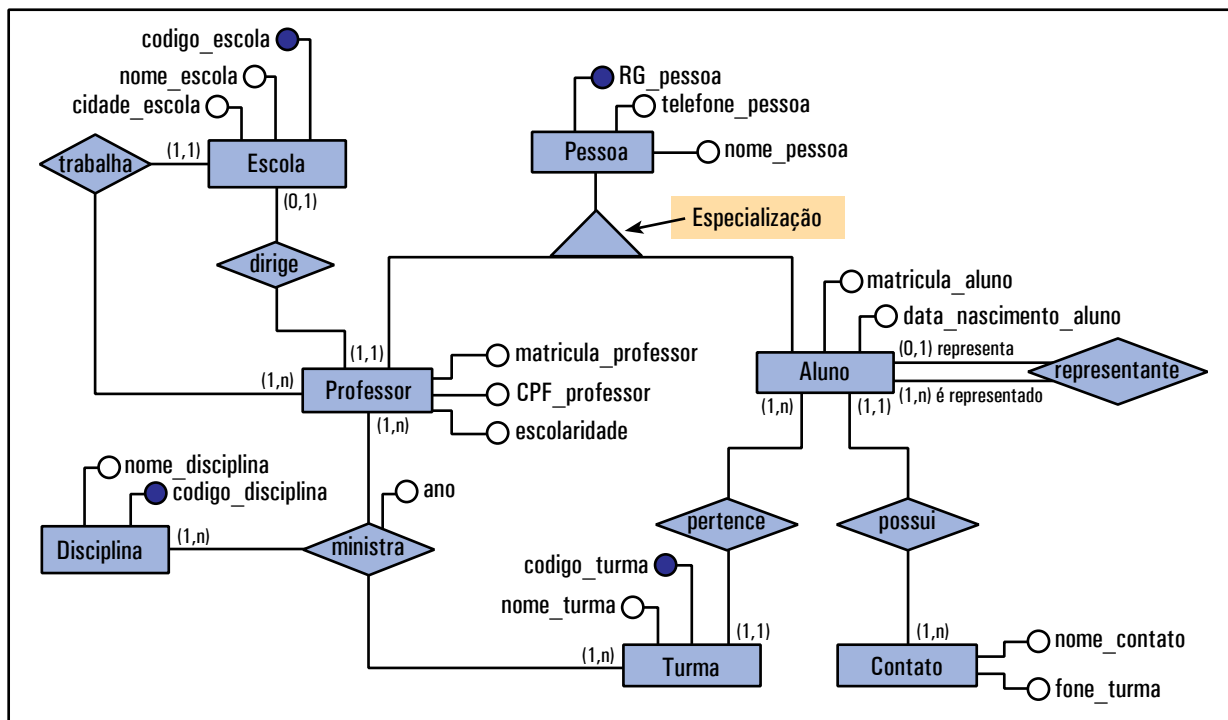


Figura 3.9 – Exemplo de uma especialização

Observe que a especialização é indicada no diagrama por um triângulo, e as entidades filhas estão relacionadas com a entidade pai por meio do triângulo.

As entidades filhas “herdam” todos os atributos da entidade pai e, portanto, não se devem repetir os atributos da entidade pai nas entidades filhas. Isso significa que os atributos que aparecem na entidade pai são os atributos que existem em comum entre as entidades filhas. No diagrama da figura 3.9 os atributos da entidade “Pessoa” (RG\_pessoa, telefone\_pessoa e nome\_pessoa) serão herdados pelas entidades filhas “Professor” e “Aluno”.

Também não é necessário indicar uma chave primária para as entidades filhas. A chave primária para as entidades filhas será definida no modelo relacional.

Para utilizar uma especialização, deve-se analisar antes se as entidades filhas possuem atributos específicos ou relacionamentos específicos ou ainda outra especialização. Observe que na figura 3.9 as entidades filhas possuem tanto atributos específicos quanto relacionamentos específicos a elas.

Se a entidade filha não tiver nem atributo específico, nem relacionamento específico ou nem outra especialização, como mostra a figura 3.10, então ela não deve ser especializada. Neste caso, dizemos que o modelo deve ser generalizado, ou seja, deve passar por um processo de **generalização**.



A generalização é o processo inverso da especialização. Em vez de subdividir a entidade, cria-se uma entidade mais genérica e adiciona-se um atributo denominado “tipo” que identifica o tipo do objeto, como mostra a figura 3.11. Para o exemplo apresentado na figura 3.11, o atributo “tipo” identificará se o telefone é do tipo “celular” ou “residencial”.

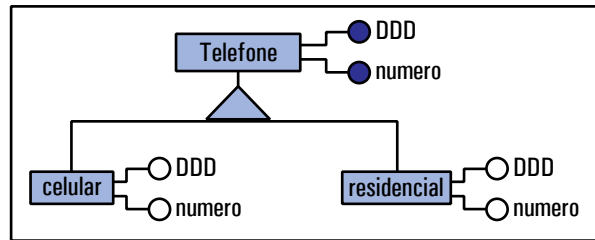


Figura 3.10 – Exemplo de uma especialização sem necessidade

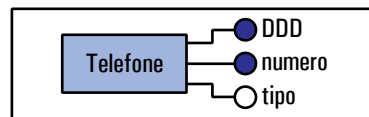


Figura 3.11 – Exemplo de uma generalização

No exemplo da figura 3.9, a especialização ocorreu entre duas entidades filhas. No entanto, uma especialização pode ter quantas entidades filhas forem necessárias, inclusive apenas uma, se for o caso. Uma entidade filha pode ser entidade pai para outra especialização, como mostra a figura 3.12, em que a entidade “Alimento” é uma entidade filha de “Produto” e é entidade pai de “Perecível”.

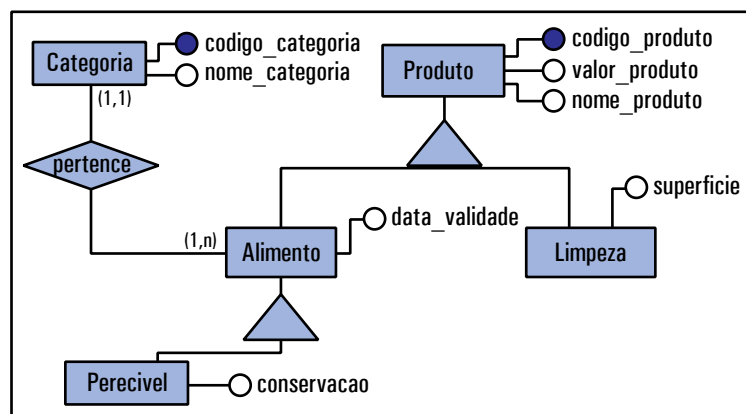


Figura 3.12 – Exemplo de uma entidade filha especializada

# Agregação

A **agregação**, ou **entidade associativa**, ocorre quando precisamos relacionar dois relacionamentos entre si.

Imagine que tenhamos duas entidades “Cliente” e “Produto” ligadas pelo relacionamento “Compra”. Agora, suponha que tenhamos que modificar esse modelo de modo que seja necessário saber quantas prestações serão pagas em uma compra. Relacionar a entidade “Prestação” com “Cliente” ou com “Produto” não faz sentido, uma vez que as prestações serão referentes à compra efetuada. Sendo assim, a entidade “Prestação” deve se relacionar à “Compra”, como mostra a figura 3.13. O retângulo desenhado em volta do relacionamento indica a agregação.

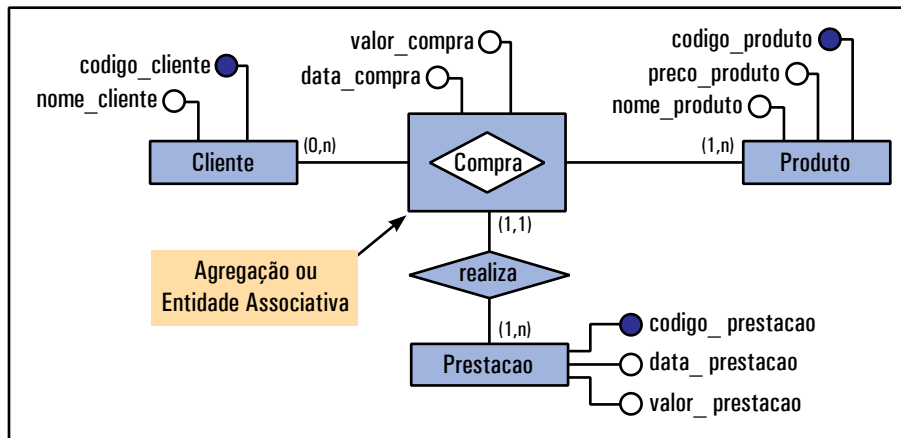


Figura 3.13 – Exemplo de agregação

Podemos também reescrever o modelo sem utilizar agregação. Nesse caso, o relacionamento “Compra” seria transformado em uma entidade que poderia ser relacionada à “Prestação”, como mostra a figura 3.14.

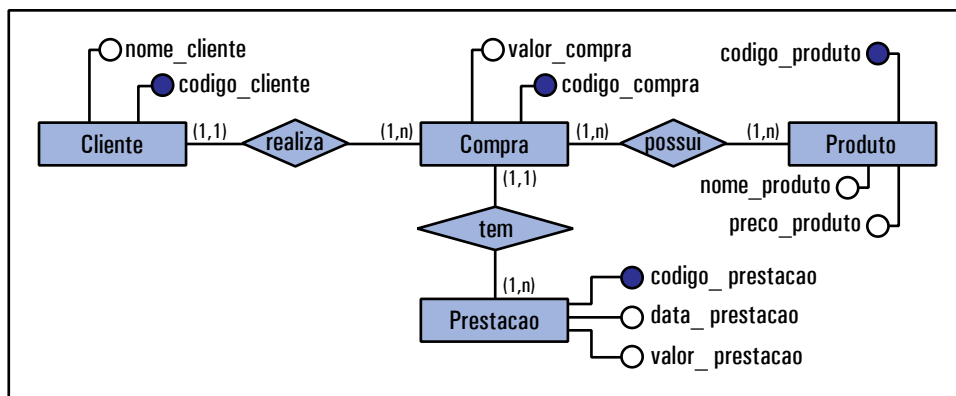


Figura 3.14 – Exemplo sem usar agregação

É importante ressaltar que um mesmo problema pode ter diferentes interpretações, e assim gerar diagramas diferenciados. Isso não significa que apenas um dos diagramas está certo.

Isso ocorre porque muitas vezes a informação que o usuário passa tem mais de uma interpretação. É interessante, portanto, que o desenvolvedor do sistema tenha muita atenção ao que o usuário precisa para poder desenvolver algo que atenda às necessidades desse cliente.



## Atividades

- 1) Apresente uma situação diferente da apresentada no capítulo 3, na qual você tenha um relacionamento com atributos descritivos.
- 2) Apresente duas situações em que você utilizaria um relacionamento ternário.
- 3) Para as situações do exercício 2, reescreva o modelo sem utilizar o relacionamento ternário. Destaque os problemas, se houver, desse novo modelo.
- 4) Apresente duas situações nas quais um relacionamento recursivo seria necessário.
- 5) Seria possível reescrever as situações do exercício 4 sem utilizar relacionamento recursivo? Explique.
- 6) Em que situações devemos utilizar uma especialização? Dê 3 exemplos de especialização.
- 7) Em que situação devemos fazer uma generalização? Dê um exemplo diferente do apresentado no capítulo 3.
- 8) Dê um exemplo de uma situação, diferente da apresentada neste capítulo, em que você poderia utilizar uma agregação.
- 9) Para a situação do exercício 8, você poderia reescrever o modelo sem utilizar a agregação? Se for possível, reescreva o modelo, e se não for possível, explique por que não conseguiu.
- 10) Suponha que você tenha uma entidade denominada “Empregado”, que armazena dados dos empregados. Agora, você recebe a informação de que é necessário armazenar também os gerentes. Os gerentes também são empregados. Como você faria o modelo para que mais tarde fosse possível identificar quem é gerente e quem não é?

- 11) Dada a seguinte situação: desenhe o DER, coloque os atributos para cada entidade e determine a chave primária para cada entidade.

Uma turma de segundo ano do Ensino Médio resolveu formar um clube do livro. Como esses alunos são do curso Técnico em Informática, eles resolveram desenvolver um sistema para controlar isso. O clube vai funcionar assim: Cada aluno deve selecionar alguns livros que tenha em casa para disponibilizar ao clube. Esses livros serão colocados em caixas separadas por área (por exemplo: romance, ficção, policial, etc.) Sobre o livro, é necessário saber: nome do livro, nome do autor, editora, ano de publicação, código da área, nome da área e ISBN. Não pode haver livros repetidos. Sobre os alunos, precisamos saber a matrícula do aluno, nome, telefone e turma a que ele pertence. Os livros podem ser emprestados pelos alunos cadastrados, e quando isso acontece é necessário saber a data do empréstimo e a data da devolução. Se um aluno atrasar a devolução mais de uma vez ele será banido do grupo.

- 12) Dada a seguinte situação: desenhe o DER, coloque os atributos para cada entidade e determine a chave primária para cada entidade.

Uma concessionária que trabalha com venda de veículos deseja criar uma base de dados para o seu negócio. Para qualquer veículo, sabemos o número do chassi, número da placa, cor, ano de fabricação, quilometragem, código da marca, nome da marca, código do modelo e nome do modelo. Todo carro pertence a um modelo, e este modelo pertence a uma marca. Como a concessionária vende veículos usados de diferentes marcas, é interessante haver um cadastro para as marcas e um cadastro para os modelos. Uma pessoa pode assumir um dos seguintes papéis em relação a concessionária: corretor ou comprador. Sobre o comprador do veículo, tem-se CPF, nome, estado civil e, se for casado, os dados do cônjuge (como nome e CPF). Sobre os corretores, tem-se número da matrícula, nome e data de admissão. Um corretor negocia com um comprador a venda de um veículo. Sobre a venda, são necessárias as seguintes informações: data da venda, valor da venda e valor da comissão do corretor.

# Introdução ao Modelo Relacional

O **modelo relacional** é um modelo lógico, utilizado em banco de dados relacionais. Nesse modelo, começamos a nos preocupar em **como** os dados devem ser armazenados e em **como** criaremos os relacionamentos do modelo conceitual. É também nessa etapa que definiremos o SGBD que será utilizado, bem como os tipos de dados para cada atributo.

Este modelo tem por finalidade representar os dados como uma coleção de tabelas e cada linha de uma tabela representa uma coleção de dados relacionados. Para descrever uma tabela no modelo relacional, usamos o nome da tabela seguida dos atributos entre parênteses. Para identificar a chave primária, devem-se sublinhar o(s) atributo(s) correspondente(s) a ela. O tipo de cada atributo também deve aparecer no modelo relacional, como mostra o exemplo abaixo:

```
tbAluno(matricula_aluno: inteiro, nome_aluno: caracter(100), data_nascimento_aluno: data)
```

37

Introdução ao Modelo Relacional

O nome da tabela e das colunas são utilizados para facilitar a interpretação dos valores armazenados em cada linha da tabela. Todos os valores em uma coluna são necessariamente do mesmo tipo. Na terminologia do modelo relacional, tabela é a mesma coisa que uma relação; linha é a mesma coisa que um registro; coluna é igual a um campo ou um atributo; e tipo de dado é igual a um domínio.

## Chave Estrangeira e Integridade Referencial

Um conceito muito importante quando se fala de modelo relacional é o conceito de **chave estrangeira** (ou *Foreign Key* ou **FK**).

Uma chave estrangeira é um atributo da tabela que faz referência a uma chave primária de outra tabela ou da própria tabela. Por exemplo, suponha que tenhamos as tabelas “Aluno” e “Turma”, representadas na descrição do modelo relacional a seguir:

```
tbTurma(codigo_turma: inteiro, nome_turma: caracter(5))

tbAluno(matricula_aluno: inteiro, nome_aluno: caracter(200), data_nascimento_aluno:
data, codigo_turma: inteiro)
```

Observe que a tabela tbAluno possui o atributo codigo\_turma. Esse atributo é chave primária na tabela tbTurma e, portanto, é uma chave estrangeira na tabela tbAluno. O atributo que é a chave estrangeira deve ser do mesmo tipo e do mesmo tamanho que a sua primária correspondente.

Uma chave estrangeira pode ser identificada por um asterisco (\*) na frente do atributo, como mostra o modelo relacional abaixo:

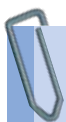
```
tbAluno(matricula_aluno:inteiro, nome_aluno: caracter(200), data_nascimento_aluno:
data, *codigo_turma: inteiro)
```

No entanto, esse tipo de notação não nos permite identificar a que chave primária de qual tabela a chave estrangeira faz referência. Para termos essa informação de forma clara, devemos usar a notação, conforme exemplo abaixo:

```
tbAluno(matricula_aluno:inteiro, nome_aluno: caracter(200), data_nascimento_aluno:
data, codigo_turma: inteiro)

codigo_turma referencia tbTurma
```

Quando dizemos “codigo\_turma referencia tbTurma” estamos dizendo que o atributo codigo\_turma faz referência à chave primária da tabela tbTurma.



Uma chave estrangeira **sempre** faz referência a uma chave primária. A chave estrangeira **nunca** fará referência a um atributo que não seja uma chave primária.

No modelo relacional é a chave estrangeira que especifica o relacionamento entre as tabelas. É através da chave estrangeira que conseguimos descobrir, por exemplo, que a aluna Anna pertence à turma do 1º ano do Curso Técnico em Informática (1TI), como mostram as figuras 4.1 e 4.2.

matricula_aluno	nome_aluno	data_nascimento_aluno	codigo_turma
100	Anna	12/05/1997	1
101	Gustavo	15/04/1996	2
102	Elaini	22/09/1995	3
103	Maria	27/06/1997	2
104	Pedro	03/12/1997	1

Figura 4.1 – Exemplo de registros para a tabela tbAluno

codigo_turma	nome_turma
1	1TI
2	2TI
3	3TI
4	1TP

Figura 4.2 – Exemplo de registros para a tabela tbTurma

O valor para uma chave estrangeira deve ser um valor que já tenha sido cadastrado na chave primária correspondente ou um valor nulo. Na tabela da figura 4.1, não poderíamos cadastrar que a aluna Maria pertence à turma de código 6, uma vez que não existe nenhum código 6 cadastrado em tbTurma (figura 4.2).

Essa restrição é o que garante a **integridade referencial** do modelo relacional. Ou seja, ela garante que não se faça referência a valores que não existam na base de dados. Imagine a confusão que seria se fosse permitido cadastrar a aluna Maria na turma de código 6. Quando fôssemos procurar a que turma Maria pertence não teríamos essa informação. Isso tornaria a base de dados inconsistente. Sendo assim, a implementação de uma chave estrangeira garante a integridade referencial da base.

Uma chave estrangeira pode também fazer referência a uma chave primária dentro da mesma tabela. Isso ocorre quando temos relacionamentos recursivos. Por exemplo:

```
tbAluno(matricula_aluno: inteiro, nome_aluno: caracter(200),
data_nascimento_aluno:data, matricula_aluno_representante: inteiro)
matricula_aluno_representante referencia tbAluno
```

Nesse caso, o atributo matricula\_aluno\_representante poderá receber o valor nulo, caso o aluno seja o próprio representante, como mostrado na figura 4.3. O valor nulo para a chave estrangeira fere a restrição de integridade referencial, mas, em algumas situações, ele é necessário (especialmente quando ainda não conhecemos os valores cadastrados na primária correspondente).

matricula_aluno	nome_aluno	data_nascimento_aluno	matricula_aluno_representante
100	Anna	12/05/1997	Null
101	Gustavo	15/04/1996	Null
102	Elaini	22/09/1995	100
103	Maria	27/06/1997	100
104	Pedro	03/12/1997	100

Figura 4.3 – Exemplo de registros para a tabela tbAluno com chave estrangeira recursiva

Observe ainda que o nome da chave estrangeira não precisa ser igual ao nome da chave primária correspondente.

# Conversão entre o Modelo de ER e o Modelo Relacional

O modelo relacional é definido usando como base o modelo de ER. Lembre-se de que o modelo relacional consiste em uma coleção de tabelas e na definição de chaves estrangeiras para relacionar essas tabelas. Sendo assim, construir o modelo relacional consiste em definir as tabelas e as chaves estrangeiras.

Existem algumas regrinhas que devem ser aplicadas para fazer a conversão de um modelo no outro. Vamos entender cada uma dessas regras.

## Atenção!

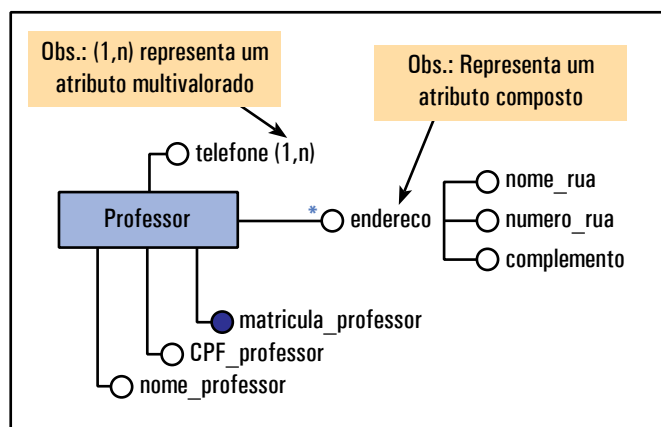
Nos exemplos que serão apresentados daqui para frente, vamos omitir o tipo de dados da descrição do modelo relacional.

## Entidade

Toda entidade do modelo de ER vira uma tabela no modelo relacional. Essa tabela terá a mesma chave primária e os mesmos atributos definidos na entidade.

Caso a entidade tenha atributos compostos, eles devem ser decompostos (se ainda não foram).

Caso a entidade tenha atributos multivalorados, para cada atributo multivalorado cria-se uma nova tabela. A tabela correspondente ao atributo multivalorado vai ter como atributos o atributo multivalorado em si, mais a chave primária da tabela onde o atributo multivalorado estava inserido (que vai passar como chave estrangeira para a nova tabela). A figura 4.4 apresenta um exemplo dessa situação.





**No modelo relacional fica:**

```
tbProfessor(matricula_professor, nome_professor, CPF_professor,  
nome_rua_professor, numero_rua_professor, complemento_professor)
```

O atributo endereço foi decomposto em atributos simples.

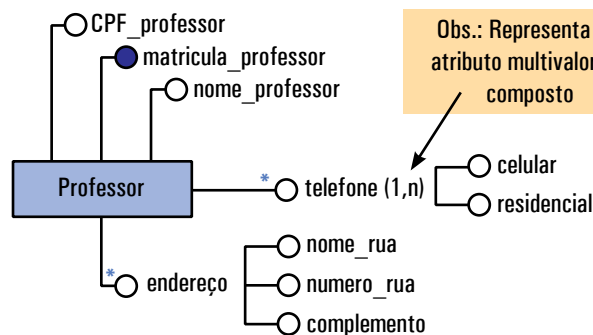
Foi criada uma tabela para o atributo multivalorado “telefone”

```
tbTelefoneProfessor(matricula_professor, telefone_professor)  
matricula_professor referencia tbProfessor
```

A chave primária de tbProfessor passou para tbTelefoneProfessor como chave estrangeira e ajuda a formar a PK dessa tabela.

Figura 4.4 – Conversão de entidade, atributo composto e multivalorado para o modelo relacional

Se o atributo multivalorado for multivalorado composto, a nova tabela (correspondente ao atributo multivalorado) deverá conter o atributo multivalorado decomposto e sua chave primária será a combinação da chave estrangeira com um ou mais atributos da nova tabela, como mostra a figura 4.5.



**No modelo relacional fica:**

```
tbProfessor(matricula_professor, nome_professor, CPF_professor, nome_rua_  
professor, numero_rua_professor, complemento_professor)
```

```
tbTelefoneProfessor(matricula_professor, telefone_residencial, telefone_celular)  
matricula_professor referencia tbProfessor
```

Figura 4.5 – Conversão de atributo multivalorado composto para o modelo relacional

## Entidade Fraca

Para cada entidade fraca no modelo ER, é criada uma tabela no modelo relacional, incluindo todos os atributos da entidade fraca, mais a chave primária da entidade com a qual a entidade fraca se relaciona (que passa como uma chave estrangeira). A chave primária desta nova tabela será composta por um ou mais atributos da entidade fraca mais a chave estrangeira. A figura 4.6 mostra um exemplo de conversão de entidade fraca para o modelo relacional.

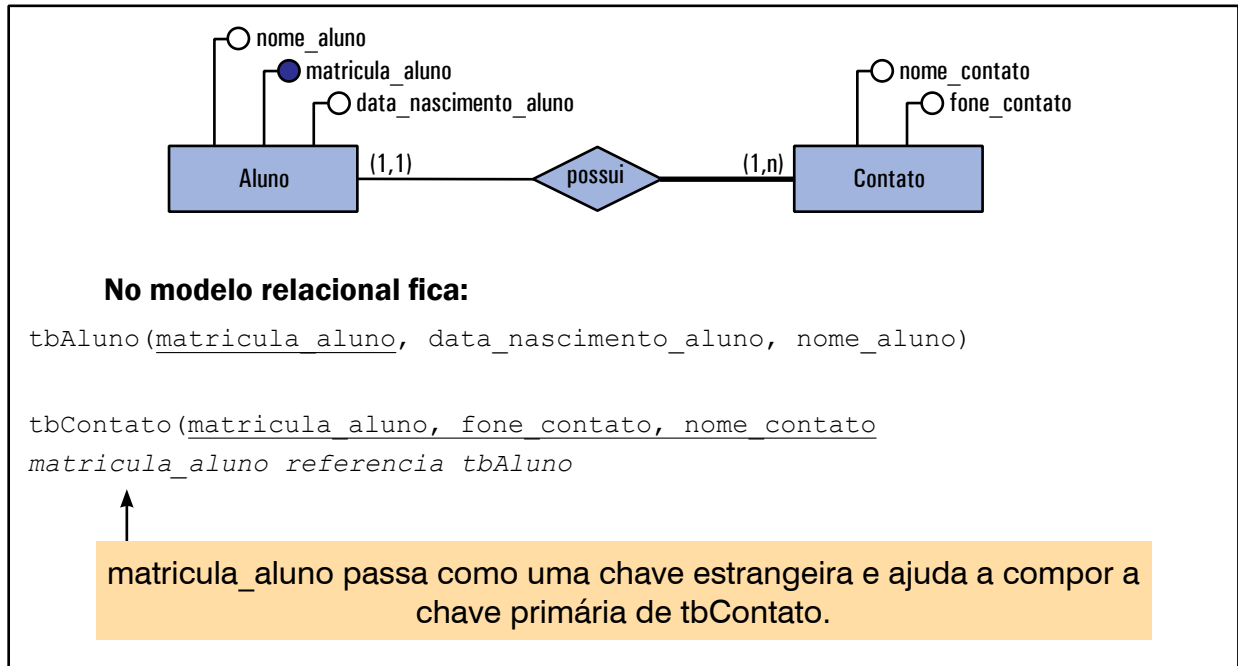


Figura 4.6 – Conversão de entidade fraca para o modelo relacional

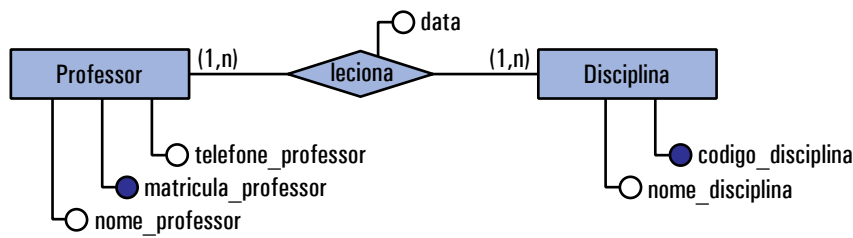
No exemplo da figura 4.6, aconteceu de a chave primária de tbContato ser composta de todos os atributos. No entanto, se apenas dois atributos pudessem identificar um único registro da tabela, não seria necessário utilizar todos os atributos na composição da chave primária.

### Atenção!

É importante ressaltar que devemos sempre escolher o menor número de atributos possíveis para compor uma chave primária.

## Relacionamento N para N

Todo relacionamento com cardinalidade – Muitos para Muitos – entre duas entidades, vira uma nova tabela. Essa nova tabela irá conter todos os atributos descritivos do relacionamento (se houver) mais as chaves primárias das entidades que fazem parte desse relacionamento. As chaves primárias que vão passar para a nova tabela passam como chaves estrangeiras. A chave primária da nova tabela será composta pelas chaves estrangeiras e, se houver necessidade, por algum atributo descritivo. A figura 4.7 mostra um exemplo de conversão de relacionamento N:N para o modelo relacional.



### No modelo relacional fica:

`tbProfessor(matricula_professor, nome_professor, telefone_professor)`

`tbDisciplina(codigo_disciplina, nome_disciplina)`

`tbProfessorDisciplina(matricula_professor, codigo_disciplina, data)`

*matricula\_professor referencia tbProfessor*

*codigo\_disciplina referencia tbDisciplina*

Foi criada uma tabela  
para o relacionamento  
N:N

Os atributos chave primária das entidades que ligam o  
relacionamento passam para a nova tabela como chaves  
estrangeiras

Figura 4.7 – Conversão de relacionamento N:N para o modelo relacional

Neste exemplo, utilizamos como nome da nova tabela o nome das entidades que participavam do relacionamento para que fique claro para quem for ler o modelo relacional que essa tabela teve origem num relacionamento Muito para Muitos.

A chave primária, nesse caso, foi composta de três atributos uma vez que só o código da disciplina e a matrícula do professor podem se repetir porque um professor pode lecionar uma disciplina várias vezes em anos diferentes. Por isso, a data foi utilizada como parte da chave primária.

## Relacionamento 1 para N

Relacionamentos com cardinalidade 1:N não geram nova tabela. No entanto, para que se possa manter o relacionamento cria-se uma chave estrangeira na entidade que possui a cardinalidade N. Se o relacionamento tiver atributos descritivos, esses atributos irão “seguir” a chave estrangeira, ou seja, ficarão na mesma tabela que a chave estrangeira ficar (a de cardinalidade N). Veja o exemplo apresentado na figura 4.8.

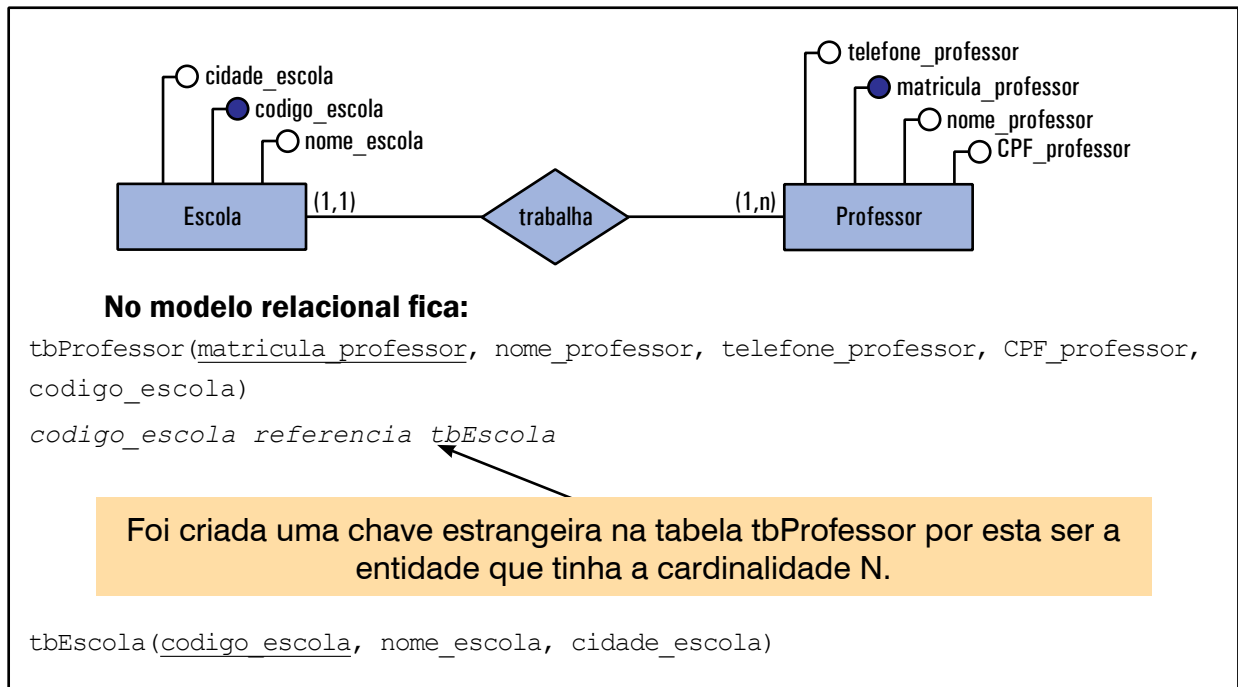


Figura 4.8 – Conversão de relacionamento 1:N para o modelo relacional

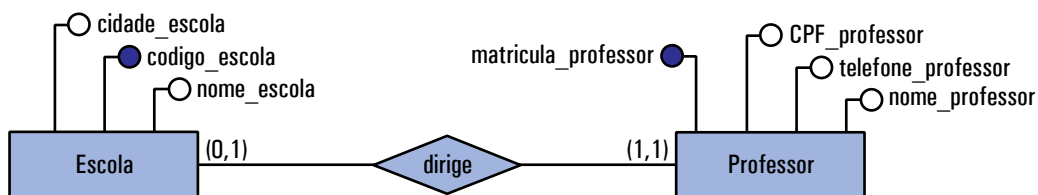
## Relacionamento 1 para 1

Relacionamentos com cardinalidade 1:1 entre duas entidades não geram uma nova tabela. No entanto, deve-se escolher a chave primária de uma das entidades ligadas ao relacionamento e inseri-la como chave estrangeira na outra tabela.

A questão aqui é a seguinte: Qual tabela deve receber a chave estrangeira já que a cardinalidade máxima das duas entidades é 1? Para que possamos decidir quem recebe a chave estrangeira, é necessário considerar o tipo de participação das entidades no relacionamento. O tipo de participação pode ser **total** ou **parcial**.

A **participação total** ocorre quando todos os objetos de uma entidade participam do relacionamento e a **participação parcial** ocorre quando apenas alguns objetos da entidade participam do relacionamento.

Por exemplo, suponha que tenhamos as entidades “Escola” e “Professor”, nas quais se percebe que uma escola sempre tem um professor que é diretor, mas nem todo professor é um diretor, como mostra a figura 4.9. Nesse tipo de relacionamento, a entidade “Escola” tem **participação total**, uma vez que **toda** escola terá um diretor. A entidade “Professor” tem **participação parcial**, uma vez que nem todo professor é diretor. Sendo assim, a entidade que tem participação total “Escola” é que deve receber a chave estrangeira.



### No modelo relacional fica:

`tbProfessor(matricula_professor, nome_professor, telefone_professor, CPF_professor)`

`tbEscola(codigo_escola, nome_escola, cidade_escola, matricula_professor_diretor)`

`matricula_professor_diretor referencia tbProfessor`

A tabela tbEscola recebeu a chave estrangeira porque a entidade “Escola” tem participação total no relacionamento.

Figura 4.9 – Conversão de relacionamento 1:1 para o modelo relacional

Note que a escolha pela entidade que tem participação total é feita para evitarmos valores nulos na tabela. Uma vez que toda escola tem um diretor, não teremos valor nulo para a chave estrangeira. Diferentemente, se escolhêssemos a entidade com participação parcial para receber a chave estrangeira, teríamos muitos valores nulos, uma vez que nem todo professor é um diretor.

No caso das duas entidades terem participação total, fica a critério do desenvolvedor escolher quem receberá a chave estrangeira.

Se as duas entidades tiverem participação parcial, também é o desenvolvedor quem decide para onde vai a chave estrangeira, devendo ele fazer uma análise de qual tabela que receberia menos valores nulos e adicionando a chave estrangeira nessa tabela.

## Importante!

Se o relacionamento tiver atributos descritivos, os atributos “seguem” a chave estrangeira, ou seja, os atributos descritivos ficarão na mesma tabela que a chave estrangeira.

## Relacionamento Recursivo

Todo relacionamento recursivo gera uma chave estrangeira que faz referência à chave primária da própria tabela, como mostra a figura 4.10.

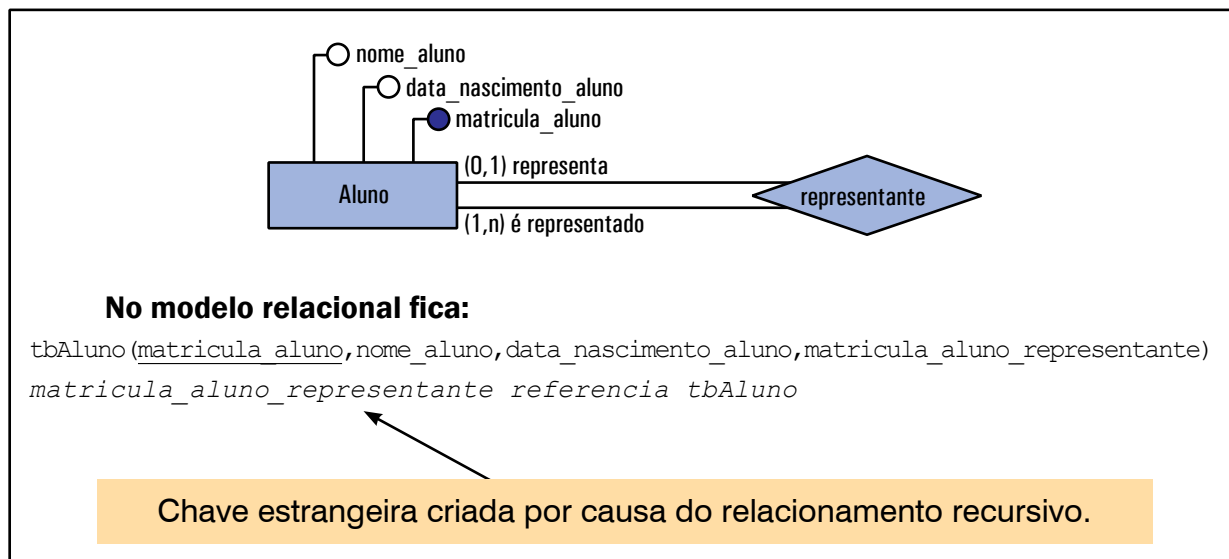


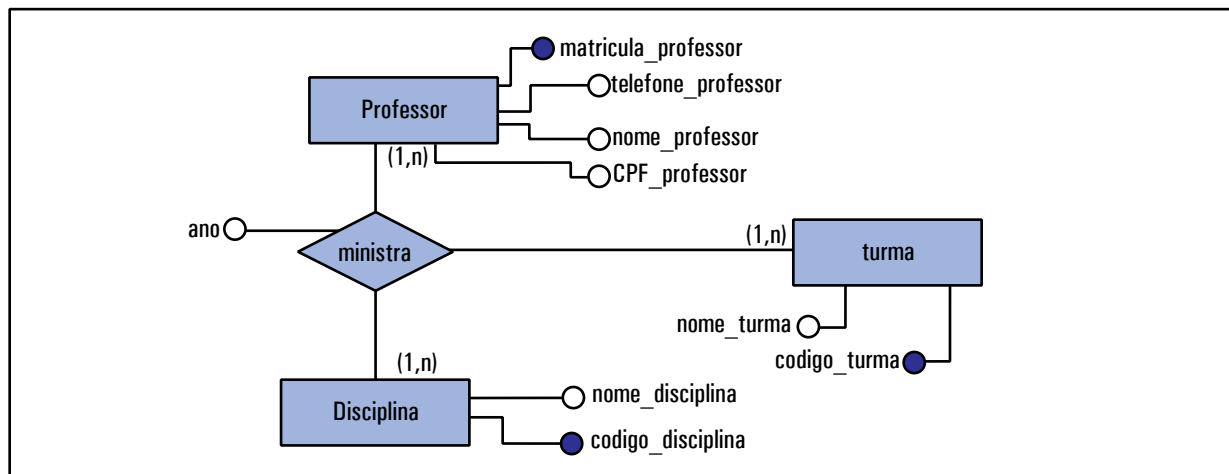
Figura 4.10 – Conversão de relacionamento recursivo para o modelo relacional

Observe que na descrição do modelo relacional da figura 4.10 foi adicionado o atributo **matricula\_aluno\_representante** para especificar o relacionamento recursivo. Esse atributo é uma chave estrangeira que faz referência ao atributo **matricula\_aluno** da própria tabela **tbAluno**.

## Relacionamentos Ternários ou Maiores

Para cada relacionamento entre mais de duas entidades, cria-se uma tabela contendo todos os atributos descritivos do relacionamento (se houver) mais as chaves primárias de todas as entidades ligadas ao relacionamento (que passam como chaves estrangeiras).

A chave primária da nova tabela, será composta pelos atributos chaves das entidades participantes do relacionamento que tiverem cardinalidade **N** e, se houver necessidade, mais algum atributo descritivo. A figura 4.11 mostra um exemplo de conversão de relacionamento ternário para o modelo relacional.



### No modelo relacional fica:

tbProfessor(matricula\_professor, nome\_professor, telefone\_professor, CPF\_professor)

tbDisciplina(codigo\_disciplina, nome\_disciplina)

tbTurma(codigo\_turma, nome\_turma)

Criou-se uma tabela para o relacionamento ternário.

tbProfessorTurmaDisciplina(matricula\_professor, codigo\_disciplina, codigo\_turma, ano)

matricula\_professor referencia tbProfessor

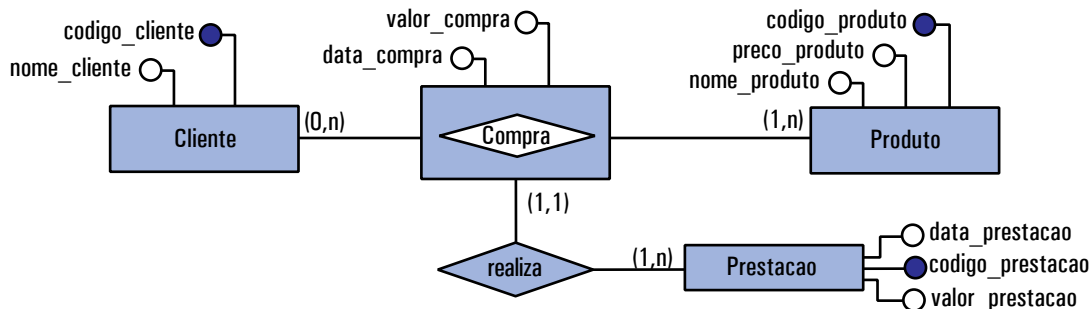
codigo\_disciplina referencia tbDisciplina

codigo\_turma referencia tbTurma

Figura 4.11 – Conversão de relacionamento ternário para o modelo relacional

## Agregação

Uma agregação no modelo de ER vira uma tabela no modelo relacional e irá conter seus próprios atributos, mais as chaves estrangeiras de acordo com os seus relacionamentos, como mostra a figura 4.12.



### No modelo relacional fica:

tbCliente(codigo\_cliente, nome\_cliente)

tbProduto(codigo\_produto, nome\_produto, preco\_produto)

A agregação é criada como uma tabela, e a chave primária dessa tabela é formada pelas estrangeiras mais um atributo da própria tabela.

tbCompra(codigo\_cliente, codigo\_produto, data\_compra, valor\_compra)

codigo\_cliente referencia tbCliente

codigo\_produto referencia tbProduto

tbPrestacao(codigo\_prestacao, data\_prestacao, valor\_prestacao, codigo\_cliente, codigo\_produto, data\_compra)

(codigo\_cliente, codigo\_produto, data\_compra) referencia tbCompra

Chave estrangeira composta.

Figura 4.12 – Conversão de agregação para o modelo relacional

Observe que a tabela tbPrestacao possui uma chave estrangeira composta. Essa chave estrangeira é composta porque a sua primária correspondente também é composta.

Quando temos uma chave primária composta, não podemos passar para outra tabela apenas parte da chave, e por esse motivo sua estrangeira correspondente também será composta.



## Atividades

- 1) O que você entende por chave estrangeira? Explique, com base naquilo que você leu no capítulo 4. Dê 3 exemplos.
- 2) O que você entende por integridade referencial da base de dados?
- 3) Como a chave estrangeira garante a integridade referencial do BD?
- 4) Uma chave estrangeira pode assumir o valor nulo ou repetir-se? Explique.
- 5) Por que o valor nulo para uma chave estrangeira fere a restrição de integridade referencial?
- 6) Suponha que você tenha o modelo relacional abaixo:

```
tbTurma(codigo_turma: inteiro, nome_turma: caracter(5))  
tbAluno(matricula_aluno: inteiro, nome_aluno: caracter(200), data_nascimento_  
aluno: data, codigo_turma: inteiro)  
codigo_turma referencia tbTurma
```

O que o SGBD deve fazer se for excluída a Turma “1T1” para que a base de dados continue mantendo a restrição de integridade referencial?

- 7) Para a descrição do modelo relacional a seguir, defina as chaves primárias, as chaves estrangeiras e o tipo de dados para cada atributo.

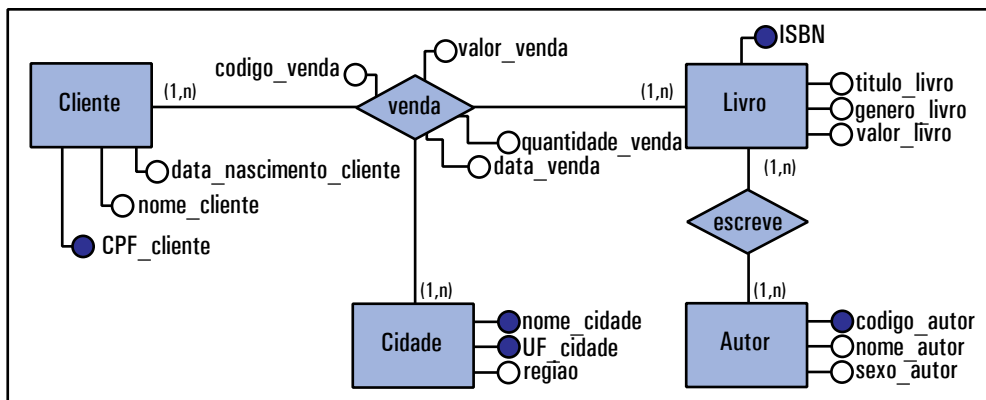


```

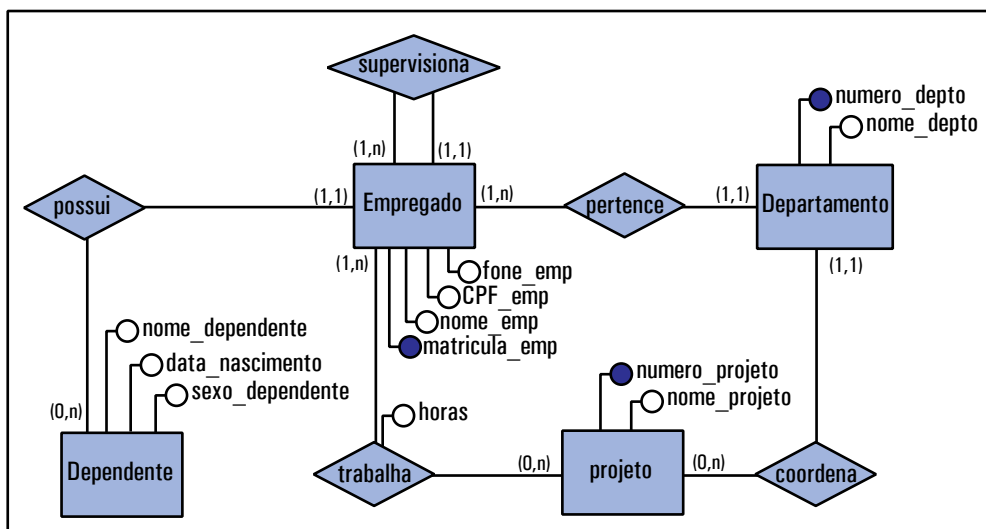
tbFuncionario (matricula, RG, nome, sexo, telefone, codigo_loja)
tbLoja (codigo, nome, telefone, codigo_cidade, matricula_funcionario_gerente)
tbCidade (codigo, nome, UF)
tbFabricante (codigo, nome, CNPJ, fone, codigo_cidade)
tbProduto (codigo, descricao, preco_unitario, codigo_fabricante)
tbVenda (codigo_venda, valor_total, data_venda, matricula_funcionario)
tbItens_venda (codigo_venda, codigo_produto, data_venda, valor_item,
quantidade)

```

- 8) Desenhe o DER para o modelo relacional do exercício 7.
- 9) Passe o Diagrama de ER abaixo para o modelo relacional. Especifique os tipos de dados para cada atributo.



- 10) Passe o Diagrama de ER abaixo para o modelo relacional. Especifique os tipos de dados para cada atributo.



# Modelo Relacional: Tópicos Avançados

## Especialização

A transformação de uma especialização do modelo de ER para o modelo relacional pode ser feita de 3 diferentes modos:

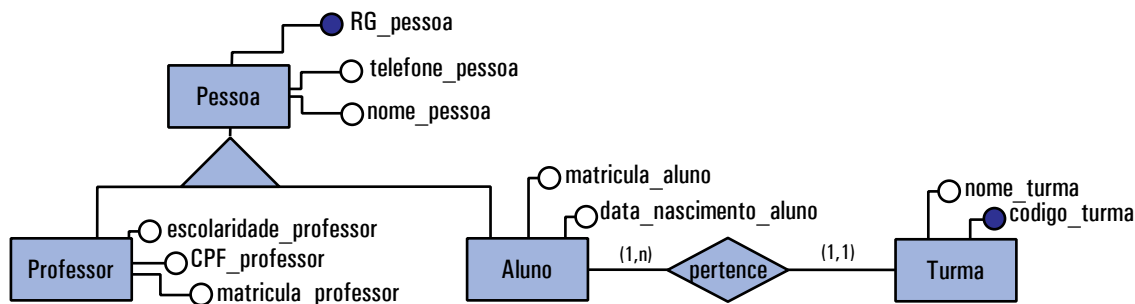
1. Criando uma tabela apenas para a entidade pai;
2. Criando tabelas apenas para as entidades filhas;
3. Criando uma tabela para cada entidade (tanto para a entidade pai, quanto para as filhas).

Na primeira situação, será criada uma tabela única com o nome da entidade pai e essa tabela irá conter: todos os atributos das entidades pai (genérica), os atributos da(s) entidade(s) filha(s) (entidades especializadas), atributos referentes a possíveis relacionamentos e um atributo chamado “tipo” que identificará qual entidade especializada está sendo representada em uma linha. A chave primária dessa tabela será a própria chave primária da entidade pai (ver figura 5.1).

Na segunda situação, serão criadas tabelas apenas para as entidades filhas. Cada entidade filha que virar uma tabela terá como atributos tantos os seus atributos específicos e de seus relacionamentos diretos, quanto os atributos da entidade pai, mais os atributos dos relacionamentos de outras entidades com a entidade pai. A chave primária de cada uma das tabelas especializadas será a chave primária da entidade pai, como mostra a figura 5.1. As tabelas criadas serão completamente independentes umas das outras.

Na terceira situação, serão criadas tabelas para todas as entidades (pai e filhas). Cada tabela terá seus atributos específicos, e os atributos dos seus relacionamentos. As tabelas referentes às entidades filhas também receberão como chave estrangeira a chave primária da entidade pai.

A chave primária para cada entidade filha será a chave estrangeira, que neste caso terá as duas funções (**PK** e **FK**). Caso exista algum atributo que identifique unicamente a entidade filha, ele poderá ser escolhido como chave primária e a chave primária da entidade pai passa apenas como chave estrangeira, como mostra a figura 5.1.



**1º modo de conversão:** Criando uma única tabela para a especialização, teremos:

```
tbTurma(codigo_turma, nome_turma)
```

```
tbPessoa(RG_pessoa, telefone_pessoa, nome_pessoa, matricula_aluno, data_nascimento_aluno, codigo_turma, escolaridade_professor, CPF_professor, matricula_professor, tipo_pessoa)
```

```
codigo_turma referencia tbTurma
```

**2º modo de conversão:** Criando tabelas apenas para as entidades filhas (especializadas), teremos:

```
tbTurma(codigo_turma, nome_turma)
```

```
tbAluno(RG_pessoa, nome_pessoa, telefone_pessoa, matricula_aluno, data_nascimento_aluno, codigo_turma)
```

```
codigo_turma referencia tbTurma
```

```
tbProfessor(RG_pessoa, nome_pessoa, telefone_pessoa, matricula_professor, CPF_professor, escolaridade_professor)
```

**3º modo de conversão:** Criando tabelas apenas para as entidades pai e filhas, teremos:

```
tbTurma(codigo_turma, nome_turma)
```

```
tbPessoa(RG_pessoa, nome_pessoa, telefone_pessoa)
```

```
tbProfessor(matricula_professor, RG_pessoa, CPF_professor, escolaridade_professor)  
RG_pessoa referencia tbPessoa
```

```
tbAluno(matricula_aluno, RG_pessoa, data_nascimento_aluno, codigo_turma)
```

```
codigo_turma referencia tbTurma
```

```
RG_pessoa referencia tbPessoa
```

Figura 5.1 – Conversão de especialização para o modelo relacional

Na descrição do modelo relacional da figura 5.1, para o 3º modo de conversão, observe que foi escolhido como atributo chave para as tabelas especializadas (tbAluno e tbProfessor) os atributos matricula\_aluno e matricula\_professor respectivamente. O atributo RG\_pessoa funcionará apenas como chave estrangeira, uma vez que temos um atributo em cada tabela especializada que identifica uma única linha da tabela. Caso não existisse tal atributo, a chave primária seria o próprio RG\_pessoa que seria chave estrangeira e primária ao mesmo tempo. A chave primária não será composta nessa situação.

A pergunta que aparece nesse momento é a seguinte: qual abordagem eu devo utilizar quando o modelo apresenta uma especialização?

A primeira abordagem irá conter muitos valores nulos, uma vez que dado o tipo do objeto somente os atributos referentes àquele objeto serão preenchidos. Por isso, nem todos os atributos serão obrigatórios. Por outro lado, essa primeira abordagem tem a vantagem de dispensar a necessidade de junção entre tabelas, uma vez que os dados estão todos na mesma tabela.

A segunda abordagem é pouco recomendada, porque pode gerar redundância de dados, uma vez que os dados da entidade genérica são repetidos em todas tabelas especializadas. Assim, se uma pessoa for tanto professor como aluno, teremos as informações referentes a essa pessoa repetida nas duas tabelas. Portanto, essa abordagem só deve ser utilizada quando tivermos uma especialização exclusiva, ou seja, uma pessoa ou é do tipo aluno ou do tipo professor.

A terceira abordagem tem a vantagem de evitar os valores nulos que aparecem na primeira abordagem e ainda a de não permitir a duplicidade como na segunda abordagem.

Assim, o desenvolvedor da base de dados deve analisar todos os aspectos referentes à situação que se está modelando e optar pela solução que seja mais adequada ao problema.

## Diagrama do Modelo Relacional

O modelo relacional pode ser descrito, como fizemos nos exemplos anteriores, ou pode ser diagramado (forma mais comum).

No Diagrama do Modelo Relacional, tudo que virou tabela, aplicando-se as regrinhas de conversão entre modelos, será representado por um retângulo. Esse retângulo irá conter o nome da tabela, seus atributos, os tipos dos atributos, a identificação da chave primária, a identificação da chave estrangeira e a cardinalidade do modelo.

A cardinalidade é atribuída considerando-se o Modelo de ER. Cardinalidade do tipo 1:1 e 1:N são representadas da mesma forma que no modelo de ER. Já a cardinalidade N:N não aparece no diagrama do modelo relacional, uma vez que o relacionamento N:N virou uma tabela. Portanto, todo relacionamento N:N dará origem a dois relacionamentos do tipo 1:N.

O exemplo da figura 5.2 apresenta o Diagrama do Modelo Relacional para o Diagrama de ER da figura 3.7. A ferramenta **brModelo** chama esse diagrama de Modelo Lógico (ou Esquema Lógico).

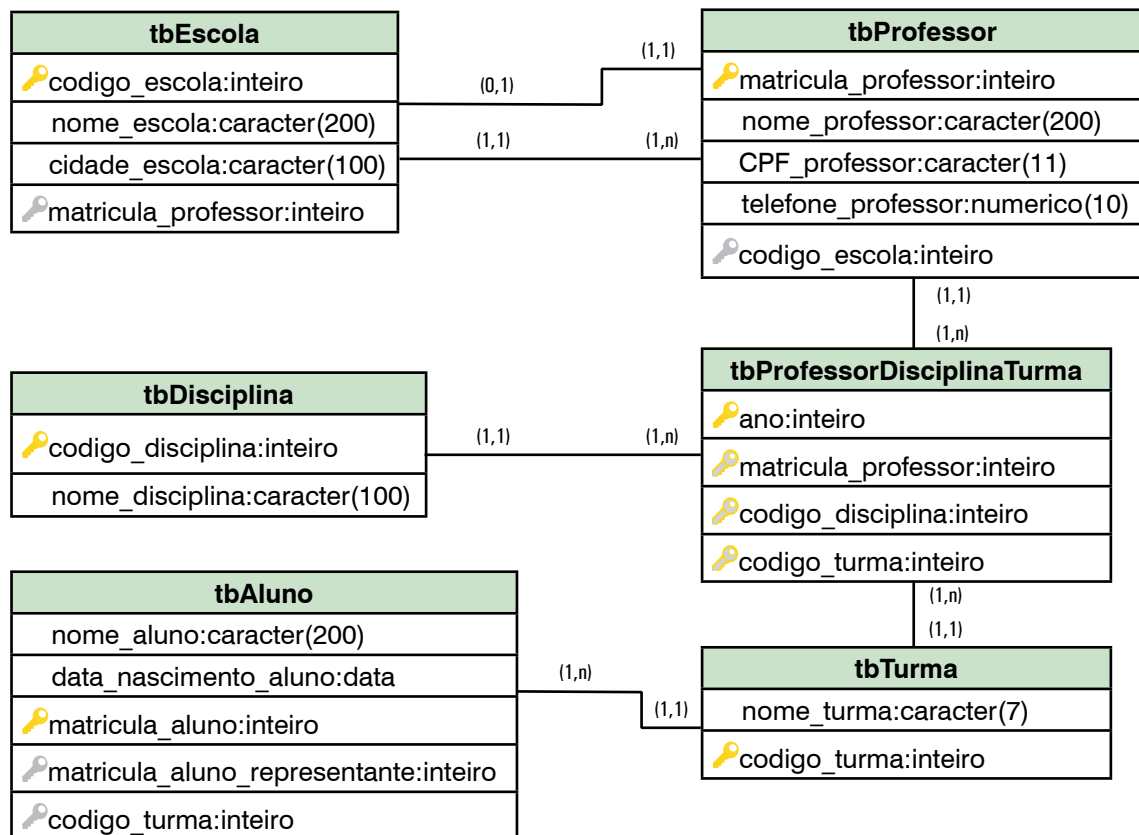


Figura 5.2 – Diagrama do modelo relacional para o diagrama de ER da figura 3.7

A chave primária nesse diagrama é indicada pela chave dourada, mas também poderia ser indicada grifando-se os atributos que formam a chave primária. A chave estrangeira é indicada pela chave de cor prata, mas também poderia ser indicada pelo asterisco. Aqueles atributos que são chaves estrangeiras e também ajudam a formar a chave primária, como mostrado na tabela tbProfessorDisciplinaTurma, são indicados pela chave dourada com o interior prateado.

## Dicionário de Dados da Base de Dados

Após o modelo relacional ter sido descrito ou diagramado, é necessário criar o Dicionário de Dados para a base de dados.

O Dicionário de Dados da Base de Dados tem por objetivo descrever as propriedades de uma tabela, sua estrutura física e as restrições que cada atributo possui. Assim, o desenvolvedor que irá implementar o banco de dados saberá exatamente como a base deve ser criada.

No Dicionário de Dados da Base de Dados, cada tabela do modelo relacional deverá ser descrita e deverá conter os seguintes campos: Nome do Atributo, Descrição do Atributo, Tamanho, Tipo e Restrições (Valor Nulo, Regra de Domínio, Chaves, Valor *Default* e *Unique*).

A figura 5.3 apresenta um exemplo do Dicionário de Dados para a tbAluno. Foram acrescentados alguns atributos nessa tabela para que seja possível exemplificar cada uma das restrições citadas anteriormente.

Nome	Descrição	Tipo	Tamanho	Nulo	Regra (check)	Chave	Default	Unique
matricula_aluno	Armazena a matrícula do aluno	Numérico	5	Não	—	PK	—	Não
RG_aluno	Armazena o RG do aluno	Caracter	11	Não	—	—	—	Sim
nome_aluno	Armazena o nome do aluno	Caracter	100	Não	—	—	—	Não
data_nascimento_aluno	Armazena a data de nascimento do aluno	Data	—	Não	—	—	—	Não
cidade_aluno	Armazena a cidade em que o aluno mora	Caracter	20	Sim	—	—	Curitiba	Não
matricula_aluno_representante	Armazena a matrícula do aluno representante	Numérico	5	Sim	—	—	—	Não
codigo_turma	Armazena o código da turma do aluno	Inteiro	—	Não	—	FK que referencia tbTurma	—	Não
sexo_aluno	Armazena o sexo a que o aluno pertence	Caracter	1	Não	M – Masculino F – Feminino	—	—	Não

Figura 5.3 – Dicionário de dados para a tabela tbAluno

Para alguns tipos de dados, não é possível definirmos o tamanho, como por exemplo o tipo Data, porque esses tipos já têm tamanho pré-definido pelo SGBD. As restrições aplicadas a um atributo definem as propriedades desse atributo.

A restrição de **Nulo** define se um atributo permite ou não o valor nulo, ou seja, define se o atributo será obrigatório ou não.

Uma restrição de **Domínio** ou **Regra de Domínio** define quais valores serão permitidos cadastrar para um atributo. No exemplo da figura 5.3, temos uma regra de domínio que diz que os valores permitidos para sexo são apenas “M” ou “F”.

As restrições de **chave** permitem identificar a chave primária (**PK**) e as chaves estrangeiras (**FK**). É interessante que na definição da chave estrangeira também seja identificado a que tabela ela referencia.

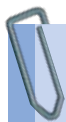
A restrição de **default** permite que seja inserido um valor padrão caso o usuário não digite nada para o campo. No nosso exemplo, definiu-se que se o usuário não digitar nada para o campo cidade\_aluno, o próprio SGBD armazena o valor “Curitiba” para esse campo.

A última restrição é a de **unicidade**. Essa restrição é aplicada apenas para atributos que não são chave primária e que não podem se repetir. No exemplo da figura 5.3, o atributo RG\_aluno não é chave primária e não pode se repetir. Sendo assim, pode-se definir o atributo como *unique* (único). É redundante dizer que uma chave primária é *unique*, já que ela não se repete.

## Normalização

O processo de normalização geralmente é aplicado quando temos uma base de dados que foi criada antes da existência de um banco de dados relacional ou foi desenvolvida sem considerar a existência de um banco de dados relacional.

Na maioria dos casos, esses sistemas são antigos e seus arquivos de dados possuem muitas informações redundantes e inconsistentes. Além disso, normalmente não existe nenhum documento que especifique o modelo de dados, o que torna muito difícil a manutenção desses sistemas ou as migrações para um banco de dados relacional.



Vale ressaltar que o modelo conceitual da base de dados agiliza bastante o processo de manutenção, porque permite que pessoas que não participaram do desenvolvimento do projeto possam entendê-lo mais rapidamente.

Às vezes, também encontramos sistemas não muito antigos e que, no entanto, foram implementados usando uma única tabela. Nesses casos, as informações estarão repetidas, haverá muitos valores nulos na base e será muito difícil executar consultas que retornem valores consistentes com o esperado.

Para resolver ou minimizar os problemas apresentados, pode-se fazer o que chamamos de engenharia reversa do projeto, utilizando para isso o processo de normalização. A engenharia reversa do projeto consiste em, a partir dos dados armazenados, obter um modelo conceitual da base de dados e eliminar as redundâncias.

Também podemos utilizar o processo de normalização para conferir se o nosso modelo de dados está normalizado e, caso não esteja, pode-se normalizá-lo antes da implementação da base de dados no SGBD.

O processo de normalização consiste em um conjunto de regras, denominadas formas normais. A literatura apresenta 6 formas normais: 1FN, 2FN, 3FN, 4FN, 5FN e a de Boyce/Codd. Alguns autores tratam a forma normal de Boyce/Codd como sendo um caso específico da 3FN.

No entanto, para a maioria das bases de dados, a aplicação até a 3FN é suficiente. Sendo assim, só iremos abordar nesse livro as três primeiras formas normais.

A tabela tbPedido da figura 5.4 será utilizada para demonstrar o processo de normalização.

Atributo multivalorado

tbPedido(codigo\_pedido, valor\_total, data\_pedido, (telefone\_contato),  
(codigo\_produto, nome\_produto, preco\_unitário, quantidade, valor\_pago\_por\_produto))

Grupo de valores repetidos

O modelo relacional acima poderia ser escrito como mostra a tabela:

Código pedido	Valor Pedido	Data pedido	Telefone de contato	Produto				
				Codigo produto	Nome produto	Valor unitário do produto	Quantidade	Valor pago por produto
100	3300,00	30/11/2009	2222-2222 9999-9999	1	Computador	1500,00	1	1500,00
				5	Impressora	600,00	2	1200,00
				6	Papel A4	12,00	50	600,00
101	3800,00	15/12/2009	2121-2121 9191-9191	2	Mouse	30,00	10	300,00
				5	Impressora	600,00	5	3000,00
				7	Teclado	50,00	10	500,00

Figura 5.4 – Exemplo de uma tabela não normalizada

## Primeira Forma Normal (1FN)

Toda tabela está na 1FN se os seus atributos forem atômicos. Isso significa que não serão permitidos atributos compostos, multivalorados ou grupos repetidos de dados (também conhecidos como tabelas aninhadas).

Os grupos repetidos de dados ocorrem quando uma tabela aparece dentro de outra tabela. Por exemplo, na tabela tbPedido da figura 5.4, temos uma única tabela que armazena os dados de pedidos e de produtos. Assim, os dados referentes ao produto pertencem a uma tabela que está dentro da tabela pedido.

É possível perceber que a tabela tbPedido não está na 1FN porque além de grupos repetidos ela também tem um atributo multivalorado.

Para deixar uma tabela na 1FN, é necessário fazer o seguinte:

- Os atributos compostos devem ser decompostos e armazenados como atributos simples.
- Para cada atributo multivalorado será criada uma tabela que irá conter o atributo multivalorado mais a chave primária da tabela inicial, que passa como chave estrangeira. A chave primária da nova tabela será composta.
- Para cada grupo repetido será criada uma tabela que irá conter os atributos do grupo repetido mais a chave primária da tabela inicial, que passa como chave estrangeira e irá ajudar a compor a chave primária.



A figura 5.5 mostra como ficaria o modelo da figura 5.4 após aplicarmos a 1FN.

```
tbPedido(codigo_pedido, valor_total, data_pedido)

tbTelefoneContatoPedido(codigo_pedido, telefone_contato)
codigo_pedido referencia tbPedido

tbPedidoProduto(codigo_pedido, codigo_produto, nome_produto, preco_unitario,
quantidade, valor_pago_por_produto)
codigo_pedido referencia tbPedido
```

Figura 5.5 – Aplicação da 1FN para eliminar grupo repetido e atributo multivalorado

## Segunda Forma Normal (2FN)

A 2FN só é aplicável para tabelas que possuem uma chave primária composta e que, além disso, tenham outros atributos que não façam parte da chave primária.

Uma tabela está na 2FN se estiver na 1FN e todo atributo que não compõe a chave primária deve ter dependência funcional total em relação à chave primária.

No modelo relacional, a **dependência funcional** entre dois atributos, A e B, ocorre quando, em todas as linhas da tabela, para cada valor de A irá aparecer sempre o mesmo valor de B. Por exemplo, na tabela da figura 5.4 sempre que aparecer o código de produto “5” teremos como nome do produto “Impressora”. Assim, o nome do produto depende funcionalmente do código do produto.

Para denotar a dependência funcional, usa-se uma expressão na forma “codigo\_produto → nome\_produto”. Isso significa que o nome do produto depende funcionalmente do código do produto.

Para que a dependência funcional seja total, o atributo não chave deve depender de toda a chave primária composta. Por exemplo, no caso da tabela tbPedidoProduto o atributo “quantidade” representa a quantidade de um produto que foi solicitada em um pedido. Assim, o atributo “quantidade” depende tanto do codigo\_produto quanto do codigo\_pedido, caracterizando uma **dependência funcional total**. O atributo nome\_produto, por outro lado, tem uma **dependência funcional parcial**, porque depende apenas de parte da chave primária, ou seja, depende apenas do codigo\_produto.

Para deixarmos o modelo na 2FN devemos:

- Repetir as tabelas que tenham chave primária simples, pois já estão na 2FN.
- Repetir as tabelas que tenham chave primária composta, mas que não tenham nenhum outro atributo além dos que compõem a chave primária.

- Para cada tabela que tiver chave primária composta e pelo menos um atributo que não faz parte da chave, deve-se verificar se cada um dos atributos não chave têm **dependência funcional total**. Caso a dependência não seja total, deve-se criar uma tabela com o atributo que depende parcialmente, mais o atributo do qual ele depende (que será chave primária na nova tabela e será chave estrangeira na tabela inicial).

A tabela 5.6 mostra como ficaria o modelo da figura 5.4 após aplicarmos a 2FN.

```
tbPedido(codigo_pedido, valor_total, data_pedido)

tbTelefoneContatoPedido(codigo_pedido, telefone_contato)
codigo_pedido referencia

tbPedidoProduto(codigo_pedido, codigo_produto, quantidade,
valor_pago_por_produto)
codigo_pedido referencia tbPedido
codigo_produto referencia tbProduto

tbProduto(codigo_produto, nome_produto, preco_unitario)
```

Figura 5.6 – Aplicação da 2FN para eliminar redundâncias

## Terceira Forma Normal (3FN)

Uma tabela está na 3FN se estiver na 2FN e ela não possuir dependências transitivas.

Dependência Transitiva ocorre quando existe um atributo que não é chave e nem faz parte da chave, mas que identifica outros atributos. Ou seja, existe um atributo não chave que depende de outro atributo não chave.

Como no modelo da figura 5.6 não existem dependências transitivas, ele já está na 3FN. No entanto, para entendermos melhor dependências transitivas, vamos supor a situação da figura 5.7.

```
tbDepartamento(codigo_depto, nome_depto, codigo_gerente, nome_gerente)
```

Figura 5.7 – Tabela com dependência transitiva

A tabela da figura 5.7 está na 1FN porque todos os seus atributos são atômicos e está na 2FN porque não tem chave primária composta. No entanto, temos uma dependência transitiva que ocorre com o atributo nome\_gerente que depende do atributo codigo\_gerente que não é chave e nem faz parte da chave primária.

Para eliminar dependências transitivas, deve-se criar uma nova tabela que irá conter o atributo que depende (ex.: nome\_gerente) mais o atributo do qual ele é dependente (ex.: codigo\_gerente). A figura 5.8 mostra como ficaria o modelo.

```
tbDepartamento(codigo_depto, nome_depto, codigo_gerente)
codigo_gerente referencia tbGerente

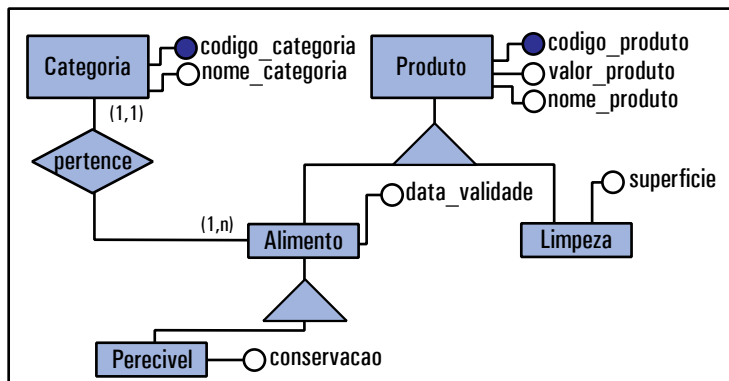
tbGerente (codigo_gerente, nome_gerente)
```

Figura 5.8 – Aplicação da 3FN para eliminar dependência transitiva



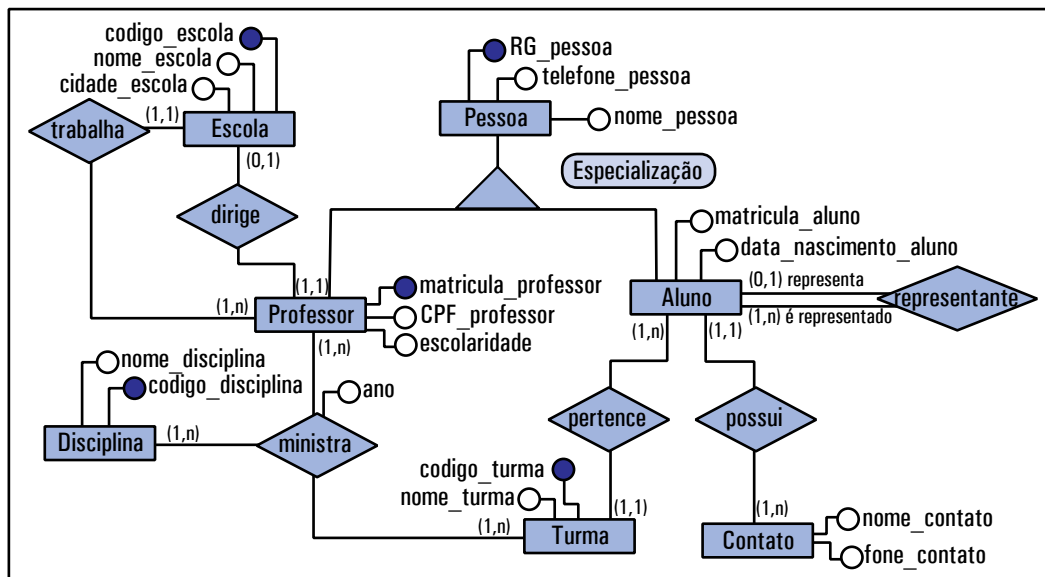
## Atividades

- 1) Explique os três modos de converter uma especialização para o modelo relacional.
- 2) Dê um exemplo de quando você aplicaria o primeiro modo de conversão da especialização.
- 3) Dê um exemplo de quando você aplicaria o segundo modo de conversão da especialização.
- 4) Dê um exemplo de quando você aplicaria o terceiro modo de conversão da especialização.
- 5) Passe o Diagrama de ER abaixo para o modelo relacional. Especifique os tipos de dados para cada atributo.



- 6) Desenhe o Diagrama do Modelo Relacional para o exercício 5.
- 7) Explique o que é cada uma das restrições para os atributos (Valor Nulo, Regra de Domínio, Chaves, Valor Default e Unique) apresentadas no Dicionário de Dados. Em seguida, compare as suas explicações com as que foram dadas no texto e verifique se estão corretas.

- 8) Passe o Diagrama de ER abaixo para o Diagrama do Modelo Relacional. Especifique os tipos de dados para cada atributo.



- 9) Verifique se as tabelas abaixo estão nas três formas normais. Se não estiverem, normalize-as e especifique qual forma normal está sendo aplicada.

a. `tbAcidente (numero_placa_carro, CPF_motorista, nome_motorista, total_danos_acidente, data_acidente)`

b. `tbPaciente(codigo_paciente, nome_paciente, (fone_paciente), (CRM_medico, nome_medico, data_consulta), codigo_convenio, nome_convenio, (codigo_exame, nome_exame, diagnostico_principal))`

- 10) Normalize as seguintes tabelas de dados, especificando qual forma normal está sendo aplicada em cada passo da normalização. Em seguida, desenhe o diagrama ER correspondente ao esquema relacional obtido.

a. `tbAluno (cod_aluno, nome_aluno, sexo_aluno, data_nascimento_aluno, codigo_curso, nome_curso, nome_diretor, (codigo_disciplina, nome_disciplina, nota_disciplina))`

b. `tbNotaFiscal (Numero_nota, data_emissao_nota, codigo_cliente, nome_cliente, endereço_cliente: rua; numero; e complemento, CPF_cliente, (codigo_produto, nome_produto, quantidade_vendida_produto, valor_unitario_produto, valor_total_por_item_vendido, valor_total_da_nota))`

# Um Exemplo Prático

Neste capítulo, vamos colocar em prática todos os conceitos estudados até o capítulo 5. Será apresentada uma situação e a partir dela vamos construir o Diagrama de ER, a Descrição do Modelo Relacional, o Diagrama do Modelo Relacional, o Dicionário de Dados e aplicar as regras de normalização para validar se o modelo está normalizado.

O objetivo desse capítulo é conseguir entender e integrar todas as etapas de modelagem de uma base de dados num único projeto. Imagine que você vá desenvolver um sistema de informação que necessite de um banco de dados. Depois de algumas entrevistas com o usuário, você decide começar a modelar a base de dados. O problema apresentado pelo usuário está descrito abaixo.

## 0 Problema

Uma ONG, sem fins lucrativos, deseja informatizar o seu processo de trabalho. Existem algumas pessoas que colaboram com a ONG e é necessário que se consiga identificar quem são essas pessoas e como elas contribuem.

As pessoas envolvidas (voluntários) podem ser de dois tipos: uma, é pessoa física e doa seu tempo para desenvolver algum projeto para a entidade e outra, é pessoa jurídica, e faz as doações em dinheiro para a entidade.

Quando uma pessoa se inscreve para ajudar a ONG, é necessário armazenar o código do voluntário, nome do voluntário e telefone do voluntário. Se esta pessoa for uma pessoa física, deve-se armazenar também o CPF da pessoa. Se ela for uma pessoa jurídica, é necessário saber o CNPJ da empresa. Para todas as pessoas, deve-se cadastrar o telefone e o nome de pelo menos duas outras pessoas para contato.

Existem dois tipos de colaboração que se pode fazer à ONG: doações e participação em projetos.

A participação em projetos é feita por pessoas que doam seu tempo para participar de um projeto, que pode ter várias pessoas envolvidas. Quando um projeto termina, as pessoas envolvidas podem participar de um novo. Um projeto terá sempre um coordenador, que é uma pessoa física cadastrada. Além disso, sobre cada projeto é necessário saber: o número dele (que é único), nome, data de início e data prevista para o fim do projeto.

Um projeto pode durar até um ano e as pessoas que trabalham nele, exceto o coordenador, não precisam estar envolvidas no processo durante toda a sua existência. As pessoas podem trabalhar apenas alguns dias ou apenas algumas semanas no projeto. Por isso, é necessário cadastrar quando a pessoa começou a trabalhar, quando sua participação foi encerrada e qual atividade ele desenvolveu durante esse tempo.

Para cada pessoa física, deve-se cadastrar a sua especialidade (por exemplo: informática, enfermagem, artes, dança, etc.). Existem várias especialidades previamente cadastradas, mas deve ser possível cadastrar uma especialidade nova. As especialidades pertencem a áreas, que também são pré-cadastradas (por exemplo: área de saúde, área de tecnologia, etc.). É importante esse cadastro de especialidades e áreas para que se possa procurar na base de dados por uma pessoa que tenha habilidade para trabalhar em um projeto e convidá-la a participar dele.

As pessoas jurídicas fazem doações em dinheiro. Essas doações são encaminhadas especificamente para um projeto, e este projeto pode receber várias doações. Quando uma doação é realizada, é necessário saber o número dessa doação, o valor da doação e a data em que ela foi realizada.

## O Diagrama de Entidade e Relacionamento

Com base nas informações obtidas, é possível fazer o modelo conceitual da base de dados. Se houver ambiguidade de interpretação nas informações é necessário conversar novamente com o usuário para esclarecer a situação.

O Diagrama de ER para o problema da ONG é apresentado na figura 6.1.

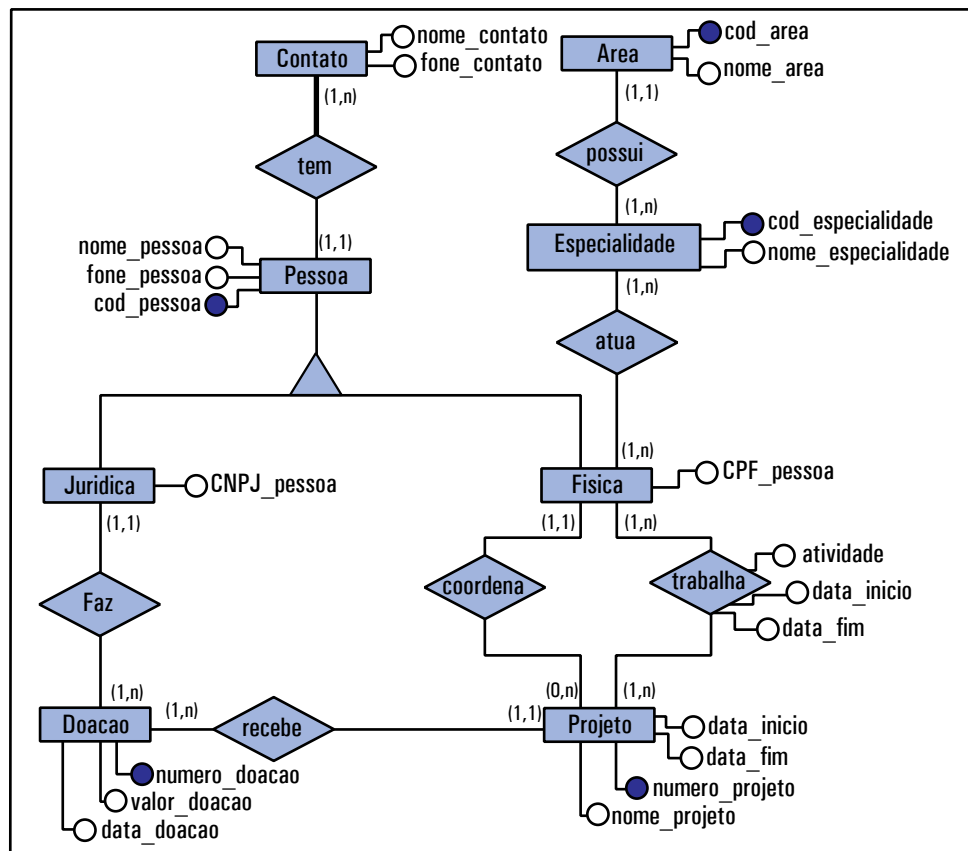


Figura 6.1 – DER para o problema da ONG

Duas observações importantes sobre o nosso modelo:

- Os atributos `nome_contato` e `fone_contato` poderiam ter sido representados como atributos multivalorados na entidade “Pessoa”. Mas como sabemos que atributos multivalorados vão virar tabela no modelo relacional, podemos representá-los no modelo de ER como uma entidade fraca.
- Existem dois relacionamentos entre “Projeto” e “Pessoa Física”. Um deles (“coordena”) para relacionar a pessoa que é coordenadora do projeto e outro (“trabalha”) para representar as pessoas que estão envolvidas com o projeto.

O próximo passo depois de criado o Diagrama de ER é fazer a descrição do modelo relacional da base de dados.

## A Descrição do Modelo Relacional

A descrição do modelo relacional consiste basicamente em definir o que vira e o que não vira tabela, usando as regrinhas apresentadas nos capítulos 4 e 5.

Lembre-se de que todas as entidades e todos os relacionamentos N:N tornam-se tabelas no modelo relacional.

No nosso exemplo, temos uma especialização e vamos optar por criar tabelas para a entidade pai e para as entidades filhas. Essa decisão foi tomada, considerando que tanto as entidades filhas quanto a entidade pai possuem relacionamentos específicos com outras entidades do modelo. Além disso, a abordagem escolhida evita valores nulos e redundantes.

Sendo assim, teremos 10 tabelas no nosso modelo relacional. A descrição do modelo relacional para o nosso problema é apresentada na figura 6.2. Os tipos de dados para cada atributo serão apresentados apenas no Diagrama do Modelo Relacional.

```
tbPessoa(cod_pessoa, nome_pessoa, fone_pessoa)
tbContato(cod_pessoa, nome_contato, fone_contato)
cod_pessoa referencia tbPessoa

tbArea(cod_area, nome_area)

tbEspecialidade(cod_especialidade, nome_especialidade, cod_area)
cod_area referencia tbArea

tbPFisica(CPF_pessoa, cod_pessoa)
cod_pessoa referencia tbPessoa

tbPFisicaEspecialidade(cod_especialidade, CPF_pessoa)
cod_especialidade referencia tbEspecialidade
CPF_pessoa referencia tbPFisica

tbProjeto(numero_projeto, nome_projeto, data_inicio, data_fim,
CPF_coordenador)
CPF_coordenador referencia tbPFisica

tbPFisicaProjeto(CPF_pessoa, numero_projeto, data_inicio, atividade,
data_fim)
CPF_pessoa referencia tbPFisica
numero_projeto referencia tbProjeto
```

```

tbPJuridica(CNPJ_pessoa, cod_pessoa)
cod_pessoa referencia tbPessoa

tbDoacao(numero_doacao, valor_doacao, data_doacao, numero_projeto,
CNPJ_pessoa)
numero_projeto referencia tbProjeto
CNPJ_pessoa referencia tbPJuridica

```

Figura 6.2 – Descrição do modelo relacional para o problema da ONG

As chaves estrangeiras são decorrentes do tipo de relacionamento entre as entidades.

## O Diagrama do Modelo Relacional

Uma vez que a descrição do modelo relacional foi feita, fica fácil desenhar o diagrama do modelo relacional. Como não especificamos os tipos de dados dos atributos na descrição do modelo relacional, temos que fazê-lo no Diagrama do Modelo Relacional.

A figura 6.3 apresenta o diagrama para o problema da ONG.

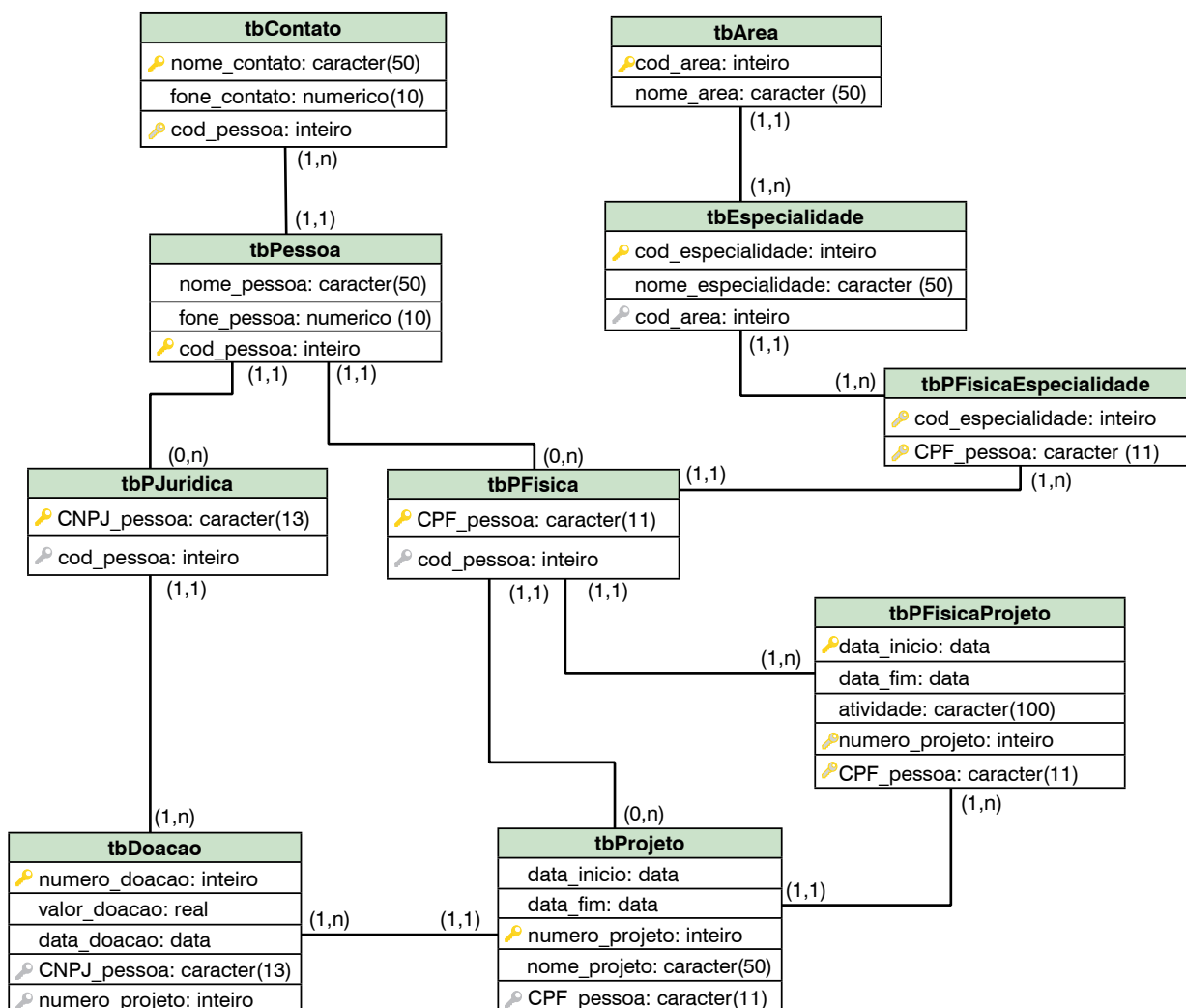


Figura 6.3 – Diagrama do Modelo Relacional para o problema da ONG



# Verificação se o Modelo Está Normalizado

Após desenharmos o diagrama lógico (relacional), é interessante conferir se as tabelas estão atendendo às três Formas Normais apresentadas no capítulo 5.

Analisando o nosso modelo, percebe-se que todas as tabelas estão na 1FN, uma vez que não possuem atributos multivalorados, não possuem atributos compostos e não possuem grupos repetidos de dados.

A 2FN é aplicada apenas às tabelas que possuem chave primária composta. Portanto, vamos verificar as tabelas: tbContato, tbPFisicaEspecialidade e tbPFisicaProjeto. Temos que analisar, para cada uma das tabelas, se existe algum atributo que não compõe a chave primária e que possui dependência funcional parcial em relação à chave primária. A tabela tbPFisicaEspecialidade já está na 2FN porque não possui atributos que não façam parte da chave primária.

Então, vamos analisar as outras duas tabelas que possuem chave primária composta.

Para descobrir se o atributo fone\_contato da tabela tbContato é totalmente ou parcialmente dependente, é necessário se perguntar: Se eu quiser saber o telefone de um contato específico, eu consigo descobrir isso só pelo nome do contato ou só pelo código da pessoa? Como o nome de um contato pode se repetir, se eu tentar descobrir o telefone do contato usando apenas o nome, poderá vir como resultado mais de um telefone. A mesma coisa irá acontecer se eu tentar descobrir o telefone usando apenas o código da pessoa, uma vez que a pessoa pode ter mais de um contato. Sendo assim, fone\_contato é dependente de toda a chave primária e a tabela tbContato já está na 2FN.

Usando o mesmo raciocínio, percebemos que a tabela tbPFisicaProjeto também está na 2FN porque os atributos data\_fim e atividade são dependentes de toda a chave primária.

Agora, temos que analisar se as tabelas estão na 3FN. Para isso, verificamos se alguma tabela possui dependência transitiva, ou seja, se existem atributos não chave e que dependem de outro atributo não chave. Nenhuma das 10 tabelas do modelo relacional possui dependência transitiva e, portanto, todas estão na 3FN.

É importante ressaltar que se, nessa etapa, o modelo estiver desnormalizado é necessário normalizá-lo. Nesse caso, teremos que alterar o Diagrama do Modelo Relacional, a Descrição do Modelo Relacional e o Diagrama de ER para que fiquem coerentes com a implementação.

## O Dicionário de Dados

Após conferir se o modelo está normalizado, a próxima etapa, antes de entregar para o desenvolvedor os modelos da base de dados, é escrever o Dicionário de Dados para a base, como será mostrado a seguir. Lembre-se de que o Dicionário de Dados deve ser feito para todas as tabelas do modelo relacional.

Nome	Descrição	Tipo	Tamanho	Nulo	Regra (check)	Chave	Default	Unique
cod_pessoa	Armazena o código do voluntário	Inteiro	—	Não	—	PK	—	Não
nome_pessoa	Armazena o nome do voluntário	Caracter	50	Não	—	—	—	Não
fone_pessoa	Armazena o telefone do voluntário	Caracter	10	Sim	—	—	—	Não

Tabela tbPessoa

Nome	Descrição	Tipo	Tamanho	Nulo	Regra (check)	Chave	Default	Unique
cod_pessoa	Armazena o código do voluntário	Inteiro	—	Não	—	PK, FK que referencia tbPessoa	—	Não
nome_contato	Armazena o nome da pessoa de contato	Caracter	50	Não	—	PK	—	Não
fone_contato	Armazena o telefone do contato	Caracter	10	Não	—	—	—	Não

Tabela tbContato

Nome	Descrição	Tipo	Tamanho	Nulo	Regra (check)	Chave	Default	Unique
cod_area	Armazena o código da área	Inteiro	—	Não	—	PK	—	Não
nome_area	Armazena o nome da área	Caracter	50	Não	—	—	—	Não

Tabela tbArea

Nome	Descrição	Tipo	Tamanho	Nulo	Regra (check)	Chave	Default	Unique
cod_especialidade	Armazena o código de uma especialidade	Inteiro	—	Não	—	PK	—	Não
nome_especialidade	Armazena o nome da especialidade	Caracter	50	Não	—	—	—	Não
cod_area	Armazena o código da área de uma especialidade	Inteiro	—	Não	—	FK que referencia tbArea	—	Não

Tabela tbEspecialidade

Nome	Descrição	Tipo	Tamanho	Nulo	Regra (check)	Chave	Default	Unique
CPF_pessoa	Armazena o CPF de uma pessoa física	Caracter	11	Não	—	PK	—	Não
cod_pessoa	Armazena o código do voluntário	Inteiro	—	Não	—	FK que referencia tbPessoa	—	Não

Tabela tbPFisica

Nome	Descrição	Tipo	Tamanho	Nulo	Regra (check)	Chave	Default	Unique
cod_especialidade	Armazena o código de uma especialidade	Inteiro	—	Não	—	PK, FK que referencia tbEspecialidade	—	Não
CPF_pessoa	Armazena o CPF de uma pessoa física	Caracter	11	Não	—	PK, FK que referencia tbPFisica	—	Não

Tabela tbPFisicaEspecialidade

Nome	Descrição	Tipo	Tamanho	Nulo	Regra (check)	Chave	Default	Unique
numero_projeto	Armazena o número do projeto	Inteiro	—	Não	—	PK	—	Não
nome_projeto	Armazena o nome do projeto	Caracter	50	Não	—	—	—	Não
data_inicio	Armazena a data em que o projeto iniciou	Data	—	Não	—	—	—	Não
data_fim	Armazena a data em que o projeto terminou	Data	—	Não	—	—	—	Não
CPF_pessoa	Armazena o CPF do voluntário	Caracter	11	Não	—	FK que referencia tbPFisica	—	Não

Tabela tbProjeto

Nome	Descrição	Tipo	Tamanho	Nulo	Regra (check)	Chave	Default	Unique
numero_projeto	Armazena o número do projeto	Inteiro	—	Não	—	PK, FK que referencia tbProjeto	—	Não
CPF_pessoa	Armazena o CPF do voluntário	Caracter	11	Não	—	PK, FK que referencia tbPFisica	—	Não
data_inicio	Armazena a data em que a pessoa começou a trabalhar no projeto	Data	—	Não	—	PK	—	Não
data_fim	Armazena a data em que a pessoa deixou de trabalhar no projeto	Data	—	Não	—	—	—	Não
atividade	Armazena a atividade que uma pessoa desenvolveu no projeto	Caracter	100	Não	—	—	—	Não

Tabela tbPFisicaProjeto

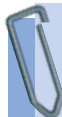
Nome	Descrição	Tipo	Tamanho	Nulo	Regra (check)	Chave	Default	Unique
CNPJ_pessoa	Armazena o CNPJ de uma pessoa jurídica	Caracter	13	Não	—	PK	—	Não
cod_pessoa	Armazena o código do voluntário	Inteiro	—	Não	—	FK que referencia tbPessoa	—	Não

Tabela tbPJuridica

Nome	Descrição	Tipo	Tamanho	Nulo	Regra (check)	Chave	Default	Unique
numero_doacao	Armazena o número da doação	Inteiro	—	Não	—	PK	—	Não
valor_doacao	Armazena o valor da doação	Real	—	Não	—	—	—	Não
data_doacao	Armazena a data em que a doação foi feita	Data	—	Não	—	—	—	Não
CNPJ_pessoa	Armazena o CNPJ da empresa que fez a doação	Caracter	13	Não	—	FK que referencia tbPJuridica	—	Não
numero_projeto	Armazena o número do projeto que recebeu a doação	Inteiro	—	Não	—	FK que referencia tbProjeto	—	Não

Tabela tbDoacao

Após terminar o projeto da base de dados, é necessário implementar essa base em um SGBD. Só depois de criada a base de dados no SGBD é que os dados poderão ser cadastrados e recuperados pela aplicação.



Como dito anteriormente, o melhor é evitar chaves do tipo caracter. Veja pelo exemplo apresentado que as chaves CPF\_pessoa e CNPJ\_pessoa seriam propagadas para as tabelas dependentes. Na prática, cria-se uma chave primária numérica e um índice *unique* nos atributos CPF\_pessoa e CNPJ\_pessoa.



## Atividades

- 1) Para o problema abaixo, desenhe o diagrama de ER, coloque as cardinalidades no diagrama e identifique as chaves primárias. Você pode acrescentar atributos ao modelo, se o texto que descreve a situação-problema não apresentar todos os atributos necessários.

Suponha que você deseja criar um sistema de banco de dados para uma farmácia na cidade de Curitiba. Esta farmácia deve possuir um cadastro de clientes, pois os clientes normalmente compram parcelado e, portanto, é necessário que existam informações como: nome, endereço, telefone, etc. Os clientes que pagam à vista não são cadastrados, e recebem um desconto (automaticamente) de 15% sobre o valor total da compra.

A farmácia deve ter um controle de estoque para que possa solicitar os produtos que estão faltando junto aos fornecedores. Os produtos podem ser: medicamentos controlados e medicamentos comuns. Sobre os medicamentos, é necessário saber o código do medicamento, o valor unitário do medicamento, o nome do medicamento e a data de validade.

Um cadastro de fornecedores é importante para que se possa saber qual fornecedor possui qual produto. Sobre os fornecedores, deve-se saber: o nome do fornecedor, telefone, endereço e CNPJ.

Os funcionários também devem ser cadastrados pelo sistema. Informações importantes sobre os funcionários são: nome, matrícula, RG, CPF e telefone. Cada funcionário que trabalha na farmácia recebe no final do mês uma comissão de 15% sobre o total de vendas que fez.

Sobre uma compra, é necessário saber: o que foi comprado (pode-se comprar vários itens numa mesma compra), data da compra, o total que foi pago e o funcionário que fez a venda. Toda venda terá um pagamento associado a ela. No caso de pagamentos à vista, deve-se saber: o valor pago e o total de desconto. Para pagamentos parcelados, é necessário saber: quem foi o cliente que fez a compra, o número de parcelas, o valor de cada parcela e a data de vencimento de cada parcela.

- 2) Para o Diagrama de ER desenhado no exercício 1, faça a Descrição do Modelo Relacional, o Diagrama do Modelo Relacional e aplique as 3 Formas Normais para verificar se o Modelo Relacional está normalizado. Caso não esteja normalizado, você deve normalizá-lo.
- 3) Para o problema abaixo, desenhe o diagrama de ER, coloque as cardinalidades no diagrama e identifique as chaves primárias. Você pode acrescentar atributos ao modelo se o texto que descreve a situação-problema não apresentar todos os atributos necessários.

Um hospital possui um corpo médico para atender os pacientes. Um paciente pode ser atendido por vários médicos e um médico pode atender vários pacientes.

Os pacientes possuem um prontuário único no hospital. Nesse prontuário, cada médico que atende o paciente registra os sintomas dele naquela consulta, o diagnóstico do médico, os exames que o paciente deve fazer e os medicamentos que o doente deve tomar.

Se o paciente for do SUS (Sistema Único de Saúde), o médico pode verificar se o medicamento está disponível na farmácia do hospital e conseguir amostras grátis para o paciente. Quando o paciente vai até a farmácia do hospital buscar o medicamento, é necessário registrar o medicamento, quem o recebeu e qual médico prescreveu o remédio.

Os medicamentos gratuitos são fornecidos por empresas especializadas (fornecedores). Sobre os fornecedores, é necessário saber: nome, CNPJ, código, telefone.

Pacientes do SUS devem fornecer a carteirinha do SUS, e outros documentos necessários. Os pacientes que não são atendidos pelo SUS são considerados particulares, e devem pagar pela consulta.

- 4) Para o Diagrama de ER desenhado no exercício 3, faça a Descrição do Modelo Relacional, o Diagrama do Modelo Relacional e aplique as 3 Formas Normais para verificar se o Modelo Relacional está normalizado. Caso não esteja normalizado, você deve normalizá-lo.
- 5) Para o problema a seguir, desenhe o diagrama de ER, coloque as cardinalidades no diagrama e identifique chaves primárias. Você pode acrescentar atributos ao modelo se o texto que descreve a situação-problema não apresentar todos os atributos necessários.

Um grupo de alunos resolveu desenvolver um sistema que será comercializado em fábricas de automóveis. Cada fábrica de automóveis possui suas concessionárias autorizadas, e para manter esse controle a fábrica disponibiliza uma licença a cada concessionária.

Uma fábrica pode ter várias concessionárias associadas, mas uma concessionária só pertence a uma fábrica.

Sobre as fábricas, é necessário saber: CNPJ, nome, endereço e telefone. Sobre as concessionárias, deve-se armazenar: nome, CNPJ, endereço, telefone e cidade.

Sabe-se que a fábrica é a responsável por contratar uma transportadora para levar os carros à concessionária. Deve ser possível saber quais carros foram para qual concessionária e que transportadora fez a viagem.

Sobre as transportadoras, sabe-se: nome, CNPJ, endereço, telefone.

A concessionária que recebe os novos carros deve armazená-los, e posteriormente vendê-los. Sempre que um carro é vendido é necessário saber qual foi o carro, quem foi o cliente que o comprou, qual valor foi pago e a data da compra. É necessário também armazenar dados sobre os clientes e os carros.

- 6) Para o Diagrama de ER desenhado no exercício 5, faça a Descrição do Modelo Relacional, o Diagrama do Modelo Relacional e aplique as 3 Formas Normais para verificar se o Modelo Relacional está normalizado. Caso não esteja normalizado, você deve normalizá-lo.
- 7) Para o problema a seguir, desenhe o diagrama de ER, coloque as cardinalidades no diagrama e identifique as chaves primárias. Você pode acrescentar atributos ao modelo se o texto que descreve a situação-problema não apresentar todos os atributos necessários.

Uma escola oferece curso para pessoas que desejam aprender a fazer massagem e tornar essa a sua profissão. Para facilitar o trabalho da secretaria, deseja-se informatizar os cadastros e processos desse curso. A primeira coisa que o desenvolvedor terá que fazer é modelar uma base de dados para armazenar todas as informações.

A escola possui instrutores, e cada instrutor é responsável por até 15 alunos. É necessário manter os cadastros desses instrutores (como nome, código, telefone, e-mail, etc.).

Os alunos que fazem o curso atendem gratuitamente pessoas da comunidade. Estas podem se inscrever (se houver vagas) e fechar um pacote de massagens. Na inscrição, é necessário que a pessoa interessada forneça dados como: nome, CPF, RG, endereço, telefone e uma descrição de problemas de saúde (caso tenha algum problema). O sistema vai gerar para essa pessoa um cartão com um código de participante e o nome da pessoa.

Cada pacote é composto de várias sessões de 50 minutos. Todo pacote tem um código, uma descrição do pacote e o número de sessões. A definição do número de sessões por pessoa depende da avaliação do massagista.

A cada sessão é feito um tipo de massagem. As sessões possuem um código e são marcadas em dias e horários específicos. Elas estão sempre relacionadas ao pacote e às massagens.

Sobre as massagens, é necessário saber o seu código e o nome da massagem. As massagens estão divididas em modalidades (como terapêutica, estética, etc.). Existem várias modalidades previamente cadastradas, mas deve ser possível cadastrar uma modalidade nova.

Dentro de cada pacote podem ter massagens de mais de uma modalidade.

- 8) Para o Diagrama de ER desenhado no exercício 7, faça a Descrição do Modelo Relacional, o Diagrama do Modelo Relacional e aplique as 3 Formas Normais para verificar se o Modelo Relacional está normalizado. Caso não esteja normalizado, você deve normalizá-lo.
- 9) Para o problema a seguir, desenhe o diagrama de ER, coloque as cardinalidades no diagrama e identifique chaves primárias. Você pode acrescentar atributos ao modelo se o texto que descreve a situação-problema não apresentar todos os atributos necessários.



Um estacionamento deseja informatizar alguns dos seus processos manuais. Atualmente, todo o trabalho é feito manualmente pelo funcionário responsável.

Quando um carro chega ao estacionamento é necessário cadastrar a placa do carro, a cor, o modelo e a marca. Um carro é sempre de um modelo, mas um modelo pode pertencer a vários carros. Um modelo pertence a uma marca específica e uma marca pode ter vários modelos.

É necessário que para os carros que utilizam o estacionamento também seja registrada a data e a hora que o carro chegou. Quando um carro vai deixar o estacionamento, deve-se registrar data e hora de saída e o valor pago.

O estacionamento possui tarifas diferenciadas, dependendo do horário, dia da semana e tipo do cliente. Deve ser possível cadastrar essas tarifas e alterá-las sempre que necessário.

Quando o cliente é horista, não é necessário fazer o cadastro dele (apenas o do carro). No entanto, o estacionamento também trabalha com clientes mensalistas.

Quando o cliente é mensalista, é necessário fazer um cadastro desse cliente e associar esse cadastro ao carro do cliente. Um cliente mensalista pode cadastrar também duas pessoas para retirar o carro do estacionamento em seu nome. Sobre essas pessoas, é necessário saber o nome e o CPF.

O estacionamento também possui convênio com algumas lojas que estão localizadas perto. Quando um cliente vai a uma dessas lojas, a loja carimba o *ticket* do estacionamento e o dono do carro não precisará fazer o pagamento para o estacionamento. Nesse caso, a loja conveniada fará o pagamento. É necessário armazenar informações sobre as lojas conveniadas. Quando um cliente (horista, mensal ou conveniado) efetua o pagamento é necessário saber quanto ele pagou e a data que efetuou o pagamento.

- 10) Para o Diagrama de ER desenhado no exercício 9, faça a Descrição do Modelo Relacional, o Diagrama do Modelo Relacional e aplique as 3 Formas Normais para verificar se o Modelo Relacional está normalizado. Caso não esteja normalizado, você deve normalizá-lo.

# Implementação do Modelo Lógico: Linguagem SQL

Uma vez que os modelos de dados conceitual e lógico estão prontos, a próxima etapa do desenvolvimento de um projeto de banco de dados é a implementação do modelo em um SGBD relacional. Para fazermos a implementação, teremos que utilizar uma linguagem própria para esse tipo de banco de dados, chamada Linguagem SQL.

A Linguagem SQL (*Structured Query Language* ou Linguagem de Consulta Estruturada) é uma linguagem para banco de dados relacional. Ela foi desenvolvida na década de 70 e tem sido aprimorada e padronizada desde então. As duas entidades responsáveis pela sua padronização são: ANSI (*American National Standards Institute*) e ISO (*International Standards Organization*). Os padrões mais recentes são SQL-99 e SQL-2003.

Atualmente, nenhum SGBD adota integralmente o padrão desenvolvido pela ANSI. Normalmente, as empresas que desenvolvem o SGBD fazem uma customização do padrão para a sua ferramenta, mas a cada versão absorvem mais características do padrão.

Embora a tradução do nome SQL seja “linguagem de consulta”, essa linguagem possui vários recursos, além da consulta a uma base de dados, como por exemplo, meios para a definição da estrutura de dados, para modificação, para inserção, para exclusão de dados, para a especificação de restrições de segurança, etc.

A linguagem SQL possui subdivisões de comandos. Essas subdivisões consistem numa separação dos comandos pela função que desempenham. As duas principais subdivisões são:

1. DDL (*Data Definition Language*) ou Linguagem de Definição de Dados: disponibiliza comandos para a definição e criação do esquema da base de dados; comandos para criação, exclusão e alteração de objetos na base de dados (exemplos de objetos: tabelas, visões, índices, procedimentos armazenados, etc.) comandos que especificam direitos de acesso; e comandos que permitem criar restrições de integridade (*triggers*).
2. DML (*Data Manipulation Language*) ou Linguagem de Manipulação de Dados: disponibiliza comandos para inserção, exclusão e alteração de dados na base de dados. Além disso, possui comandos que permitem que o usuário ou o programa de aplicação recuperem as informações previamente armazenadas na base de dados.

Neste capítulo e nos capítulos 8, 9 e 10 abordaremos os principais comandos SQL DDL e DML.

O modelo relacional utilizado nos exemplos desses capítulos será o descrito na figura 7.1.

```
tbCliente (nome_cli:caracter(100), codigo_cli:inteiro, CPF_cli:caracter(11),  
data_cadastro:data, cidade_cli:caracter(50), UF_cli:caracter(2))  
  
tbCategoria (codigo_categoria:inteiro, nome_categoria: caracter(20))  
  
tbClasse (codigo_classe:inteiro, nome_classe: caracter(20), preco_classe:real)  
  
tbTitulo (codigo_titulo:inteiro, nome_titulo: caracter(50), ano:numeric(4),  
codigo_categoria:inteiro, codigo_classe:inteiro)  
codigo_categoria referencia tbCategoria  
codigo_classe referencia tbClasse  
  
tbFilme(codigo_filme:inteiro, codigo_titulo:inteiro, nome_distribuidor:caracter(20))  
codigo_titulo referencia tbTitulo  
  
tbEmprestimo_devolucao (codigo_cli:inteiro, codigo_filme:inteiro,  
data_emprestimo:data, data_devolucao_prevista:data, data_devolucao_  
efetiva:data, valor_multa:real)  
codigo_cli referencia tbCliente  
codigo_filme referencia tbFilme
```

Figura 7.1 – Modelo relacional para uma locadora de filmes

Esse modelo é um modelo clássico e bastante didático na área de banco de dados. Ele representa uma base de dados para uma Locadora de Filmes. Esta locadora possui vários títulos de filmes e para cada título têm-se vários DVDs dos filmes daquele título. Os clientes emprestam os DVDs que contém os filmes. Os filmes pertencem a uma categoria (por exemplo: romance, policial, terror, etc.) e possuem uma classe. Essa classe define se um filme é lançamento ou catálogo e define o preço da locação.

## Comandos DDL

A primeira ação que devemos fazer é selecionar o SGBD que será utilizado para criar a base de dados e suas respectivas tabelas.

Neste livro, foi selecionado o SGBD **MySQL 5.1**. A escolha desse SGBD deve-se ao fato dele ser bastante utilizado pela comunidade de informática e por ser um SGBD gratuito.

### História do MySQL

MySQL começou em 1995, com o nome de MySQL AB. Seus fundadores são Michael Widenius, David Axmark e Allan Larsson. Durante os anos, várias versões foram desenvolvidas para acrescentar funcionalidades. As principais versões são a 3.23 (de 2001), a 4.0 (de 2003), a 4.1 (de 2004), a 5.0 (de 2005) e a 5.1 (de 2008).

O ano de 2008 foi importante, pois a empresa foi vendida para a Sun Microsystems, por 1 bilhão de dólares. Em 2010, a Oracle Corporation, a maior empresa de banco de dados do mundo, comprou a Sun Microsystems por 7 bilhões de dólares, levando junto o MySQL.

Mesmo após a aquisição pela Oracle, o MySQL continua sendo um sistema *open source* (código livre).

# Criação de Base de Dados e Esquemas

Depois de selecionado o SGBD, é necessário criar a base de dados no SGBD escolhido. Para criar uma base de dados no MySQL usamos o comando **CREATE**.

Sintaxe:

```
CREATE DATABASE nome da base de dados;
```

Vamos criar a base de dados tbLocadora para o nosso modelo relacional da figura 7.1.

Comando:

```
CREATE DATABASE dbLocadora;
```

Também é possível excluir uma base de dados. Para excluir uma base de dados utiliza-se o comando **DROP**.

Sintaxe:

```
DROP DATABASE nome da base de dados;
```

Vamos excluir a base de dados tbLocadora criada anteriormente.

Comando:

```
DROP DATABASE dbLocadora;
```

É importante ressaltar que o comando **DROP DATABASE** deve ser executado com cuidado, já que irá eliminar também todas as tabelas existentes na base. Este comando é dificilmente executado depois que temos a base de dados em produção.

Lembre-se de que podemos criar tantas bases de dados quantas forem necessárias no SGBD. Por esse motivo, antes de criarmos as tabelas para a base de dados, é necessário definir qual base de dados iremos utilizar. Para isso, usamos o comando **USE**. Ele indica que os comandos subsequentes irão ser executados sobre a base de dados escolhida.

Comando selecionar a base de dados:

```
USE dbLocadora;
```

# Criação de Tabelas e Definição de Constraints (Restrições)

Depois de criada a base de dados, é necessário criarmos as tabelas para essa base de dados. Não se esqueça de indicar a base de dados que será utilizada antes de criar as tabelas. A sintaxe de criação de uma tabela é:

```
CREATE TABLE Nome da Tabela
(
  Nome do atributo1 Tipo de dado [NOT NULL] [DEFAULT(expressão_padrão)],
  Nome do atributo2 Tipo de dado [NOT NULL] [DEFAULT(expressão_padrão)],
  ...
  [Demais restrições]
);
```

O que está entre colchetes na sintaxe acima significa opcional. Ou seja, pode existir ou não quando se cria uma tabela.

O tipo de dados de um atributo vai depender do SGBD escolhido. No caso do SGBD MySQL, os principais tipos são:

Tipo	Nome	Descrição
Caractere	char(n)	String de tamanho fixo
Caractere	varchar(n),	String de tamanho variável
Numérico de ponto fixo	decimal(p,e) ou numeric(p,e)	Número que tem precisão e escala fixas (“p” representa o número total de dígitos; “e” representa o número de casas decimais)
Numérico aproximado	float, real	Número com ponto flutuante
Numérico inteiro	int, smallint, tinyint, bigint	Números que usam dados inteiros
Data e hora	Datetime, smalldatetime, timestamp	Tipo de dado para armazenar data e hora no mesmo atributo
Data	Date	Tipo de dado para armazenar data
Hora	Time	Tipo de dado para armazenar hora

A restrição **NOT NULL**, quando aplicada a um atributo, indica que este atributo deve ter o valor obrigatoriamente preenchido. Por exemplo, se definirmos que o atributo “ano” na tabela tbTitulo é **NOT NULL**, então teremos que digitar um valor para esse campo todas as vezes que preencheremos uma linha dessa tabela. O padrão no MySQL é **NULL**, ou seja, se não for colocado explicitamente o valor **NOT NULL** para o atributo ele permitirá valores nulos.

A restrição **DEFAULT** indica qual valor deverá ser atribuído a um atributo caso o usuário não especifique algum valor. Por exemplo, se o atributo `cidade_cli` da tabela `tbCliente` tiver uma restrição de **DEFAULT** com o valor padrão “Curitiba” e na hora de preencher os dados do cliente não for digitado nenhum valor para `cidade_cli`, então o valor “Curitiba” será automaticamente preenchido pelo SGBD para aquela tabela. A restrição **DEFAULT** não pode ser associada a um atributo que é chave primária (**PK**).

Existem ainda outras quatro restrições, conhecidas como *constraints*, que podem ser aplicadas aos atributos de uma tabela. São elas:

1. **UNIQUE**: é utilizada para manter os dados inseridos com valores únicos. Se um campo estiver definido com a restrição **UNIQUE** nenhum valor repetido poderá ser digitado para este campo. Podemos ter várias restrições **UNIQUE** numa única tabela, mas para cada atributo que possui essa restrição só podemos inserir o valor nulo (*null*) uma vez.
2. **CHECK**: permite criar uma regra que verifica se o valor inserido para o atributo é um valor permitido para o atributo.
3. **PRIMARY KEY (PK)**: utilizada para definir a chave primária. Não pode se repetir e não aceita valor nulo.
4. **FOREIGN KEY (FK)**: implementa o conceito de chave estrangeira e garante a integridade referencial. O conteúdo de um campo que tem uma **FK** deve se referenciar a outro campo que possua ou uma chave primária ou uma *constraint* **UNIQUE**. Atributos com a restrição **FOREIGN KEY** permitem o valor nulo.

A sintaxe para a definição das *constraints* é dado a seguir:

CONSTRAINT de PRIMARY KEY:

*CONSTRAINT nome da constraint PRIMARY KEY (atributo que recebe a constraint)*

CONSTRAINT de FOREIGN KEY:

*CONSTRAINT nome da constraint FOREIGN KEY (atributo que recebe a constraint) REFERENCES nome da tabela referenciada (nome do atributo referenciado)*

*[ON DELETE CASCADE]*

*[ON UPDATE CASCADE]*

CASCADE: se a linha da tabela que tem a PK for apagada/modificada, a linha da tabela que tem a FK também o será.

CONSTRAINT de UNIQUE:

*CONSTRAINT nome da constraint UNIQUE (atributo que recebe a constraint)*

CONSTRAINT de CHECK:

*CONSTRAINT nome da constraint CHECK (regra)*

## Cuidado!

O nome da *constraint* é o nome que se dá para a restrição. O nome da *constraint* não muda o nome do atributo.

Quando uma tabela é criada, cria-se um objeto dentro do SGBD. Qualquer alteração nesse objeto deve ser feito por meio de um comando **ALTER** (vamos comentar sobre ele daqui a pouco). Não adianta executar novamente o comando **CREATE** que ele não vai funcionar. Vai dar um erro dizendo que o objeto já foi criado.

Vamos criar as tabelas para o nosso exemplo da Locadora. É importante observar que as tabelas que não têm **FK** devem ser criadas primeiro. Isso ocorre porque uma **FK** faz referência a atributos de outra tabela, e para fazer essa referência a outra tabela deve existir.

```
CREATE TABLE tbCliente
(codigo_cli INT,
CPF_cli CHAR(11),
nome_cli VARCHAR(20) NOT NULL,
data_cadastro DATE,
cidade_cli VARCHAR(20),
UF_cli CHAR(2) DEFAULT 'PR',
CONSTRAINT un_CPFcli UNIQUE (CPF_cli),
CONSTRAINT pk_tbCliente PRIMARY KEY (codigo_cli)
);

CREATE TABLE tbCategoria
(codigo_categoria INT,
nome_categoria VARCHAR(20) NOT NULL,
CONSTRAINT pk_tbCategoria PRIMARY KEY (codigo_categoria)
);

CREATE TABLE tbClasse
(codigo_classe INT,
nome_classe VARCHAR(20) NOT NULL,
preco_classe NUMERIC(4,2) NOT NULL,
CONSTRAINT pk_tbClasse PRIMARY KEY (codigo_classe),
CONSTRAINT ck_NomeClasse CHECK (nome_classe IN ('Lançamento', 'Catálogo'))
);

CREATE TABLE tbTitulo
(codigo_titulo INT,
nome_titulo VARCHAR(50) NOT NULL,
ano NUMERIC(4),
codigo_categoria INT,
```

Valores não numéricos, como caractere, data e hora devem estar entre aspas simples.

A última linha não tem vírgula

NUMERIC(4,2) = Números com até 4 dígitos e 2 casas decimais

Só aceita os valores "Lançamento" ou "Catálogo"

```

codigo_classe INT,
CONSTRAINT pk_tbTitulo PRIMARY KEY (codigo_titulo),
CONSTRAINT fk_tbTitulo_tbCategoria FOREIGN KEY (codigo_categoria)
    REFERENCES tbCategoria (codigo_categoria)
    ON DELETE CASCADE ON UPDATE CASCADE,
CONSTRAINT fk_tbTitulo_tbClasse FOREIGN KEY (codigo_classe) REFERENCES
tbClasse (codigo_classe)
    ON DELETE CASCADE ON UPDATE CASCADE
);

CREATE TABLE tbFilme
(codigo_filme INT,
codigo_titulo INT,
nome_distribuidor VARCHAR(20),
CONSTRAINT pk_tbFilme PRIMARY KEY (codigo_filme),
CONSTRAINT fk_tbFilme_tbTitulo FOREIGN KEY (codigo_titulo) REFERENCES
tbTitulo (codigo_titulo)
    ON DELETE CASCADE ON UPDATE CASCADE
);

CREATE TABLE tbEmprestimoDevolucao
(codigo_cli INT,
codigo_filme INT,
data_emprestimo DATE NOT NULL,
data_devolucao_prevista DATE NOT NULL,
data_devolucao_efetiva DATE,
valor_multa NUMERIC(4,2) DEFAULT 0,
CONSTRAINT pk_tbEmpDev PRIMARY KEY (codigo_cli,codigo_filme,data_emprestimo),
CONSTRAINT fk_tbEmpDev_tbCliente FOREIGN KEY (codigo_cli) REFERENCES
tbCliente (codigo_cli)
    ON DELETE CASCADE ON UPDATE CASCADE,
CONSTRAINT fk_tbEmpDev_tbFilme FOREIGN KEY (codigo_filme) REFERENCES
tbFilme (codigo_filme)
    ON DELETE CASCADE ON UPDATE CASCADE
);

```

Definição de uma Chave Primária Composta (atributos que compõem a PK devem estar entre parênteses)

Os atributos "codigo\_cli" e "codigo\_filme" fazem PARTE da PK e individualmente são FK

Para definirmos um campo de autoincremento na tabela, usamos o comando **AUTO\_INCREMENT** no MySQL. Para isso, o atributo deverá ser numérico. Quando se define um autoincremento para a coluna, o valor para essa coluna será inserido automaticamente pelo SGBD. O comando para definir o autoincremento muda dependendo do SGBD.



```
CREATE TABLE tbCategoria
(codigo_categoria INT AUTO_INCREMENT,
nome_categoria VARCHAR(20) NOT NULL,
CONSTRAINT pk_tbCategoria PRIMARY KEY (codigo_categoria)
);
```

Para excluir uma tabela, utiliza-se o comando **DROP**. Quando você exclui uma tabela, você exclui **todos** os dados que foram armazenados nela.

#### Sintaxe:

```
DROP TABLE nome da tabela;
```

#### Comando:

```
DROP TABLE tbEmprestimoDevolucao;
```

Não é possível excluir uma tabela que tenha uma **PK** que é referenciada por outra tabela, por exemplo, você não pode excluir a tabela tbClasse porque o codigo\_classe é uma **FK** em tbTitulo. Nesse caso, temos que primeiramente excluir a *constraint* de **FK**, para depois excluirmos a tabela. Para excluirmos uma *constraint* de **FK**, usamos o comando **ALTER**.

## Comando ALTER

O comando **ALTER** nos permite alterar a estrutura de uma tabela. Com esse comando, podem-se adicionar atributos, excluir atributos, alterar o tipo ou tamanho do atributo, adicionar e excluir *constraints*, etc.

A sintaxe do comando **ALTER** muda bastante entre os SGBD. A seguir, vamos ver alguns exemplos usando o comando **ALTER** no MySQL.

#### Adicionando um novo atributo:

```
ALTER TABLE tbCliente
ADD Celular_Cli varchar(8);
```

#### Excluindo um atributo de uma tabela:

```
ALTER TABLE tbCliente
DROP Celular_Cli;
```

#### Alterando o tipo e o tamanho de um atributo:

```
ALTER TABLE tbCliente
MODIFY Cidade_Cli char(25);
```

Alterou o tipo varchar para char e o tamanho de 20 para 25

#### Adicionando/Alterando Default:

```
ALTER TABLE tbCliente
ALTER COLUMN UF_cli SET DEFAULT 'SP';
```

#### Excluindo Default:

```
ALTER TABLE tbCliente  
ALTER COLUMN UF_cli DROP DEFAULT;
```

#### Alterando de NULL para NOT NULL:

```
ALTER TABLE tbCliente  
MODIFY Cidade_Cli char(25) NOT NULL;
```

#### Alterando de NOT NULL para NULL:

```
ALTER TABLE tbCliente  
MODIFY Cidade_Cli char(25) NULL;
```

Quando se altera de NULL para NOT NULL e vice-versa, deve-se repetir o nome do atributo e o tipo antes de permitir ou não nulos.

#### Excluindo uma constraint de chave estrangeira:

```
ALTER TABLE tbTitulo  
DROP foreign key fk_tbTitulo_tbClasse;
```

#### Adicionando uma constraint de chave estrangeira:

```
ALTER TABLE tbTitulo  
ADD CONSTRAINT fk_tbTitulo_tbClasse foreign key (codigo_categoria)  
REFERENCES tbCategoria(codigo_categoria);
```

#### Adicionando constraint Unique:

```
ALTER TABLE tbCliente  
ADD CONSTRAINT UN_CPFcli UNIQUE (cpf_cli) ;
```

#### Excluindo uma constraint Unique:

```
ALTER TABLE tbCliente  
DROP INDEX UN_cpfcli;
```

O MySQL considera uma restrição Unique como um índice. Por isso, para excluir essa restrição, deve-se excluir o índice.

## Índices

Um índice é definido sobre um atributo para melhorar o desempenho das consultas, ou seja, para que as consultas executem-se mais rapidamente.

A criação de índices em uma tabela é o principal método de otimização que os SGBD oferecem. Isso porque a execução de consultas que utilizam atributos indexados são, na maioria dos casos, mais rápidas do que as buscas por campos não indexados.

Um índice pode ser associado a um atributo ou a um conjunto de atributos, podendo ser restrito a valores únicos ou permitir repetições. Um índice também pode ser simples, quando possui apenas um atributo; ou composto, quando possui mais de um atributo na sua composição.

Índices **PRIMARY** são índices únicos e a indexação é feita por meio do atributo que é chave primária. Neste caso, o arquivo de dados é ordenado pela chave primária. Esses índices são, normalmente, criados quando o usuário define a chave primária da tabela e estão sempre associados a essa chave primária. Só pode haver um índice **PRIMARY** por tabela.

Um índice **UNIQUE** é um índice que pode ser associado a qualquer atributo da tabela. No entanto, ele tem a particularidade de não permitir que os valores do campo indexado apareçam mais de uma vez. Pode haver vários índices **UNIQUE** por tabela.

Um índice do tipo **INDEX** cria um índice básico que não possui restrições de unicidade, ou seja, permite valores repetidos. Pode haver vários desses índices por tabela. É o SGBD quem define o número máximo de índices por tabela.

A criação de um índice em uma tabela pode ser feita pelo comando **CREATE** ou pelo comando **ALTER TABLE**.

#### Sintaxe do CREATE:

```
CREATE [UNIQUE] INDEX nome do índice  
ON nome da tabela (nome do atributo1 [{, nome do atributo n }]);
```

#### Sintaxe do ALTER:

```
ALTER TABLE nome da tabela  
ADD [UNIQUE] INDEX nome do índice (nome do atributo1 [{, nome do atributo n }]);
```

A “{” indica que pode haver vários atributos no índice

Vamos fazer um exemplo de criação de um índice:

#### Utilizando o CREATE:

```
CREATE INDEX idx_NomeCli  
ON tbCliente (nome_cli);
```

#### Utilizando o ALTER:

```
ALTER TABLE tbCliente  
ADD INDEX idx_NomeCli (nome_cli);
```

Um índice pode ser excluído por meio do comando **DROP**.

#### Excluindo diretamente o índice:

```
DROP INDEX nome do índice ON nome da tabela;
```

#### Utilizando o ALTER para excluir o índice:

```
ALTER TABLE nome da tabela  
DROP INDEX nome do índice;
```

No MySQL, podemos ver os índices já existentes em uma tabela, pelo comando:

```
SHOW INDEX FROM nome da tabela;
```

## Algumas Dicas para Utilização de Índices:

- Os índices devem ser criados para atributos usados como condição em uma consulta (no **WHERE**), ordenamento e agrupamento. Os índices não precisam estar associados às colunas que serão exibidas ao usuário (por meio do **SELECT**).
- Em tabelas que tem poucos registros não há a necessidade de índice, e o uso do índice pode causar perda de desempenho. Quando a tabela é pequena, o trabalho envolvido em acessar o índice, pegar o endereço e acessar a tabela é maior do que o esforço de ler a tabela inteira.
- Deve-se indexar atributos que tenham poucos valores que se repetem. Quanto mais seletivo for o atributo, mais vantajoso será indexar o campo. Se os valores para um atributo repetem-se em ao menos 30% das linhas, o índice pode não ser eficiente. Por exemplo, não adianta criar um índice para o atributo sexo.
- Quanto menor o espaço ocupado por um tipo de dado, mais eficiente a indexação. Tipos de dados grandes requerem mais espaço em disco e tornam mais lentas as inserções, exclusões e atualizações nos campos da tabela. Por exemplo, não usar *bigint* quando *int* for suficiente.
- A ordem dos atributos no índice dá diferença de desempenho. Por exemplo, uma lista telefônica organizada em ordem alfabética composta de sobrenome e nome é mais eficiente do que se for organizada por nome e sobrenome.

### Atividades

- 1) Desenhe o DER para o modelo relacional da figura 7.1.
- 2) Explique para que serve um índice e quando devemos utilizá-lo.
- 3) Por que quando criamos um índice para o atributo sexo esse índice não será eficiente?

Para os exercícios de 4 a 10, utilize o modelo relacional abaixo:

```
tbEmpregado (cod_emp, nome_emp, rua_emp, cidade_emp, fone_emp,  
cod_gerente)  
cod_gerente referencia tbGerente  
  
tbTrabalha (cod_emp, cod_companhia, salário)  
cod_emp referencia tbEmpregado  
cod_companhia referencia tbCompanhia  
  
tbCompanhia (cod_companhia, nome_companhia, cidade_companhia)  
  
tbGerente (cod_gerente, nome_gerente)
```

- 4) Crie a base de dados usando comando SQL.
- 5) Defina a ordem de criação das tabelas e explique como você escolheu a ordem das tabelas.
- 6) Escreva os comandos SQL para criar todas as tabelas, na ordem definida no exercício 5. As chaves primárias e estrangeiras, bem como qualquer restrição para um atributo deverão ser definidas diretamente no momento de criação das tabelas.
- 7) Crie um índice do tipo INDEX (usando comando SQL) para o nome do empregado na tabela de empregados.
- 8) Crie um índice único (usando comando SQL) para o nome da companhia na tabela da companhia.
- 9) Usando comando SQL, acrescente o atributo telefone\_gerente em tbGerente.
- 10) Usando comando SQL, exclua a tabela tbGerente.

# Linguagem SQL – DML

Depois que o esquema da base de dados, com suas tabelas e restrições, foi criado usando os comandos DDL, podemos começar a inserir os dados dentro da nossa base de dados. Para fazermos isso, usamos os comandos DML. Os comandos dessa linguagem permitem-nos inserir, alterar, excluir e consultar dados na nossa base de dados.

Os comandos DDL variam de SGBD para SGBD. Para os comandos DML, a sintaxe não têm muita diferença entre os SGBD. Portanto, os comandos dos capítulos 8, 9 e 10 podem ser utilizados na maioria dos SGBD.

## Comando INSERT

O comando **INSERT** é usado em banco de dados, quando queremos inserir dados na base. Podemos especificar uma linha a ser inserida ou escrever uma consulta cujo resultado é um conjunto de linhas a inserir.

Os valores de atributos a serem inseridos devem estar na ordem que foram definidos no **CREATE TABLE**. Caso não estejam (ou o usuário não lembre a ordem correta), é necessário especificar a ordem, como mostram os exemplos a seguir:

Sintaxe quando não se conhece a ordem dos atributos:

```
INSERT INTO nome da tabela (atributo_1, ..., atributo_n)
```

```
VALUES (valor_1, ... , valor_n);
```

Sintaxe quando se conhece a ordem dos atributos (neste caso, podem-se omitir os atributos):

```
INSERT INTO nome da tabela
```

```
VALUES (valor_1, ... , valor_n);
```

Como exemplo, vamos inserir dados na tabela tbClasse.

Sintaxe, quando não se conhece a ordem dos atributos:

```
INSERT INTO tbClasse (nome_classe, preco_classe, codigo_classe)
VALUES ('Lançamento', 7.50, 1);
```

Sintaxe, quando se conhece a ordem dos atributos:

```
INSERT INTO tbClasse
VALUES (1, 'Lançamento', 7.50);
```

Use o ponto (.) para separar casas decimais.

Se você quiser inserir nulo para algum atributo que permita esse valor ou se quiser que o SGBD insira o valor *default* definido na criação da tabela, pode fazer assim:

Explicitando *null* e *default*:

```
INSERT INTO tbCliente (codigo_cli, CPF_cli, nome_cli, data_cadastro,
cidade_cli, UF_cli)
VALUES (1, '12345678911', 'Pedro', null, null, default);
```

Explicitando apenas os atributos que serão preenchidos:

```
INSERT INTO tbCliente (codigo_cli, CPF_cli, nome_cli)
VALUES (1, '12345678911', 'Pedro');
```

É importante lembrar que o valor de uma **FK** deve ser um valor nulo ou um valor cadastrado na sua **PK** correspondente. Sendo assim, você precisa preencher primeiro os dados das tabelas que tem **PK** referenciadas por **FK**. Caso você tente inserir para a **FK** um valor que não tenha correspondência na **PK**, o SGBD vai emitir um erro dizendo não ser possível inserir, pois isso fere a integridade referencial da base de dados.

Como foi dito anteriormente, além de inserir uma linha em uma tabela, o comando **INSERT** pode ser utilizado junto a um comando de consulta para retornar um conjunto de linhas que serão inseridos em outra tabela.

Por exemplo, suponha que você queira criar uma nova tabela na base de dados dbLocadora para armazenar o título de um filme e o nome da categoria do filme. Para essa nova tabela, vamos utilizar os dados que já temos cadastrados nas tabelas tbTitulo e tbCategoria, como mostra o exemplo a seguir.

Criando a nova tabela:

```
CREATE TABLE tbTituloCategoria  
(nome_titulo varchar (50),  
  nome_categoria varchar (20)  
);
```

Fazendo uma consulta em que as linhas do resultado serão inseridas na tabela tbTituloCategoria:

```
INSERT INTO tbTituloCategoria(nome_titulo, nome_categoria)  
SELECT nome_titulo, nome_categoria  
FROM tbtitulo  
INNER JOIN tbCategoria ON tbCategoria.codigo_categoria = tbTitulo.  
codigo_categoria;
```

O **SELECT** dentro do comando **INSERT** vai retornar o nome do título e o nome da categoria e os valores retornados serão inseridos dentro da tabela tbTituloCategoria. Os comandos (**SELECT**, **FROM** e **INNER JOIN**) referentes à consulta apresentada dentro do **INSERT** serão explicados no decorrer deste capítulo.

## Comando DELETE

O comando **DELETE** é utilizado para excluir registros de uma tabela. Este comando não exclui dados de atributos específicos e sim linhas inteiras da tabela. Se a sua intenção for excluir valores de um atributo específico (ou seja, deixar o atributo com valor *null*), você deve utilizar o comando **UPDATE** e mudar o valor do atributo para **NULL**.

Após remover os registros usando o comando **DELETE**, você não poderá desfazer a operação.

Sintaxe do comando DELETE:

```
DELETE FROM nome da tabela[;]  
  
[WHERE condição;]
```

O comando **WHERE** permite aplicar uma condição que seleciona quais linhas da tabela serão excluídas.

Excluir todos os empréstimos feitos para o filme de código 21:

```
DELETE FROM tbEmprestimoDevolucao  
  
WHERE codigo_filme = 21;
```



Quando você usa o **DELETE** sem a condição **WHERE**, **todos** os registros da sua tabela serão excluídos.

Excluindo todos os dados da tabela tbEmprestimoDevolucao:

```
DELETE FROM tbEmprestimoDevolucao;
```

Se você não usou o **ON DELETE CASCADE** na criação da tabela, não será possível excluir registros de uma tabela que tenham uma **PK** que é referenciada por uma **FK** em outra tabela. Isso daria um erro porque a tabela que tem a chave estrangeira perderia a referência, e isso comprometeria a integridade da base de dados.

Por exemplo, suponha que você queira excluir o cliente Pedro da sua base de dados. Se você fizer simplesmente um **DELETE** na tabela tbCliente, o SGBD vai emitir uma mensagem de erro, porque o cliente Pedro é referenciado na tabela tbEmprestimoDevolucao. Nesse caso, você terá que excluir primeiramente todos os registros referentes ao Pedro na tabela tbEmprestimoDevolucao e em seguida excluir o registro da tabela tbCliente, como mostra o exemplo abaixo:

1º passo:

```
DELETE FROM tbEmprestimoDevolucao  
WHERE codigo_cli = (SELECT codigo_cli FROM tbCliente WHERE nome_cli = 'Pedro');
```

2º passo:

```
DELETE FROM tbCliente  
WHERE nome_cli = 'Pedro';
```

Se você utilizar o comando **ON DELETE CASCADE** na criação da tabela e se a linha da tabela que tem a **PK** for apagada, o SGBD se encarregará de apagar também a linha da tabela que tem a **FK** correspondente. Nesse caso, só teríamos de executar o **DELETE** em tbCliente (2º passo do exemplo acima).

# Comando UPDATE


O comando **UPDATE** modifica valores inseridos dentro de uma tabela.

Sintaxe do comando UPDATE:

```
UPDATE nome da tabela  
  
SET nome do atributo1 = Novo valor [{, nome do atributo_n = novo  
valor};]  
  
[WHERE condição;]
```

O comando **SET** é um comando de atribuição. Deve-se especificar quais atributos terão seus valores alterados e o novo valor.

Pode-se utilizar ainda uma condição especificando qual (ou quais) registro deverá ser alterado por meio do comando **WHERE**. Com o **WHERE** será feita uma procura por registros que satisfazem a condição e as alterações serão aplicadas apenas nesses registros.



**Cuidado:** o uso do UPDATE sem a condição WHERE modifica os valores de todos os registros (linhas) da tabela.

Suponha que você queira alterar o preço de filmes da Classe “Lançamento” de R\$ 7,50 para R\$ 9,50. Para fazer isso, usamos o comando abaixo:

```
UPDATE tbClasse  
  
SET preco_classe = 9.50  
  
WHERE nome_classe = 'Lançamento';
```

Se você não usou o **ON UPDATE CASCADE** na criação da tabela e tentar alterar o valor de uma **PK** que é referenciada por uma **FK**, o SGBD irá emitir um erro. Se ele permitisse a alteração, estaria ferindo a regra de integridade referencial entre as tabelas.

Para que fique mais claro, imagine que você queira alterar o valor de `codigo_categoria` de filmes de terror de 2 para 200 na tabela `tbCategoria`. Como o código da categoria é chave estrangeira na tabela `tbTitulo`, você teria que alterar o código da categoria também nessa tabela. No entanto, se você não usou **UPDATE CASCADE**, o SGBD não fará isso automaticamente para você. Nesse caso, você teria que inserir um novo registro na tabela `tbCategoria` com o código 200, depois alterar na tabela `tbTitulo` todos os códigos de 2 para 200 e finalmente apagar na tabela `tbCategoria` o registro que tem o código 2, como mostra o exemplo a seguir:

1º passo:

```
INSERT INTO tbCategoria  
VALUES (200, 'Terror');
```

2º passo:

```
UPDATE tbTitulo  
SET codigo_categoria = 200  
WHERE codigo_categoria = 2;
```

3º passo:

```
DELETE FROM tbCategoria  
WHERE codigo_categoria = 2;
```

## Comando SELECT

Depois que inserimos dados em uma tabela, podemos começar a fazer consultas nessa tabela. A estrutura básica de uma consulta em SQL consiste em três comandos: **SELECT**, **FROM** e **WHERE**.

O comando **SELECT** é usado para selecionar os atributos desejados no resultado de uma consulta.

O comando **FROM** define quais tabelas serão usadas na consulta.

O comando **WHERE** descreve a condição da consulta e não é obrigatório. Por exemplo, se você quiser retornar o nome de todos os clientes da base de dados tbLocadora, você não precisará usar o comando **WHERE**.

Exemplo: Encontre o nome de todos os clientes.

```
SELECT tbCliente.nome_cli  
FROM tbCliente;
```

O nome de um atributo pode ser o mesmo em mais de uma tabela. Por exemplo, podemos ter o atributo “nome” na tabela tbProfessor e em tbAluno. Para permitir que o SGBD saiba qual é o atributo que você está se referindo, ou seja, para evitar ambiguidade, usamos NomeDaTabela.NomeDoAtributo numa consulta SQL.

Todos os atributos usados em uma consulta (tanto no **SELECT** quanto no **WHERE**) devem pertencer às tabelas que aparecem no **FROM**.

O resultado de uma consulta em SQL é sempre uma tabela, mesmo que ela tenha apenas uma linha e uma coluna.

Uma consulta SQL permite que no seu resultado apareçam valores duplicados. Para forçar a eliminação de duplicidade no resultado de uma consulta, usa-se o comando **DISTINCT**.

Suponha que na tabela tbTitulo existam três títulos com código da categoria igual a 3 cadastrados e dois títulos com o código da categoria igual a 1. Se não usarmos o **DISTINCT** o resultado da consulta será:

Exemplo: Encontre os códigos das categorias dos títulos dos filmes cadastrados.

```
SELECT tbTitulo.codigo_categoria  
FROM tbTitulo;
```

1
1
3
3
3

Utilizando o **DISTINCT**, teremos como resultado:

Exemplo: Encontre os códigos das categorias dos títulos dos filmes cadastrados.

```
SELECT DISTINCT tbTitulo.codigo_categoria  
FROM tbTitulo;
```

1
3

O asterisco **\*** pode ser usado junto ao comando **SELECT**, quando queremos listar todos os atributos de uma tabela.

Exemplo: Retorne todos os dados da tabela tbCliente.

```
SELECT *  
FROM tbCliente;
```

Neste exemplo acima, serão retornados todos os registros (porque a consulta não tem condição **WHERE**) e todas as colunas (por causa do **\***) da tabela tbCliente.

Um comando **SELECT** pode conter também expressões aritméticas (por exemplo: **+**, **-**, **\*** e **/**).

Exemplo: Verifique como ficariam os preços de filmes se aumentássemos em 5% o preço para cada classe.

```
SELECT tbClasse.nome_classe, tbClasse.preco_classe * 1.05  
FROM tbClasse;
```

## Importante!

Utilizar expressões aritméticas no **SELECT** não altera o valor que está inserido na tabela. Lembre-se de que para alterar o valor de um atributo na base de dados é necessário utilizar o comando **UPDATE**.

# Comando WHERE

O comando **WHERE** sempre terá associado a ele uma condição que determina quais registros deverão ser retornados pela consulta.

Por exemplo, se quisermos saber o nome de todos os clientes que moram em Curitiba, devemos colocar uma condição para retornar apenas as pessoas dessa cidade.

Exemplo: Encontre o nome de todos os clientes da cidade de Curitiba.

```
SELECT tbCliente.nome_cli  
FROM tbCliente  
WHERE tbCliente.cidade_cli = 'Curitiba';
```

Pode-se utilizar junto ao comando **WHERE** os operadores lógicos **AND**, **OR** e **NOT**. Além disso, também podem ser usados os operadores relacionais **<**, **<=**, **>**, **>=**, **=** e **<>**.

Exemplo: Encontre o nome de todos os clientes da cidade de Curitiba em que o código seja menor que 200.

```
SELECT tbCliente.nome_cli  
FROM tbCliente  
WHERE tbCliente.cidade_cli = 'Curitiba' AND tbCliente.codigo_cli < 200;
```

A SQL disponibiliza ainda um operador de comparação, chamado **BETWEEN**, que pode ser usado para simplificar a cláusula **WHERE**. Esse operador especifica que a consulta deve retornar linhas em que o valor do campo esteja dentro de um intervalo de valores.

Exemplo: Encontre o nome de todos os clientes da cidade de Curitiba que tenham data de cadastro no ano de 2009.

```
SELECT tbCliente.nome_cli
FROM tbCliente
WHERE tbCliente.cidade_cli = 'Curitiba'
AND tbCliente.data_cadastro BETWEEN '2009-01-01' AND '2009-12-31';
```

O comando **NOT BETWEEN** também pode ser utilizado para retornar linhas que estejam fora de um intervalo de valores.

Exemplo: Encontre o nome de todos os clientes da cidade de Curitiba que tenham data de cadastro em qualquer ano, exceto 2009.

```
SELECT tbCliente.nome_cli
FROM tbCliente
WHERE tbCliente.cidade_cli = 'Curitiba'
AND tbCliente.data_cadastro NOT BETWEEN '2009-01-01' AND '2009-12-31';
```

## Comando FROM

O comando **FROM** define quais tabelas serão utilizadas em uma consulta, ou seja, de quais tabelas devemos buscar os dados.

Até agora, em nossos exemplos, usamos apenas uma tabela no **FROM**. No entanto, é muito comum termos que acessar dados de duas ou mais tabelas para que possamos obter informações interessantes.

É importante sempre usar numa consulta o menor número de tabelas possível por questão de desempenho. Por isso, sempre faça uma análise de quais tabelas são realmente necessárias na sua consulta. Lembre-se de que as tabelas se relacionam por meio das chaves estrangeiras e primárias.

Quando precisamos utilizar duas ou mais tabelas, podemos fazer o **produto cartesiano** dessas tabelas. O produto cartesiano permite combinar informações de várias tabelas fazendo a combinação de todos os dados de uma tabela com todos os dados da outra tabela.

Para entender o produto cartesiano, imagine que tenhamos os seguintes dados nas tabelas tbCategoria e tbTitulo.

codigo_categoria	nome_categoria
1	Terror
2	Drama
3	Comédia

tbCategoria

<b>codigo_titulo</b>	<b>nome_titulo</b>	<b>codigo_categoria</b>
1	Mortos Vivos	1
2	Superando Desafios	3
3	A Hilariante	2
4	O Bicho Papão	1

tbTitulo

Suponha que queiramos fazer uma consulta que retorne os dados sobre títulos e a qual categoria pertencem. Para isso, teremos que usar as tabelas tbCategoria e tbTitulo, como mostrado abaixo:

```
SELECT*
FROM tbTitulo, tbCategoria;
```

O produto cartesiano é indicado pela ,

Como o produto cartesiano fará a combinação de todos os registros da tabela tbCategoria com todos os registros da tabela tbTitulo, o resultado do produto cartesiano para o nosso exemplo será uma tabela com 12 registros (multiplica-se o número de registros de uma tabela pelo número de registros da outra,  $3 \times 4 = 12$ ), que combina uma linha de tbTitulo com cada linha de tbCategoria.

<b>codigo_titulo</b>	<b>nome_titulo</b>	<b>codigo_categoria</b>	<b>codigo_categoria</b>	<b>nome_categoria</b>
1	Mortos Vivos	1	1	Terror
1	Mortos Vivos	1	2	Drama
1	Mortos Vivos	1	3	Comédia
2	Superando Desafios	3	1	Terror
2	Superando Desafios	3	2	Drama
2	Superando Desafios	3	3	Comédia
3	A Hilariante	2	1	Terror
3	A Hilariante	2	2	Drama
3	A Hilariante	2	3	Comédia
4	O Bicho Papão	1	1	Terror
4	O Bicho Papão	1	2	Drama
4	O Bicho Papão	1	3	Comédia

tbTitulo X tbCategoria

Observe que a consulta deveria retornar apenas as linhas da tabela que estão pintadas de cinza. As outras informações retornadas não são verdadeiras. Por exemplo, o filme “Mortos Vivos” é apenas da categoria “Terror”, mas no nosso resultado diz que ele pertence também a categoria “Drama” e “Comédia”.

A questão é: Como eu faço para resolver isso? Você deve ter observado que as linhas que estão corretas no nosso resultado são aquelas que têm chave primária em uma tabela (tbCategoria.codigo\_categoria) igual à chave estrangeira da outra tabela (tbTitulo.codigo\_categoria). Portanto, para resolver isso temos que colocar uma condição **WHERE** que diga que o valor da **PK** deve ser igual ao da **FK**, como mostra o exemplo:

```
SELECT *
FROM tbTitulo, tbCategoria
WHERE tbCategoria.codigo_categoria = tbTitulo.codigo_categoria;
```

Essa consulta nos retornaria o seguinte resultado:

codigo_titulo	nome_titulo	codigo_categoria	codigo_categoria	nome_categoria
1	Mortos Vivos	1	1	Terror
2	Superando Desafios	3	3	Comédia
3	A Hilariante	2	2	Drama
4	O Bicho Papão	1	1	Terror

Portanto, se usarmos o produto cartesiano a execução da consulta primeiramente irá gerar a tabela com todas as possíveis combinações de valores para depois aplicar a condição do **WHERE**.

Atualmente, a forma mais utilizada de se escrever uma consulta que utilize duas ou mais tabelas é usando o comando **INNER JOIN**. A diferença é que esse comando possui uma condição de junção. Essa condição é a mesma que fizemos no **WHERE** (**PK = FK**). No entanto, como ela é aplicada no **FROM** ela permite que os dados sejam “filtrados” durante a execução do **FROM**, diminuindo o tempo de execução da consulta, uma vez que o número de registros pesquisados será menor agora que só temos os dados que são corretos. Por exemplo:

```
SELECT *
FROM tbTitulo
INNER JOIN tbCategoria ON tbCategoria.codigo_categoria = tbTitulo.
codigo_categoria;
```

Essa consulta retornaria o mesmo resultado obtido anteriormente, porém o tempo de execução é mais rápido. É importante ressaltar que o otimizador de alguns SGBD (como por exemplo, o do SQLServer) quando se deparam com o produto cartesiano otimizam a consulta e executam o **INNER JOIN**. Portanto, não haverá diferença de desempenho nesse caso.

Observe que para verificar a diferença no tempo de execução quando usamos o **INNER JOIN** e o produto cartesiano, as tabelas devem ter um número grande de linhas.



A Sintaxe de um comando **INNER JOIN** é:

```
SELECT atributo1, ... atributo_n
FROM tabela1
INNER JOIN tabela2 ON condição da junção entre as tabelas 1 e 2
[{{INNER JOIN tabela 3 ON condição da junção entre as tabelas 1 e 3}}];
```



## Atividades

Resolva os exercícios a seguir, utilizando a descrição do modelo relacional abaixo. Utilize os comandos SQL. Se houver a necessidade de utilizar mais de uma tabela em uma consulta, utilize o INNER JOIN, em vez do produto cartesiano.

```
tbSala(numero_sala: inteiro, descricao_sala: caracter(20), capacidade: inteiro)

tbSalaFilme(numero_sala: inteiro, codigo_filme: inteiro, data: date, horario: time)
numero_sala referencia tbSala
codigo_filme referencia tbFilme

tbFilme(codigo_filme: inteiro, nome_filme: caracter(50), ano_lancamento: inteiro, categoria_filme: caracter(20), codigo_diretor: inteiro)
codigo_diretor referencia tbDiretor

tbDiretor(codigo_diretor: inteiro, nome_diretor: caracter(20))

tbPremio(codigo_premio: inteiro, nome_premio: caracter(20), ano_premiacao: inteiro, codigo_filme: inteiro)
codigo_filme referencia tbFilme
```

- 1) Escreva os comandos para criar todas as tabelas do modelo relacional anterior. As chaves primárias e estrangeiras, bem como qualquer restrição para um atributo deverão ser definidas diretamente no momento de criação das tabelas. Não se esqueça de que a ordem de criação das tabelas é importante.
- 2) Insira 3 registros para cada tabela criada.
- 3) Faça uma consulta que retorne o nome de todos os diretores cadastrados na base de dados.
- 4) Faça uma consulta que retorne o nome de todos os filmes da categoria “terror”.
- 5) Atualize a base de dados de forma que a capacidade da sala de número 8 passe de 150 lugares para 200 lugares.
- 6) Atualize a base de dados para que todos os filmes que seriam exibidos na sala 12 no dia 15/11/2010 sejam transferidos para a sala 8.
- 7) Exclua o diretor de nome “Pedro Paulo Matos”. Suponha que você não tenha criado as tabelas usando ON DELETE CASCADE.
- 8) Encontre o nome de todos os filmes dirigidos pelo diretor “Jorge da Penha”.
- 9) Faça uma consulta que retorne o nome de todos os filmes e horários de exibição para o dia 20/04/2010.
- 10) Faça uma consulta que retorne todos os prêmios que o filme “Titanic” ganhou.
- 11) Faça uma consulta que retorne o nome de todos os filmes que estão sendo exibidos em salas com capacidade maior que 200 lugares.
- 12) Faça uma lista com o nome de todos os filmes, número da sala e horários de exibição para o mês de maio de 2010.
- 13) Faça uma lista com o nome de todos os filmes que receberam o prêmio de melhor diretor nos anos de 2007, 2008, 2009 e 2010.
- 14) Faça uma consulta que retorne o nome de todos os filmes da categoria “comédia” que serão exibidos no mês de junho de 2010.
- 15) Faça uma consulta que retorne a descrição da sala, a categoria e o nome dos filmes que foram dirigidos pelo diretor “Severino Juca”.

# Outros Comandos SQL – DML

Neste capítulo, vamos ver alguns outros comandos muito úteis para fazer consultas em SQL.

## Aliases

Podemos criar apelidos tanto para atributos quanto para tabelas. Um apelido pode ser criado tanto no **SELECT** quanto no **FROM**.

Para criar um apelido, utiliza-se a palavra “**AS**”. No entanto, a palavra “**AS**” é opcional. Pode-se simplesmente colocar o apelido depois do nome da tabela ou do atributo.

O apelido para um atributo é particularmente útil quando usamos uma expressão aritmética ou função no atributo, porque nesse caso a coluna resultante ficará sem um nome. Também podemos usá-lo simplesmente para mudar o nome da coluna que irá aparecer no resultado da consulta. O exemplo abaixo cria um apelido para a coluna “tbClasse.preço\_classe x 1,05”. Note que o apelido está entre aspas porque temos uma frase. Se o apelido tivesse uma única palavra não haveria necessidade de aspas.

```
SELECT tbClasse.nome_classe,  
       tbClasse.preço_classe * 1.05 AS 'Preço com aumento de 5%'  
FROM tbClasse;
```

Outro uso do apelido é para tabelas. Podemos criar um apelido para uma tabela e utilizarmos esse apelido tanto no **SELECT** quanto no **WHERE**.

Exemplo: Encontre o nome dos títulos e o nome de sua respectiva categoria para filmes de 2009.

```
SELECT T.nome_titulo, C.nome_categoria  
FROM tbTitulo T  
INNER JOIN tbCategoria C ON C.codigo_categoria = T.codigo_categoria  
WHERE T.ano = 2009;
```

Finalmente, um apelido pode ser usado quando precisamos comparar dados de uma mesma tabela entre si. Por exemplo, suponha que queiramos saber todos os filmes que possuem o mesmo nome, mas foram lançados em anos diferentes. Para isso, temos que pegar um registro da tabela `tbTitulo` e comparar com os registros da mesma tabela, como mostra o exemplo abaixo:

```
SELECT T1.nome_titulo, T1.ano
FROM tbTitulo T1
INNER JOIN tbTitulo T2 ON T1.nome_titulo = T2.nome_titulo
WHERE T2.ano <> T1.ano;
```

## Comando LIKE

O comando **LIKE** é utilizado quando queremos comparar *strings* em uma consulta. No entanto, diferentemente dos operadores relacionais de igual (=) e diferente (<>) que comparam a *string* exata, o comando **LIKE** permite que utilizemos operadores que comparam se parte de uma *string* está contida em algum registro de uma tabela.

Os operadores utilizados são:

- Porcentagem (%): para comparar parte de uma *string*.
- Sublinhado (\_): para comparar um caractere.

Selecione todos os títulos que comecem com a Letra 'M' e o ano em que foram lançados:

```
SELECT T.nome_titulo, T.ano
FROM tbTitulo T
WHERE T.nome_titulo LIKE 'M%';
```

Selecione todos os títulos que contenham a substring 'ama':

```
SELECT T.nome_titulo
FROM tbTitulo T
WHERE T.nome_titulo LIKE '%ama%';
```

Selecione todos os títulos que tenham apenas 5 caracteres:

```
SELECT T.nome_titulo
FROM tbTitulo T
WHERE T.nome_titulo LIKE '_____';
```

Selecione todos os títulos que tenham pelo menos 5 caracteres e termine com 'a':

```
SELECT T.nome_titulo
FROM tbTitulo T
WHERE T.nome_titulo LIKE '_____a';
```

Podemos utilizar o caractere `\` se quisermos comparar caracteres especiais. Para pesquisar diferenças em *strings* pode-se usar o comando **NOT LIKE**. Por exemplo:

Selecione todos os títulos que não tenham a substring 'ama%':

```
SELECT T.nome_titulo
FROM tbTitulo T
WHERE T.nome_titulo NOT LIKE '%ama\%%';
```

## Comando ORDER BY

O comando **ORDER BY**, como o próprio nome diz, é utilizado para ordenar o resultado de uma consulta. Ele não altera a ordem dos dados na tabela física, somente o resultado da consulta aparece ordenado.

Por padrão, quando você usa o **ORDER BY**, a maioria dos SGBD ordenam em ordem crescente (do menor para o maior). No entanto, é possível determinar a ordem utilizando explicitamente:

- **ASC** para ordem crescente (do menor para o maior);
- **DESC** para ordem decrescente (do maior para o menor);

Selecione o nome dos títulos e o ano em ordem crescente de nome e decrescente de ano.

```
SELECT T.nome_titulo, T.ano
FROM tbTitulo T
ORDER BY T.nome_titulo ASC, T.ano DESC;
```

No exemplo acima, o resultado foi ordenado em ordem crescente pelo nome do título e em caso de nomes iguais de títulos ordena-se pelo ano em ordem decrescente.

É importante destacar que quando você usa o **ORDER BY**, o SGBD terá um trabalho adicional para fazer a ordenação do resultado, o que pode comprometer o desempenho da consulta. Por isso, só use o comando **ORDER BY** quando ele for realmente necessário.

## Funções Agregadas

Todo SGDB oferece um conjunto de funções para tratamento de data, hora, caracteres, etc. Outro uso dessas funções é o cálculo estatístico. Para isso, existem 5 funções comuns a qualquer SGDB, chamadas funções agregadas, que são bastante utilizadas em consultas SQL.

A seguir, vamos descobrir quais são elas, como funcionam e onde podem ser aplicadas.

## Função AVG

Essa função calcula a média aritmética, dado um conjunto de valores numéricos. Por exemplo, podemos calcular a média de preços de filmes das classes.

```
SELECT AVG(tbClasse.preco_classe) AS Media
FROM tbClasse;
```

## Função SUM

Assim como a função **AVG**, a função **SUM** também é uma função numérica. Essa função faz o somatório de um conjunto de valores numéricos. Por exemplo, podemos somar todos os anos da tabela tbTitulo, como mostrado abaixo.

```
SELECT SUM(tbTitulo.ano) AS 'Soma dos Anos'
FROM tbTitulo;
```

## Função MIN e Função MAX

As funções **MIN** e **MAX** são funções que retornam, respectivamente, o valor mínimo e o valor máximo de um campo. Essas funções funcionam com conjuntos de dados numéricos e não numéricos.

Selecione a data de empréstimo mais antiga.

```
SELECT MIN (tbEmprestimoDevolucao.data_emprestimo) AS 'Data mais Antiga'
FROM tbEmprestimoDevolucao;
```

Selecione a data de cadastro do último cliente.

```
SELECT MAX(tbCliente.data_cadastro) AS 'Data do último cadastro'
FROM tbCliente;
```

## Função COUNT

A função **COUNT** tem por objetivo contar o número de ocorrências de um atributo na base de dados. Esta função pode ser aplicada a qualquer tipo de atributo (numérico e não numérico). Por exemplo, suponha que queiramos contar o número de filmes que o cliente Pedro emprestou em 2009.

```
SELECT COUNT(ED.data_emprestimo) AS 'Numero de Locações', C.nome_cli
FROM tbEmprestimoDevolucao ED
INNER JOIN tbCliente C ON ED.codigo_cli = C.codigo_cli
WHERE ED.data_emprestimo BETWEEN '2009-01-01' AND '2009-12-31';
```

Ou ainda podemos querer descobrir quantos títulos existem da categoria “Terror”.

```
SELECT COUNT(T.codigo_titulo) AS 'Quantidade de Títulos'
FROM tbTitulo T
INNER JOIN tbCategoria C ON T.codigo_categoria = C.codigo_categoria
WHERE C.nome_categoria LIKE 'Terror';
```

Para contar o número total de registros em uma tabela, utilizamos a função **COUNT(\*)**. Por exemplo, vamos contar quantas linhas existem na tabela tbFilme.

```
SELECT COUNT(*) AS 'Quantidade de Linhas em tbFilme'
FROM tbFilme;
```

O comando **DISTINCT** pode ser utilizado para eliminar repetições em qualquer função. No entanto, a SQL não permite o uso do **DISTINCT** com **COUNT(\*)**, uma vez que o **DISTINCT** serve para eliminar redundâncias e o **COUNT(\*)** somente conta o número de linhas.

Todas as funções agregadas, exceto o **COUNT(\*)**, ignoram os valores nulos dos seus conjuntos de valores de entrada.

## Comando GROUP BY

O comando **GROUP BY** é usado para formar grupos e categorizar os resultados por meio desses grupos. Este comando sempre estará associado a uma função agregada uma vez que seu objetivo é o de aplicar uma função agregada, a um grupo de registros. Por exemplo, suponha que você queira descobrir quantos títulos existem por categoria. A consulta ficaria como mostrado a seguir.

```
SELECT COUNT(T.codigo_titulo) AS 'Quantidade de Títulos', C.nome_categoria
FROM tbTitulo T
INNER JOIN tbCategoria C ON T.codigo_categoria = C.codigo_categoria
GROUP BY C.nome_categoria;
```

A consulta anterior agrupa a quantidade de títulos pelo nome da categoria. Assim, registros que possuem os mesmos valores de categoria são contados e colocados em um grupo.

Se quiséssemos saber a quantidade total de títulos, não seria necessário utilizar a cláusula **GROUP BY**. A consulta ficaria assim:

```
SELECT COUNT(T.codigo_titulo) AS 'Quantidade de Títulos'
FROM tbTitulo T
INNER JOIN tbCategoria C ON T.codigo_categoria = C.codigo_categoria;
```

Como dito anteriormente, pode-se utilizar o **DISTINCT** com uma função agregada. Por exemplo, vamos descobrir a quantidade de títulos que cada cliente já tomou emprestado.

```
SELECT C.nome_cli, COUNT(DISTINCT F.codigo_titulo) AS 'Quantidade de Títulos'
FROM tbCliente C
INNER JOIN tbEmprestimoDevolucao ED ON C.codigo_cli = ED.codigo_cli
INNER JOIN tbFilme F ON F.codigo_filme = ED.codigo_filme
GROUP BY C.nome_cli
ORDER BY COUNT(DISTINCT F.codigo_titulo);
```

O **DISTINCT** foi utilizado na consulta para impedir que no caso do cliente que pegou um mesmo título em duas ou três diferentes oportunidades, esse título seja contado mais de uma vez. Isso porque não queremos saber a quantidade de empréstimos de um cliente e sim quantos títulos diferentes ele já emprestou.

Note que foi feito o **INNER JOIN** de 3 tabelas embora as informações que eram necessárias para a consulta estivesse nas tabelas tbFilme e tbCliente. A tabela tbEmprestimoDevolucao teve que ser utilizada na consulta uma vez que é ela quem relaciona as duas outras tabelas (através das **FK**).

Também foi utilizado um **ORDER BY** a fim de que o resultado seja ordenado do cliente que emprestou mais títulos para o que emprestou menos títulos.

## Comando HAVING

Vimos que o **WHERE** é o comando que permite definir condições para uma consulta. No entanto, é muito comum precisarmos aplicar uma condição aos grupos formados pelo **GROUP BY** em vez de aplicar a condição a todos os registros da tabela. Para aplicar a condição aos grupos, usamos o comando **HAVING**.



Sendo assim, o comando **HAVING** sempre irá aparecer junto a um comando **GROUP BY**. Por exemplo, suponha que você queria descobrir o nome dos clientes que tiveram mais de 100 empréstimos (agora pode ser o mesmo título várias vezes).

```
SELECT C.nome_cli, COUNT(ED.data_emprestimo) AS 'Quantidade de Empréstimos'
FROM tbCliente C
INNER JOIN tbEmprestimoDevolucao ED ON C.codigo_cli = ED.codigo_cli
GROUP BY C.nome_cli
HAVING COUNT(ED.data_emprestimo) > 100;
```

Primeiramente, deve-se contar o número de empréstimos por cliente, para, em seguida, aplicar a condição do **HAVING** e retornar apenas aqueles clientes que tiveram mais de 100 locações.

Se uma cláusula **WHERE** e **HAVING** aparecem na mesma consulta, o predicado que aparece em **WHERE** é aplicado primeiro, depois são criados os grupos para, em seguida, aplicar-se o **HAVING**. Por exemplo, imagine que agora você queira descobrir o nome dos clientes que tiveram 30 ou mais empréstimos em janeiro de 2010.

```
SELECT C.nome_cli, COUNT(ED.data_emprestimo) AS 'Quantidade de Empréstimos'
FROM tbCliente C
INNER JOIN tbEmprestimoDevolucao ED ON C.codigo_cli = ED.codigo_cli
WHERE ED.data_emprestimo BETWEEN '2010-01-01' AND '2010-01-31'
GROUP BY C.nome_cli
HAVING COUNT(ED.data_emprestimo) >= 30;
```

Nesse caso, primeiro foram selecionados todos os empréstimos em janeiro de 2010, em seguida foram contados o número de empréstimos somente para as pessoas que emprestaram filmes em janeiro de 2010 e, finalmente, foram selecionados apenas aqueles registros em que o cliente teve 30 ou mais empréstimos.

## Valores Nulos

Como visto anteriormente, o valor nulo indica a ausência de informação. Para testar se um atributo possui valores nulos, ou não, é necessário utilizar o operador **IS** junto à palavra **NULL**. Por exemplo, se quisermos saber quais clientes ainda estão em atraso, podemos escrever:

```
SELECT DISTINCT C.nome_cli 'Clientes com Filmes em Atraso'
FROM tbCliente C
INNER JOIN tbEmprestimoDevolucao ED ON C.codigo_cli = ED.codigo_cli
WHERE ED.data_devolucao_prevista < CURDATE()
AND ED.data_devolucao_efetiva IS NULL;
```

A função **CURDATE()** que aparece na consulta acima, retorna a data atual do servidor. Essa função está sendo utilizada para verificar se a data atual é maior do que a data cadastrada no atributo referente à devolução prevista do filme. Caso isso ocorra verifica-se se o campo referente à devolução efetiva do filme está preenchido e se não estiver caracteriza que o cliente não entregou o filme e portanto está em atraso.

Não podemos utilizar os operadores relacionais **=** e **<>** para testar a ausência de um valor. Isso ocorre porque **NULL** não é valor (indica a ausência dele) e só podemos utilizar os operadores relacionais para testar valores.

O predicado **IS NOT NULL** pode ser utilizado para verificar se existem valores para um atributo.

Numa expressão aritmética, (**+**, **-**, **\*** e **/**), se qualquer um dos valores de entrada for nulo, o resultado da expressão também o será.

## Tabelas Derivadas

Você já sabe que no **FROM** sempre definimos as tabelas que iremos utilizar numa consulta e que o resultado de uma consulta é sempre uma tabela. Sendo assim, quando você escreve uma consulta dentro do **FROM** e essa consulta é executada, gera-se uma tabela que poderá ser consultada normalmente. A essa consulta que escrevemos dentro do **FROM** é que damos o nome de tabela derivada.

O SGBD primeiro executa o que está no **FROM**, gera a tabela em memória e, em seguida, executa o **WHERE** da consulta principal. Por exemplo, vamos reescrever usando uma tabela derivada, a consulta apresentada no primeiro *box* da página 105, que retorna o nome dos clientes que tiveram mais de 100 empréstimos.

```
SELECT tbNovaTabela.nome_cli AS 'NOME DO CLIENTE', tbNovaTabela.
quantidade
FROM (SELECT C.nome_cli, COUNT(ED.data_emprestimo) AS quantidade
FROM tbCliente C
INNER JOIN tbEmprestimoDevolucao ED ON C.codigo_cli = ED.codigo_cli
GROUP BY C.nome_cli) AS tbNovaTabela
WHERE tbNovaTabela.quantidade > 100;
```

Observe que tivemos que criar um apelido (tbNovaTabela) para a tabela gerada no **FROM**, para que pudéssemos referenciar essa tabela tanto no **SELECT** mais externo, quanto no **WHERE**.



## Atividades

Para resolver os exercícios, utilize a descrição do modelo relacional abaixo e os comandos SQL aprendidos. Se houver a necessidade de utilizar mais de uma tabela em uma consulta, utilize o INNER JOIN, em vez do produto cartesiano.

```
tbNovela(codigo_novela:inteiro, nome_novela:caracter(30),
data_primeiro_capitulo:date, data_ultimo_capitulo:date,
horario_exibicao:time)

tbNovelaPersonagem(codigo_novela:inteiro,
codigo_personagem:inteiro)
    codigo_novela referencia tbNovela
    codigo_personagem referencia tbPersonagem

tbPersonagem(codigo_personagem:inteiro, nome_
personagem:caracter(50), idade_personagem:inteiro,
situacao_financeira_personagem:caracter(20), codigo_ator:inteiro)
    codigo_ator referencia tbAtor

tbAtor (codigo_ator:inteiro, nome_ator:caracter(20),
idade:inteiro, cidade_ator:caracter(20), salario_ator:real,
sexo_ator:caracter(1))

tbCapitulos(codigo_capitulo:inteiro, nome_capitulo:caracter(50),
data_exibicao_capitulo:date, codigo_novela:inteiro)
    codigo_novela referencia tbNovela
```

- 1) Escreva os comandos para criar todas as tabelas do modelo relacional acima. As chaves primárias e estrangeiras, bem como qualquer restrição para um atributo deverão ser definidas diretamente no momento de criação das tabelas. Não se esqueça de que a ordem de criação das tabelas é importante.
- 2) Insira 3 registros para cada tabela criada.
- 3) Encontre a data de exibição do último capítulo para a novela “Mistérios de uma Vida”.
- 4) Encontre todas as novelas que tenham o valor do horário de exibição vazio (NULL).

- 5) Encontre o nome de todos os atores que morem em cidades que comecem com a letra "M".
- 6) Encontre a quantidade de novelas que tenham como parte do nome a palavra "vida".
- 7) Encontre a quantidade de novelas que o ator "Fernando Souza" participou.
- 8) Selecione todos os campos da tabela tbPersonagem ordenados por nome em ordem crescente.
- 9) Selecione todos os campos da tabela tbPersonagem ordenados por idade em ordem decrescente.
- 10) Selecione quantos atores existem cadastrados.
- 11) Selecione quantas novelas existem cadastradas.
- 12) Selecione quantos capítulos existem por novela. Retorne o nome da novela e a quantidade de capítulos para a novela.
- 13) Selecione quantos atores são do sexo feminino.
- 14) Faça uma consulta que retorne a idade média dos personagens.
- 15) Selecione quantos personagens têm menos de 15 anos.
- 16) Selecione o nome dos atores que têm a mesma idade do seu personagem.
- 17) Encontre qual o maior salário.
- 18) Encontre qual o menor salário.
- 19) Faça o somatório de todos os salários.
- 20) Selecione a quantidade de personagens representados para cada ator.
- 21) Encontre o nome de todas as novelas quem tem mais de 40 capítulos.
- 22) Encontre o nome de todos os atores que atuaram como personagens ricos (situação financeira) em mais de 15 novelas.

# SQL – DML: Subconsultas e Tipos de Junção

Neste capítulo, vamos aprender alguns outros recursos para escrever uma consulta SQL.

## Operador IN e NOT IN

O operador **IN** testa se dados de um conjunto são membros de outro conjunto de dados. Por exemplo, suponha que você queira saber o nome de todos os títulos das categorias: drama, terror, suspense e comédia.

```
SELECT T.nome_titulo AS Titulo, C.nome_categoria AS Categoria
FROM tbTitulo T
INNER JOIN tbcategoria C ON C.codigo_categoria = T.codigo_categoria
WHERE C.nome_categoria IN ('Drama','Terror','Suspense','Comedia');
```

Neste exemplo, o nome da categoria da tabela tbCategoria é comparado com um conjunto de dados que contém valores de categorias. Se o nome da categoria estiver contido no conjunto de dados, então a consulta retorna o título do filme e o nome da categoria.

Para utilizar o operador **IN**, não é necessário que o conjunto de dados tenha apenas valores fixos. Pode-se executar uma subconsulta e ela gerar o conjunto de dados que será comparado. Uma subconsulta é uma consulta do tipo **SELECT-FROM-WHERE** aninhada dentro de um **WHERE** mais externo. Por exemplo, suponha que queiramos descobrir o nome dos títulos que foram entregues pelo distribuidor “ABC Distribuidora”.

```

SELECT T.nome_titulo
FROM tbTitulo T
WHERE T.codigo_titulo IN (SELECT F.codigo_titulo
                           FROM tbFilme F
                           WHERE F.nome_distribuidor = 'ABC Distribuidora'
                           );

```

Nesse caso, a consulta mais interna (subconsulta), que descobre os códigos dos títulos de filmes do distribuidor “ABC Distribuidora”, é executada antes e gera como resultado um conjunto de dados que serão comparados com os dados da consulta mais externa, aquela que retorna o nome dos títulos.

Observe que o atributo utilizado na comparação é o `codigo_titulo` que aparece nas duas consultas (no **WHERE** da consulta mais externa e no **SELECT** da consulta mais interna). Não se pode, por exemplo, comparar `nome_titulo` com `codigo_titulo` porque não seria encontrada nenhuma igualdade entre eles, uma vez que um armazena números e o outro armazena caracteres.

O conectivo **NOT IN** testa a ausência de membros de um conjunto. Por exemplo, encontre o nome de todos os títulos lançados entre os anos de 1995 e 2009 que **nunca** foram emprestados.

```

SELECT T.nome_titulo
FROM tbTitulo T
WHERE T.codigo_titulo IN
    (SELECT F.codigo_titulo
     FROM tbFilme F
     INNER JOIN tbEmprestimoDevolucao ED ON F.codigo_filme = ED.codigo_filme
     )
AND T.ano BETWEEN '1995' AND '2009';

```

O atributo utilizado na comparação dessa consulta é o atributo `codigo_titulo`. Observe que podemos continuar a consulta normalmente acrescentando outras condições, agrupamentos, ordenação, etc. (desde que necessário).

Uma pergunta muito comum quando falamos sobre o operador **IN** é se não podemos usar o operador relacional de igualdade (**=**) em uma subconsulta em vez de utilizarmos o **IN**. A resposta para essa pergunta é “depende”.

A igualdade assim como todos os operadores relacionais (<, <=, >, >=, <>, =) sempre comparam um único valor. Isso significa que se você tem certeza de que a sua subconsulta vai retornar apenas um valor, você pode usar o sinal de igual no lugar do **IN**. No entanto, se você não tem essa certeza ou sabe que a subconsulta pode retornar um conjunto de valores não deve usar o sinal de igual porque dará erro de sintaxe.

Por exemplo, se quisermos saber o títulos dos filmes que possuem o preço igual a média de preços de todos os filmes, podemos usar o sinal de igual porque temos certeza de que só será retornado um valor de média na subconsulta. O exemplo abaixo apresenta essa situação.

```
SELECT T.nome_titulo, C.preco_classe
FROM tbTitulo T
INNER JOIN tbClasse C ON T.codigo_classe = C.codigo_classe
WHERE C.preco_classe = (SELECT AVG (C.preco_classe)
                        FROM tbClasse C
                        );
```

## Operador EXISTS

O operador **EXISTS** é utilizado para verificar se o resultado de uma subconsulta possui algum registro, ou seja, ele retorna **verdadeiro** se a subconsulta encontrar algum registro que a satisfaça.

Por exemplo, suponha que você queira descobrir quais clientes possuem algum empréstimo e a data do seu cadastro. Para isso, pode-se fazer assim:

```
SELECT C.nome_cli, C.data_cadastro
FROM tbCliente C
WHERE EXISTS (SELECT ED.codigo_cli
              FROM tbEmprestimoDevolucao ED
              WHERE C.codigo_cli = ED.codigo_cli
              );
```

Essa consulta irá retornar todos os registros da tabela tbCliente em que há pelo menos um registro na tabela tbEmprestimoDevolucao com o mesmo código do cliente (codigo\_cli).

Neste exemplo, é executado primeiro a subconsulta que verifica quais clientes tem um empréstimo. Caso a subconsulta retorne alguma linha, ou seja, se o resultado é verdadeiro, será retornado ao usuário o nome do cliente e a data de cadastro do cliente que aparece na subconsulta.

Pode-se reescrever essa consulta, usando o operador **IN**:

```
SELECT C.nome_cli, C.data_cadastro
FROM tbCliente C
WHERE codigo_cli IN (SELECT ED.codigo_cli
                     FROM tbEmprestimoDevolucao ED
                     );
```

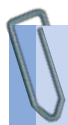
A principal diferença entre os operadores **EXISTS** e o **IN** é que o operador **IN** compara valores e o **EXISTS** verifica a existência de resultados em uma consulta. Quanto ao desempenho, normalmente é igual para os dois operadores. No entanto, se a maioria das condições estiver na subconsulta o comando **IN** poderá ser mais eficiente.

Ainda podemos reescrever a mesma consulta utilizando **INNER JOIN**.

```
SELECT DISTINCT C.nome_cli
FROM tbCliente C
INNER JOIN tbEmprestimoDevolucao ED ON
ED.codigo_cli = C.codigo_cli;
```

Nesse caso, tivemos que usar o operador **DISTINCT** para eliminar a repetição porque, uma vez que estamos fazendo a junção, os nomes dos clientes que tiverem mais de um empréstimo vão aparecer em várias linhas, portanto vão se repetir.

Normalmente, o **INNER JOIN** executa mais rapidamente que uma subconsulta. No entanto, existem outros fatores que determinam o desempenho da consulta. A utilização do **DISTINCT**, por exemplo, produz um trabalho adicional na execução da consulta e diminui o desempenho dela. Nesse caso, pode-se utilizar um dos operadores (**IN** ou **EXISTS**) para eliminar o **DISTINCT** e obter melhor desempenho.



Quando o problema é desempenho, a dica é sempre testar cada um dos métodos possíveis porque o resultado para cada um deles pode variar dependendo da consulta criada.

O operador **NOT EXISTS** verifica se o resultado de uma subconsulta é **falso**, ou seja, ele vai retornar todos os registros da tabela mais externa onde não há registros na tabela da subconsulta. Por exemplo, suponha que agora queremos encontrar todos os clientes que não possuem empréstimo. Para isso, podemos fazer:



```

SELECT C.nome_cli, C.data_cadastro
FROM tbCliente C
WHERE NOT EXISTS (SELECT ED.codigo_cli
                  FROM tbEmprestimoDevolucao ED
                  WHERE C.codigo_cli = ED.codigo_cli
                  );

```

Neste exemplo, a consulta irá retornar todos os registros da tabela tbCliente onde não há registros na tabela tbEmprestimoDevolucao.

Podemos utilizar os operadores **IN**, **NOT IN**, **EXISTS** e **NOT EXISTS** em comandos como **UPDATE** e **DELETE**, como mostram os exemplos abaixo.

**Exemplo 1:** Atualize a base de dados para que a data da devolução prevista para as locações feitas em fevereiro de 2010 receba dois dias a mais.

```

UPDATE tbEmprestimoDevolucao ED
SET ED.data_devolucao_prevista = ED.data_devolucao_prevista + 2
WHERE ED.data_devolucao_prevista BETWEEN '2010-02-01' AND '2010-02-28'
      AND ED.codigo_filme IN
      (SELECT F.codigo_filme
       FROM tbFilme F
       INNER JOIN tbTitulo T ON F.codigo_titulo = T.codigo_titulo
       INNER JOIN tbClasse C ON T.codigo_classe = C.codigo_classe
       WHERE C.nome_classe = 'Lancamento'
      );

```

**Exemplo 2:** Exclua todos os clientes que não emprestaram nenhum filme em 2008, 2009 e 2010.

```

DELETE FROM tbCliente
WHERE codigo_cli NOT IN
      (SELECT ED.codigo_cli
       FROM tbEmprestimoDevolucao ED
       WHERE ED.data_emprestimo BETWEEN '2008-01-01' AND '2010-12-31'
      );

```

# Operadores ALL e SOME

Os operadores relacionais só comparam um valor e não um conjunto de valores. No entanto, podemos usar junto com os operadores relacionais os operadores **ALL** e **SOME** para compararmos conjuntos de dados.

O operador **ALL** é o equivalente ao operador **ANY** em alguns SGBD. O significado do **ALL** junto com os operadores relacionais é:

Operador + ALL	Significado
=ALL	Igual a todos
<>ALL	Diferente de todos
>ALL	Maior que todos
>=ALL	Maior ou igual a todos
<ALL	Menor que todos
<=ALL	Menor ou igual a todos

Por exemplo, suponha que queiramos descobrir quem foi o cliente que pediu emprestado o maior número de filmes durante o ano de 2009.

```
SELECT C.codigo_cli, C.nome_cli, COUNT(ED.data_emprestimo) AS
'Quantidade de filmes locados'
FROM tbCliente C
INNER JOIN tbEmprestimoDevolucao ED ON C.codigo_cli = ED.codigo_cli
WHERE ED.data_emprestimo BETWEEN '2009-01-01' AND '2009-12-31'
GROUP BY C.codigo_cli
HAVING COUNT(ED.data_emprestimo) >=ALL
    (SELECT COUNT(ED.data_emprestimo)
     FROM tbEmprestimoDevolucao ED
     WHERE ED.data_emprestimo BETWEEN '2009-01-01' AND '2009-12-31'
     GROUP BY ED.codigo_cli
    );
```

Neste exemplo, a subconsulta retorna um conjunto de dados contendo a quantidade de filmes locados por cliente no ano de 2009. A consulta mais externa também agrupa a quantidade de empréstimos por cliente, mas além disso, ela compara se essa quantidade é maior ou igual à obtida na subconsulta.

Essa mesma consulta poderia ser reescrita usando o comando **LIMIT** com o **ORDER BY**.

```
SELECT C.codigo_cli, C.nome_cli, COUNT(*) AS 'Quantidade de filmes
locados'
FROM tbCliente C
INNER JOIN tbEmprestimoDevolucao ED ON C.codigo_cli = ED.codigo_cli
GROUP BY C.codigo_cli
ORDER BY COUNT(*) DESC
LIMIT 1;
```

O comando **LIMIT** limita o número de linhas que será retornada pela consulta. Como ordenamos em ordem decrescente pela quantidade de empréstimos, o primeiro valor do conjunto de resultado é o que tem o maior valor e, portanto, é o que nos interessa.

No primeiro exemplo, se existem dois ou mais clientes com a mesma quantidade máxima de empréstimos, a consulta irá retornar todos os que têm a quantidade máxima. No segundo exemplo, teríamos que ficar testando o **LIMIT** até descobrir quantas linhas devem ser retornadas. Por isso, o segundo exemplo só deve ser utilizando quando você tiver certeza de quantas linhas serão retornadas ou se você quiser, por exemplo, apenas a primeira ocorrência.

O operador **SOME** e seus significados são apresentados abaixo.

Operador + SOME	Significado
=SOME	Igual a pelo menos um
<>SOME	Diferente de pelo menos um
>SOME	Maior que pelo menos um
>=SOME	Maior ou igual a pelo menos um
<SOME	Menor que pelo menos um
<=SOME	Menor ou igual a pelo menos um

Por exemplo, suponha que queiramos descobrir os nomes dos cliente que tiveram pelo menos um empréstimo em 2009.

```
SELECT C.codigo_cli, C.nome_cli, COUNT(*) AS 'Quantidade de filmes
locados'
FROM tbCliente C
INNER JOIN tbEmprestimoDevolucao ED ON C.codigo_cli = ED.codigo_cli
WHERE ED.data_emprestimo BETWEEN '2009-01-01' AND '2009-12-31'
GROUP BY C.codigo_cli
HAVING COUNT(*) =SOME (SELECT COUNT(*)
                        FROM tbEmprestimoDevolucao ED
                        WHERE ED.data_emprestimo BETWEEN '2009-01-01'
                        AND '2009-12-31'
GROUP BY ED.codigo_cli
);
```

## Tipos de Junção

O **INNER JOIN** faz a junção de duas tabelas obedecendo a uma condição de junção (normalmente chave primária = chave estrangeira). Isso significa que serão retornadas todas as linhas de uma tabela que se relacionam com as linhas de outras tabelas.

No entanto, além do **INNER JOIN** que é uma junção interna, a SQL oferece também as junções externas.

Uma junção externa (**OUTER JOIN**) é uma junção que retorna as linhas de uma tabela, mesmo que essas linhas não possuam registros equivalentes em outra tabela, ou seja, mesmo que as linhas retornadas não satisfaçam a condição da junção.

Existem 3 tipos de junção externa: **LEFT OUTER JOIN** (junção externa à esquerda), **RIGHT OUTER JOIN** (junção externa à direita), e **FULL OUTER JOIN** (junção externa total).

A junção externa **LEFT OUTER JOIN** retorna todas as linhas que satisfazem a condição da junção mais as linhas da tabela que ficam à esquerda da consulta, mesmo que não satisfaçam a condição.

Para que possamos entender melhor a junção externa, imagine que tenhamos os seguintes dados nas tabelas tbCliente e tbEmprestimoDevolucao.

codigo_cli	nome_cli	CPF_cli	data_cadastro	cidade	UF
1	Anna	12345678911	2006-03-12	Curitiba	PR
2	Pedro	12345678912	1995-04-17	Curitiba	PR
3	Janaína	12345678913	2010-02-03	Curitiba	PR
4	Elaine	12345678914	2009-05-08	Curitiba	PR
5	Gustavo	12345678915	2006-07-28	Curitiba	PR

tbCliente

codigo_cli	codigo_filme	data_emprestimo	data_devolucao_prevista	data_devolucao_efetiva	Multa
1	1	2006-03-15	2006-03-17	2006-03-17	0,00
2	2	1996-03-23	1996-03-27	1996-03-28	3,00
3	1	2010-02-03	2010-02-07		
3	5	2009-05-08	2009-05-11	2009-05-11	0,00

tbEmprestimoDevolucao

Agora vamos fazer uma consulta, usando o **LEFT JOIN**, que retorne o nome de todos os clientes, mesmo aqueles que não possuem empréstimo.

```
SELECT C.nome_cli, ED.data_emprestimo
FROM tbCliente C
LEFT OUTER JOIN tbEmprestimoDevolucao ED ON C.codigo_cli = ED.codigo_cli;
```

A tabela resultante para a consulta acima será:

nome_cli	data_emprestimo
Anna	2006-03-15
Pedro	1996-03-23
Janaína	2010-02-03
Janaína	2009-05-08
Elaine	NULL
Gustavo	NULL

No resultado da consulta anterior, estão sendo mostrados todos os clientes (tabela esquerda), mesmo aqueles que não satisfazem à condição da junção. Quando um cliente não possui empréstimos, as colunas da tabela tbEmprestimoDevolucao irão mostrar o valor **NULL**.

A junção externa, à direita e à esquerda, podem ser usadas para substituir o **NOT EXISTS** e o **NOT IN**. Mas, nesses casos, deve-se lembrar de incluir uma condição que checa a condição **NULL**. Por exemplo, vamos refazer a consulta retornando agora somente os clientes que não possuem um empréstimo.

```
SELECT C.nome_cli
FROM tbCliente C LEFT OUTER JOIN tbEmprestimoDevolucao ED ON
C.codigo_cli = ED.codigo_cli
WHERE ED.data_emprestimo IS NULL;
```

A junção **RIGHT OUTER JOIN** faz a mesma coisa que a **LEFT OUTER JOIN**, mas agora para a tabela que fica à direita, ou seja, retorna todas as linhas relacionadas mais às linhas à direita que não possuem correspondente na tabela da esquerda. Por exemplo, vamos fazer a mesma consulta, mas agora alterando a ordem das tabelas e o tipo da junção. Essa nova consulta irá retornar os mesmos dados da consulta anterior.

```
SELECT C.nome_cli
FROM tbEmprestimoDevolucao ED
RIGHT OUTER JOIN tbCliente C ON C.codigo_cli = ED.codigo_cli
WHERE ED.data_emprestimo IS NULL;
```

Se for utilizada a junção **FULL OUTER JOIN**, todas as linhas de ambas as tabelas são incluídas, mesmo as que não estão relacionadas entre si. Por exemplo, poderíamos querer descobrir qual título não tem categoria e qual categoria não tem nenhum título. Para isso, faríamos:

```
SELECT C.nome_categoria, T.nome_titulo
FROM tbCategoria C
FULL OUTER JOIN tbTitulo T ON C.codigo_categoria = T.codigo_categoria
WHERE C.nome_categoria IS NULL
      AND T.nome_titulo IS NULL;
```

Alguns SGBD, como o caso do MySQL, não possuem a junção **FULL JOIN**. Nesse caso, podemos utilizar o comando **UNION** para simular o **FULL JOIN**, como segue.

```
(SELECT C.nome_categoria, T.nome_titulo
FROM tbCategoria C LEFT OUTER JOIN tbTitulo T ON C.codigo_categoria =
T.codigo_categoria
WHERE T.nome_titulo IS NULL)
UNION
(SELECT C.nome_categoria, T.nome_titulo
FROM tbCategoria C RIGHT OUTER JOIN tbTitulo T ON C.codigo_categoria
= T.codigo_categoria
WHERE C.nome_categoria IS NULL)
```

O operador **UNION**, na SQL, funciona da mesma forma que na teoria dos conjuntos na matemática. Ou seja, dado dois conjuntos, A e B, a união desses conjuntos retorna todos os dados de A, todos os dados de B, e mais o que aparece em A e B ao mesmo tempo, como mostra a figura 10.1.

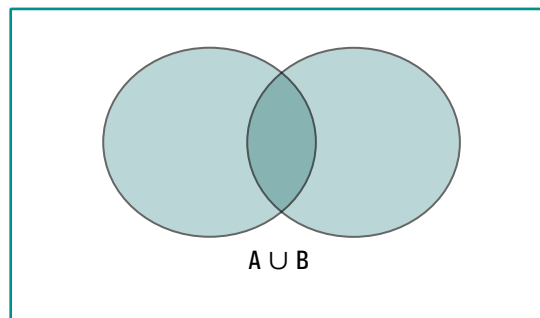


Figura 10.1 – União de A com B

Em SQL, o operador **UNION** combina os resultados de duas ou mais consultas em um único resultado, que inclui todas as linhas de todas as consultas da união. A única observação que se faz na utilização do **UNION** é que o número e a ordem dos atributos que aparecem no **SELECT** das consultas devem ser idênticos.

## Atividades

- 1) Explique, com base nas leituras e debates da sala de aula, o que é o **LEFT OUTER JOIN** e o **RIGHT OUTER JOIN**.
- 2) Dê exemplos de situações (diferentes das apresentadas no capítulo 10) onde você usaria **LEFT OUTER JOIN** e o **RIGHT OUTER JOIN**.

**Observação:** Para os próximos exercícios, vamos utilizar o mesmo modelo relacional dos exercícios do capítulo 9. Este modelo é apresentado abaixo.

Para resolver as questões, utilize comandos **SQL**. Se houver a necessidade de utilizar mais de uma tabela em uma consulta, utilize o **INNER JOIN** em vez do produto cartesiano.

```
tbNovela(codigo_novela:inteiro, nome_novela:caracter(30),
data_primeiro_capitulo:date, data_ultimo_capitulo:date,
horario_exibicao:time)

tbNovelaPersonagem(codigo_novela:inteiro, codigo_personagem:inteiro)
codigo_novela referencia tbNovela
codigo_personagem referencia tbPersonagem

tbPersonagem(codigo_personagem:inteiro, nome_personagem:caracter(50),
idade_personagem:inteiro,
situacao_financeira_personagem:caracter(20), codigo_ator:inteiro)
codigo_ator referencia tbAtor

tbAtor (codigo_ator:inteiro, nome_ator:caracter(20),
idade:inteiro, cidade_ator:caracter(20), salario_ator:real,
sexo_ator:caracter(1))

tbCapitulos(codigo_capitulo:inteiro, nome_capitulo:caracter(50),
data_exibicao_capitulo:date, codigo_novela:inteiro)
codigo_novela referencia tbNovela
```

- 3) Selecione o nome dos atores que não participam de nenhuma novela.
- 4) Selecione o nome e a idade dos atores que participam da novela “Ser Estranho”.
- 5) Selecione o nome de todos os atores que tiveram personagens com o nome “Anna”.
- 6) Selecione o nome dos atores que trabalharam nas mesmas novelas que a atriz “Joaquina Penteado”.
- 7) Selecione o nome dos atores que não trabalharam nas mesmas novelas que a atriz “Joaquina Penteado”.
- 8) Selecione o nome e a idade do personagem mais novo.
- 9) Selecione o nome e o salário do ator que recebe o menor salário.
- 10) Quais os nomes dos atores que nunca representaram personagens pobres nas novelas.
- 11) Selecione o nome dos atores que trabalharam em pelo menos uma novela das 18 horas.

## Referências Bibliográficas

CÂNDIDO, C. H. *brModelo 2.0*. Disponível em: <<http://sis4.com/brModelo/>>. Acesso em: 10 jan. 2010.

CODD, E. F. Relational model of data for large shared Data Banks. In: *Communications of the ACM*, v. 13, p. 377-387, 1970.

DATE, C. J. *Introdução a Sistemas de Banco de Dados*. Rio de Janeiro: Elsevier, 2003.

ELMASRI, R.; NAVATHE, S. B. *Sistema de Banco de Dados*. Ed. Pearson, 2005.

HEUSER, C. A. *Projeto de Banco de Dados*. 6. ed. Editora Bookman, 2009.

KRIEGL, A.; TRUKHNOV, B. M. *SQL bible*. Indianápolis: Wiley, 2003.

SILBERCHATZ, A. et al. *Sistema de Banco de Dados*. 3. ed. Makron Books, 1999.