



# Banco de Dados I

*Edson Marchetti da Silva*

Curso Técnico em Planejamento e  
Gestão em Tecnologia da Informação





# Banco de Dados I

*Edson Marchetti da Silva*



Belo Horizonte - MG  
2012

Presidência da República Federativa do Brasil  
Ministério da Educação  
Secretaria de Educação Profissional e Tecnológica

© Centro Federal de Educação Tecnológica de Minas Gerais  
Este Caderno foi elaborado em parceria entre o Centro Federal de Educação  
Tecnológica de Minas Gerais e a Universidade Federal de Santa Catarina para a  
Rede e-Tec Brasil.

**Equipe de Elaboração**

Centro Federal de Educação Tecnológica  
de Minas Gerais – CEFET-MG

**Coordenação do Curso**

Adelson de Paula Silva/CEFET-MG

**Professor-autor**

Edson Marchetti da Silva/CEFET-MG

**Comissão de Acompanhamento e Validação**

Universidade Federal de Santa Catarina – UFSC

**Coordenação Institucional**

Araci Hack Catapan/UFSC

**Coordenação do Projeto**

Silvia Modesto Nassar/UFSC

**Coordenação de Design Instrucional**

Beatriz Helena Dal Molin/UNIOESTE e UFSC

**Coordenação de Design Gráfico**

Juliana Tonietto/UFSC

**Design Instrucional**

Gustavo Pereira Mateus/UFSC

**Web Master**

Rafaela Lunardi Comarella/UFSC

**Web Design**

Beatriz Wilges/UFSC

Mônica Nassar Machuca/UFSC

**Diagramação**

Bárbara Zardo/UFSC

Luiz Fernando Tomé/UFSC

Marília Cerioli Hermoso/UFSC

Roberto Gava Colombo/UFSC

**Revisão**

Júlio César Ramos/UFSC

**Projeto Gráfico**

e-Tec/MEC

**Catalogação na fonte pela Biblioteca Universitária da  
Universidade Federal de Santa Catarina**

**S586b      Silva, Edson Marchetti da  
Banco de dados I / Edson Marchetti da Silva. - Belo Horizonte : CEFET- MG, 2012.  
138 p.. : il., tabs.**

**Inclui bibliografia.**

**Elaborado para o Curso Técnico em Planejamento e Gestão  
em Tecnologia da Informação da Rede e-Tec Brasil.  
ISBN: 978-85-99872-26-0**

**1.Tecnologia da informação. 2. Banco de dados. 3. Linguagem  
de programação (Computadores). I. Título.**

# Apresentação e-Tec Brasil

Prezado estudante,

Bem-vindo ao e-Tec Brasil!

Você faz parte de uma rede nacional pública de ensino, a Escola Técnica Aberta do Brasil, instituída pelo Decreto nº 6.301, de 12 de dezembro 2007, com o objetivo de democratizar o acesso ao ensino técnico público, na modalidade a distância. O programa é resultado de uma parceria entre o Ministério da Educação, por meio das Secretarias de Educação a Distância (SEED) e de Educação Profissional e Tecnológica (SETEC), as universidades e escolas técnicas estaduais e federais.

A educação a distância no nosso país, de dimensões continentais e grande diversidade regional e cultural, longe de distanciar, aproxima as pessoas ao garantir acesso à educação de qualidade, e promover o fortalecimento da formação de jovens moradores de regiões distantes, geograficamente ou economicamente, dos grandes centros.

O e-Tec Brasil leva os cursos técnicos a locais distantes das instituições de ensino e para a periferia das grandes cidades, incentivando os jovens a concluir o ensino médio. Os cursos são ofertados pelas instituições públicas de ensino e o atendimento ao estudante é realizado em escolas-polo integrantes das redes públicas municipais e estaduais.

O Ministério da Educação, as instituições públicas de ensino técnico, seus servidores técnicos e professores acreditam que uma educação profissional qualificada – integradora do ensino médio e educação técnica, – é capaz de promover o cidadão com capacidades para produzir, mas também com autonomia diante das diferentes dimensões da realidade: cultural, social, familiar, esportiva, política e ética.

Nós acreditamos em você!

Desejamos sucesso na sua formação profissional!

Ministério da Educação  
Janeiro de 2010

Nosso contato  
[etecbrasil@mec.gov.br](mailto:etecbrasil@mec.gov.br)



# Indicação de ícones

Os ícones são elementos gráficos utilizados para ampliar as formas de linguagem e facilitar a organização e a leitura hipertextual.



**Atenção:** indica pontos de maior relevância no texto.



**Saiba mais:** oferece novas informações que enriquecem o assunto ou “curiosidades” e notícias recentes relacionadas ao tema estudado.



**Glossário:** indica a definição de um termo, palavra ou expressão utilizada no texto.



**Mídias integradas:** sempre que se desejar que os estudantes desenvolvam atividades empregando diferentes mídias: vídeos, filmes, jornais, ambiente AVEA e outras.



**Atividades de aprendizagem:** apresenta atividades em diferentes níveis de aprendizagem para que o estudante possa realizá-las e conferir o seu domínio do tema estudado.



# Sumário

<b>Apresentação e-Tec Brasil</b> .....	<b>3</b>
<b>Indicação de ícones</b> .....	<b>5</b>
<b>Sumário</b> .....	<b>7</b>
<b>Palavra do professor-autor</b> .....	<b>9</b>
<b>Apresentação da disciplina</b> .....	<b>11</b>
<b>Projeto instrucional</b> .....	<b>13</b>
<b>Aula 1 – Conceitos de modelos de dados</b> .....	<b>15</b>
1.1 Introdução.....	15
1.2 Conceitos sobre modelagem de dados.....	16
1.3 Padronização.....	29
<b>Aula 2 – Diagrama de Entidade e Relacionamento</b> .....	<b>33</b>
2.1 Introdução.....	33
2.2 Modelo de Entidade e Relacionamento.....	33
2.3 Diagrama de Entidade e Relacionamento.....	34
<b>Aula 3 – Diagrama de Entidade e Relacionamento – casos práticos</b> .....	<b>49</b>
3.1 Primeiro problema .....	50
3.3 Terceiro problema .....	57
<b>Aula 4 – Estrutura física</b> .....	<b>63</b>
4.1 Estrutura física.....	63
4.2 Características do MySQL.....	71
4.3 Características das tabelas MySQL.....	73
4.4 Tipos de Dados MySQL.....	73
4.5 Controle de transação.....	77
4.6 Controle de concorrência.....	78

<b>Aula 5 – Structured Query Language – SQL.....</b>	<b>83</b>
5.1 Persistência de dados.....	83
5.2 Arquivos convencionais <i>versus</i> SGBDs.....	83
5.3 Linguagem de consulta.....	84
5.4 Comandos SQL.....	85
<b>Aula 6 – Visões e restrições.....</b>	<b>99</b>
6.1 Implementando restrições.....	99
6.2 Visões ( <i>View</i> ).....	102
6.3 Esquema ( <i>Schema</i> ).....	105
6.4 Restrição de unicidade.....	106
6.5 Uniões ( <i>Union</i> ).....	107
<b>Aula 7 – Estudo de caso – Fórmula I.....</b>	<b>109</b>
7.1 Problema proposto.....	109
7.2 Possíveis soluções para as entidades Piloto, Equipe e País.....	112
7.3 Possíveis soluções para as entidades Campeonato, Autodromo, Prova e Pontuacao.....	115
7.4 Solução final.....	117
<b>Aula 8 – Integridade referencial – estudo de caso – Fórmula I.....</b>	<b>129</b>
8.1 Integridade referencial.....	129
<b>Referências.....</b>	<b>137</b>
<b>Curriculum do professor-autor.....</b>	<b>138</b>

# Palavra do professor-autor

Prezado estudante!

Nesta disciplina você será apresentado aos conceitos básicos relacionados: à modelagem de dados e construção de Diagrama de Entidade e Relacionamento (DER); aos sistemas gerenciadores de banco de dados (SGBDs) com ênfase no uso da linguagem *Structured Query Language* (SQL). Os assuntos abordados nesta disciplina são de extrema importância para a formação do profissional técnico em Planejamento de Gestão da Tecnologia da Informação (PGTI). Entre eles estão as maneiras como os sistemas de informação armazenam e recuperam os seus dados. Esses conhecimentos são vitais para o desenvolvimento e manutenção dos sistemas, pois, a grande maioria das linguagens de programação, C++, Java, Delphi, entre outras, utiliza a linguagem SQL para trocar dados entre os programas e o SGBD. Aqui você encontrará um material de referência amplo, dividido por semana de estudo. Outros materiais se agregam à disciplina, tais como vídeos e textos adicionais, o que de forma alguma esgota o assunto, mas dá uma base necessária para a atuação do profissional técnico em PGTI.

Ao preparar este caderno, procurei fazê-lo com um conteúdo bastante abrangente e aprofundado. A partir daí chega-se a duas conclusões. A primeira é que você, estudante desta disciplina, terá uma ótima oportunidade de realmente aprender, de forma teórica e prática, o conteúdo sobre banco de dados. A segunda é que este curso, conforme foi montado, exigirá extrema dedicação de sua parte. Será necessária uma leitura atenta dos textos e execução minuciosa das práticas. Portanto, siga esse conselho desde o início, reserve muitas horas de estudo semanais e não desanime. Conte com o meu apoio para buscar o conhecimento.

Sucesso e bom trabalho!

Prof. Edson Marchetti da Silva



# Apresentação da disciplina

Banco de Dados é uma disciplina do Curso de PGTI de fundamental importância para o desenvolvimento do trabalho diário dos técnicos de PGTI, tanto para aqueles que irão desenvolver sistemas quanto para os que vão gerenciar o uso por parte dos demais usuários. É uma disciplina essencialmente técnica, pois faz uso de produtos de *software*, tais como os sistemas gerenciadores de banco de dados (SGBDs), que servem para lidar com os dados e informações empresariais que serão processados pelos sistemas de informação. Nesta disciplina iremos trabalhar especificamente com os SGBDs relacionais. Esse tipo de *software* implementa vários conceitos teóricos do modelo relacional, baseado na teoria matemática de conjunto a fim de permitir que os dados sejam armazenados e organizados de forma segura e evitando ambiguidades e redundâncias. Ou seja, a ideia é que os dados armazenados tenham uma estrutura que expresse o mesmo significado que eles possuem no mundo real.

Nesta disciplina, temos como objetivo maior a instrumentalização do aluno, através do estudo dos conceitos básicos de modelagem de dados: entidades, atributos, relacionamentos, chaves primárias e estrangeiras, regras de normalização e padronização de nomes. Isso o levará ao desenvolvimento das habilidades necessárias para elaboração de uma representação de dados que devem ser armazenados para dar suporte a um sistema de informação, construção dos diagramas de entidade e relacionamento (DER). Também o capacitará para usar a linguagem de acesso e recuperação dos dados armazenados, entendendo alguns aspectos tecnológicos envolvidos nesse ambiente de *software*.

Você estará apto também a entender a estrutura física de um SGBD, o controle de transação e o de concorrência. Poderá ainda perceber as diferenças entre sistemas de arquivos convencionais e os SGBDs, o funcionamento básico da linguagem SQL, o conceito de instância de banco de dados, e compreender as vantagens do uso de visões, índices e uniões.



# Projeto instrucional

**Disciplina:** Banco de Dados (carga horária: 60h).

**Ementa:** Introdução ao conceito de banco de dados. Banco de dados. Fases de um projeto de banco de dados. Modelo entidade-relacionamento. Normalização. Modelo de dados. Modelo orientado a objetos. *Structure Query Language*. Segurança e integridade.

AULA	OBJETIVOS DE APRENDIZAGEM	MATERIAIS	CARGA HORÁRIA (horas)
1. Conceitos de modelo de dados	Entender o porquê de modelar dados, os principais conceitos envolvidos e quais os benefícios para o processo de desenvolvimento de sistemas.	PowerPoint, IntroduçãoBD.ppt	8
2. Diagrama de Entidade e Relacionamento	Entender os principais conceitos envolvidos com a construção dos Diagramas de Entidade e Relacionamento.	PowerPoint , ModeloER.ppt	8
3. Diagrama de Entidade e Relacionamento – casos práticos	Entender os principais aspectos da modelagem de dados e elaboração dos DERs.	Caderno de atividades e de conteúdos	7
4. Estrutura física	Entender a estrutura física de um SGBD e controle de transação e concorrência.	RoteiroPratica01.pdf	7
5. Structure Query Language	Entender as diferenças entre sistemas de arquivos convencionais e os SGBDs e o funcionamento básico da linguagem SQL.	VideoAula5.avi	8
6. Implementando restrições e visões	Entender o conceito de instância de banco de dados, o uso das visões, índices e das uniões.	VideoAula6.avi	7
7. Estudo de caso Fórmula I	Apresentar um estudo de caso prático.	Caderno de atividades e de conteúdos	7
8. Integridade referencial – Estudo de Caso Fórmula I	Entender como funciona na prática a integridade de dados referencial.	Caderno de atividades e de conteúdos	8



# Aula 1 – Conceitos de modelos de dados

## Objetivo

Entender o porquê de modelar dados, os principais conceitos envolvidos e quais são os benefícios para o processo de desenvolvimento de sistemas.

### 1.1 Introdução

O objetivo deste texto é oferecer a você, estudante do Curso Técnico de Planejamento e Gestão da Tecnologia da Informação, e também aos profissionais da área, uma visão abrangente e profunda sobre o tema. A ideia é fazer uma abordagem pragmática e objetiva, proporcionando a transmissão de conhecimentos teóricos aplicados de forma crítica a partir da experiência do autor no desenvolvimento, manutenção e implantação de diversos tipos de sistemas de informação, em empresas dos mais variados portes e ramos de atividades.

O texto utiliza uma metodologia em que um conteúdo teórico é seguido de um estudo de caso que serve como pano de fundo para discussão das possíveis soluções e de seus impactos no resultado final de um produto de software. O MySQL foi utilizado como banco de dados de referência para apresentação dos exemplos práticos. No decorrer das aulas, o objetivo é mostrar e justificar para o leitor quais são as vantagens em projetar um modelo de dados com rigor, utilizando todo o poder oferecido pelos SGBDs atuais. Afinal os SGBSs são não somente um local seguro para armazenar e recuperar dados, mas principalmente um local capaz de garantir a riqueza semântica dos dados. Esta estratégia procura trazer parte do conhecimento ou das regras do negócio, que normalmente ficam espalhadas pelo código-fonte das linguagens de programação para dentro do banco de dados. Isto certamente trará ganhos de produtividade se considerado todo o processo de desenvolvimento de software.

Estudar este tema com afinco é vital para a formação do profissional. Portanto, espero ajudá-lo alcançar esse objetivo. Vamos ao trabalho!

## 1.2 Conceitos sobre modelagem de dados

Esta aula começa apresentando uma discussão da importância dos dados como a base para o desenvolvimento do *software*. A partir daí estuda-se o que é e quais são os principais conceitos envolvidos no processo de modelagem de dados. Em destaque, uma discussão sobre as três formas normais e suas aplicações, pois afinal normalização é um importante conceito da modelagem de dados, que evita as anomalias, tais como redundância e perda de informação. Bom! Mas isso é o que veremos a seguir.

### 1.2.1 Por que devemos modelar dados?

Durante as últimas décadas, o processo de desenvolvimento de sistemas para fins comerciais vem ocorrendo tanto para uso específico de alguma empresa quanto elaborado por empresas desenvolvedoras de soluções para seus clientes. Nesse período surgiram diversas metodologias para o desenvolvimento de sistemas, diversas formas de abordar o problema e propor solução: análise estruturada, essencial, orientada a objetos, etc. Sem falar no lado tecnológico, que sofre uma nova onda a cada período e com ciclos cada vez menores. Foram várias as linguagens de programação e ambiente de desenvolvimento: Cobol, Clipper e a família dBase, o Delphi, o VisualBasic, PHP, Java, etc. Convivemos com diversas formas de interface com o usuário: texto, semigráfica, gráfica, web, etc. Durante todo esse período houve muitos erros e acertos. O sucesso da informática é inexorável; a prova disso é que ela está em toda a parte. O uso da tecnologia da informação é visto pelas empresas não apenas como necessário para condução das tarefas, mas, como uma vantagem competitiva. Em contrapartida, tivemos também muitos casos de insucesso na implantação de sistemas, em que um enorme volume de recursos foi gasto sem que se houvesse um retorno econômico esperado. Não é tão raro presencermos cancelamentos de projetos nos quais já se investiram valores significativos.

Diante desse mundo da TI, em que tudo é efêmero, vivenciamos situações em que sistemas de informação têm de ser substituídos, não porque suas funcionalidades não atendem aos processos de negócio, mas apenas porque estão defasados tecnologicamente. Isso leva à reconstrução de sistemas inteiros, trocando o ambiente de desenvolvimento, reescrevendo todos os programas para obter um novo resultado final, em um novo paradigma tecnológico. Mudanças tecnológicas normalmente levam a mudanças da metodologia de desenvolvimento de sistemas, pois, cada uma delas é mais adequada a um determinado arcabouço tecnológico. Entretanto, diante de tudo isso, podemos observar, que independe da arquitetura utilizada na constru-

ção da solução: sistema multiusuário, cliente-servidor ou em camadas. Ou ainda, no tocante à interface do *front end*: texto, gráfico ou web. O fato é que, desde a década de 1970, os bancos de dados estão aí como solução de repositório de dados e informações. Eles garantem integridade, segurança, confiabilidade, etc. Dessa forma, podemos afirmar sem medo de estarmos cometendo algum equívoco, que a parte mais estável no tocante às tecnologias que envolvem as soluções de TI está relacionado com os dados e com a forma em que eles são armazenados e disponibilizados.

A seguir apresentaremos mais alguns argumentos de que a partir do entendimento do problema, uma boa forma de propor uma solução é tomar como base a elaboração de um modelo de **persistência** de dados com grande riqueza **semântica**.

- O dado é a origem da informação. Muitas das vezes não é possível, independentemente da tecnologia, alterar programas para obter uma informação desejada. Ou seja, armazenando apenas o todo não é possível conhecer as partes que o compõem.
- O dado é a infraestrutura do sistema e está por baixo da estrutura. Ele suporta a estrutura. Portanto, mexer na forma de armazenar os dados impacta toda a estrutura. O mesmo não é sempre verdade quando temos mudança na regra de negócio contida nos programas ou na forma visual da interface.
- Normalmente, as falhas de normalização irão gerar esforço de programação a fim de evitar inconsistência dos dados.
- Definir semântica no modelo de dados reduz a quantidade necessária de código- fonte a ser escrito no programa aplicativo.

Embutir regras de negócio no banco de dados melhora o desempenho do sistema, garante a unicidade do código, além de simplificar o programa de aplicação e criar maior independência da interface e do aplicativo.

Mas, como nem tudo são louros, obviamente, utilizar o banco de dados em sua plenitude aumenta o grau de dificuldade na sua administração, entretanto, de uma forma bastante controlada e justificável. Por fim, o trabalho é menor e o resultado será sempre melhor.

## A-Z

### Persistência

Nesse contexto, está relacionada à garantia de permanência ou gravação dos dados em um meio seguro.

### Semântica

Nesse contexto, está relacionada a levar para o banco de dados informações a respeito do significado dos dados armazenados. Ex.: garantir que no campo com informações sobre o sexo do cliente só aceite M ou F, significando masculino ou feminino, respectivamente.

## 1.2.2 Modelo de dados ER

O modelo de dados é uma forma de abstração do mundo real. É um recorte no qual criamos um “minimundo” do que se deseja representar. O modelo expressa para o sistema a ser elaborado um entendimento de quais dados devem ser armazenados, qual é a relação de dependência que existe entre eles e qual é o significado semântico desses dados e de suas relações.

O modelo ou Diagrama de Entidade e Relacionamento (DER) é uma das formas de abordagem semântica mais conhecida e provavelmente das mais utilizadas atualmente. Ele foi proposto por Peter Chen em 1976, sendo, a partir daí, refinado e ampliado por diversos autores. Um DER é composto por Entidades (tabela que agrupam os dados de algo que se deseja armazenar no banco de dados). Ex.: (Aluno, Curso) e Relacionamentos (representa a forma de como as entidades se relacionam entre si). É importante destacar que as entidades são compostas de atributos, os quais descrevem as características da própria entidade. (Ex.: nome do aluno, número da matrícula do aluno, nome do curso, etc.).

## 1.2.3 Definindo entidade/atributos

Segundo Korth e Silberschatz (1989), uma entidade é um objeto que existe e se distingue de outros objetos. Um CNPJ. de uma empresa 17.111.222.0001-44 é uma entidade, uma vez que identifica univocamente uma empresa/filial. As várias empresas formam o conjunto de entidades do mesmo tipo que formam a **relação** Empresa. Elmasri e Navathe (2000) corroboraram utilizando essa mesma nomenclatura. Date (2004) descreve o termo entidade como sendo normalmente utilizado na área de banco de dados para indicar qualquer objeto distinguível que seja neste representado.

Conforme definido pelos autores o termo entidade está relacionado a uma linha ou ocorrência em uma tabela do banco de dados, enquanto que, o termo conjunto de entidades estaria associado à tabela propriamente dita. Na prática, nenhum autor faz distinção entre esses termos, utilizando apenas o termo entidade como sinônimo de tabela.

Uma entidade pode ser: um objeto com existência física (entidade concreta), ex.: um empregado, um aluno, um carro; ou conceitual (entidade abstrata), ex.: uma empresa, uma matrícula, um relacionamento entre duas ou mais entidades.

Uma entidade pode ser descrita por propriedades particulares que a distinguem e a descrevem. Essas propriedades são chamadas de atributos da en-

A-Z

Relação

O termo “relação” pode ser entendido nesse contexto como sinônimo de tabela.

tidade. Fazendo associação às tabelas dos bancos de dados, as linhas são as ocorrências e os atributos são as colunas da tabela. Por exemplo: na entidade Piloto mostrada na Figura 1.1, temos dois atributos (colunas): Cod\_Piloto, Nom\_Piloto e três linhas nas quais é mostrado o conteúdo para cada piloto cadastrado na tabela. Cada ocorrência ou linha da entidade possui valores (atributos) que identificam um único piloto.

Tabela - Piloto

Cod_Piloto	Nom_Piloto
ALO	Fernando Alonso
MAS	Felipe Massa
RUB	Rubens Barrichello

**Figura 1.1: Tabela piloto**

Fonte: Elaborada pelo autor

Os atributos podem ser:

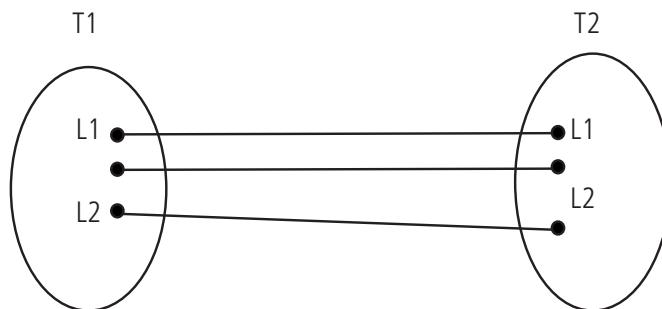
- **Simples** (atômicos) – quando não fizer sentido semântico dividir o atributo em subpartes. Exemplo: a altura ou o peso de uma pessoa.
- **Composto** – quando o atributo puder ser dividido em subpartes com significado semântico. Exemplo: endereço de uma pessoa pode ser dividido em Cep, cidade, bairro, logradouro, número e complemento.
- **Multivalorado** – quando o atributo possuir simultaneamente mais de um valor para uma mesma ocorrência da entidade. Exemplo: a cor de um carro pode ser parte preta e parte branca.
- **Derivado** – quando o atributo puder ser calculado dinamicamente sem que haja necessidade de manter o dado armazenado no banco de dados. Exemplo: em uma entidade com itens de uma nota fiscal. A partir do preço unitário e da quantidade vendida o valor total do item pode facilmente ser calculado pela multiplicação dos dois termos.
- **Obrigatório** – quando não for possível incluir uma linha na tabela sem que haja alguma informação/valor nos atributos obrigatórios dela.
- **Não obrigatório** – quando ao inserir uma linha na tabela isto possa ser feito independentemente da obrigação de informar dados aos atributos que aceitam valores nulos.

- **Complexo** – quando juntamos os conceitos de atributos composto e multivalorados.
- **Delimitado** – quando existir definição no banco de dados de uma regra de domínio que delimita os valores válidos possíveis a serem armazenados.

#### 1.2.4 Definindo relacionamentos

O relacionamento é uma representação da forma de como as entidades se relacionamumas com as outras. Ele expressa certas restrições existentes no mini-mundo a ser modelado que delimitam como uma ou mais ocorrências de uma entidade se relacionam com uma ou mais ocorrências de outra entidade. Este conceito é denominado cardinalidade do relacionamento. Para um conjunto de relacionamento binário, entre duas tabelas T1 e T2, a cardinalidade pode ser:

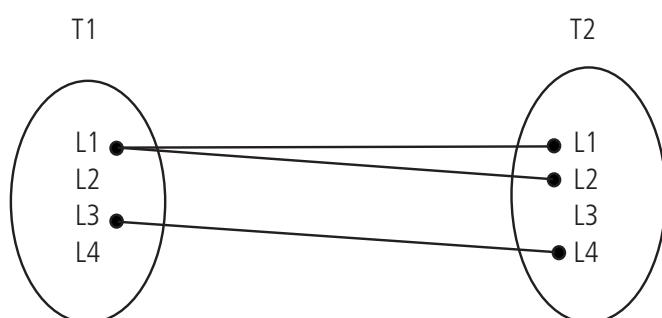
**Um para um** – uma linha da tabela T1 está associada com no máximo uma linha da tabela T2 e vice-versa, conforme mostrado na Figura 1.2 a seguir.



**Figura 1.2: Representação da cardinalidade um para um**

Fonte: Elaborada pelo autor

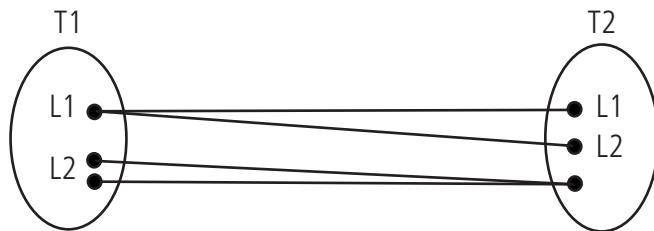
**Um para muitos** – uma linha da tabela T1 está associada com de zero a “n” linhas da tabela T2, enquanto que, uma linha da tabela T2 está associada com no máximo uma linha da tabela T1, conforme mostrado na Figura 1.3. Observe que relacionamentos muitos para um é o inverso de um para muitos.



**Figura 1.3: Representação da cardinalidade um para muitos**

Fonte: Elaborada pelo autor

**Muitos para muitos** – uma linha da tabela T1 está associada com zero a “n” linhas da tabela T2 e vice-versa, conforme mostrado na Figura 1.4 a seguir.



**Figura 1.4: Representação da cardinalidade muitos para muitos**

Fonte: Elaborada pelo autor

### 1.2.5 Atributo chave (Candidata, Primária e Estrangeira)

Outra importante tarefa da modelagem é determinar dentre os atributos de uma entidade/tabela quais são aqueles que de forma individual ou agrupada são capazes de identificar uma única ocorrência da entidade. Esse conjunto de atributos é chamado de chaves candidatas. Ou seja, são os possíveis candidatos a serem definidos como chave primária. Entende-se como chave primária a menor quantidade de atributos capaz de identificar uma ocorrência da tabela de forma unívoca.

Podem ocorrer casos em que tenhamos diversos conjuntos distintos de chaves candidatas. Cabe ao projetista escolher dentre as possibilidades aquela que for mais adequada como solução para o problema proposto. Vejamos um exemplo da entidade Veículo. Dentre os atributos de um veículo, tais como marca, modelo, ano de fabricação, cor, placa e chassi, Renavam, etc., existem pelo menos três possíveis chaves candidatas: placa, chassi e Renavam, pois, cada uma delas, individualmente, identifica de forma única um veículo. Para um sistema que trabalhe apenas com veículos licenciados para rodar no Brasil, o uso da placa como chave é provavelmente o mais apropriado, pois placa pode ser representada por um menor número de caracteres que o Renavam e o chassi. Portanto, é mais fácil de digitar e mais fácil de visualizar. O chassi fica posicionado no veículo em locais mais difíceis de ver do que a placa, e o Renavam fica informado apenas no documento do veículo. Já o uso do chassi como chave primária poderia ser utilizado em um sistema de controle de veículos novos, não emplacados, que estão sendo transferidos, importados e exportados, entre diversos países. O que queremos dizer é que a motivação para a tomada de decisão depende de fatores externos às regras de modelagem.

É também possível que uma entidade não possua atributos suficientes para formação de uma chave primária. Nesse caso poderá ser arbitrado um novo atributo sem significado semântico, mas que sirva com chave ou parte da chave. Normalmente uma sequência numérica serve como atributo complementar nas entidades fracas, que herdam a chave da tabela pai, entidade forte. Esse atributo numérico sequencial sempre começa de um (1) para cada registro da tabela associado à tabela pai. Ele representa uma sequência de ocorrências na tabela filha, relacionada a apenas uma ocorrência da tabela pai. Veja um exemplo de uma entidade Funcionário (entidade forte) e Dependente (entidade fraca) mostrado na Figura 1.5. O funcionário “José dos Santos” tem três dependentes, sequência (1, 2 e 3).

Funcionario			Dependente		
Cod_Funcionario	Nom_Funcionario	Nro_Telefone	Cod_Funcionario	Seq_Dependente	Nom_Dependente
1	José dos Santos	(31) 2222-2222	1	1	José dos Santos Junior
2	Maria da Silva	(31) 3333-3333	1	2	Antônio dos Santos
			1	3	Joana dos Santos
			2	1	João Pedro

**Figura 1.5: Representação das tabelas Funcionário e Dependente**

Fonte: Elaborada pelo autor

## A-Z

### Autoincrementado

É uma sequência numérica gerada automaticamente pelo SGBD.

### Chave primária

Pode ser definida como a menor quantidade de atributos capaz de identificar uma linha de uma tabela de forma unívoca.

### Chave estrangeira

Corresponde a um conjunto de um ou mais atributos de uma linha de uma entidade com o objetivo de referenciar uma ou mais linhas em outra entidade na qual esses mesmos atributos são uma chave primária parcial ou completa.

Outro exemplo que podemos citar é o de uma entidade que não possui nenhum atributo natural capaz de representar de forma unívoca uma linha da tabela. Nesse caso, a chave primária pode ser definida como um atributo **autoincrementado**.

Um importante conceito é o de **chave estrangeira**. A chave estrangeira é utilizada para ligar uma ocorrência de uma entidade a uma ou mais ocorrências de outra entidade. Ela se caracteriza por aparecer em uma entidade com atributo originado de outra entidade na qual ela é uma **chave primária**. Ou seja, a chave estrangeira é uma referência numa entidade que aponta para a chave primária de outra entidade, possibilitando dessa forma as junções entre as entidades envolvidas. É ela que informa ao banco de dados a integridade referencial entre as tabelas envolvidas no relacionamento.

## 1.2.6 Normalização

A normalização é um conjunto de regras criado para ajudar os projetistas de modelos de dados a definir qual é a melhor forma de agrupar os atributos em uma tabela e de como essas tabelas se relacionam criando um esquema que evite as anomalias de atualização dos dados na inserção, remoção ou atualização. As anomalias, características indesejáveis, que devemos eliminar do modelo são: redundância da informação; incapacidade de junção adequada da informação e perda da informação.



Veja o PowerPoint com explicações adicionais disponível no ambiente virtual de ensino-aprendizagem (AVEA).

Edgard F. Codd (apud DATE,2002), em 1972, foi o primeiro a escrever sobre o conceito de decomposição sem perdas e dependência funcional, que é a base para o procedimento de normalização. Inicialmente Codd propôs as três formas normais, conhecido pelo acrônimo 3FN. A dependência funcional é uma propriedade que nos leva a decompor uma relação com o objetivo de fazer com que todos os atributos em uma tupla (linha) dependam exclusivamente da sua chave primária. O conceito de decomposição sem perdas significa dizer que ao decompor uma relação em duas ou mais relações, não poderá haver perdas de informação. Todos os atributos deverão permanecer no banco de dados. A decomposição apenas rearranja esses atributos em novas tabelas a fim de evitar as anomalias citadas anteriormente.

Vamos usar como referência para explicar o tema o seguinte exemplo: suponha que desejamos armazenar dados dos empregados de uma empresa em um banco de dados. Os seguintes atributos devem ser persistidos:

- código do empregado (Cod\_Emp);
- nome do empregado (Nom\_Emp);
- data de admissão (Dat\_Admis);
- nome do dependente (Nom\_Dep);
- data de nascimento do dependente (Dat\_Nasc);
- descrição da função (Des\_Funcao);
- percentual de participação no lucro (Per\_Par).

A seguir, na Figura. 1.6, apresentamos um exemplo de um conjunto de dados definidos para a tabela Empregado.

Cod_Emp	Nom_Emp	Dat_Admis	Nom_Dep	Dat_Nasc	Des_Funcao	Per_Par
00001	José	01/05/2000	Alex Marina	05/03/2000 20/07/2005	Gerente	200
00002	Marcos	15/08/2000			Engenheiro	100
00003	Francisco	01/09/2000	Pedro Mateus Carla	12/02/1995 17/03/1998 07/08/2002	Encarregado	50
00004	Manuel	01/09/2000	Beatriz	28/06/2006	Servente	50

**Figura 1.6: Tabela Empregado**

Fonte: Elaborada pelo autor

**Primeira forma normal 1FN** – visa eliminar os atributos multivalorados, atributos compostos e suas combinações. Ou seja, para cada ocorrência da chave primária, só pode corresponder a uma ocorrência dos demais atributos não chave. Os atributos de uma relação devem ser atômicos (indivisíveis) não repetitivos. O uso dessa regra gera alguma controvérsia. Existem alguns produtos no mercado chamados NFNF, que significa *Non First Normal Form*, que permitem definir atributos multivalorados. Na prática, quando temos um controle bem definido dos valores de domínio de um atributo, sabemos o número exato de repetições e sendo a quantidade de repetições constante e bem reduzida é possível, por opção do projetista do modelo de banco de dados, violar a 1FN sem ficar com nenhum remorso.

Como pudemos ver no exemplo proposto pela Figura 1.6, cada empregado pode ter de nenhum até vários dependentes. Nesse caso existem três possibilidades em criar um modelo que atenda à 1FN.

A primeira solução seria o uso de atributos multivalorados ou, na inexistência destes, definir o nome seguido de uma sequência numérica, como por exemplo: Nom\_Dep1, Nom\_Dep2, ..., NomDepN. Ou seja, criar espaço para armazenar os 'N' dependentes de cada empregado. Entretanto, as condições apresentadas anteriormente para abrir uma exceção quanto ao uso de atributos repetitivos não existem neste caso. Portanto, é fácil perceber que esta solução não é a mais adequada. Não parece viável, de antemão, criar, por exemplo, dez atributos para cadastrar dependentes sabendo que muitos empregados podem ter nenhum e outros podem ter mais de dez dependentes. Quanto espaço e trabalho seriam desperdiçados nesses casos? A solução ideal para cada caso depende das demandas do minimundo que queremos representar e do bom senso do projetista. Portanto, iremos descartá-la.

A segunda alternativa é que como existem vários dependentes para cada empregado e não é razoável criar atributos repetitivos, a opção seria repetir os dados de um determinado empregado em tantas tuplas quantos fossem o número de dependentes dele. Como existirão várias tuplas de um mesmo empregado, a chave primária não poderá ser composta apenas no Cod\_Emp. Será necessário criar uma chave composta agregando, por exemplo: uma sequência numérica para o cada dependente incluído por empregado. Também nos parece óbvio que esta solução, mesmo atendendo 1FN, gera muita redundância dos dados do empregado tais como Cod\_Emp, Nom\_Emp, Dat\_Admis e Des\_Funcao.

Portanto, a solução mais adequada seria remover o atributo multivalorado Nom\_Dep da entidade empregado criando uma nova entidade Dependente. A chave de Dependente é composta pela propagação da chave primária de Empregado, agregada a uma sequência numérica dos dependentes. A entidade Dependente é uma entidade fraca, pois ela herda a chave da entidade pai Empregado (entidade forte). Nela deve-se acrescentar um atributo a mais para compor sua chave primária. Pois, como não é possível determinar com exatidão quantos dependentes cada empregado pode ter, a entidade Dependente terá para um determinado empregado exatamente uma tupla para cada dependente. Veja a solução proposta na Figura 1.7 a seguir.

Empregado	Dependente
# Cod_Emp	# Cod_Emp
Nom_Emp	# Seq_Dep
Des_Funcao	Nom_dep
Per_Part	Dat_Nasc

**Figura 1.7: Representação das entidades Empregado e Dependente**

Fonte: Elaborada pelo autor

**Segunda forma normal 2FN** – uma entidade está na 2FN se, e somente se, estiver na 1FN e se cada atributo não chave depender funcionalmente da totalidade da chave primária. Basicamente, um determinado atributo tem dependência funcional com relação à chave, quando para realizar seu acesso é necessário conhecer a chave. Vejamos o atributo Des\_Funcao na entidade Empregado. A descrição de uma função não dependente do empregado que exerce a função. Podemos ter vários empregados que têm a mesma função. Portanto, a descrição da função está ligada a uma chave primária que identifica a função. Nesse caso, a entidade Empregado deve possuir como atributo apenas uma referência (chave estrangeira) a esta chave primária. Portanto, para adequarmos o modelo à 2FN teremos de remover os atributos da entidade Empregado que não tenham dependência funcional com a chave primária como um todo. Veja a solução proposta na Figura 1.8 a seguir.

Empregado	Dependente	Funcao
# Cod_Emp	# Cod_Emp	
Nom_Emp	# Seq_Dep	# Cod_Funcao
* Cod_Funcao	Nom_dep	Des_Funcao
Per_Part	Dat_Nasc	

**Figura 1.8: Representação das entidades Empregado, Dependente e Função**

Fonte: Elaborada pelo autor

É importante esclarecer que ao criarmos a entidade Função, foi necessário definir um atributo para ser chave primária da entidade. Como proceder neste caso? Bem, primeiro devemos identificar pela natureza dos dados associados à entidade em questão, se existe alguma chave natural. Entenda como chave natural algum atributo que já é reconhecido no mundo real como único para identificação de uma ocorrência ou linha da entidade. No nosso caso podemos utilizar a Classificação Brasileira de Ocupações (CBO) como uma possível chave candidata ou definir uma chave autoincrementada sem significado semântico, entre outras soluções. Mais uma vez o leitor deve estar se perguntando: qual será a melhor solução? A resposta mais uma vez é: depende. Ao adotarmos o CBO como chave, não teremos liberdade para separar ou agragar funções que são trabalhadas no minimundo modelado. Ao adotarmos um número sequencial, estaremos criando uma chave “burra” para entidade e provavelmente teremos de criar o CBO como um atributo não chave, caso tenhamos de enviar dados para o Ministério do Trabalho. Cabe ao projetista definir a melhor opção caso a caso, dependendo do contexto.

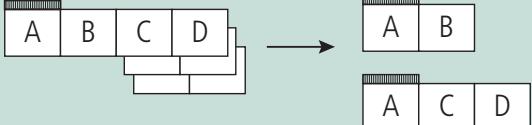
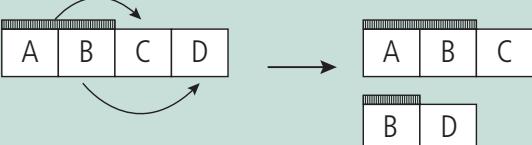
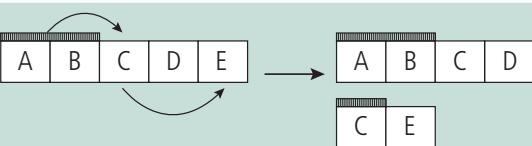
**Terceira forma normal 3FN** –uma entidade está na 3FN se, e somente se, estiver na 2FN e não existam atributos que dependam do(s) atributo(s) chave que não sejam chave primária, ou se já não exista dependência transitiva entre os atributos. No nosso exemplo, o percentual de participação do lucro não depende do empregado e sim da função que ele exerce. Se ele for promovido, mudar de função, automaticamente ele poderá ter seu percentual de participação alterado. Na entidade Empregado existe uma dependência transitiva entre a chave de função e o percentual de participação. Portanto, o modelo não atende à 3FN. Veja a alteração feita na Figura 1.9 para resolver o problema.

Empregado	Dependente	Função
# Cod_Emp	# Cod_Emp	
Nom_Emp	# Seq_Dep	
* Cod_Funcao	Nom_dep	# Cod_Funcao
	Dat_Nasc	Des_Funcao
		Per_Part

**Figura 1.9: Adequação das entidades à 3FN**

Fonte: Elaborada pelo autor

A Figura 1.10 apresenta um resumo do processo de normalização apresentado.

Regra	Esquema	Descrição
1FN		Eliminar os subgrupos repetitivos, decompondo a relação em duas ou mais relações.
2FN		Separar os atributos que dependem de um subconjunto da chave, decompondo a relação em duas ou mais entidades.
3FN		Separar os atributos que dependem de um outro atributo não pertencente à chave, decompondo a relação em duas ou mais entidades.

**Figura 1.10: Resumo adaptado das três formas normais**

Fonte: Sousa; Loureiro (2012)

É importante ressaltar que o conhecimento e aplicação das regras de normalização são fundamentais para construir ou mesmo refinar o modelo de dados. Entretanto, na prática, existem algumas honrosas exceções em que se justifica a violação de alguma dessas regras: por exemplo, para melhoria do desempenho na recuperação de um dado, ou quando se deseja manter informações históricas, portanto estáticas, que perdem a integridade referencial, podendo transferir atributos referenciados em outra tabela como sendo atributos da própria tabela.

## Modelo Relacional

### Normalização

#### Diagramas E-R e Tabelas Originadas

## Vejamos um outro exemplo do uso da 3FN

Normalmente, um bloco de pedidos em uma loja qualquer tem um formulário em duas vias onde se deve preencher à mão os dados do pedido, do cliente, da forma de pagamento, condições de entrega, itens relacionados entre outros. Para simplificar, estamos representando apenas parte desses dados na tabela de cabeçalho do pedido (CabecPedido) conforme mostrado na Figura 1.1 a seguir.

Nro_Pedido	Dat_Pedido	Nom_Cliente	Des_Endereco	Nro_Telefone
00001	05/01/2011	José	R. Itu, 100	3222-2222
00002	05/01/2011	Antônio	R. Sabará, 322	3111-3333
00003	06/01/2011	José	R. Itu, 100	3222-2222
00004	06/01/2011	Antônio	R. Sabará, 322	3111-3333

**Figura 1.11: Tabela de CabecPedido**

Fonte: Elaborada pelo autor

Como podemos perceber, um mesmo cliente pode fazer vários pedidos; isto faz com que seus dados sejam repetidos em diferentes folhas do bloco de pedidos. Mas ao projetarmos o modelo de um banco de dados capaz de persistir as informações do pedido, ele não terá exatamente a mesma estrutura em que se encontra no mundo real. Teremos de fazer uma decomposição sem perdas de dados de tal forma a evitar anomalias, tais como: se o cliente fizer um novo pedido, tenho que lhe pedir novamente seus dados de cadastro? Se o cliente pedir para cancelar todos os seus pedidos, tenho que pedir que ele informe o número de cada pedido? Ou senão? Tenho que pesquisar em todos os pedidos quem tem o nome José? Poderá haver outros clientes com o mesmo nome? E se ao cadastrar o cliente forem utilizadas grafias diferentes: Jose, JOSE, Jozé, etc. Problemas semelhantes podem ocorrer caso o cliente solicite para alterar o endereço de todos os seus pedidos.

Como podemos ver, se fizermos uma conversão direta do bloco de pedidos em um modelo de banco de dados, diversas anomalias ocorrerão no modelo proposto. Como um cliente pode fazer vários pedidos, os atributos de cliente não dependem funcionalmente na chave primária de CabecPedido. Os atributos de cliente devem ser separados em uma nova relação. Para tal, devemos identificar ou criar entre os atributos de cliente quais são o(s) conjunto(s) de atributos que podem ser considerados como chaves candidatas. E entre as chaves candidatas, qual é a mais indicada para ser chave primária. Neste ponto, cabe um debate sobre o assunto: Qual é a forma correta de escolher a chave primária? Resposta: depende. Sempre quando estamos definindo um modelo de banco de dados, estamos fazendo uma representa-

ção do minimundo. Portanto, nem sempre é possível, ao cadastrar um cliente, obtermos todos os seus dados. Suponha que o cliente está comprando em dinheiro e pagou à vista; ele pode não querer fornecer seu número de CPF ou sua carteira de identidade. Se um desses campos for chave primária da tabela, como cadastrar os diferentes clientes que tenham essa mesma conduta. Nesse caso, não seria melhor pesquisar o cliente pelo nome, buscando identificar se ele já está cadastrado? Dessa forma, talvez a melhor saída seja a geração de uma chave primária automática “burra” sem nenhum significado semântico. Apenas um número sequencial. Os demais atributos, chaves candidatas: CPF ou carteira de identidade poderiam ser atributos que aceitem valores nulos. Desse modo, torna-se possível fazer um cadastramento do cliente de forma incompleta para uma venda à vista. Para os casos de venda a prazo, seria possível criar uma regra de negócio que verifique se o cadastro do cliente está completo, não permitindo a realização da venda sem o preenchimento completo e verificação de aprovação de crédito do cliente.



Assista ao vídeo com informações introdutórias complementares em [http://youtu.be/qgnuH\\_qSI9o](http://youtu.be/qgnuH_qSI9o) e <http://youtu.be/tEEAIs6aB2s>

## 1.3 Padronização

Ao iniciar um processo de desenvolvimento de um sistema, assim como qualquer outra atividade na qual se pretende chegar a um produto final, é de fundamental importância estabelecer previamente regras de como chegar lá. Qual será o processo utilizado? Quais serão as atividades dentro do processo? Enfim, qual será a metodologia, composta de quais artefatos, gerando quais produtos intermediários, para possibilitar a validação das várias fases do processo de desenvolvimento. Todo esse questionamento nos remete à área da Engenharia de Software, sobre a qual na verdade não é o objetivo deste texto discorrer. Mas, independentemente da metodologia utilizada, o que importa nesse contexto é que todos dados identificados na etapa de levantamento do problema, que devam ser persistidos no banco de dados, sejam definidos a partir de um padrão de nomes.

Em um SGBD, ao criarmos um banco de dados, será necessário atribuir a ele um nome. O banco de dados, depois de criado, possibilita definir quais tabelas ele conterá. Cada tabela deve possuir um nome único dentro de um mesmo banco de dados. Uma tabela é formada por um conjunto de atributos (partes indivisíveis de informação). Cada atributo deverá ter um nome único dentro de uma mesma tabela. Diversas regras semânticas poderão ser definidas no banco de dados, tais como: chaves primárias, visões, restrição de domínio, etc. Cada uma dessas restrições, ao ser definida, recebe um nome que a identifique de forma unívoca. Enfim! Esses assuntos serão mais bem detalhados no decorrer do texto.

Na verdade, o que importa neste momento é dizer que mesmo na fase do projeto conceitual é importante definir nomes de tabelas e atributos seguindo alguma regra de padronização. Isto irá ajudar muito na organização e principalmente no mapeamento do modelo conceitual para o esquema físico do banco de dados, fazendo com que os ajustes da sua estrutura física sejam mínimos em relação ao projeto lógico.

Após defendermos a bandeira da padronização, afinal, qual deve ser padrão? Na verdade, a resposta para esta pergunta é que não existe uma regra única e obrigatória. Normalmente, cada grupo de desenvolvedores dentro de uma empresa normatiza e escreve as suas próprias regras. A título de exemplo, iremos apresentar algumas sugestões de padronização.

Mesmo apesar de alguns SGBDs aceitarem características da língua portuguesa, tais como: acentos, cedilha e espaços em branco. Essas situações devem ser evitadas na criação de nomes de tabelas, atributos, etc. Agindo dessa forma você estará evitando possíveis problemas de incompatibilidades. Além disso, um detalhe importante é atentarmos para as limitações de nomenclatura do próprio SGBD que será utilizado como repositório. Ao propor uma regra o importante é utilizar o bom senso. Afinal! E o que é bom senso? Vamos apresentar nossa visão.

### **Nome de entidade/tabela**

- Deve-se evitar uso de nomes codificados, como por exemplo, S01010, pois, ao olharmos para o nome, ele não tem nenhum significado.
- Os nomes devem ter um limite de tamanho; normalmente 18 caracteres é o máximo.
- Deve-se procurar utilizar nomes que traduzam de forma clara o sentido da entidade, normalmente no singular. Exemplo: cliente, aluno, etc.
- Deve-se evitar vincular ao nome de uma tabela a associação a um determinado sistema. Pode ocorrer que, em uma nova versão do sistema, a mesma tabela passe a ser utilizada por dois ou mais sistemas. Nesse caso, seria necessário trocar o seu nome, ou mantê-la no banco de dados com um nome que não expressa a realidade. Exemplo: o sistema de contabilidade tem um prefixo “contab”; então, usamos para nomear as tabelas este prefixo: contab\_plano\_conta.

- Uma possível solução para reduzir o tamanho do nome é criar um mnemônico com um tamanho fixo de caracteres, por exemplo, seis caracteres, sendo a formação do nome composta pelas iniciais do nome por extenso. Exemplo: fornecedor = fornec; agência bancária = ageban, etc. Nesse caso foi criada uma codificação, mas que mantém ainda assim certo teor semântico.

## Nome de atributos

- Deve-se evitar uso de nomes codificados. Também se deve evitar usar o nome da tabela como parte do nome do atributo ex.: S01\_Cod, Cliente\_Cod, Cliente\_Nome.
- Os nomes devem ter um limite de tamanho; normalmente 18 caracteres é o máximo.
- Deve-se dividir o nome do atributo em pelo menos duas partes. Um prefixo identificador, para identificar o tipo do atributo, seguido de um *underscore* (caracter de sublinhado) e uma sequência de caracteres para qualificar o atributo. Veja o Quadro 1.1 com os identificadores sugeridos.

**Quadro 1.1: Sugestão de identificadores dos atributos**

Identificador	Descrição
Aut	Para identificar atributos que são autoincrementados. Normalmente é uma sequência numérica que funciona como chaves primárias na relação. Exemplo: Aut_Pedido; Aut_OS.
Cod	Para identificar atributos que são codificados. Normalmente são chaves primárias na relação. Exemplo: Cod_Disciplina; Cod_Depto.
Dat	Para identificar atributos que representam datas. Exemplo: Dat_Admissao.
Des	Para identificar descrição. Exemplo: Des_Peça; Des_Serviço.
Hor	Para representar um identificador horas. Exemplo: Hor_Inicio.
Idt	Para representar um identificador que possui restrição de domínio de valores. Exemplo: Idt_Situacao; Idt_Sexo. Possui como valores válidos "M" ou "F".
Nro	Para identificar campos numéricos. Exemplo: Nro_Filhos.
Nom	Para identificar nomes. Exemplo: Nom_Cliente; Nom_Cidade.
Txt	Para identificar um texto. Exemplo: Txt_Aviso_Cobranca.
Qtd	Para identificar quantidade. Exemplo: Qtd_Vendida.
Vlr	Para identificar valor ou preço. Exemplo: Vlr_unitario.

Fonte: Elaborado pelo autor

## Resumo

Nesta aula discutimos a importância que os dados têm para um sistema de informação. Na verdade, os sistemas de informação são criados para melhor lidar com os dados existentes nos processos de negócio das organizações. Nesse sentido foram apresentados os conceitos de entidades e de relacionamentos que podem ser representados através de um modelo DER que expressa como os dados se relacionam no “minimundo” que desejamos modelar. Além disso, foram apresentados aspectos de normalização dos dados a fim de minimizar a redundância e a inconsistência dos dados. Finalizamos apresentando a importância da padronização dos nomes das entidades e atributos.

## Atividades de aprendizagem

1. Apresente pelo menos três vantagens dentro do ciclo de desenvolvimento de um *software* que justifique o esforço na elaboração de um modelo de dados bem feito?
2. Explique quais são as características dos atributos simples, compostos multivalorados, derivados e obrigatórios.
3. Explique quais são os tipos possíveis de relacionamento.
4. Defina o que é uma chave candidata, chave primária e chave estrangeira.
5. Conceitue o que é normalização.
6. Identifique quais são as formas normais e explique como se aplica cada uma delas no processo de normalização.
7. Explique com exemplos quais são as boas práticas de padronização dos nomes de tabelas e atributos.

Poste as respostas no ambiente virtual de ensino-aprendizagem (AVEA).

# Aula 2 – Diagrama de Entidade e Relacionamento

## Objetivo

Entender os principais conceitos envolvidos com a construção dos Diagramas de Entidade e Relacionamento.

### 2.1 Introdução

Nesta aula são apresentados os conceitos do Modelo de Entidade e Relacionamento (MER). Serão discutidas as entidades e seus tipos, o grau, a cardinalidade e os tipos dos relacionamentos. Serão tratadas também formas de expressar esses conceitos utilizando desenhos gráficos denominados Diagrama de Entidade e Relacionamento (DER). Serão apresentadas ainda diretrizes que orientam a construção dos DERs, seguidas de exemplos resolvidos e comentados sobre o tema.

### 2.2 Modelo de Entidade e Relacionamento

Para entendermos o Modelo de Entidade e Relacionamento (DER), vamos reforçar alguns conceitos envolvidos. Vamos começar definindo o que vem a ser uma entidade. Segundo Elmasri e Navathe (2005, p. 39) entidade é:

[...] uma entidade ‘algo’ do mundo real com uma existência independente. Uma entidade pode ser um objeto com existência física (por exemplo, uma pessoa, um carro, uma casa ou um funcionário) ou um objeto com uma existência conceitual (por exemplo, uma empresa, um trabalho ou um curso universitário).

Podemos propor uma definição de entidade como um conjunto de dados inter-relacionados que representam um conceito de algo físico ou abstrato que deve ser armazenado e manipulado pelos sistemas de informação. Os dados referidos anteriormente que compõem a entidade são chamados de atributos. Eles descrevem as propriedades ou características das entidades. Como exemplo, uma entidade cliente poderia ter como atributos: o código do cliente, nome, endereço, telefone, etc. São todos dados associados a um cliente específico. Já o conceito de relacionamento, se refere a como ou por meio de quais atributos que as entidades se relacionam.

Peter Chen, em meados da década de 1970, não propôs apenas um modelo conceitual; ele introduziu uma técnica de diagramação a fim de expressar o significado dos dados que ficou conhecida como Diagrama de Entidade e Relacionamento (DER). Embora a grande maioria das formas de representação do DER trabalhem com os mesmos conceitos, entidades e seus relacionamentos, existem diversas representações gráficas com diferente poder semântico de representação. Date (2004) utiliza em sua obra uma representação estendida do DER, em que são mostrados também os atributos das entidades em forma de elipses. Dessa forma, aumenta-se o poder semântico do diagrama; entretanto, aumenta sobremaneira o número de símbolos gráficos a serem representados, tornando o diagrama mais confuso. Na prática, torna-se visualmente poluído quando estamos tratando de um modelo que tem mais que uma dezena de entidades. Normalmente, por mais que se divida o DER por área de negócio, no intuito de reduzir a complexidade, a grande maioria dos sistemas comerciais envolvem centenas ou até milhares de entidades.

## 2.3 Diagrama de Entidade e Relacionamento

O DER é uma representação gráfica do modelo de dados. Devido a sua forma de apresentação visual, através de símbolos, ele é de fácil compreensão, podendo ser utilizado como uma importante ferramenta de trabalho para interação com o usuário. Diante de um DER fica mais fácil modelar o minimundo. Portanto, o DER pode ser utilizado como um importante instrumento na fase de levantamento de requisitos, criando modelos conceituais preliminares que vão se refinando e incorporando novos requisitos. Depois de criado o modelo conceitual de dados, o mapeamento para o esquema de construção física pode ser feito dependendo do nível de detalhe que se chegou de forma quase que direta. Cada entidade irá virar uma tabela implementada no banco de dados. Portanto, o DER é uma ferramenta que transita desde os primórdios do levantamento de requisitos, servindo também como documentação de um sistema já implantado.

Existem no mercado diversas ferramentas CASE – *Computer-Aided Software Engineering* – que permitem trabalhar em nível conceitual, em que a partir do desenho do diagrama seja possível gerar o esquema DDL – *Data Dictionary Language* (conjunto de comandos que permite gerar a construção física da estrutura do banco de dados, criando as tabelas, atributos e as restrições). A maior parte dessas ferramentas permite que a partir do esquema exportado de um banco de dados existente seja possível gerar o desenho de forma automática. A esse processo é dado o nome de engenharia reversa.

Obviamente que, nesse caso, dependendo do número de tabelas, da ferramenta e da versão do *software*, o desenho gerado pode se tornar um pouco ilegível, sendo necessário fazer alguns ajustes na distribuição das tabelas pelo desenho, evitando assim que as linhas dos relacionamentos apareçam cruzadas ou sobrepostas.

### 2.3.1 Entidades

Uma entidade é representada no modelo por um retângulo. O nome da entidade escrita dentro do retângulo deve ser único no contexto. Por exemplo, ao criarmos uma entidade Cliente, ela deverá ser representada conforme mostrado na Figura 2.1 a seguir.



**Figura 2.1: Representação de uma entidade – Cliente**

Fonte: Elaborada pelo autor

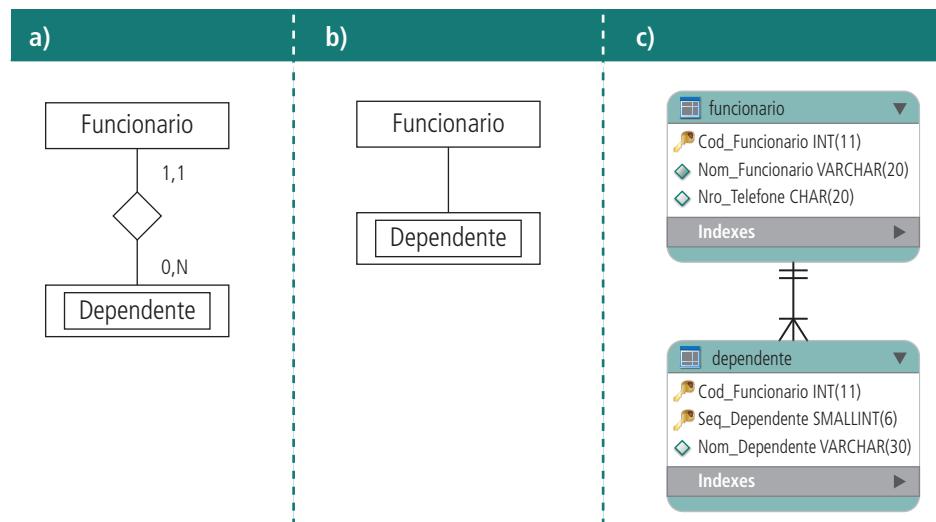
Conforme já foi dito, entidade é um conjunto de dados inter-relacionados que representam um conceito de algo físico ou abstrato que deve ser armazenado e manipulado pelos sistemas de informação. Esses conjuntos de dados inter-relacionados são denominados atributos da entidade. Os atributos são átomos de informação que habitam as entidades. São elementos de dados que conferem identificação e qualificação aos objetos e seus relacionamentos. Na implementação física, os atributos são definidos por nome, tipo/tamanho, obrigatoriedade ou não e por restrição de domínio.

Uma entidade pode ser qualquer conjunto de atributos que represente algo que necessite ser persistido/armazenado no banco de dados, independentemente de ser algo tangível como uma peça, um fornecedor, um cliente, ou intangível como uma conta contábil. Cada entidade deve possuir uma chave de identificação única, chave primária, que é composta por um atributo ou um conjunto mínimo de atributos que identificam uma ocorrência da entidade de forma unívoca. As entidades podem ainda ser classificadas como:

**Entidades fortes** – são aquelas entidades independentes com relação a sua existência de identificação. A formação de sua chave primária é composta por atributos próprios, exclusivos da entidade.

**Entidades fracas** – são aquelas entidades que possuem dependência de

existência em relação a uma entidade forte. A formação de sua chave primária é composta em parte pelos atributos da chave primária da entidade forte com a qual ela está associada, possuindo outros atributos próprios na sua composição. A Figura 2.2 esboça uma representação de uma entidade fraca. A notação de entidade fraca é representada por um retângulo duplo, e o traço que a liga à entidade forte conecta-se ao retângulo mais interno. Essa representação tem um significado semântico expressando que a chave primária da entidade forte é levada para compor a chave primária da entidade fraca. Note que, quando houver outros tipos de relacionamentos, eles são representados ligados ao retângulo mais externo da entidade fraca.



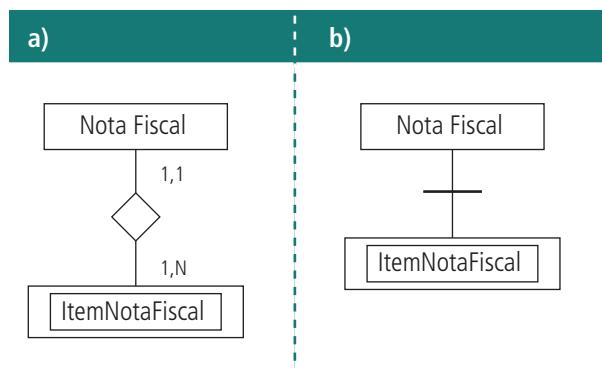
**Figura 2.2: Representação de uma entidade fraca Dependente**

Fonte: Elaborada pelo autor

Como podemos observar, a Figura 2.2 apresenta três formas de representar o relacionamento entre as entidades Funcionário e Dependente. A opção (b) é uma simplificação da opção (a) e a opção (c) gerado pelo MySQL Workbench.

Todo relacionamento que possui uma dependência de existência, a cardinalidade poderá ser de “zero” a “N” (não há obrigatoriedade de existir linhas associadas à tabela dependente) ou de “um” a “N” (sempre haverá pelo menos uma linha na tabela dependente associada a cada linha da tabela entidade forte). Portanto, por questão de simplificação do modelo, usualmente representamos as entidades com dependência de existência, conforme mostrado na opção (b) da Figura 2.2.

Para representarmos a obrigatoriedade de existência, em que ao existir uma ocorrência da entidade forte implica que deve haver pelo menos uma ocorrência na entidade fraca, grafamos um risco conforme mostrado na Figura 2.3, onde a opção (b) é uma simplificação da opção (a).

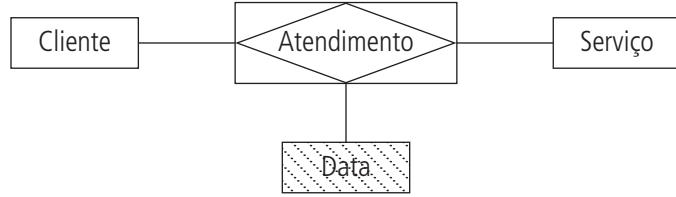


**Figura 2.3: Representação de uma entidade fraca ItemNotaFiscal**

Fonte: Elaborada pelo autor

Na verdade, definir se uma entidade é forte ou fraca está totalmente baseado no contexto semântico no qual se deseja modelar. A princípio, sem conhecer as regras do negócio e a representação que cada entidade tem no mundo real, não é possível preconceber que uma determinada entidade é fraca ou forte. A condição que cada entidade terá depende do minimundo que se deseja modelar.

**Entidades e relacionamentos** – são aquelas criadas para representar relacionamentos entre duas ou mais tabelas. Possuem atributos de identificação das entidades envolvidas nos relacionamentos por elas representados. Podem possuir atributos próprios denominados atributos de interseção ou ter como atributos apenas as chaves primárias das relações envolvidas no relacionamento. Nesse caso é chamada de entidade *all key*. Podem existir alguns casos em que o relacionamento entre as chaves primárias de duas relações podem ocorrer mais de uma vez ao longo do tempo. Nesse caso, o que estamos representando é que está ocorrendo um relacionamento ternário, em que uma das dimensões é a entidade virtual temporal. Essa situação pode ser representada no DER, conforme mostrado na Figura 2.4 e Figura 2.5. Note que os traços (relacionamentos) que se ligam à entidade atendimento o fazem sempre nos vértices do losango interno, significando que esses relacionamentos compõem a chave primária. Os demais relacionamentos, se houver, deverão estar ligados às demais partes do retângulo externo.



**Figura 2.4: Representação da entidade virtual Data, compondo um relacionamento ternário**

Fonte: Elaborada pelo autor



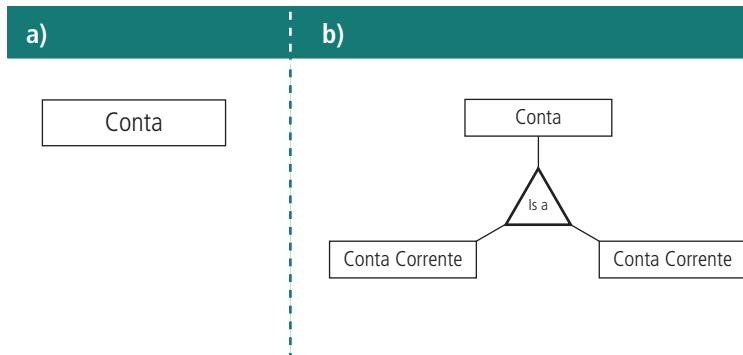
**Figura 2.5: Representação da entidade virtual Data utilizando MySQL WorkBench**

Fonte: Elaborada pelo autor

**Entidade Evento** – são associadas a ciclos temporais de vida e tendem a permanecer na base de dados por um período determinado, após o qual podem ser descartadas ou transferidas para bases de dados históricas. Deve-se observar, entretanto, que esse tipo de categorização a que estamos nos referindo normalmente não faz parte do modelo conceitual de dados. Esse tipo de avaliação da entidade tem muito mais a ver com o seu ciclo de vida depois do sistema implantado. Os puristas nem a consideram, pois partem do princípio que o projeto lógico trabalha com um banco de dados “ideal”, onde não existem problemas de espaço e desempenho. Na prática, não podemos ser tão simplistas separando o lógico do físico. Portanto, ao criarmos o modelo de dados, devemos ter em mente uma ideia de qual serão as características do uso da tabela, tais como: volatilidade; taxa de crescimento diário, mensal, anual; número de acessos, etc., até mesmo porque, ao conhecer tais informações, o analista de sistemas poderá orientar o administrador do banco de dados a definir a melhor forma física da alocação da tabela no SGBD.

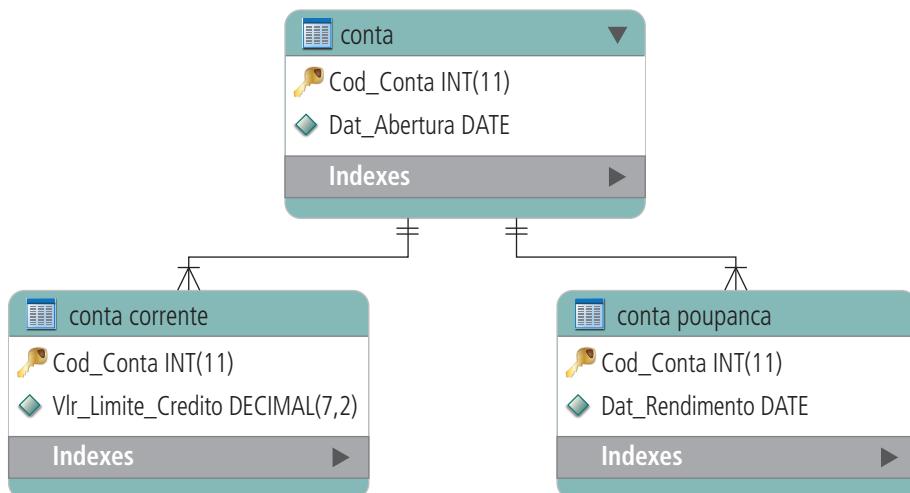
**Entidade de Tipo e Subtipo** – este tipo de entidade está relacionado ao conceito de generalização e especialização. A generalização significa tornar geral, ou seja, agrupar em uma entidade as entidades correlatas que possuem um conceito similar, mas que, entretanto, possuem alguns atributos distintos. A especialização significa exatamente o inverso. Significa separar uma entidade única em duas ou mais entidades específicas. A decisão entre até que ponto se deve generalizar ou especializar é uma decisão de projeto. Quando se opta pela especialização, uma entidade geral contém os atributos comuns e as entidades especializadas contêm os atributos específicos. Veja o exemplo da Figura 2.6 e Figura 2.7. Note que a expressão “is a” dentro

do triângulo, traduzindo para o português, significa “é um”. No exemplo: Conta é uma Conta Corrente ou Conta é uma Conta Poupança.



**Figura 2.6: Representação gráfica de generalização (a) e especialização (b)**

Fonte: Elaborada pelo autor



**Figura 2.7: Representação gráfica de generalização (a) e especialização (b) utilizando MySQL Workbench.**

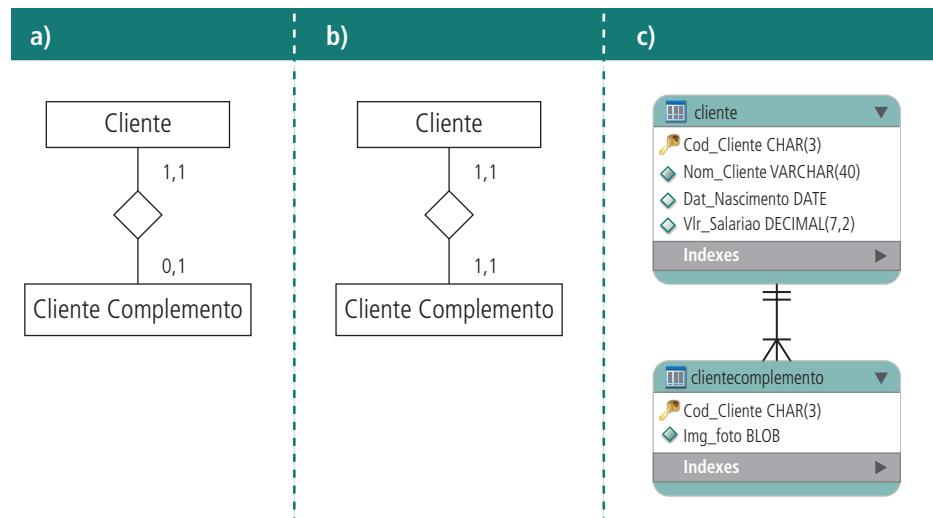
Fonte: Elaborada pelo autor

### 2.3.2 Relacionamento

Definida a entidade, seus atributos e a sua chave primária, o primeiro questionamento que surge é como uma entidade se relaciona com as demais. Essa tarefa está totalmente baseada no contexto semântico no qual se deseja modelar. Em outras palavras, as formas de como as entidades se relacionam dependem de como isso ocorre no contexto que se deseja modelar, pois o DER deve ser a expressão da realidade. Antes de falarmos em relacionamento, é importante entender o conceito de cardinalidade. Cardinalidade de um relacionamento é o número de ocorrências de uma entidade que se relaciona com o número de ocorrências de uma ou mais entidades.

O número de entidades participantes de um relacionamento define o grau do relacionamento. Os relacionamentos poderão ser binários, ternários ou  $n$ -ários. Vejamos inicialmente quais os tipos de relacionamento binários podem ocorrer, considerando duas entidades  $A$  e  $B$ :

**Relacionamento um para um** – significa que para cada ocorrência da entidade  $A$  pode existir de zero a um elementos na entidade  $B$ . Neste caso a chave primária de ambas as entidades é a mesma. Quando se diz que o relacionamento é de zero até um, escreve-se  $(0,1)$  significa dizer que nem toda ocorrência de uma entidade estará relacionada com uma ocorrência da outra entidade. Isto somente irá ocorrer quando a cardinalidade do relacionamento estiver representada por  $(1,1)$ . Como a chave primária de ambas as relações é a mesma, não faz muito sentido implementarmos isso na prática, a não ser por questões de redução de espaço de armazenamento ou de *performance*. Em um relacionamento um para um, quer dizer que as entidades deveriam se tornar uma só. Podemos entender este tipo de relacionamento como um caso particular do um para muitos. Veja na Figura 2.8 que a entidade Cliente é considerada principal e a entidade ClienteComplemento é considerada dependente. Mostra-se em (a) uma cardinalidade em que as ocorrências da entidade ClienteComplemento não existem obrigatoriamente para todas as ocorrências da entidade Cliente; em (b) a obrigatoriedade existe.



**Figura 2.8: Representação gráfica do relacionamento um para um**

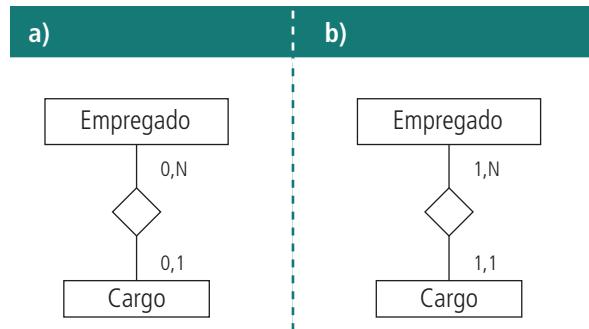
Fonte: Elaborada pelo autor

Na Figura 2.8 podemos ver que o relacionamento está representado pelo losango. Muitos autores inserem dentro do losango um texto que representa

o relacionamento. Nesse exemplo poderíamos usar “possui” ou “tem”. O modelo da letra (a) pode ser lido como: um Cliente possui de zero a um ClienteComplemento. Um ClienteComplemento pertence a um, e somente um, Cliente. No exemplo (b) um Cliente possui um, e somente um, ClienteComplemento. Um ClienteComplemento pertence a um, e somente um, Cliente. Neste texto não adotaremos esta conduta de inserir um texto dentro do losango por achar desnecessária essa representação. Note que a cardinalidade de uma entidade que está sendo considerada sempre está representada no lado oposto da entidade em questão em relação ao losango. O desenho da letra (c) representa a mesma informação, só que utilizando o MySQL Workbench.

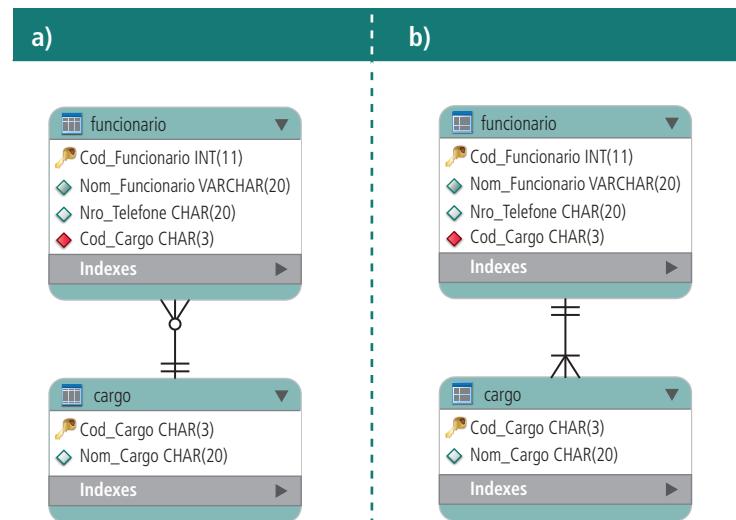
**Relacionamento um para muitos** – significa que, para cada ocorrência da relação A podem existir de zero a muitas ocorrências na relação B. Nesse caso a chave primária da entidade referenciada aparece como chave estrangeira na entidade que faz a referência. Quando se diz que o relacionamento é de zero até N escreve-se (0,N) e isso significa dizer que a chave estrangeira que faz referência a outra entidade referenciada aceita nulo. Portanto, o relacionamento é parcial. Nesse caso nem toda ocorrência de uma entidade possui uma referência para a outra. Entretanto, quando o relacionamento for de um até N, escreve-se (1,N), e isso significa dizer que o relacionamento é total. O atributo chave estrangeira sempre estará preenchido. Veja na Figura 2.9 que a entidade Empregado possui uma referência para a entidade Cargo. Para saber o cargo que um empregado ocupa, temos na entidade Empregado, apenas uma referência às informações do cargo. Como vários empregados podem ter um mesmo cargo, todos eles irão referenciar a mesma ocorrência da entidade Cargo. Caso haja qualquer alteração na descrição ou atribuição, basta acertar as informações em um único lugar, pois a entidade Empregado não possui as informações completas de cargo, possui apenas uma referência sobre onde as informações estão armazenadas.

Na Figura 2.9 mostra-se em (a) uma cardinalidade em que a ocorrência da chave estrangeira na entidade Empregado não é obrigatória, enquanto que em (b) ela é obrigatória.



**Figura 2.9: Representação gráfica do relacionamento um para muitos**  
Fonte: Elaborada pelo autor

Na Figura 2.10 é apresentada a mesma representação utilizando o MySQL Workbench.



**Figura 2.10: Representação gráfica do relacionamento um para muitos**  
Fonte: Elaborada pelo autor

Na Figura 2.9 podemos perceber que o relacionamento está representado pelo losango. No modelo da letra (a) pode ser lido como: um Empregado ocupa nenhum ou um Cargo. Um Cargo é ocupado por nenhum ou até por vários Empregados. No exemplo (b) um Empregado ocupa um e somente um cargo. Um Cargo é ocupado por pelo menos um ou até vários Empregados. Note que, no caso (a), dizermos que um Empregado pode ocupar nenhum cargo significa dizer que o atributo chave estrangeira que faz referência à entidade cargo irá aceitar nulos significando relacionamento parcial. No caso (b), dizermos que um cargo é ocupado por pelo menos um Empregado significa relacionamento total. Isso implica que nenhum Empregado pode ser cadastrado sem informar um cargo para ele. Nesse caso, normalmente primeiro cadastramos todos os cargos existentes na empresa e somente depois é que iremos cadastrar os empregados tornando possível fazermos referências ao cargo. Na Figura 2.10

(a) o fato de a chave estrangeira (cod\_cargo) aceitar nulo é representado por um pequeno círculo desenhado no relacionamento próximo ao cargo, o que indica o relacionamento parcial.

**Relacionamento muitos para muitos** – significa que para cada ocorrência da relação *A* podem existir muitas ocorrências na relação *B* e vice-versa. Nesse caso a chave primária de cada uma das entidades participantes do relacionamento irá compor a chave primária de uma nova entidade que será criada a partir desse relacionamento. Essa nova entidade terá como atributo as chaves estrangeiras, que são as chaves primárias de cada uma das entidades que compõem o relacionamento, considerando que este pode ser *n*-ário. O relacionamento muitos para muitos, escreve-se (*N,M*), é gerado a partir de dois ou mais relacionamentos (1,N). Veja na Figura 2.11 e Figura 2.12.



**Figura 2.11: Representação gráfica do relacionamento muitos para muitos**

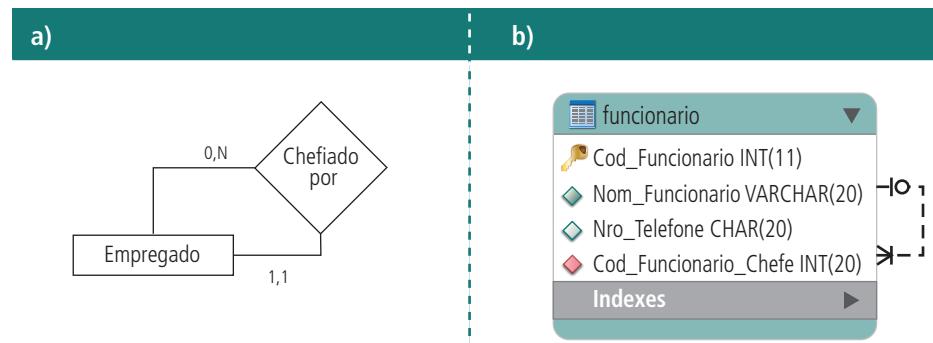
Fonte: Elaborada pelo autor



**Figura 2.12: Representação gráfica do relacionamento muitos para muitos pelo MySQL Workbench**

Fonte: Elaborada pelo autor

**Autorrelacionamento (recursividade)** – significa que uma ocorrência da entidade relaciona-se com uma ocorrência diferente da mesma entidade que pode estar relacionada à de zero a ‘n’ ocorrências. A Figura 2.13 mostra um exemplo da entidade Empregado em que um de seus atributos aponta para a chave primária do empregado que chefia o empregado corrente. Na opção (b) temos a mesma representação implementada pelo MySQL Workbench.

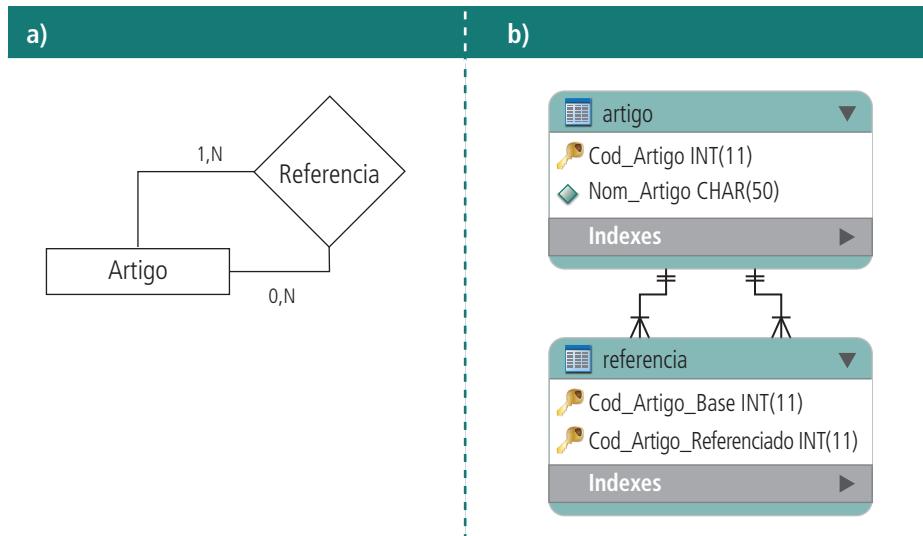


**Figura 2.13: Representação gráfica do autorrelacionamento da entidade Empregado**

Fonte: Elaborada pelo autor

Note que se lê: um empregado é chefiado por de zero a um chefe. Um empregado chefia de zero a “n” empregados. Nesse caso optamos em escrever um texto dentro do losango a fim de deixar mais claro qual é o autorrelacionamento que se deseja representar.

**Autorrelacionamento muitos para muitos (recursividade)** – significa que uma ocorrência da entidade relaciona-se com muitas ocorrências diferentes da mesma entidade, que podem estar relacionadas a muitas outras ocorrências. Vejamos um exemplo que ilustra bem essa necessidade. Vamos pensar em uma situação em que fosse necessário montar uma base de dados com as publicações científicas. Como poderemos representar em um modelo de dados que registre que um artigo faz referência a outros artigos e ao mesmo tempo pode ser referenciado por de nenhum até ‘n’ outros artigos. A Figura 2.14 mostra um exemplo da entidade artigo e o relacionamento muitos para muitos que está representado pela entidade referencia. Note que se lê: olhando a cardinalidade (1,N), significa dizer que um artigo faz referência de um a vários outros artigos, e a cardinalidade (0,N), significa dizer que um artigo pode ser referenciado por nenhum ou até muitos artigos.



**Figura 2.14: Representação gráfica do autorrelacionamento da entidade Artigo/Referência**  
Fonte: Elaborada pelo autor

### 2.3.3 Diretrizes para construção de um DER

A notação apresentada até aqui é suficiente para construir DERs complexos. Entretanto, durante o desenvolvimento de um sistema, não se deve esperar que o primeiro diagrama desenhado seja a versão final. Pelo contrário, o diagrama, dependendo do grau de complexidade que ele tenta representar, deverá ser refeito e aperfeiçoado diversas vezes no decorrer do projeto, por meio de refinamentos sucessivos. O DER inicial normalmente será criado a partir de entrevistas iniciais com o usuário e a partir do conhecimento prévio da área de negócio à qual está se modelando. Isso, naturalmente, proporcionar-lhe-á uma boa indicação de como identificar os principais objetos e relacionamentos.

Existem diversas formas de representações gráficas para desenhar um DER. Neste texto apresentamos até aqui mais de uma forma. A partir deste ponto, iremos adotar a que nos parece ser aquela que melhor oferece um balançamento adequado entre a capacidade de expressão semântica, simplicidade e rigor formal. Portanto, não adotamos o modelo estendido proposto por Elmasri e Navathe, que representa também os atributos dentro de elipses. No modelo utilizado neste texto, boa parte dessa representatividade se dá através da posição no desenho onde a ligação ocorre.

Utilizamos para complementar o desenho um anexo, que é uma forma de dicionário de dados simplificado no qual são relacionadas todas as entidades e atributos envolvidos identificando as chaves primárias, as chaves estrangeiras e as restrições de domínio existentes. No dicionário de dados, identificam-se os atributos que são chave primária com (#), e os atributos que são a chave estrangeira com (\*). Obviamente que se estivermos utilizando uma ferramenta Case para construção do desenho, o próprio software tem alternativas de mostrar ou ocultar atributos de forma automática e de usar na construção do DER diferentes alternativas de desenho correspondente aos diversos autores.

Desde que tenhamos consistência e formalismo semântico no DER, não existem regras rígidas sobre qual a melhor forma de fazer o desenho. Obviamente que se busca sempre uma melhor clareza visual que facilite o entendimento. Portanto, como boa prática, deve-se colocar as entidades principais no centro da folha, pois isso possibilita maior capacidade de ligações ao seu entorno. Sempre que possível, deve-se subdividir o modelo por área de negócio, diminuindo a quantidade de entidades que necessitam estar em um mesmo modelo. A fim de se evitar o cruzamento de traços, pode-se optar em redundar uma entidade marcando-a com um traço diagonal em seu canto inferior direito, conforme pode ser visto na Figura 2.15 a seguir.



**Figura 2.15: Representação de uma entidade duplicada no DER**

Fonte: Elaborada pelo autor



Diz-se que a chave primária da entidade do lado que tem “0” ou “1” vai como chave estrangeira para o lado que tem “N”.

## Resumo

Nesta aula foram apresentados os principais conceitos de entidades (fortes e fracas) e de relacionamentos (um para um, um para muitos, muitos para muitos, autorrelacionamento e especialização/generalização dos relacionamentos). Todo esse arcabouço teórico é que dá a sustentação ao processo de modelagem e representação semântica dos dados em um banco de dados relacional.



Para mais informações sobre a forma de representação de Elmasri e Navathe, veja o material em PowerPoint disponibilizado no ambiente virtual de ensino-aprendizagem (AVEA)

## Atividades de aprendizagem

Considere que as tabelas mostradas na Figura 2.16 representam duas relações de um BD de um consultório médico. Utilize essas informações para responder às questões a seguir.

Convenio		Paciente		
NumConvenio	Nome	numProntuario	nomPaciente	NumConvenio
1	Unimed	234342	Filipe	1
2	Casu	437639	Robson	1
3	Golden Cross	543535	Natalia	2
		643982	Tereza	1
		756635	Debora	3

**Figura 2.16: Representação das entidades Convenio e Paciente**

Fonte: Elaborada pelo autor

1. Faça um desenho DER que represente as tabelas mostradas.
2. Ao tentar realizar as seguintes operações de atualização no BD, verifique se ocorre violação de algum tipo de integridade. Assinale a opção correta e indique qual restrição é violada (no caso em que houver).
  - a) Inserir a seguinte linha na tabela Paciente: <**466655, Antônio Souza, 2**> viola algum tipo de restrição? Sim ou não? Se sim, qual?
  - b) Inserir a seguinte linha na relação Paciente: <**843981, Maria Fontes, 5**> viola algum tipo de restrição? Sim ou não? Se sim, qual?
  - c) Inserir a seguinte linha na tabela Convenio: <**3, Bradesco Saúde**> viola algum tipo de restrição? Sim ou não? Se sim, qual?
  - d) Inserir a seguinte linha na relação Paciente: <**null, Emilia Faria Soares, 2**> viola algum tipo de restrição? Sim ou não? Se sim, qual?
3. Com base na relação Paciente, responda aos seguintes itens:
  - a) Quais são seus atributos?
  - b) Qual é a chave primária?
  - c) Qual é a chave estrangeira?

**4.** Julgue as alternativas assinalando V (verdadeiro) ou F (falso).

- a)**  Uma tabela pode ter atributos com o mesmo nome.
- b)**  Todos os valores de uma coluna são do mesmo tipo de dados.

Poste suas respostas no ambiente virtual de ensino-aprendizagem (AVEA).

# Aula 3 – Diagrama de Entidade e Relacionamento – casos práticos

## Objetivo

Entender os principais aspectos da modelagem de dados e elaboração dos DERs.

Nesta aula iremos praticar os conceitos até aqui apresentados. Portanto, trabalharemos com exercícios resolvidos e comentados. Em todos os problemas apresentados considere que você é um analista de sistemas que trabalha em uma empresa desenvolvedora de *software*. Você foi escalado para visitar um cliente em potencial e terá de entrevistar pessoas ligadas à empresa contratante. A partir das informações obtidas, você irá criar um DER e mostrar quais serão as tabelas e os atributos necessários para armazenar informações em um banco de dados. Os dados dessa entrevista, na prática, serão obtidos através do texto que explica cada problema.

Os dados lhe darão uma base para entender a dificuldade do problema, a fim de que você possa apresentar um orçamento para o cliente. Mas, essa é uma outra história...

Antes de iniciar, faremos alguns comentários sobre esse tipo de situação que, na prática, podemos chamar de **levantamento de dados ou levantamento de requisitos**. Sempre ao entrevistar pessoas para obter informações sobre o funcionamento de algum processo ou necessidade do cliente, é muito difícil conseguir as informações de forma precisa. Normalmente, esse é um processo de refinamentos sucessivos, em que normalmente é necessário recorrer aos entrevistados mais de uma vez para coletar dados adicionais que não foram bem esclarecidos na primeira entrevista. Isso é uma questão de prática profissional: quanto mais informações formos capazes de captar e registrar a cada vez, melhor será.



Para resolver os exercícios propostos, iremos nos basear no texto fornecido. Propositalmente, os textos apresentados aqui são às vezes ambíguos e incompletos. Isso é uma forma de nos aproximarmos da realidade.

Cabe aqui observar que a sequência de passos utilizada para solucionar os problemas não é obrigatória ser feita dessa forma, seguindo uma ordem passo a passo; o que importa é apenas o resultado final: o DER e o esquema das tabelas ou dicionário de dados. O esquema de solução apresentado aqui é apenas uma tentativa didática de reproduzir a elaboração mental envolvida no processo de modelagem de dados.

### 3.1 Primeiro problema

Deseja-se representar a hierarquia da estrutura funcional dos empregados de uma empresa. Sabe-se que um empregado está sempre lotado em um departamento/divisão/seção. Uma seção pertence a uma única divisão. Uma divisão pertence a um único departamento. Enquanto um departamento pode ter várias divisões e uma divisão pode ter várias seções.

#### Solução

**1º Passo** – Inicialmente devemos identificar quais são as entidades que devemos criar.

Ao ler o texto relacionado ao problema, verificamos que as informações importantes para representar a hierarquia da estrutura funcional da empresa são: empregado, departamento, divisão e seção. Observação: empresa não precisa ser armazenada, pois o minimundo que está sendo representado já é a própria empresa.

**2º Passo** –Identificar as chaves candidatas e os atributos de cada entidade.

Identificadas as entidades, temos que buscar quais são as informações que se deseja persistir. Em outras palavras, quais dados são importantes para a empresa e que, portanto, devem ser armazenados. Obviamente que definir com exatidão quais são esses dados é uma questão que depende de quais serão as funcionalidades que o sistema da empresa terá de desempenhar. É importante destacar que na solução dos nossos exercícios hipotéticos iremos arbitrar alguns atributos principais que podem ser elencados. Portanto, esse tipo de problema com que estamos lidando não tem uma única solução, pois mais de uma alternativa pode estar correta. A Figura 3.1 apresenta uma proposta de solução.

Empregado	Departamento	Divisão	Seção
Código Empregado	Código Departamento	Código Divisão	Código Seção
Nome do Empregado	Nome do Departamento	Nome da Divisão	Nome da Seção

**Figura 3.1: Representação das entidades Empregado/Departamento/Divisão/Seção**

Fonte: Elaborada pelo autor

**3º Passo** – Verificar as três formas normais.

1<sup>a</sup> FN – Eliminar subgrupos repetitivos, decompondo a relação em duas ou mais.

As tabelas atendem à 1<sup>a</sup> FN.

2<sup>a</sup> FN – Separar os atributos que dependem de um subconjunto da chave, decompondo a relação em duas ou mais.

As tabelas atendem à 2<sup>a</sup> FN.

3<sup>a</sup> FN – Separar os atributos que dependem de um atributo não pertencente à chave, decompondo a relação em duas ou mais.

As tabelas atendem à 3<sup>a</sup> FN.

**4º Passo** – Padronizar os nomes das entidades e dos atributos, conforme mostrado na Figura 3.2.

Empregado	Departamento	Divisao	Secao
Nro_Empregado	Cod_Departamento	Cod_Divisao	Cod_Secao
Nom_empregado	Nom_Departamento	Nom_Divisao	Nom_Secao

**Figura 3.2: Representação das entidades com nomes padronizados**

Fonte: Elaborada pelo autor

**5º Passo** – Avaliar as chaves candidatas, primárias e os relacionamentos.

Para cada uma das quatro entidades foi definido um atributo com objetivo de identificar uma única ocorrência na tabela. Para Empregado, o nro\_empregado representa o número da matrícula do empregado na empresa, que é único para cada empregado. Nas demais entidades foi criado um atributo código para cada uma delas. Como não existe nenhuma chave natural para elas, definiu-se um código único. Entretanto, o problema aponta que uma seção pertence a uma única divisão. Uma divisão pertence a um único departamento. Enquanto que um departamento pode ter várias divisões e

uma divisão pode ter várias seções. Ou seja, existe uma hierarquia de departamento que contém divisão, que por sua vez contém seções. Portanto, a chave primária deverá ser passada da entidade do topo da hierarquia para a entidade dependente. A Figura 3.3 apresenta as entidades com a chaves primárias representadas pelo símbolo (#) antes do nome dos atributos.

<b>Empregado</b>	<b>Departamento</b>	<b>Divisao</b>	<b>Secao</b>
#Nro_Empregado Nom_Empregado	#Cod_Departamento Nom_ Departamento	#Cod_Departamento #Cod_Divisao Nom_Divisao	#Cod_Departamento #Cod_Divisao #Cod_Secao Nom_Secao

**Figura 3.3: Representação das entidades com definição das chaves primárias**

Fonte: Elaborada pelo autor

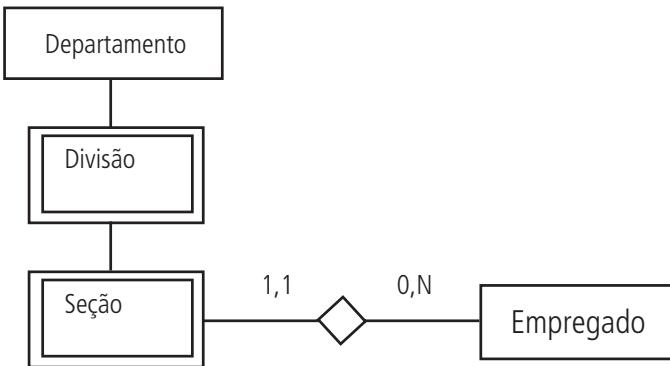
Entretanto, o problema também diz que um empregado está sempre lotado em um departamento/divisão/seção. Para que possamos persistir essa informação no banco de dados, é necessário que tenhamos a chave primária da Seção em Empregado, pois essa é a forma de cada ocorrência de Empregado apontar para o Departamento/Divisão/Seção que está lotado. Portanto, temos a chave primária de Seção como chave estrangeira em Empregado. A Figura 3.4 apresenta um esboço final da definição das tabelas, onde o símbolo (\*) representa os atributos que são chave estrangeira.

<b>Empregado</b>	<b>Departamento</b>	<b>Divisao</b>	<b>Secao</b>
#Nro_Empregado Nom_empregado *Cod_Departamento *Cod_Divisao *Cod_Secao	#Cod_Departamento Nom_Departamento	#Cod_Departamento #Cod_Divisao Nom_Divisao	#Cod_Departamento #Cod_Divisao #Cod_Secao Nom_Secao

**Figura 3.4: Representação das entidades com definição das chaves primárias e estrangeiras**

Fonte: Elaborada pelo autor

A seguir apresentamos, na Figura 3.5, a solução proposta, esboçada através do DER.



**Figura 3.5: Representação das entidades no DER**

Fonte: Elaborada pelo autor

### Comentários:

Como existe uma dependência de existência da entidade Setor para Divisão e dessa para Departamento, a chave primária de Departamento é passada para Divisão e a chave primária de Divisão é passada para Seção. Note que essa situação é mostrada no desenho através do traço (relacionamento) que sai de Departamento e entra até o retângulo mais interno de Divisão, o mesmo ocorrendo de Divisão para Seção. Como nesse caso, um funcionário tem de estar lotado em um departamento/divisão/seção, cada ocorrência de funcionário tem de indicar tal fato. Observando a cardinalidade do relacionamento, diz-se “a chave primária do lado que tem ‘1’ vai para o lado que tem ‘N’ ”. Portanto, a chave primária de Seção vai para Empregado.

## 3.2 Segundo problema

Deseja-se registrar as informações das vendas de uma empresa. Deve-se criar uma nota fiscal com vários itens, sendo que cada item está associado a um único produto. Sabe-se também que uma nota fiscal pertence a um único cliente.

### Solução

**1º Passo** – Inicialmente devemos identificar quais são as entidades que devemos criar.

Ao ler o texto relacionado ao problema, verificamos que as informações importantes para representar as vendas da empresa são: Nota Fiscal e seus itens, Cliente. Na verdade, cada item da Nota Fiscal está relacionado a um produto que está sendo vendido.

**2º Passo** – Identificar as chaves candidatas e os atributos de cada entidade. A Figura 3.6 apresenta um esboço desses dados

Empregado	Departamento
Número da Nota	Código do Cliente
Número de Série	Nome do Cliente
Data de Emissão	
Descrição Item 1	
Qtde. Comprada 1	
Preço Unit.	
Descrição Item 2	
Qtde. Comprada 2	
Preço Unit.	

**Figura 3.6: Representação das entidades no DER**

Fonte: Elaborada pelo autor

**3º Passo** – Verificar as três formas normais.

1<sup>a</sup> FN – Eliminar subgrupos repetitivos, decompondo a relação em duas ou mais.

Conforme já sabemos, devemos evitar definição de campos repetitivos. Afinal, quantos itens podem ter em uma Nota Fiscal? Talvez 10 ou 15, um pouco mais ou um pouco menos. Ao definirmos os atributos da forma que aí está (descrição item 1, ..., descrição item N), na prática, estamos criando uma forte restrição ao modelo de dados, pois ele só será capaz de lidar com uma quantidade predefinida de itens. Essa é uma violação da 1<sup>a</sup> FN. Portanto vamos criar uma nova entidade ItemNF com dependência de existência de Nota Fiscal. Ao fazermos dessa forma estaremos associando os itens de uma mesma Nota. A Figura 3.7 apresenta um esboço das tabelas após colocá-las na 1<sup>a</sup>FN.

Nota Fiscal	ItemNF	Cliente
Número da Nota	Número da Nota	Código do Cliente
Número de Série	Número de Série	Nome do Cliente
Data de Emissão	Sequência do Item	
	Descrição do Item	
	Qtde. Comprada	
	Preço Unit.	

**Figura 3.7: Representação das tabelas na 1<sup>a</sup> FN**

Fonte: Elaborada pelo autor

2<sup>a</sup> FN – Separar os atributos que dependem de um subconjunto da chave, decompondo a relação em duas ou mais.

Quanto à 2<sup>a</sup> FN, ao nos atermos ao que está proposto até o momento, podemos perceber que a descrição do item não depende exclusivamente de um item específico. A quantidade comprada, o preço unitário da venda, tudo isso são informações que caracterizam uma venda (um item de uma Nota Fiscal), mas a descrição do item depende, na prática, de uma entidade que contenha todos os itens (produtos) que estão disponíveis para ser vendidos. A Figura 3.8 apresenta um esboço das tabelas após colocá-las na 2<sup>a</sup>FN.

Nota Fiscal	ItemNF	Cliente	Produto
Número da Nota	Número da Nota	Código do Cliente	Código do Produto
Número de Série	Número de Série	Nome do Cliente	Nome do Produto
Data de Emissão	Sequência do Item		Preço de Compra
	Código de Produto		Preço de Venda
	Qtde. Comprada		Qtyde. em Estoque
	Preço Unit.		

**Figura 3.8: Representação das tabelas na 2<sup>a</sup> FN**

Fonte: Elaborada pelo autor

3<sup>a</sup> FN – Separar os atributos que dependem de um atributo não pertencente à chave, decompondo a relação em duas ou mais.

As tabelas atendem à 3<sup>a</sup> FN.

**4º Passo** – Padronizar os nomes das entidades e dos atributos.

NotaFiscal	ItemNF	Cliente	Produto
Nro_NF	Nro_NF	Cod_Cliente	Cod_Produto
Nro_Serie	Nro_Serie	Nom_Cliente	Nom_Produto
Dat_Emissao	Seq_Item		Prc_Compra
	Cod_Produto		Prc_Venda
	Qtd_Compra		Qtd_Estoque
	Prc_Unit		

**Figura 3.9: Representação das tabelas com os nomes padronizados**

Fonte: Elaborada pelo autor

**5º Passo** – Avaliar as chaves candidatas, primárias e os relacionamentos.

Para cada uma das quatro entidades foi definido um atributo com objetivo de identificar uma única ocorrência na tabela. Para NotaFiscal, o Nro\_NF e o Nro\_Serie formam o conjunto mínimo de atributos capaz de identificar uma única ocorrência da tabela, que também pode ser chamado de linha ou tupla, de uma Nota Fiscal. Em ItemNF, por ser dependente de NotaFiscal, recebendo a chave primária completa de NotaFiscal. E, para cada ocorrência de NotaFiscal poderão existir de ‘um’ até ‘N’ ItensNF. Portanto, foi utilizado o atributo

**Código de barras**

É uma representação gráfica de dados numéricos ou alfanuméricos. A decodificação (leitura) dos dados é realizada por um tipo de scanner (leitor de código de barras) que emite um raio vermelho que percorre todas as barras. Onde a barra for escura, a luz é absorvida; onde a barra for clara (espaços), a luz é refletida novamente para o leitor.

Os dados capturados nessa leitura ótica são compreendidos pelo computador, que por sua vez converte-os em letras ou números humano-legíveis (CÓDIGO..., 2011).

Seq\_Item que irá armazenar essa sequência numérica. Nas demais entidades foi criado um atributo código para cada uma delas. Nesses dois últimos casos teremos de pesquisar se existe alguma chave natural para cada uma das entidades. Podemos propor CPF para Cliente e o **código de barras** para Produto.

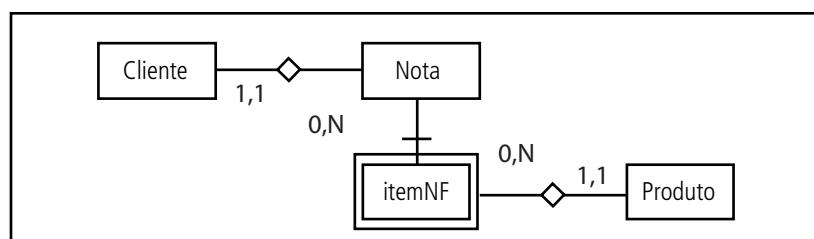
Para identificar qual Cliente está associado a uma NotaFiscal temos de colocar a chave primária do Cliente como chave estrangeira em NotaFiscal. A Figura 3.10 apresenta o resultado final.

NotaFiscal	ItemNF	Cliente	Produto
#Nro_NF	#Nro_NF	#Cod_Cliente	#Cod_Produto
#Nro_Serie	#Nro_Serie	Nom_Cliente	Nom_Produto
Dat_Emissao	#Seq_Item		Prc_Compra
*Cod_Cliente	*Cod_Produto		Prc_Venda
	Qtd_Compra		Qtd_Estoque
	Prc_Unit.		

**Figura 3.10: Representação das tabelas com as chaves primárias e estrangeiras**

Fonte: Elaborada pelo autor

Já a Figura 3.11 apresenta essas mesmas tabelas esboçadas através de um Diagrama de Entidade e Relacionamento (DER).



**Figura 3.11: Representação das tabelas no DER**

Fonte: Elaborada pelo autor

**Comentários:**

**Cliente X NotaFiscal** – A cardinalidade desse relacionamento expressa que um cliente pode ter de zero a “N” Notas Fiscais e que uma Nota Fiscal é de um e somente um cliente. Portanto, a entidade NotaFiscal recebe a chave primária da entidade Cliente, onde vira uma chave estrangeira. Diz-se que a chave primária da entidade do lado que tem “0” ou “1” vai como chave estrangeira para o lado que tem “N”. Se a cardinalidade for igual a “0”, significa que o relacionamento é parcial, não obrigatório. Portanto, o atributo chave estrangeira deverá aceitar valor igual a nulo. Se a cardinalidade for igual a “1” implica em dizer que o atributo chave estrangeira é obrigatório.

**NotaFiscal X ItemNF** – Nesse relacionamento ocorre uma dependência de existência. Isso se dá porque os itens de uma Nota Fiscal, para existirem, dependem, ou seja, estão diretamente ligados ao cabeçalho da Nota Fiscal. Note que, nesse caso, uma Nota Fiscal tem de ter de 1 a “N” itens de Nota Fiscal; portanto, é obrigatória a existência de pelo menos um item, pois não pode haver uma Nota Fiscal vazia sem item. Por isso um risco foi grafado logo acima da entidade ItemNF, expressando essa obrigatoriedade. Podemos ainda, através da representação expressa no modelo sobre as entidades NotaFiscal e ItemNF, saber que a chave primária de NotaFiscal foi levada para compor parte da chave primária de ItemNF.

**ItemNF X Produto** – Este relacionamento expressa que um item de Nota Fiscal tem um e somente um produto e que um produto pode estar de “0” a “N” itens de nota fiscal. A cardinalidade é “0”, pois pode haver produtos cadastrados os quais ainda não foram emitidos em nenhuma Nota Fiscal. Para que no item da Nota Fiscal possa ser identificado o produto, a chave primária de Produto vai para ItemNF como chave estrangeira. Diz-se a chave do lado que tem “1” vai para o lado que tem “N”.

Note que existem três relacionamentos no modelo e, portanto, são três linhas expressas no desenho. Existe uma associação direta entre o que está representado no DER e a forma em que as tabelas foram definidas no dicionário de dados.



### 3.3 Terceiro problema

Uma empresa de venda, por via telefônica, de produtos de beleza deseja cadastrar os seus clientes com as seguintes informações: nome, endereço completo, categoria do cliente, vendedor responsável pelo cliente.

#### Solução

**1º Passo** – Inicialmente devemos identificar quais são as entidade que devemos criar.

Como queremos cadastrar clientes, parece-nos intuitivo pensar em primeiro lugar na entidade Cliente.

## 2º Passo – Identificar as chaves candidatas.

Entre as informações que se deseja persistir propostas pelo problema, nenhuma delas pode identificar de forma única uma ocorrência de cliente. Poderíamos pensar no nome, mas... e os homônimos e as diversas formas de escrever o mesmo nome, com letra maiúscula, com um ou mais espaços, abreviando partes do nome, etc.? Portanto, nesse caso temos de propor um atributo para servir de chave. Aí devemos nos perguntar: será que podemos sempre pedir ao cliente o seu CPF na hora da venda? Essa parece ser uma solução razoável, pois em venda por telefone o pagamento não pode ser à vista, em dinheiro; sendo assim, só podemos vender para pessoas previamente cadastradas. Com o CPF podemos verificar o cadastro do Serviço de Proteção ao Crédito (SPC) para liberar a compra.

## 3º Passo – Identificar os atributos, conforme mostrado na Figura 3.12.

Cliente
CPF
Nome
Endereço completo
Categoria do cliente
Vendedor responsável pelo cliente

**Figura 3.12: Representação dos atributos**

Fonte: Elaborada pelo autor

## 4º Passo – Verificar as três formas normais.

1ª FN – Eliminar subgrupos repetitivos, decompondo a relação em duas ou mais.

A tabela como está atende à 1ª FN.

2ª FN – Separar os atributos que dependem de um subconjunto da chave, decompondo a relação em duas ou mais.

Nesse caso, temos problemas. A categoria do cliente não depende exclusivamente do CPF do cliente. Ou seja, supondo termos as seguintes categorias: Preferencial, Regular, Novo, Bloqueado, podemos ter vários clientes em uma mesma categoria. Observe que o mesmo não acontece com nome e endereço, que pertencem a um CPF. Isso nos leva a concluir que precisamos criar uma entidade Categoria.

O mesmo raciocínio podemos ter para vendedor, ao qual o cliente está associado. Precisamos criar uma entidade Vendedor. A Figura 3.13 apresenta um esboço dessas novas tabelas e atributos identificados.

Cliente	Categoria	Vendedor
CPF	Categoria do Cliente	Vendedor responsável pelo cliente
Nome		
Endereço completo		

**Figura 3.13: Representação das tabelas e atributos**

Fonte: Elaborada pelo autor

3<sup>a</sup> FN – Separar os atributos que dependem de um atributo não pertencente à chave, decompondo a relação em duas ou mais.

Nesse caso está *ok*. O problema é que temos de repetir a verificação das três formas normais a cada nova tabela criada. Ou seja, temos de definir as chaves candidatas e, por sua vez, a chave primária para cada uma das tabelas. No caso de categoria, a qual foi criada para classificar os clientes, definida pela própria empresa, não existe uma chave natural. Portanto, podemos criar um código da categoria definindo-o com uma ou duas posições alfanuméricas. Para Vendedor podemos considerar código do vendedor como o seu número de matrícula registrado pelo Departamento de Pessoal no momento da contratação do funcionário, sendo por exemplo um campo numérico; portanto, podemos alterar as entidades e seus atributos conforme mostrado na Figura 3.14.

Cliente	Categoria	Vendedor
CPF	Código da Categoria	Código do Vendedor
Nome	Categoria do Cliente	Nome do vendedor
Endereço completo		

**Figura 3.14: Representação das tabelas e dos atributos**

Fonte: Elaborada pelo autor

**5º Passo** – Verificar se não existe a necessidade de subdividir algum atributo. Além disso, vamos utilizar o padrão de atribuição de nomes.

Deve-se pensar que o atributo é a menor informação tratada pelo banco de dados. Nesse caso, o endereço completo pode ser dividido em várias subpartes, a fim de organizar melhor os dados. A Figura 3.15 apresenta um maior refinamento das tabelas e atributos identificados.

Cliente	Categoria	Vendedor
#Cod_CPF	#Cod_Categoria	#Cod_Vendedor
Nom_Cliente	Des_Categoria	Nom_Vendedor
Des_Endereço		
Nro_Endereço		
Des_Complemento		
Nom_Bairro		
Nom_Cidade		
Nom_UF (unidade da federação)		
Cod_CEP		
Nro_Telefone		

**Figura 3.15: Representação das tabelas e dos atributos**

Fonte: Elaborada pelo autor

**6º Passo** – Verificar se a decomposição dos atributos não viola alguma das formas normais.

Verificamos, nesse caso, que o nome da cidade e o nome da unidade da federação, o CEP e bairro não dependem exclusivamente do CPF. Sabemos que a partir do CEP podemos encontrar os demais dados; basta ter acesso ao CEP eletrônico dos Correios. Em nosso caso vamos considerar apenas a decomposição de unidade da federação (UF) e Cidade. Como sabemos, uma cidade para existir sempre está dentro de uma unidade da federação. Portanto, nesse caso existe uma dependência de existência de cidade para UF conforme mostrado na Figura 3.16 em que a chave primária de UF vai para a tabela Cidade para compor a sua chave primária.

Cliente	Categoria	Vendedor	UF	Cidade
#Cod_CPF	#Cod_Categoria	#Cod_Vendedor	#Cod_UF	#Cod_UF
Nom_Cliente	Des_Categoria	Nom_Vendedor	Nom_UF	#Cod_Cidade
Des_Endereço				Nom_Cidade
Nro_Endereço				
Des_Complemento				
Nom_Bairro				
Nom_Cidade				
Nom_UF				
Cod_CEP				
Nro_Telefone				

**Figura 3.16: Representação das tabelas e dos atributos após refinamento**

Fonte: Elaborada pelo autor

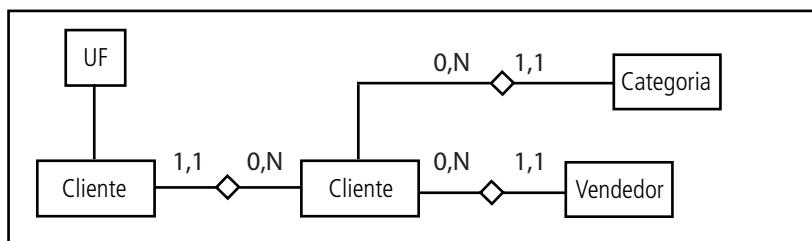
**7º Passo** – Agora temos de definir as chaves estrangeiras a fim de tornar possível acessar os dados referenciados nas tabelas que foram decompostas. Vejamos:

Cliente	Categoria	Vendedor	UF	Cidade
#Cod_CPF	#Cod_Categoria	#Cod_Vendedor	#Cod_UF	#Cod_UF
Nom_Cliente	Des_Categoria	Nom_Vendedor	Nom_UF	#Cod_Cidade
Des_Endereço				Nom_Cidade
Nro_Endereço				
Des_Complemento				
Nom_Bairro				
Cod_Cidade				
*Cod_UF				
*Cod_CEP				
Nro_Telefone				
*Cod_Categoria				
*CodVendedor				

**Figura 3.17: Representação das tabelas e dos atributos com as chaves primária e estrangeira**

Fonte: Elaborada pelo autor

A Figura 3.18 apresenta tabelas e atributos definidos através de uma representação em um DER.



**Figura 3.18: Representação das tabelas no DER**

Fonte: Elaborada pelo autor

## Resumo

Nesta aula foram apresentados três casos práticos em que a partir da descrição de um problema é discutido cada passo necessário para se desenvolver o processo de modelagem de dados a fim de se chegar ao esboço final do Diagrama de Entidade e Relacionamento (DER).

## Atividades de aprendizagem

Desenhe um Diagrama de Entidade e Relacionamento para cada um dos problemas propostos. Poste as respostas no AVEA.

- 1.** Deseja-se cadastrar um cliente com informações sobre: nome, endereço completo, categoria do cliente, vendedor responsável.
- 2.** Crie um modelo para um sistema acadêmico. Sabe-se que um curso (Engenharia de Computação, Ciência da Computação, Sistema de Informação) possui várias disciplinas (nome, horas-aula). Uma disciplina pertence a um determinado departamento e pode atender a vários cursos em um determinado período formando a grade do curso.
- 3.** Complemente o modelo acadêmico anterior, sabendo que uma disciplina de um curso é oferecida em um determinado ano/semestre por um professor. Os alunos (nome, end., dat\_entrada, tipo de entrada) interessados poderão se matricular.

# Aula 4 – Estrutura física

## Objetivo

Entender a estrutura física de um SGBD e controle de transação e concorrência.

### 4.1 Estrutura física

Esta aula apresenta conceitos da estrutura física de um banco de dados e discute quais são as principais partes que compõem um Sistema Gerenciador de Banco de Dados (SGBD). Serão abordados temas como controle de transação e controle de concorrência. Com relação à concorrência, serão abordadas as estratégias otimista e pessimista.

#### 4.1.1 Sistemas de banco de dados

Um sistema de banco de dados tem por finalidade armazenar dados, possibilitando que os usuários possam compartilhá-los, realizando consultas e atualizações de forma segura e garantindo a sua integridade. Independente mente de falhas que porventura ocorram (falta de energia, queda do sistema, erro na aplicação, etc.), qualquer requisição de atualização em um banco de dados só pode ser feita de forma completa. O banco de dados passa de um estado íntegro antes da atualização para outro estado íntegro após a atualização. Se durante a mudança de estado ocorrer uma falha e não for possível finalizar a atualização por completo, o banco retorna sempre ao estado inicial, garantindo assim a integridade dos dados.

O termo “sistema de banco de dados” que estamos tratando aqui está inserido num contexto macroscópico, que pode ser entendido como um conjunto de dados/informações, *hardware*, *software* e usuários.

**Dados/Informações** – Dentro deste texto podemos considerar as palavras dados e informações, como correlatas. Alguns autores fazem distinção entre “dado”, como um valor bruto e atômico, e informação, como o resultado dos dados trabalhados, agregados, composta pela soma das partes. Portanto, definir o significado de dado e informação depende da perspectiva de quem analisa.

**Hardware** – São os componentes físicos que armazenam os dados: a memória secundária de massa, enfim, toda a estrutura de computadores servidores, estações de trabalho e de redes. Toda essa parte, apesar de extremamente importante para o funcionamento de um sistema de banco de dados, não será discutida aqui, pois foge do escopo proposto.

**Software** – O Sistema Gerenciador de Banco de Dados (SGBD) nada mais é do que um *software*. Portanto, um conjunto de programas elaborados por uma empresa ou grupo de pessoas que o torna um produto comercial ou não. Exemplo de SGBDs disponíveis no mercado: DB2, Informix, Oracle, Sybase, MySQL, SqlServer, etc. Em um SGBD, o *engine* é o núcleo principal. Ele é responsável por criar a estrutura física e controlar todo o acesso aos dados. Ele funciona como uma camada entre os dados e os aplicativos de acesso aos dados, que recebem requisições e retornam respostas. Existem ainda, dentro dos pacotes distribuídos pelos produtores de SGBDs, softwares utilitários para administração da estrutura das tabelas, execução de requisições SQL, recuperação e reconstrução de tabelas e índices, importação/exportação dos dados, cópia de segurança, geradores de relatórios de alto nível, aplicativos de *tunning* (análise de desempenho), *drivers* de conexão para acesso aos dados, tanto os nativos quanto os baseados no padrão ODBC – *Open Database Connectivity*.

**Usuários** – São vários os usuários envolvidos diretamente com o sistema de banco de dados.

O Administrador de Banco de Dados (DBA) é o responsável por instalar, configurar e monitorar o funcionamento do SGBD. Normalmente, somente o DBA tem acesso à senha do administrador. Ele é o responsável por garantir o funcionamento de toda a infraestrutura: gestão de segurança (delegar autorização de acesso, controle e definição de regras); e gestão de desempenho (avaliar *log*, comportamento da carga do sistema, estado de segmentação física das tabelas). Com base nessas informações, o DBA pode identificar problemas e, a partir daí, reconfigurar parâmetros, reconstruir índices e/ou reorganizar tabelas, etc.

O Administrador de Dados (AD) é o responsável por validar os modelos de dados criados pelos analistas de aplicação, verificando quais são os dados relevantes da organização e por que prazo ele devem ser mantidos, garantindo assim a integridade e definindo as interdependências entre os diferentes sistemas de aplicação. Essa função, em algumas organizações pode ser delegada ao DBA ou distribuída à responsabilidade entre os próprios analistas de aplicações.

O analista/desenvolvedor de aplicações é o responsável por definir os modelos de dados e por criar a estrutura física das tabelas dentro do banco de dados. Ele é responsável também por desenvolver e manter as aplicações utilizadas para automatizar os processos de negócio das empresas.

O analista de negócio é normalmente um usuário oriundo da área de Sistemas de Informação, com conhecimentos da área de negócio da empresa e visão da estrutura interna do banco de dados e das ferramentas de consulta *ad hoc*. Geralmente este profissional trabalha mais próximo do usuário final provendo informações não rotineiras demandadas extraídas por meio da SQL ou de ferramentas de alto nível de geração de relatórios. Essa função em algumas organizações é realizada pelo próprio usuário final ou pelo analista desenvolvedor, dependendo do grau de dificuldade e da frequência.

De forma indireta ainda existe a necessidade de um profissional de suporte ao sistema operacional, além de outro para infraestrutura de rede.

Para complementar, aparece aquele que é a razão de tudo: o usuário final. É ele que entra com dados e retira as informações necessárias através do uso dos aplicativos desenvolvidos.

### 4.1.2 Vantagens do uso dos SGBDs

Utilizar produtos de SGBDs como forma de repositório de dados em comparação com sistema de arquivos convencionais pode trazer uma série de vantagens. Como regras semânticas que podem ser definidas na estrutura física do banco de dados, isso permite garantir:

- integridade referencial – que é uma amarração da relação de dependência entre os dados de uma tabela em relação a outra;
- restrição de domínio – permite definir qual são os valores válidos que um determinado atributo pode assumir;
- controle de redundância – por meio do uso de *triggers* (gatilhos), mesmo que redundâncias ocorram, é possível definir gatilhos para atuar no momento da atualização de um dado, disparando a propagação automática dos dados redundantes.

Um SGBD possui ainda outras características intrínsecas ao seu conceito, tais como:

**Controle de transação** – faz com que um determinado fato gerador (um evento), que motiva a atualização de uma ou mais tabelas em um banco de dados, seja sempre realizado de forma completa. Ou seja, caso um evento tenha de atualizar mais de uma linha em uma ou mais tabelas, o SGBD sempre se encarregará de fazê-lo de forma completa, garantindo assim a consistência dos dados. Se ocorrerem erros ou falhas durante a atualização e ela não puder ser completada, o SGBD se encarregará de desfazer a parte que porventura já tiver sido atualizada.

**Controle de segurança** – define quais tabelas e/ou atributos um usuário conectado poderá acessar e de que forma ele poderá acessar (somente leitura, permissão de atualização), e se é permitido criar ou alterar estruturas do banco de dados. Normalmente, o controle de segurança possui recursos que permitem controlar a quantidade de acessos simultâneos.

**Independência de dados** – é a imunidade que as aplicações têm quando ocorre uma alteração na estrutura física de uma tabela. Numa aplicação que acessa sistema de arquivos convencionais, uma simples alteração na estrutura de índice de um arquivo ou inclusão de um novo campo leva a uma revisão e compilação dos aplicativos. A ocorrência do mesmo fato em um banco de dados, desde que o atributo acrescentado não seja utilizado pela aplicação, não demanda nenhuma alteração ou recompilação do aplicativo. Uma outra característica relacionada ao conceito de independência de dados em um SGBD possibilita que uma tabela ou um atributo físico possa possuir mais de um nome lógico, a fim de atender a aplicações distintas, compatibilizando diferenças decorrentes de uma migração ou conversão de sistemas.

**Independência de localização dos dados** – num aplicativo que acessa arquivos de dados convencionais, a aplicação tem de conhecer o local físico onde o arquivo se encontra, para, a partir do *path* (caminho) seguido do nome externo ser possível abrir o arquivo para utilizá-lo. Como uma aplicação se conecta diretamente em uma instância do banco de dados e não aos dados, a aplicação não necessita conhecer o local físico onde os dados estão armazenados. Portanto, a mudança de localização dos dados não demanda alteração nem recompilação da aplicação.

**Arquitetura em níveis** – um banco de dados pode ser dividido em vários níveis. O nível interno está relacionado com a forma de como os dados estão fisicamente armazenados. O nível externo, ou nível lógico do usuário, permite diferentes visões dos dados de um mesmo banco de dados de acordo com o usuário que está acessando. O nível conceitual ou lógico define a visão ou esquema externo (*schema*) de um grupo de usuários afins.

Existem ainda algumas características importantes dos bancos de dados, mas que não os distingue dos sistemas de arquivos convencionais, tendo em vista que ambos implementam essas características:

- garantia de unicidade da chave primária;
- controle de compartilhamento de dados – permite controlar o bloqueio dos dados que estão sendo alterados, evitando as chamadas leituras sujas ou perdas de atualizações.

#### **4.1.3 Principais componentes de um SGBD**

Considerando que o sistema operacional é um programa genérico que visa gerenciar o funcionamento do computador para todo e qualquer tipo de aplicação, na sua essência ele não foi concebido para trabalhar em condições extremas requeridas por um SGBD com alto consumo de memória RAM e de I/O em disco. Pensando em otimizar tais recursos, os fornecedores implementam algoritmos especializados incorporados nos SGBDs. Essas implementações alinham a forma de o computador funcionar com a real necessidade de recursos requeridos, aumentando dessa forma o desempenho das aplicações que acessam os SGBDs.

Como já dissemos, o SGBD é um *software* que funciona como uma camada que isola os dados armazenados das aplicações. De uma forma genérica, um SGBD é subdividido em vários componentes de *software*, sendo cada um deles responsável por parte da tarefa de garantir integridade, segurança, etc. Dependendo da implementação dada pelo fornecedor de um produto de banco de dados, podemos ter um menor ou maior controle do *hardware*. Existem produtos que tomam para si parte das funções do sistema operacional, como gerenciamento de memória e de disco. Os produtos mais simples, ou mesmo produtos de maior porte com versões simplificadas, consideram cada tabela e cada índice como um arquivo alocado de forma dinâmica a partir da estrutura de diretórios do sistema operacional disponibilizada através do seu sistema de arquivos.

Existem SGBDs que realizam pré-alocação de um ou mais arquivos que servirão como uma área física para armazenamento das várias tabelas. Nesse caso o sistema operacional aloca um “arquivão” através de seu sistema de arquivos, que é visto pelo banco de dados como uma área física proprietária capaz de armazenar várias estruturas de dados. O crescimento desse “arquivão” pode ocorrer de forma dinâmica, como arquivo comum, ou através de requisições de novas pré-alocações feitas normalmente pelo DBA. Existem ainda produtos que têm a capacidade de definir áreas físicas para o banco de

dados fora da área do sistema de arquivos gerido pelo sistema operacional. Para isso utilizam partições de disco cruas, não formatadas, como sistemas de arquivos. Independentemente da forma, todas essas áreas funcionam como um espaço magnético de armazenamento permanente dos dados, metadados e demais estruturas demandadas, tais como *logs* para controle de transações e registro dos estados consistentes dos dados. A diferença está no desempenho obtido. Para exemplificar, se compararmos um SGBD que para cada acesso a uma tabela precisa solicitar ao sistema operacional um *handle* para acessá-la, isso demanda muito mais tempo na recuperação de uma informação em comparação com um banco de dados que utiliza um “arquivão” que fica disponibilizado durante todo o tempo para acessar as várias tabelas. Seguindo essa linha de raciocínio, os SGBDs que gerenciam a sua própria área de disco não precisam requisitar, a cada acesso, nada ao sistema operacional; portanto, são a forma mais eficiente.

No tocante ao gerenciamento de memória, os produtos de SGBDs funcionam através de um programa gerenciador em tempo de execução (*database engine*), que é carregado para a memória no momento em que uma instância do SGBD é inicializada. Dependendo do produto, a alocação de memória para processamento pode ser fixada de forma residente. Dessa forma, o gerenciamento de memória do sistema operacional que permite retirar (*swap out*) páginas que foram instanciadas pelo SGBD a partir de critérios próprios fica inoperante. O próprio produto SGBD, a partir da memória RAM alocada, cria um *buffer* de páginas e gerencia através de algoritmo proprietário quais páginas devem permanecer e quais devem ser liberadas.

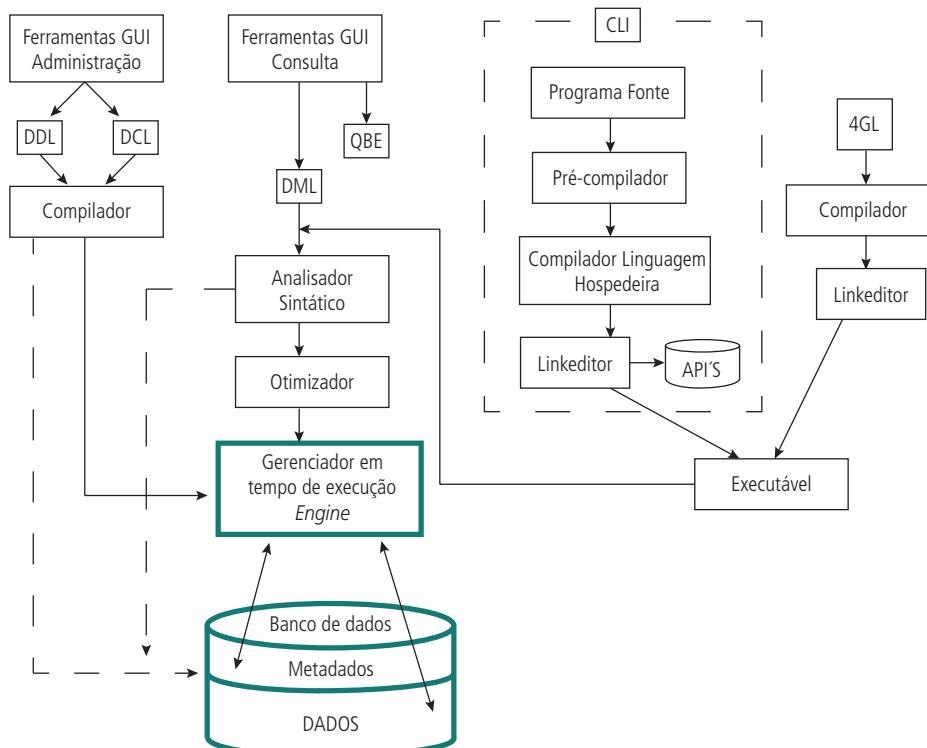
Outro recurso que os produtos costumam transferir para si é o controle de bloqueio de linhas das tabelas que estão sendo atualizadas, a fim de gerir o acesso compartilhado das informações. Alguns produtos não utilizam os semáforos implementados pelo sistema operacional. Porém, criam sua própria lista de controle de bloqueios, que normalmente pode ser definida com diferentes níveis de granularidade, tais como: por banco de dados; por tabela; por página, por linha, por atributo.

A Figura 4.1 apresenta um modelo genérico e simplificado dos componentes de *software* de um SGBD. A seguir veremos uma explicação sucinta de cada um desses componentes.

**Ferramentas GUI de administração** –As ferramentas de administração com interface de usuário gráfica normalmente acompanham opcionalmente ou não uma versão distribuída pelo fabricante. Elas servem para abstrair as dificuldades em conhecer a sintaxe dos comandos DDL, DCL de administra-

ção da estrutura do banco de dados, além de poder controlar a execução de comandos/utilitários de recuperação de tabelas, importação e exportação de dados, gestão de cópias de segurança, gestão de desempenho e configuração, etc. Esse tipo de ferramenta, através de cliques no *mouse*, monta um comando DDL ou DCL e o submete ao *engine* do banco. Ou então faz uma chamada de sistema para executar um utilitário de administração.

**Ferramentas GUI de consulta interativa** – As ferramentas de consulta interativa com interface de usuário gráficas normalmente acompanham uma versão distribuída pelo fabricante que acessa o banco de dados utilizando *drivers* nativos. Ou então podem ser ferramentas de terceiros que normalmente acessam o banco de dados utilizando o padrão de conectividade ODBC – *Open DataBase Connectivity*. Nesse caso, o usuário digita comandos SQL que são repassados ao *engine* do banco, que em seguida retorna o resultado que pode ser mostrado em forma de uma tabela, ou através de mensagem de sucesso ou falha de execução.



**Figura 4.1: Modelo simplificado dos componentes de um SGBD**

Fonte: Elaborada pelo autor

**CLI (Call Level Interface)** – Os aplicativos de usuários do tipo CLI são programas-fonte escritos em uma linguagem de programação hospedeira (Java, C, Pascal) no qual os comandos SQL estão embutidos. Nesse caso, os comandos SQL podem ser executados de forma dinâmica. Ou seja, eles podem ser formados ou modificados em tempo de execução. Para abstrair do desenvol-

vedor a complexidade de chamada das funções de interface (APIs) do banco de dados, é utilizado um pré-compilador. O pré-compilador recebe como entrada o programa-fonte e substitui os comandos mais simplificados pelas chamadas APIs. O resultado gerado é compilado pelo compilador da linguagem hospedeira. Em seguida ocorre a “*linkedição*” que gera como saída o programa executável.

**4GL (Four Generation Language)**– As linguagens de quarta geração são nativas do SGBD. São linguagens que possuem os comandos para tratar SQL incorporados em sua estrutura. Os usuários desenvolvedores podem escrever aplicativos que geram um código interpretado ou executável.

**Programa executável** – Em ambos os casos anteriores, mesmo após o programa executável estar compilado e livre de erros de sintaxe na linguagem nativa, ao ser executado, podem ocorrer erros de sintaxe SQL. Isso ocorre, pois a sintaxe SQL é verificada apenas em tempo de execução. Os comandos SQL embutidos nos programas executáveis são encaminhados para o analisador sintático, que em seguida verifica a correção do comando, submetendo-o ao SGBD. Dessa forma, o SGBD verifica a sintaxe através dos metadados mantidos no banco de dados, tais como os nomes das tabelas e dos atributos. A sintaxe estando correta, em seguida o comando é encaminhado para o otimizador, que, baseado em informações estatísticas dos metadados, decide qual é a melhor estratégia para acessar fisicamente os dados.

**Engine** – O processador em tempo de execução recebe o comando e retorna o resultado obtido. O resultado pode ser em forma de um *result set*, nos casos em que o comando executado for um *select*, ou um “ok” ou uma mensagem de erro nos demais comandos.

**Banco de dados** – A parte que estamos chamando de banco de dados é a que está armazenada no disco. Ela é composta pelos metadados, também conhecidos como catálogo do banco de dados, e pelos dados propriamente ditos, os quais estão armazenados nas tabelas.

**Instância** – Um SGBD trabalha com arquivos que permitem configurar um ou mais ambientes de banco de dados. Cada um desses ambientes é identificado por um nome de instância. Cada uma das instâncias possui um sistema de gerenciamento independente em termos de memória primária e secundária. Cada instância pode possuir um ou mais bancos de dados. Bancos de dados em instâncias diferentes são ditos distribuídos.

## 4.2 Características do MySQL

Os *storage engines* ( motores de armazenamento) são diferentes formas de armazenamento de dados do MySQL, em que cada uma delas possui características próprias. A escolha do motor de armazenamento a ser utilizado na criação de cada tabela depende da sua finalidade, ou seja, a necessidade do uso ou não de transações, desempenho, quantidade de dados a serem armazenados, etc. A forma de armazenamento é atribuída individualmente para cada tabela. O tipo de *engine* da tabela é declarado através de um comando inserido no fim da cláusula DDL *create table*. A Figura 4.2 apresenta a sintaxe do comando para criação de uma tabela.

```
create table teste  
  (campo1 integer) [type=InnoDB | engine=InnoDB];
```

**Figura 4.2: Sintaxe DDL para criação de uma tabela**

Fonte: Elaborada pelo autor

A seguir descreveremos as formas possíveis de armazenar tabelas permitidas pelo MySQL:

**MyISAM** (*Index Sequencial Access Method*) – Não possui transação.

Inclui as seguintes funcionalidades:

- Portabilidade – as tabelas podem ser transferidas entre sistemas operacionais diferentes.
- Suporte a tabelas grandes – trabalha com tabela de tamanho acima dos 4GB permitidos pela estrutura ISAM. Entretanto, a limitação do máximo tamanho do arquivo pode estar ligada ao sistema operacional = 2 GB.
- Otimização do armazenamento – uso mais eficiente da alocação do espaço em disco.
- Limite dos índices – a estrutura ISAM permitia 16 chaves por tabela, sendo o tamanho máximo da chave de 256 bytes. No MyISAM é possível criar até 64 chaves por tabela, sendo que o limite máximo do tamanho da chave foi extendido para 1024 bytes.

**InnoDB** – é um mecanismo de armazenamento rápido com transação segura. Possui as seguintes características:

- As tabelas são alocadas em uma *tablespace*.
- Trabalha com *lock* por linha da tabela.
- Dá suporte a *foreign key*.
- Aloca mais espaço no armazenamento que o tipo MyISAM.

**Archive** – é usado para armazenar grandes quantidades de dados. Só permite realizar os comandos *select* e *insert*. Os dados armazenados não podem ser removidos ou atualizados. Não trabalha com transação nem *foreign key* (chave estrangeira).

**MRG\_MyISAM (Merge)** – é uma forma de criar tabelas grandes driblando o limite de tamanho máximo de um arquivo do sistema operacional. Ou seja, é uma construção que permite tratar diversas tabelas MyISAM como sendo um tabelão único perante as instruções SQL. É quando se cria uma fragmentação física da tabela por localização geográfica ou faixa de tempo, sendo que logicamente a tabela é única. Semelhante a uma visão que contempla a união de subpartes em tabelas distintas.

**Memory (Heap)** – são tabelas cujo acesso é extremamente rápido, pois elas ficam armazenadas na memória primária, além de usarem indexação por *hash*. Entretanto, não suportam campos *auto-increment*, *blob*, *text*. Seus dados são voláteis.

**CSV** – Armazena os dados em arquivos no formato de texto com os atributos separados por vírgula.

**BLACKHOLE** – A tabela aceita os dados, mas não armazena as informações. É utilizado quando se deseja que uma aplicação distribuída em uma determinada circunstância não produza armazenamento dos dados.

## 4.3 Características das tabelas MySQL

As tabelas MySQL podem ser estáticas, dinâmicas e compactadas. Uma tabela estática é aquela que possui todas as suas linhas com o mesmo tamanho de alocação física. Isso se dá em função de ela só possuir campos de tamanho invariável, tais como: *char* e os numéricos. Enquanto que uma tabela dinâmica possui linhas com alocação física de tamanho variável em função de possuir campos de tamanho variáveis, dependendo do conteúdo armazenado, tais como: *varchar*, *text*, *blob*.

Uma tabela que possui somente campos estáticos, linha com tamanho fixo, leva vantagem no tempo de acesso em relação à dinâmica, pois é mais fácil para o SGBD pesquisar uma determinada linha a partir do índice, sabendo-se que cada linha está a um certo deslocamento vezes o tamanho da linha em relação ao início da tabela. Em contrapartida, as tabelas dinâmicas evitam o desperdício de espaço no armazenamento em disco. Entretanto são bem mais complexas de serem administradas pelo SGBD, pois quando campos de alocação de tamanho variáveis são alterados com informações maiores que as originais, a linha permanecerá armazenada no mesmo local e o espaço adicional requerido será alocado em um fragmento da página em outra parte do arquivo.

Existe também a opção de compactar as tabelas através do programa utilitário MylsamPack. Entretanto, tabelas compactadas são *read only*. O utilitário de recuperação e desfragmentação de tabela é o MylsamChk.

## 4.4 Tipos de Dados MySQL

Os tipos de dados do MySQL se subdividem em três grandes grupos principais:

### I – Tipos numéricos

Os tipos de dados numéricos podem se dividir em dois subgrupos: os que possuem a vírgula flutuante (como os decimais) e os inteiros.

**TinyInt:** é um número inteiro com ou sem sinal. Com sinal, a faixa de valores válidos é desde -128 até 127. Sem sinal, a faixa de valores é de 0 até 255 (1 byte)

**Bit ou Bool:** um número inteiro que pode ser 0 ou 1.

**SmallInt:** número inteiro com ou sem sinal. Com sinal, a faixa de valores válidos é desde -32.768 até 32.767. Sem sinal, a faixa de valores é de 0 até 65.535 (2 bytes)

**MediumInt:** número inteiro com ou sem sinal. Com sinal, a faixa de valores válidos é desde -8.388.608 até 8.388.607. Sem sinal, a faixa de valores é de 0 até 16.777.215 (3 bytes).

**Integer, Int:** número inteiro com ou sem sinal. Com sinal, a faixa de valores válidos é desde -2.147.483.648 até 2.147.483.647. Sem sinal, a faixa de valores é de 0 até 4.294.967.295 (4 bytes).

**BigInt:** número inteiro com ou sem sinal. Com sinal, a faixa de valores válidos é desde -9.223.372.036.854.775.808 até 9.223.372.036.854.775.807. Sem sinal, a faixa de valores é de 0 até 18.446.744.073.709.551.615 (8 bytes).

**Float:** número com vírgula flutuante de precisão simples com cerca de sete números significativos. Os valores válidos vão desde -3,4E+38 até 3,4E+38 (4 bytes).

**Double:** número com vírgula flutuante de dupla precisão com cerca de 15 números significativos. Os valores permitidos vão desde -1,7E+308 até 1,7E+308 (8 bytes).

**Decimal:** número em vírgula flutuante com cada dígito sendo representado em 4 bits. O número é armazenado como uma cadeia. Decimal (M,D), onde M é a quantidade de dígitos total do número e D é o tamanho da parte decimal. ((M+1)/2) bytes. Nesse caso o sinal ocupa 4 bits.

Para definir um atributo sem sinal, no campo "Attribute" escolhe-se a opção UNSIGNED.

## II – Tipos data

**Date:** tipo data, armazena uma data. A faixa de valores vai desde o 1 de janeiro do ano 0 até 31 de dezembro de 9999. Formato ano-mês-dia (3 bytes).

**DateTime:** combinação de data e hora. A faixa de valores vai desde o zero até 31 de dezembro de 9999 às 23 horas, 59 minutos e 59 segundos. O formato de armazenamento é de ano-mês-dia horas:minutos:segundos (8 bytes).

**TimeStamp:** combinação de data e hora. A faixa vai desde o 1º de janeiro de 1970 ao ano 2037 (4 bytes). O formato de armazenamento depende do tamanho do campo. A Figura 4.3 apresenta uma definição do formato armazenado, relacionando-o com a quantidade de *bytes* necessários para armazenar os dados.

Tamanho	Formato
14	AnoMesDiaHoraMinutoSegundo aaaammddhhmmss
12	AnoMesDiaHoraMinutoSegundo aammddhhmmss
8	AnoMesDia aaaammdd
6	AnoMesDia aammdd
4	AnoMes aamm
2	Ano aa

**Figura 4.3: Formatação de um campo de data em relação ao tamanho de bytes alocados**

Fonte: Elaborada pelo autor

**Time:** armazena tempo. O intervalo de tempo vai desde -838 horas, 59 minutos e 59 segundos até 838 horas 59 minutos e 59 segundos. O formato de armazenamento é 'HH:MM:SS' (3 bytes).

**Year:** armazena tempo. O intervalo de tempo vai desde o ano 1901 ao ano 2155. O campo pode ter tamanho 2 ou tamanho 4, dependendo se o ano for armazenado com dois ou quatro algarismos (1 byte).

### III – Tipos de cadeia de caracteres

**Char(n):** armazena uma cadeia de tamanho fixo. A cadeia poderá conter desde 0 até 255 caracteres.

**VarChar(n):** armazena uma cadeia de tamanho variável. A cadeia poderá conter desde 0 até 255 caracteres. Dentro dos tipos de cadeia pode-se distinguir dois subtipos: o tipo *Text* e o tipo *Blob (Binary Large Object)*. A diferença entre um tipo e outro é o tratamento que recebem na hora de ordená-los e compará-los. No tipo *Text* ordena-se sem distinguir as maiúsculas e as minúsculas; e no tipo *Blob*, ordena-se levando-se em conta as maiúsculas e minúsculas.

**TinyText e TinyBlob:** campo com um tamanho máximo de 255 caracteres.

**Blob e Text:** texto com um máximo de 65.535 caracteres.

**MediumBlob e MediumText:** texto com um máximo de 16.777.215 caracteres.

**LongBlob e LongText:** texto com um máximo de 4.294.967.295 caracteres.

**Enum:** campo que pode assumir um dos valores definidos em uma lista específica. O tipo Enum aceita até 65.535 valores diferentes.

**Set:** especifica uma lista de valores dos quais um ou mais desses valores podem ser atribuídos a uma coluna específica. A lista pode ter um máximo de 64 valores. A Figura 4.4 apresenta os tipos de campos *string* e a respectiva quantidade de *bytes* alocados por cada um desses tipos.

Tipo de campo	Tamanho de armazenamento
CHAR(n)	n bytes
VARCHAR(n)	n +1 bytes
TINYBLOB, TINYTEXT	Tamanho+1 bytes
BLOB, TEXT	Tamanho +2 bytes
MEDIUMBLOB, MEDIUMTEXT	Tamanho +3 bytes
LONGBLOB, LONGTEXT	Tamanho +4 bytes
ENUM('value1','value2',...)	1 ou dois bytes, dependendo do número de valores
SET('value1','value2',...)	1, 2, 3, 4 ou 8 bytes, dependendo do número de valores

**Figura 4.4: Formatação de um campo de string em relação ao tamanho de bytes alocados**  
Fonte: Elaborada pelo autor

## 4.5 Controle de transação

Um dos principais benefícios em utilizar um SGBD como repositório de armazenamento dos dados de um aplicativo está na sua capacidade de preservar a integridade dos dados em caso de ocorrência de falhas. O conceito de transação é que permite garantir a manutenção de estados consistentes dos dados. Entende-se por transação uma unidade lógica de trabalho, ou seja, a atualização de um fato gerador no banco de dados pode demandar a atualização de uma ou mais tabelas em uma ou mais linhas. Portanto, o processo de atualização consome um determinado tempo para ser realizado. A transação deve garantir que, ao ser finalizada sem falhas, seja criado um novo estado íntegro ou que, na ocorrência de uma falha durante o processo de atualização, o processamento seja completamente desfeito, retornando o banco de dados para o estado íntegro inicial.

Uma transação pode ser declarada de forma explícita ou implícita. Uma transação é considerada explícita quando o primeiro comando declarado em um bloco de comandos, que visa atualizar o banco de dados, é o que define o início da transação, tipicamente: *begin work* ou *begin transaction*, e o último comando do bloco é o que finaliza a transação, tipicamente: *commit work* ou *end transaction*. Uma transação é considerada implícita quando os comandos de controle de transação não são declarados. Nesse caso, cada comando de forma individualizada estará garantido por uma transação. Considerando um banco de dados num instante  $t_0$  em um estado inicial íntegro, quando ocorre um fato gerador que demanda uma atualização no banco de dados no instante  $t_1$ . O resultado dessa atualização deverá garantir a manutenção da integridade dos dados ao final. Portanto, só existem duas alternativas:

**Atualização completada** – o banco sai de um estado inicial íntegro para outro estado também íntegro, ou;

**Atualização não é completada** – na recuperação da falha, o banco desfaz a parte já atualizada, restaurando o estado íntegro anterior à ocorrência da falha.

Uma transação deve possuir um grupo de propriedades, conhecidas pelo acrônimo ACID, que significa:

**Atomicidade** – uma transação deve ser atômica. Dá-se de forma completa ou é desfeita.

**Correção** – uma transação leva o banco de um estado correto para outro estado correto, podendo passar por pontos intermediários.

**Isolamento** – pode haver várias transações ocorrendo ao mesmo tempo a partir de diferentes processos sendo executados. No entanto, elas devem ser tratadas de maneira isolada uma das outras. Durante uma transação, os dados que já foram parcialmente atualizados podem ser percebidos apenas pelo processo em questão, que disparou a transação.

**Durabilidade** – uma vez concluída a transação, suas atualizações devem permanecer no banco mesmo em caso de falhas posteriores.

Existe também um comando que tem de ser declarado de forma explícita, que desfaz uma transação, tipicamente *rollback work*. Outro dado importante é que transações não podem ser aninhadas. Apenas uma transação por vez pode ser aberta por conexão no SGBD.

Quando o fato gerador de uma aplicação de usuário demandar atualizar uma ou mais tabelas que se encontram em diferentes instâncias de banco de dados, essa transação será controlada por um processo coordenador, o qual deverá garantir que ocorra a confirmação da transação nas duas instâncias, ou que, em caso de falhas de qualquer uma delas, que ambas sejam desfeitas. A esse conceito é dado o nome de transação distribuída ou atualização em duas fases (*two phase commit*).

## 4.6 Controle de concorrência

Normalmente, um SGBD deve garantir que diversos usuários, centenas ou até milhares, consultem e atualizem dados ao mesmo tempo. Cabe ao SGBD controlar os problemas oriundos desses múltiplos acessos concorrentes. Se não houver um mecanismo de controle pelo SGBD, alguns problemas de concorrência poderão ocorrer. Podemos citar alguns casos típicos.

**Perda de atualização** – ocorre quando duas conexões leem a mesma linha de uma tabela e mostram na tela da aplicação de cada um dos usuários. Em seguida, cada usuário realiza a alteração desejada. O usuário que gravar primeiro terá seus dados perdidos quando o segundo usuário gravar, tendo em vista que no momento precedente da leitura o segundo usuário não tinha uma versão dos dados alterados pelo primeiro usuário. A Figura 4.5 esboça o exemplo.

Instante	Conexão 1	Conexão 2
T1	Lê linha	Lê linha
T2	Altera dados	
T3	Atualiza	
T4		Altera dados
T5		Atualiza

**Figura 4.5: Problema da perda de atualização**

Fonte: Elaborada pelo autor

Conforme pode ser visto na Figura 4.5, no instante T5, os dados atualizados pela conexão 1 no instante T3 são perdidos.

**Leitura com transação pendente** – o problema surge se uma conexão tiver permissão de ler e atualizar uma mesma linha que foi atualizada por uma outra transação que ainda estiver aberta. A Figura 4.6 e Figura 4.7 esboçam exemplos.

Instante	Conexão 1	Conexão 2
T1	Inicia transação	
T2	Lê a linha	
T3	Altera dados	
T4	Atualiza	
T5		Lê linha
T6		Rollback

**Figura 4.6: Leitura com transação pendente**

Fonte: Elaborada pelo autor

A conexão 2 se torna dependente de uma atualização sem confirmação no instante T5.

Instante	Conexão 1	Conexão 2
T1		Inicia transação
T2	Lê a linha	Lê a linha
T3		Atualiza
T4	Atualiza	
T5		Rollback

**Figura 4.7: Leitura dependente e perda de atualização**

Fonte: Elaborada pelo autor

No exemplo da Figura 4.7 a situação é ainda pior que a anterior, já que não apenas a conexão 1 se torna dependente de uma alteração ocorrida na conexão 2 no instante T2, mas perde a atualização feita no instante T4, no momento em que a conexão 2, em T5, processa o rollback. O rollback faz com que a linha seja restaurada para o instante T1.

**Problema da análise inconsistente** – ocorre em uma situação em que temos o processo de uma conexão calculando a soma dos valores de um atributo de uma tabela, enquanto que, um outro processo concomitantemente transfere valores entre diferentes linhas desse mesmo atributo. Considere uma tabela com saldos em três contas de número 1, 2 e 3 com os respectivos saldos 20, 30 e 40. Conexão 1 deseja obter a soma dos saldos. Portanto igual a 90. Conexão 2 deseja transferir 15 da conta 3 para a conta 1. A Figura 4.8 esboça o exemplo.

Instante	Conexão 1	Conexão 2
T1	Lê conta 1 e acumula saldo	
T2	Lê conta 2 e acumula saldo	
T3		Atualiza Conta 3 = 40 - 15
T4	Lê conta 3 e acumula saldo	
T5		Atualiza conta 1 = 20 + 15
T6	Mostra Total = 75	

**Figura 4.8: Problema de análise inconsistente**

Fonte: Elaborada pelo autor

O erro ocorreu porque a conexão 1 operou durante um estado inconsistente, no meio de uma transação. Portanto, ela efetuou uma análise inconsistente.

**Conclusão** – como podemos ver, se existirem conexões concorrentes, poderão ocorrer problemas se ambas as conexões quiserem ler ou gravar uma mesma linha de uma tabela. Existem quatro possibilidades, conforme mostrado na Figura 4.9. Considere em cada situação que primeiro é executado o comando da conexão 1 e que somente em seguida é executado o comando da conexão 2.

Conexão 1	Conexão 2	Resultado
Lê	Lê	Não há problemas
Lê	Grava	Pode ocorrer análise inconsistente
Grava	Lê	Leitura suja
Grava	Grava	Gravação suja

**Figura 4.9: Possibilidades de problemas**

Fonte: Elaborada pelo autor

Para resolver esses problemas de concorrência apresentados, os SGBDs implementam um mecanismo de bloqueio que controla o acesso dos dados. Quando a transação *A* precisa garantir que uma linha que ela está processando não seja alterada pela transação *B*, a transação *A* cria um bloqueio na linha, o que irá impedir *B* de alterá-la. Desse modo, a transação *A* será capaz de executar todo o seu processamento tendo certeza que as linhas envolvidas permanecerão em um estado estável.

Basicamente existem duas estratégias adotadas pelos desenvolvedores dos produtos de banco de dados.

**Estratégia pessimista** – considera que se uma aplicação solicitou uma linha com a intenção de fazer uma atualização, as demais aplicações somente poderão ler a linha na versão do estado inicial. As demais aplicações que desejarem atualizar uma linha bloqueada irão aguardar a liberação do bloqueio, que ocorrerá somente com a confirmação ou cancelamento da transação inicial que gerou o bloqueio. Um dos problemas desse tipo de implementação ocorre quando existe um bloqueio gerado por uma intenção de atualização interativa de dados oriunda de uma aplicação que exibe a linha na tela para que o usuário faça as devidas alterações. Nesse caso o tempo de permanência da linha bloqueada depende de fatores externos ao sistema computacional. Se o usuário que prende a linha sair para almoçar, isto pode ocasionar a espera das demais aplicações que necessitam da linha bloqueada. Esse tempo de espera, dependendo do SGBD, pode ser definido de três formas: espera infinitamente até liberar; espera por um tempo previamente definido; não espera e retorna uma mensagem de erro.

**Estratégia otimista** – não trabalha sobre a hipótese que sempre o pior caso irá acontecer. Se uma aplicação solicita uma linha com a intenção de fazer uma atualização, uma cópia da linha original é mantida na memória no contexto da transação em questão. Se uma outra aplicação também solicita a mesma linha com intenção de fazer uma atualização, o mesmo procedimento é tomado. No momento em que uma das conexões confirmar a atualização na tentativa de gravar no banco, uma leitura atualizada dessa mesma linha será feita. A linha recém-lida será verificada com a versão anterior armazenada no contexto da transação. Se não houver diferença entre as versões, a gravação é concluída. Caso contrário será realizada uma comparação entre a versão alterada pela aplicação com a versão salva no contexto da conexão. Através dessa estratégia é possível saber quais foram os atributos que a aplicação alterou. De posse dessa informação, novamente será realizada uma comparação entre a versão da atualização com a salva no contexto da aplicação. Dessa vez, entretanto, serão comparados apenas os atributos que foram alterados pela aplicação. Não havendo diferença, a transação é finalizada; se houver, uma mensagem de erro será retornada alertando da tentativa de alterar dados sujos e a transação não será completada.

## Resumo

Nesta aula foram apresentadas informações sobre os principais papéis desempenhados pelas pessoas que interagem com os SGBDs e as principais vantagens dessa utilização, demonstradas pelo entendimento das partes que os compõem. Foram apresentadas também algumas das características específicas do MySQL, tais como os seus tipos de dados e formas de armazenamento e informações sobre o bloqueio e o controle de concorrência de acesso.

## Atividades de aprendizagem

- 1.** Explique quais são os tipos de usuários que interagem com um SGBD.
- 2.** Cite pelo menos cinco vantagens no uso dos SGBDs.
- 3.** Cite pelo menos cinco tipos de *engine* do MySQL e explique as principais características de cada um deles.
- 4.** O que são os tipos de dados? Dê pelo menos cinco exemplos.
- 5.** Explique o que é uma transação em um SGBD.
- 6.** Explique o significado do termo ACID.
- 7.** Explique para que serve o controle de concorrência em um SGBD.
- 8.** Explique como funcionam as estratégias otimistas e pessimistas de concorrência.
- 9.** Faça o roteiro de prática 1 disponibilizado no arquivo RoteiroBD01.pdf.

Poste as respostas no AVEA.

# Aula 5 – *Strutured Query Language* – SQL

## Objetivo

Entender as diferenças entre sistemas de arquivos convencionais e SGBDs e o funcionamento básico da linguagem SQL.

### 5.1 Persistência de dados

Esta aula começa com uma discussão que compara os impactos na programação causados pelos tipos de repositório de dados: arquivos convencionais *versus* Sistemas Gerenciadores de Banco de Dados. Esses conceitos são importantes para obtenção de eficiência no uso da SQL no desenvolvimento dos aplicativos de usuários. Em seguida apresentamos a sintaxe dos principais comandos SQL utilizados.

### 5.2 Arquivos convencionais *versus* SGBDs

Para entendermos como se dá o processo de recuperação de dados pela SQL, vamos fazer uma comparação com o processo convencional utilizado pelas linguagens de programação que acessam arquivos de dados convencionais. Aqueles que trabalham com programação há mais tempo, devem se lembrar de linguagens tais como: Dbase III, Clipper, Cobol, que vinham com comandos próprios de acesso aos sistemas de arquivos indexados. Nesses casos, quando um programa precisava recuperar dados dos arquivos, o processamento feito era totalmente procedural, pois, só era possível extrair dados de um arquivo de cada vez, registro a registro. Nesse ambiente, normalmente era possível ler um arquivo de forma sequencial ou direta através de um índice. Na leitura indexada, é necessário mover os dados para os campos de um determinado índice a fim de posicionar a partir de que ponto os dados seriam lidos e qual seria a ordenação da leitura. Se fosse necessário associar a leitura de vários arquivos, partia-se de um arquivo principal e a cada registro lido, posicionava-se a chave e o índice desejado e fazia-se a leitura indexada dos demais arquivos. Ou seja, o processo de recuperação de dados é totalmente procedural. Toda a sequência lógica para recuperar as informações tinha de ser escrita pelo programador através da linguagem de programação.

Ao mudarmos a forma de persistir os dados, abandonando os sistemas convencionais de arquivo para os SGBDs, existe aí uma quebra de paradigma na forma de como se deve programar. Infelizmente, esse é um erro comum que alguns desenvolvedores cometem ao usar a SQL. Ao desconhecer a SQL, alguns desenvolvedores evitam o uso de recursos mais complexos como junções *subqueries*, etc. Acabam por escrever códigos procedurais dentro das linguagens de programação tais como Java, Delphi, PHP, C++, etc. Na prática a SQL engana e se mostra extremamente simples, pois utiliza apenas quatro comandos principais; entretanto, a SQL é uma linguagem extremamente rica e exige um forte embasamento formal do modelo relacional, sobre pena de se obter os resultados, porém de forma extremamente onerosa em termos da utilização dos recursos computacionais. Isso normalmente produz sistemas com problemas de *performance* nos tempos de recuperação dos dados. Portanto, não devemos negligenciar a busca pelo correto entendimento dos conceitos envolvidos.

### 5.3 Linguagem de consulta

Na prática, os sistemas gerenciadores de banco de dados relacionais comerciais precisavam de uma linguagem amigável; daí surgiu a SQL, acrônimo para *Structured Query Language*, ou linguagem de consulta estruturada. Na verdade, a SQL é mais que apenas uma linguagem de consulta dos dados do banco de dados, pois ela permite também a atualização dos dados. A SQL foi originalmente criada pela IBM nos anos 1970. Na busca de padronização, ela foi revisada várias vezes: em 1992, quando a versão ficou conhecida como: SQL-92; em 1999, para se tornar a SQL:1999; e a SQL3 em 2003, quando surgiu a versão SQL:2003. Mesmo sendo a SQL padronizada pela ANSI e ISO, existem muitas variações e extensões produzidas pelos diferentes fabricantes dos SGBDs.

A SQL é uma linguagem de pesquisa declarativa para banco de dados relacional de alto nível que teve muitas de suas características originais inspiradas na álgebra relacional. A SQL é dita declarativa pois ela especifica a forma do resultado e não o caminho para chegar a ele. Ela é dividida em três grupos principais de comandos:

- DML (*Data Manipulation Language*) ou linguagem de manipulação de dados;
- DDL (*Data Definition Language*) ou linguagem de definição de dados;
- DCL (*Data Control Language*) ou linguagem de controle de dados.

A DML é a que mais nos interessa, pois é através dela que iremos recuperar e atualizar os dados no banco de dados, através principalmente das linguagens de programação que irão interagir com os SGBDs. Ela trabalha com quatro comandos principais: *select*, *insert*, *delete* e *update*. Na versão 2003 foi incluído comando *replace*. Ele veio para resolver um antigo problema de atualização, em que era necessário executar um *select* a fim verificar: se a linha já existisse, executava-se um *update*, senão, era necessário executar um *insert*. Como a proposta deste texto não é de se tornar um manual de referência da linguagem, iremos apresentar alguns conceitos introdutórios, partindo para uma discussão prática do uso dos comandos.

Os comandos referentes a DDL, utilizados para definir e alterar tabelas, índices, visões, *procedures*, etc. e os DCL, utilizados para gerenciar quais usuários poderão acessar quais dados, não serão vistos aqui, tendo em vista que a maior parte dos produto de SGBDs possui interfaces nas quais se torna possível realizar todas as tarefas de administração sem que seja necessário conhecer a sintaxe desses comandos.

## 5.4 Comandos SQL

Ao analisar a sintaxe dos comandos SQL, considere a expressão entre colchetes como opcional e expressões com “|” (pipe ou barra vertical) como mutuamente exclusivas (uma ou outra).

**Select** – é utilizada para fazer uma consulta no banco de dados. Ao submeter um comando DML, o SGBD analisa a sintaxe, define a melhor estratégia de recuperação dos dados, executa o comando e retorna uma resposta (*result set*). Como resultado produz uma relação, que nada mais é que uma nova tabela com o resultado da consulta. A sintaxe básica do comando *select* é apresentada na Figura 5.1 a seguir.

```
Select [ All | Distinct ] coluna1 [, coluna2]
From tabela1
[Where <condições de seleção> ]
[Order by coluna1,coluna2 [asc | desc] ]
```

**Figura 5.1: Sintaxe do comando select**

Fonte: Elaborada pelo autor

**Lista de colunas** – após o comando *select*, corresponde a quais serão os atributos mostrados como resposta. A cláusula *from* mostra a lista das tabelas envol-

vidas na seleção. A cláusula *where* corresponde ao predicado com as restrições. A cláusula *order by* é um recurso adicional da SQL que faz com que os dados, após serem selecionados, possam ser ordenados de acordo com a(s) coluna(s) informada(s), para em seguida serem apresentados como resultado final.

O comando *select* possui uma enorme quantidade de opções de sintaxe que o torna extremamente poderoso e rico em termos de sua capacidade em produzir resultados. Neste texto vamos apresentar algumas das opções de sintaxe sem a pretensão de esgotar o tema.

**All e distinct** – são comandos opcionais, sendo que o *all* é o *default*. O *distinct* restringe no resultado gerado quaisquer linhas repetidas. Quando o *all* é declarado ou nada é declarado, não haverá restrições em apresentar linhas repetidas na saída.

**As** – é um operador de renomeação do atributo. Altera o nome do atributo gerado na saída. Cada uma das colunas separadas por vírgulas a serem retornadas podem ser renomeadas. Basta inserir após a descrição da coluna o comando *as <nome\_renomeado>*, ou apenas *<nome\_renomeado>*, tendo em vista que a cláusula *as* é opcional. Obs.: o nome utilizado na renomeação do atributo não pode conter espaços em branco. Se desejar fazê-lo, a expressão deverá ser informada entre aspas. A Figura 5.2 apresenta um exemplo.

```
Selec Nro_Nf as NotaFiscal, Vlr_Pago as "Valor Pago"  
from NotaFiscal;
```

```
Select Nro_Nf Nota, Vlr_Pago Total  
from NotaFiscal;
```

**Figura 5.2: Comando select que utiliza um aliás para o nome do atributo**  
Fonte: Elaborada pelo autor

Dentro da lista de colunas que podem ser retornadas no comando *select*, é possível, além dos atributos das tabelas envolvidas na cláusula *from*, definir constantes, que nada mais são que qualquer conteúdo entre aspas, resultado de operações matemáticas, etc. A lista de atributos pode ser utilizada como operando pelos operadores matemáticos e como argumentos de funções de agregação, comparação, etc. É o que veremos a seguir.

## 5.4.1 Operadores aritméticos

Os operadores aritméticos possibilitam realizar expressões aritméticas com os atributos que estão armazenados, retornando o resultado da expressão executada. A Figura 5.3 apresenta uma relação dos operadores aritméticos.

Operador	Descrição
+	Adiciona dois valores numéricos.
-	Subtrai dois valores numéricos.
*	Multiplica dois valores numéricos.
/	Divide dois valores numéricos. Divisão por zero produz resultado nulo.
%	Divide dois valores numéricos. Divisão por zero produz resultado nulo.
( )	Altera a precedência da execução da operação aritmética.

**Figura 5.3: Relação dos operadores aritméticos**

Fonte: Elaborada pelo autor

Pode-se usar a cláusula `as` para atribuir um nome ao resultado de uma expressão. A Figura 5.4 mostra um exemplo.

```
Select Nro_NF, Nro_Item, (Qtd_Comprada * Vlr_Pago) as Subtotal  
from ItemNotaFiscal;
```

**Figura 5.4: Uso da cláusula `as` no comando `select`**

Fonte: Elaborada pelo autor

## 5.4.2 Funções de agregação

As funções agregadas podem ser utilizadas de forma exclusiva na lista de atributos. Nesse caso retornam apenas um única linha como resultado final. Ou, usadas combinadas com atributos comuns originados das tabelas existentes no banco de dados. A Figura 5.5 apresenta uma lista com essas funções.

Função	Descrição
AVG([distinct]expr)	Retorna a média da <code>expr</code> . Considera apenas valores não nulos. O uso do <code>distinct</code> faz com que sejam considerados para cálculo da média apenas os valores distintos.
COUNT(expr)	Conta o número de linhas, valores não nulos, retornados pelo comando <code>select</code> .
COUNT(*)	Conta o número de valores não nulos retornados pelo comando <code>select</code> .
COUNT(distinct expr)	Conta o número de valores não nulos das linhas retornadas que são distintos.
MAX(expr)	Retorna o máximo valor não nulo encontrado na <code>expr</code> .
MIN(expr)	Retorna o mínimo valor não nulo encontrado na <code>expr</code> .

**Figura 5.5: Relação das funções de agregação do comando SQL**

Fonte: Elaborada pelo autor

No caso de a agregação ser adicionada com outros atributos na lista de seleção do comando SQL, o resultado retornado será uma linha para cada conjunto distinto dos atributos da lista, acrescida do resultado retornado pela função agregada aplicada a esse mesmo subconjunto de dados, o qual deverá estar definido de forma explícita na cláusula *group by* do comando *select*. A Figura 5.6 apresenta um exemplo.

```
Select Nro_Nf, sum(Qtd_Comprada * Vlr_Pago) as Total  
from ItemNotaFiscal  
group by Nro_Nf;
```

**Figura 5.6: Comando select com uso da cláusula group by**

Fonte: Elaborada pelo autor

Obs.: Alguns SGBDs aceitam uma sintaxe em que a lista de atributos do *group by*, pode ser informada a partir de números que representam a posição que o(s) campo(s) que está(ão) sendo agrupado(s) se encontra(m) da lista dos atributos que estão sendo selecionados. A Figura 5.7 apresenta um exemplo.

```
Select Nro_Nf, sum(Qtd_Comprada * Vlr_Pago) as Total  
from ItemNotaFiscal  
group by 1  
order by 2 desc;
```

**Figura 5.7: Comando select com uso de número que identifica a posição do atributo**

Nesse caso o resultado será ordenado por valor total de cada nota de forma decrescente.

### 5.4.3 Funções numéricas

As funções numéricas podem utilizar os atributos como argumentos, retornando o resultado dessas operações. A Figura 5.8 apresenta um lista com algumas das funções numéricas.

Função	Descrição
ABS(x)	Retorna o valor absoluto de x.
POW(x, y)	Retorna o resultado de x elevado a y.
RAND([n])	Retorna um valor aleatório de ponto flutuante no intervalo de 0.0 a 1.0. O n é o valor da semente para geração do número. Obs.: o resultado retornado sempre será o mesmo para um dado valor de n.
ROUND(x, d)	Retorna o valor de x, arredondado para d casas decimais. Sendo para d = 0, o valor retornado é um número inteiro.
SQRT(x)	Retorna a raiz quadrada de x.
TRUNCATE(x, d)	Retorna o valor de x truncado para d casas decimais.

**Figura 5.8: Relação das funções numéricas do comando SQL**

Fonte: Elaborada pelo autor

A seguir é apresentado um exemplo do uso da função *Round()* na Figura 5.9 a seguir.

```
Select Nro_NF, Nro_Item, Round(Qtd_Comprada * Vlr_Pago),0 Subtotal
from ItemNotaFiscal;
```

**Figura 5.9: Comando select utilizando função numérica**

Fonte: Elaborada pelo autor

#### 5.4.4 Funções de *string*

As funções de *string* operam nos atributos do tipo *string*, possibilitando realizar nesses atributos algumas alterações visando adequar o resultado que se deseja obter. A Figura 5.10 apresenta um lista com algumas das funções numéricas.

Função	Descrição
ASCII(string)	Retorna o código ASCII do caracter mais à esquerda da <i>string</i> .
CHAR(n1, n2, ...)	Retorna os caracteres correspondentes ao código ASCII.
CHARSET	Retorna o conjunto de caracteres da <i>string</i> fornecida.
COALESCE(expr1,expr2, ...)	Retorna o primeiro elemento não nulo da lista.
LCASE ( <i>string</i> )	Retorna a <i>string</i> com os caracteres convertidos para maiúsculas.
LENGTH( <i>string</i> )	Retorna o número de caracteres da <i>string</i> .
LTRIM( <i>string</i> )	Retorna a <i>string</i> sem os caracteres brancos iniciais.
RTRIM( <i>string</i> )	Retorna a <i>string</i> sem os caracteres brancos finais.
SUBSTRING( <i>string</i> , pos, tam)	Retorna uma parte da <i>string</i> , a partir da posição <i>pos</i> com um tamanho <i>tam</i> .
UCASE( <i>string</i> )	Retorna a <i>string</i> com os caracteres convertidos para maiúsculas.

**Figura 5.10: Relação das funções de *string* do comando SQL**

Fonte: Elaborada pelo autor

A seguir é apresentado um exemplo do uso da função Icase na Figura 5.11.

```
Select Icase(Nom_Cliente) as Nomes  
from Cliente;
```

**Figura 5.11: Comando select utilizando função de string**

Fonte: Elaborada pelo autor

#### 5.4.5 Funções de data

As funções de data operam nos atributos do tipo *date*, *datetime*, *time*, possibilitando realizar algumas alterações nesses atributos visando adequar o resultado de acordo com a necessidade. A Figura 5.12 apresenta um lista com algumas das funções de data.

Função	Descrição
DAY( <i>data</i> )	Retorna o dia do mês de 1 até 31 de uma data.
DAYOFWEEK( <i>data</i> )	Retorna um valor numérico de 1 até 7, correspondendo ao dia da semana de domingo até sábado, respectivamente.
DAYOFYEAR( <i>data</i> )	Retorna o número de dias corridos no ano de uma data. Pode variar de 1 a 366.
MONTH( <i>data</i> )	Retorna o mês de 1 até 12 de uma data.
NOW()	Retorna data e hora correntes em uma <i>string</i> no formato "AAAA-MM-DD HH:MM:SS".
WEEKDAY( <i>data</i> )	Retorna um número 0 até 6 correspondendo ao dia da semana de segunda até domingo respectivamente.
WEEKOFYEAR( <i>data</i> )	Retorna quantas semanas houve, de acordo com o calendário, desde o início do ano até a data passada como argumento. O número de semanas retornado pode variar de 1 até 53.
YEAR( <i>data</i> )	Retorna o ano de uma data com quatro algarismos.

**Figura 5.12: Relação das funções de data do comando SQL**

Fonte: Elaborada pelo autor

A seguir é apresentado um exemplo do uso da função DayOfWeek na Figura 5.13.

```
Select DayOfWeek(Dat_Emissao) as Dia_da_Semana,  
      count(*) as Total_Notas_Emitidas  
  from NotaFiscal  
 group by 1
```

**Figura 5.13: Comando select utilizando função de data**

Fonte: Elaborada pelo autor

## 5.4.6 Funções de comparação

As funções de comparação operam nos atributos possibilitando realizar algumas comparações entre eles antes de retornar o resultado. A Figura 5.14 apresenta um lista com algumas dessas funções.

Função	Descrição
GREATEST(expr1,expr2)	Retorna a maior das duas expressões.
IF(expr1,expr2,expr3)	Se expr1 for verdadeira retorna expr2. Se expr1 for falsa ou nula retorna expr3.
IFNULL(expr1, expr2)	Retorna expr2 se expr1 for nula, senão retorna a expr1.
ISNULL(expr)	Retorna 1 (verdadeiro) se a expr for nula, senão retorna 0 (falso). Isto é necessário, pois comparação de valores nulos usando o símbolo = retorna falso.
LEAST(expr1,expr2)	Retorna a menor das duas expressões.

**Figura 5.14: Relação das funções de comparação do comando SQL**

Fonte: Elaborada pelo autor

A Figura 5.15 apresenta um exemplo do uso de função de comparação.

```
Select Cod_Produto,  
      Least(Vlr_Venda, Vlr_Promocional) as Valor_Venda  
  from Produto
```

**Figura 5.15: Comando select utilizando função de comparação**

Fonte: Elaborada pelo autor

## 5.4.7 Comando case

O comando case é utilizado quando queremos testar o conteúdo de um atributo armazenado no banco de dados e retornar como resposta um outro valor. A Figura 5.16 apresentada a sintaxe utilizada.

1º Formato	
CASE expr	Compara expr com um conjunto de possibilidades de valores.
WHEN valor1 THEN Resultado	Onde a comparação for verdadeira, retorna o resultado.
[WHEN valor2 THEN Resultado]	WHEN – Especifica o valor a ser comparado com expr.
[ELSE Resultado]	THEN – Especifica o resultado a ser retornado. ELSE – É selecionado como opção do resultado, caso todas as comparações anteriores forem falsas.
END	
2º Formato	

### 1º Formato

```
CASE  
WHEN condição1 THEN Resultado  
[WHEN condição2 THEN Resultado]  
[ELSE Resultado]  
END
```

Compara as expressões booleanas e retorna o resultado da primeira condição que for verdadeira.

WHEN – Especifica a condição a ser verificada. THEN – Especifica o resultado a ser retornado. ELSE – É selecionado como opção do resultado, caso todas as condições anteriores forem falsas.

**Figura 5.16: Variações de sintaxe do comando case**

Fonte: Elaborada pelo autor

Vejamos um exemplo que retorna a quantidade de Notas Fiscais emitidas a cada dia da semana, apresentado na Figura 5.17 a seguir.

```
Select Case DayOfWeek(Dat_Emissao)  
    when 0 then "Domingo"  
    when 1 then "Segunda"  
    when 2 then "Terça"  
    when 3 then "Quarta"  
    when 4 then "Quinta"  
    when 5 then "Sexta"  
    when 0 then "Sabado"  
End as Dia_da_Semana,  
count(*) as Total_Notas_Emitidas  
from NotaFiscal  
group by 1
```

**Figura 5.17: Exemplo de uso do case**

Fonte: Elaborada pelo autor

### 5.4.8 Cláusula from

Apresentadas algumas das possibilidades de montar a lista de atributos do comando *select*, vamos passar a discutir um pouco sobre a cláusula *from*. A cláusula *from* serve para especificar os nomes das tabelas participantes e a forma de como iremos fazer as junções entre as elas. Existem algumas opções de sintaxe para se fazer a junção entre as tabelas. Pode-se relacionar as tabelas separando-as com vírgula na cláusula *from* e informar os critérios de junção na cláusula *where*, ou usar a cláusula *join*, que substitui a vírgula na lista das tabelas, complementada com a cláusula *on* ou *using* a fim de definir os critérios de junção dentro da própria cláusula *from*. Devemos considerar que ao utilizarmos a junção de forma explícita (*join*), torna mais claro o que é junção, do que é seleção. Vejamos exemplo com diferentes formas de sele-

cionar os mesmos dados utilizando diferentes opções de sintaxe.

**1<sup>a</sup> Opção** – fazendo a junção utilizando a cláusula *where*.

```
Select A.Nro_Nf, A.Nro_Item, B.Nom_produto  
from ItemNotaFiscal A, Produto B  
where B.Cod_Produto = A.Cod_Produto  
and B.Cod_Produto = "001"
```

**Figura 5.18: Exemplo de junção utilizando a cláusula *where***

Fonte: Elaborada pelo autor

**2<sup>a</sup> Opção** – fazendo a junção utilizando a cláusula *on*.

```
Select A.Nro_Nf, A.Nro_Item, B.Nom_produto  
from ItemNotaFiscal A join Produto B  
on B.Cod_Produto = A.Cod_Produto  
where B.Cod_Produto = "001"
```

**Figura 5.19: Exemplo de junção utilizando a cláusula *on***

Fonte: Elaborada pelo autor

**3<sup>a</sup> Opção** – fazendo a junção utilizando a cláusula *using*

```
Select A.Nro_Nf, A.Nro_Item, B.Nom_produto  
from ItemNotaFiscal A join Produto B  
using Cod_Produto  
where B.Cod_Produto = "001"
```

**Figura 5.20: Exemplo de junção utilizando a cláusula *using***

Fonte: Elaborada pelo autor

Obs.: Só é possível utilizar a cláusula *using* quando o nome dos atributos de junção for igual em ambas as tabelas.

Tipos de junção:

- **Join / Cross Join / Inner Join** – retorna linhas quando a junção for possível;
- **Left Join / Left Outer Join** – retorna as linhas da tabela da esquerda mesmo que não seja possível ocorrer uma junção com a tabela da direita;
- **Straight\_Join** – similar ao *join*, exceto que a tabela da esquerda é sempre lida antes que a tabela da direita. É utilizado quando se deseja forçar a forma de execução do otimizador.
- **Where** – especifica uma condição de filtro das linhas que serão retornadas. Nas condições de filtro podem trabalhar com condições combinadas que utilizam os operadores especificados a seguir.

## Operadores lógicos

Uma relação com os operadores lógicos é mostrada na Figura 5.21 a seguir.

Sintaxe	Descrição
Not (expr)	Inverte o valor booleano da expressão. Se for 1 (verdadeiro) retorna falso, se for 0 (falso) retorna verdadeiro. Exceto <i>not null</i> , retorna <i>null</i> .
(expr1) or (expr2)	Retorna verdadeiro se pelo menos uma das expressões forem verdadeiras.
(expr1) and (expr2)	Retorna verdadeiro se ambas as expressões forem verdadeiras.

**Figura 5.21: Relação com operadores lógicos**

Fonte: Elaborada pelo autor

## Operadores de comparação

Uma relação com os operadores de comparação é mostrada na Figura 5.22 a seguir.

Sintaxe	Descrição
(expr1) = (expr2)	Testa a igualdade entre os operandos.
(expr1) <> (expr2)	Testa se os dois operandos são diferentes.
(expr1) > (expr2)	Testa se o primeiro operando é maior que o segundo.
(expr1) >= (expr2)	Testa se o primeiro operando é maior ou igual que o segundo.
(expr1) < (expr2)	Testa se o primeiro operando é menor que o segundo.
(expr1) <= (expr2)	Testa se o primeiro operando é menor ou igual que o segundo.
(expr1) <=>(expr2)	Retorna verdadeiro se os dois operandos forem iguais mesmo que sendo nulos.

Sintaxe	Descrição
Is [Not] null	Testa se o valor é nulo.
(expr) [not] in (v1,v2,...vn)	Testa se o valor corresponde a algum de uma lista de valores.
(expr) between min and max	Testa se o valor da expressão está dentro de um intervalo de valores incluindo os valores limítrofes.

**Figura 5.22: Relação com operadores de comparação**

Fonte: Elaborada pelo autor

Vejamos alguns exemplos na Figuras 5.23 e 5.24 a seguir.

```
Select A.Nro_Nf, A.Nro_Item, B.Nom_produto
from ItemNF A join Produto B
on B.Cod_Produto = A.Cod_Produto
where A.Dat_Emissao > "01/01/2011"
and B.Prc_Venda < 1000;
```

**Figura 5.23: Exemplo do uso dos operadores de comparação**

Fonte: Elaborada pelo autor

```
Select A.Nro_Nf, A.Nro_Item, B.Nom_produto
from ItemNF A join Produto B
on B.Cod_Produto = A.Cod_Produto
where A.Dat_Emissao between "01/01/2011" and "17/03/2011";
```

**Figura 5.24: Exemplo do uso dos operadores de comparação**

Fonte: Elaborada pelo autor

**Having** – é semelhante à cláusula *where*. É uma forma de filtro atribuída apenas para as linhas retornadas pela cláusula *group by*. Vejamos um exemplo em que se deseja selecionar os nomes dos produtos que venderam pelo menos dez itens no primeiro trimestre do ano de 2011, ordenados pelo total vendido de forma descendente. A Figura 5.25 apresenta um exemplo.

```
Select Nom_Produto, count(*)

from ItemNF A join Produto B
on B.Cod_Produto = A.Cod_Produto
where A.Dat_Emissao between "01/01/2011" and "31/03/2011"
group by 1
having count(*) >= 10
order by 2 desc;
```

**Figura 5.25: Exemplo do uso da cláusula having**

Fonte: Elaborada pelo autor

**Insert** – A cláusula *insert* possibilita inserir dados (linha) em uma tabela. Cada comando *insert* inclui linha(s) em uma única tabela de cada vez. Ele retorna somente a confirmação da inserção ou a mensagem de erro ocorrida na tentativa de inserção de uma linha. Todos os campos que são obrigatórios e que não possuírem um valor de inserção *default*, deverão ser informados no momento da inserção. Ou seja, não poderão estar nulos. A sintaxe do comando *insert* é mostrada na Figura 5.26 a seguir.

```
Insert into tabela [coluna1, coluna2] values ( "valor1", "valor2");
```

**Figura 5.26: Sintaxe do comando *insert***

Fonte: Elaborada pelo autor

A Figura 5.27 apresenta um exemplo do uso do comando *insert*.

```
Insert into cliente (cod_cliente, nom_cliente)  
Values ("99988822200" "José da Silva");
```

**Figura 5.27: Exemplo do uso do comando *insert***

Fonte: Elaborada pelo autor

**Resultado:** Insere na tabela um novo cliente.

Obs.: Existe uma forma de combinar um comando *insert* com um *select*, em que todo o *result set* de um comando *select*, ou seja, as várias linhas, possam ser diretamente inseridas em uma tabela. A Figura 5.28 apresenta a sintaxe do comando.

```
Insert into tabela1 [coluna1 , coluna2, colunaN ]  
select [coluna1 , coluna2, colunaN ] from tabela2
```

**Figura 5.28: Sintaxe do comando *insert select***

Fonte: Elaborada pelo autor

A Figura 5.29 apresenta um exemplo.

```
Insert into cliente (cod_cliente, nom_cliente)  
Select cod_funcionario, nom_funcionario from funcionario  
Where vlr_salario > 1000;
```

**Figura 5.29: Exemplo de uso do comando *insert select***

Fonte: Elaborada pelo autor

**Resultado:** Insere na tabela cliente o código e o nome de todos os funcionários da tabela Funcionario cujo salário for maior que 1000.

**Update** – A cláusula *update* possibilita atualizar dados, (colunas), em uma ou mais linhas de uma única tabela. Ela retorna somente a confirmação da atualização ou a mensagem de erro ocorrida na tentativa de atualizar a(s) linha(s). A sintaxe do comando *update* é mostrada na Figura 5.30 a seguir.

```
Update tabela  
set campo1 = "valor1"  
[, campo2 = "valor2"]  
[ where "condições" ]
```

**Figura 5.30: Sintaxe do comando update**

Fonte: Elaborada pelo autor

Obs.: Caso a condição definida na cláusula *where* não selecione nenhuma linha para atualizar, mesmo assim o comando é considerado como executado com sucesso.

Um exemplo do uso do comando *update* é mostrado na Figura 5.31 a seguir.

```
Update produto  
Set prc_venda = prc_compra * 1.5;
```

**Figura 5.31: Exemplo do uso do comando update**

Fonte: Elaborada pelo autor

**Resultado:** Atualiza o preço de venda de todos os produtos existentes na tabela como sendo 50% maior que o preço de compra.

**Delete** – A cláusula *delete* possibilita remover dados da tabela. Ela remove uma ou mais linhas de uma única tabela. Ela retorna somente a confirmação da remoção ou a mensagem de erro ocorrida na tentativa de remover a(s) linha(s). A sintaxe do comando *delete* é mostrada na Figura 5.32 a seguir.

```
Delete from tabela  
[ Where "condições" ]
```

**Figura 5.32: Sintaxe do comando delete**

Fonte: Elaborada pelo autor

Obs.: Caso a condição definida na cláusula *where* não selecione nenhuma linha para remover, mesmo assim o comando é considerado como executado com sucesso.

Um exemplo do uso do comando *delete* é apresentado na Figura 5.33 a seguir.

```
Delete from produto  
Where qtd_estoque = 0;
```

**Figura 5.33: Exemplo do uso do comando *delete***

Fonte: Elaborada pelo autor

**Resultado:** Remove da tabela *produto* todas as linhas que a quantidade em estoque for igual a zero.

## Resumo



Assista ao vídeoAula5.avi  
disponível no ambiente virtual de  
ensino-aprendizagem (AVEA)

Esta aula começou com uma discussão sobre as formas de as aplicações persistirem os dados, através do uso de arquivo *versus* dos SGBDs. Em seguida foram apresentados os principais elementos da sintaxe dos comandos de manipulação de dados (DML).

## Atividades de aprendizagem

Aponte quais são as principais diferenças para uma aplicação em persistir os dados utilizando sistema de arquivos ou um SGBD.

1. Quais são os operadores de comparação e como podem ser usados?
2. O que pode ser colocado na lista de atributos de um comando *select*?
3. Para que servem os comandos *insert*, *update* e *delete*?
4. Qual é a função do comando *where* na SQL?
5. Faça o roteiro de prática 2 disponibilizado no arquivo RoteiroBD02.pdf.

Poste as respostas no AVEA.

# Aula 6 – Visões e restrições

## Objetivo

Entender o conceito de instância de banco de dados, o uso das visões, dos índices e das uniões.

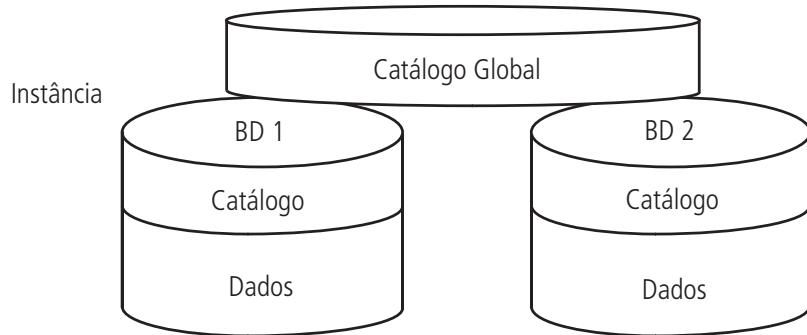
Esta aula apresenta conceitos de implementação de restrições no banco de dados. Esses recursos são importantes porque expressam um conteúdo semântico dos dados. Iremos apresentar os conceitos e discutir quando devemos utilizar recursos tais como visões e restrição de domínio.

## 6.1 Implementando restrições

Antes de entrar no tema título desta aula, vamos introduzir alguns conceitos a fim de esclarecer um pouco mais sobre a forma de como os bancos de dados estão estruturados. Um importante conceito é o de instância de banco de dados. Esse conceito está relacionado com uma cópia do *software* (SGBD) que está sendo executada no computador. Analogamente, podemos pensar em um programa qualquer que pode ser carregado na memória do computador várias vezes. Para cada execução do programa, damos o nome de uma instância de sua execução. Ou seja, cada cópia pode ser executada de forma independente. Para cada instância do SGBD executada na memória, existe uma ou mais áreas em disco, onde os dados propriamente ditos estão armazenados em um ou mais bancos de dados de usuários.

Normalmente, os SGBDs criam, para registrar todas as informações globais de uma instância, um banco de dados especial chamado de catálogo, que possui uma estrutura de tabelas proprietárias conforme o fabricante. Além disso, para cada banco de dados de usuário criado, os SGBDs criam algumas tabelas adicionais, as quais podemos considerar em linhas gerais como um catálogo local interno de cada banco de dados específico. O objetivo do SGBDs ao criar esse banco de dados global e as tabelas locais é de catalogar todas as informações necessárias ao seu gerenciamento, tais como: definição de quais bancos de usuários existem em uma instância, quais usuários estão cadastrados, quais acessos são permitidos aos usuários, quais são as

tabelas e seus atributos, chaves primárias e secundárias, índices, visões, restrições de domínio, procedimentos armazenados, etc. Esses dados registrados nos catálogos global e local são os chamados metadados. A Figura 6.1 mostra uma representação conceitual de instância.



**Figura 6.1: Representação conceitual de instância**

Fonte: Elaborada pelo autor

Fizemos esse breve parêntese para dizer que todo SGBD deve possuir, além dos dados dos usuários persistidos nas tabelas, informações sobre a estrutura dos dados e suas regras de restrições semânticas. As restrições são implementadas em um banco de dados como metadados. Cabe ao SGBD, antes de fazer qualquer manipulação nos dados do usuário, verificar as informações registradas nos metadados a fim de validar se tal manipulação não está violando nenhuma restrição definida previamente. Esse conjunto de regras aumenta a confiabilidade dos dados armazenados, pois permite expressar qual é o significado do dado. Por exemplo, um campo para armazenar a altura de uma pessoa pode ser um decimal (3,2). Isso permite informar pessoas com -9,99 metros até 9,99 metros de altura. Por ser um campo decimal, já existe uma restrição de não aceitar letras. Isso está correto; afinal, a altura é um dado numérico. Entretanto, nem toda a faixa de valores que esse campo pode armazenar faz sentido como representação da altura de uma pessoa. Como sabemos, a altura de uma pessoa tem de ser positiva e varia de alguns centímetros até pouco mais de dois metros. Uma forma de limitar a faixa dos valores válidos de um atributo de uma tabela em um banco de dados é através da criação das restrições de domínio.

Como vimos, normalmente, uma instância de um SGBD mantém parte das informações do catálogo em tabelas específicas de um banco de dados de uso global, complementadas por informações que são pertinentes a cada banco de dados existente. O banco de dados global, ou *master catalog*, é criado automaticamente no momento que se cria uma nova instância do

SGBD. Ele contém informações de uso compartilhado para todos os bancos de dados de uma mesma instância, enquanto que as demais informações necessárias são armazenadas em tabelas específicas definidas no instante da criação de cada banco de dados de usuário. Essas informações dizem respeito exclusivamente a um único banco de dados.

As restrições que estamos discutindo ficam armazenadas dentro das tabelas de catálogo do SGBD. Essas restrições são criadas para garantir a integridade dos dados. Devemos entender integridade, nesse contexto, no sentido amplo de consistência da informação. Lembre o exemplo da altura citado anteriormente. Basicamente existem dois tipos de integridade em um banco de dados:

- declarativa – utiliza restrições (*constraints*) e regras (*rules*);
- procedural – utiliza *stored procedures* (procedimentos armazenados) e *triggers* (gatilhos).

Para implementar a integridade declarativa de dados, temos os elementos chamados de *constraints* (restrições). Essas *constraints* devem ser criadas utilizando as instruções de criação e alteração de tabela CREATE TABLE e ALTER TABLE, respectivamente. Esse tipo de restrição é dita declarativa, pois, ela é registrada no catálogo do banco através da declaração dos comandos. Em geral os tipos de restrições que podem ser implementados em um banco de dados são mostrados na Figura 6.1.

Para implementar a integridade procedural, em vez declarar cláusulas que criam restrições semânticas, temos de escrever procedimentos (programas) e definir gatilhos que ficam diretamente associados às operações de *insert*, *update* e *delete* de cada tabela. Esses programas são escritos utilizando uma linguagem normalmente conhecida com *Stored Procedure Language* (SPL). A SPL é um código-fonte que utiliza comandos de desvios, estruturas de repetições e sentenças SQL que possibilitam ao usuário criar as regras. Entretanto, esses temas não serão aprofundados neste curso.

Constraint	Descrição
<i>Default</i>	Indica um valor padrão para um atributo quando este não for especificado de forma explícita no comando <i>insert</i> . Exemplo: se a tabela ItemNF possui o atributo Vlr_Desconto decimal(7,2) <i>default</i> 0. Ao inserir uma nova linha, se o valor do desconto for omitido será atribuído um valor igual a zero automaticamente na linha inserida.
<i>CHECK</i>	Permite criar uma regra para o preenchimento de um atributo qualquer em uma tabela. Na regra podemos utilizar valores de outros atributos da mesma tabela para fazer a validação dos dados. Exemplo: se a tabela ItemNF possui os campos Vlr_Venda e Vlr_Desconto. Uma possível restrição que se aplica é que Vlr_Desconto tem de ser menor que Vlr_venda.
<i>PRIMARY KEY</i>	Define um ou mais atributos como chaves primárias. A chave primária identifica unicamente uma linha em uma tabela. Podemos criar somente uma <i>PRIMARY KEY</i> por tabela e não podemos colocar o valor <i>NULL</i> nos campos que compõem a chave primária.
<i>UNIQUE</i>	Valida demais atributos de uma tabela que também devem possuir valores únicos. Diferentemente da <i>PRIMARY KEY</i> podemos colocar várias restrições <i>UNIQUE</i> por tabela.
<i>FOREIGN KEY</i>	Utilizada para implementar o conceito de chave estrangeira, serve para indicar que o conteúdo de um atributo faz referência a um outro atributo definido como chave primária ou <i>UNIQUE</i> em uma outra tabela. Ela garante a integridade referencial

**Figura 6.2: Tipos de restrições declarativas**

Fonte: Elaborada pelo autor

Nas seções a seguir veremos com mais detalhes os conceitos de restrições apresentados até aqui, exceto o de integridade referencial, que será tratada na Aula 8.

## 6.2 Visões (View)

O conceito de visão (*view*) no contexto dos SGBDs possibilita a criação de tabelas virtuais (lógicas), as quais são derivadas de outras tabelas ou através de outras visões previamente definidas. A visão amplia a forma de visualização dos dados, introduzindo uma nova possibilidade de representá-los a partir de tabelas lógicas. Desse modo, diferentemente das tabelas que existem fisicamente no banco de dados, as visões estão apenas definidas no catálogo do banco de dados de tal forma que a sua estrutura é resultado de um comando *select* qualquer. Podemos perceber que o acesso aos dados a partir de uma visão ocorre de forma indireta. Ao executar um comando SQL que contenha na cláusula *from* uma visão, o SGBD incorpora no processo de execução as restrições que definem a visão, para somente após recuperar os dados. Os dados acessados em uma visão sempre estão fisicamente ligados a uma ou mais tabelas. Como os dados da visão são os mesmos da(s) tabela(s) correspondente(s), não há redundância. Portanto, atualizar os dados através da tabela ou através da visão, desde que possível, produz sempre o mesmo efeito.

O uso de visão possibilita um nível adicional de segurança e uma melhor adequação às possibilidades de acesso conforme a área de interesse do usuário. Nem todas as pessoas em uma empresa podem ter acesso a todas as informações. As visões permitem ocultar a complexidade e o sigilo dos dados e possibilitam criar uma independência lógica de dados, sem alterar a estrutura física. Ou seja, o uso de visões permite definir formas de limitar o acesso às informações do banco de dados, dependendo do usuário, de acordo com o papel que executa na empresa. A visão possibilita criar novas tabelas lógicas, restringindo ou agrupando informações de forma a atender às diversas necessidades de acesso. Por exemplo, uma tabela de Funcionário com todos os dados cadastrais dos funcionários, incluindo salário. É de se esperar, nesse caso, que o salário, entre outros dados, é de uso restrito da área de pessoal. Entretanto, existem diversas informações dos funcionários que podem ser públicas, de acesso livre a todos, tais como nome, ramal telefônico, cargo, etc. Essa demanda de acesso público em parte dos dados de uma tabela física poderia ser resolvida com a criação de uma visão de nome Ramal. Esse é um exemplo extremamente simples, mas vamos supor que em vez de criarmos a visão Ramal, optássemos por criar uma tabela Ramal, redundando alguns atributos da tabela Funcionário. Ao criamos a tabela Ramal, pudemos perceber que cada vez que um empregado for admitido ou demitido, será necessário acertar os dados em ambas as tabelas. Dessa forma, fatalmente a lista de ramais ficaria desatualizada ou inconsistente.

As visões permitem que os mesmos dados possam ser vistos por usuários diferentes de forma diferente e simultaneamente. O uso de visão pode ser útil para evitar redundâncias e possibilitar construir alguns conceitos semânticos, por exemplo, criando visões que são a união de duas ou mais tabelas ou visões que possuem atributos virtuais. Os atributos virtuais são aqueles calculados no momento da consulta que não estão fisicamente implementados em nenhuma tabela, mas que, para quem acessa a visão, é como se existissem. Os atributos virtuais são gerados através de agrupamentos ou concatenação de outros dados.

Obviamente a virtualização limita as possibilidades de atualização dos dados das tabelas físicas a partir das visões. Para minimizar as dificuldades de atualizações a partir das visões, Elmasri e Navathe (2000) apontam duas principais abordagens, uma estratégia chamada *query modification* e outra chamada *view materialization*. Na primeira delas ocorre uma transformação do comando utilizado na criação da visão em comando para atualização direta nas tabelas base, no momento da atualização. Esse método tem a

desvantagem de ser ineficiente para visões definidas a partir de consultas complexas. A segunda estratégia envolve a criação física de uma tabela *view* temporária no primeiro acesso que passa a ser mantida na expectativa de ser necessária a fim de atender às novas requisições de consulta.

Em síntese, podemos considerar as seguintes observações:

- visão definida a partir de uma única tabela é atualizável se os atributos da visão contêm a chave primária, ou alguma outra chave candidata, da tabela base. Isso permite uma correlação direta entre uma tupla da visão com uma tupla da tabela base;
- visão definida a partir de múltiplas tabelas usando junções geralmente não é atualizável;
- visão definida a partir de funções agregadas não é atualizável.

Os comandos para definir e apagar uma visão são respectivamente apresentados na Figura 6.3 a seguir.

```
Create view <nome da visão> as <comando select>
Drop view <nome da visão>
```

**Figura 6.3: Sintaxe dos comandos para definir e apagar uma visão**

Fonte: Elaborada pelo autor

Vejamos a seguir a criação de algumas visões tomando como base a Figura 6.4 a seguir.

NotaFiscal	ItemNF	Produto
	# Nro_NF	# Cod_Produto
# Nro_NF	# Nro_Item	Nom_Produto
Dat_Emissao	* Cód_Produto	Vlr_Venda
	Qtd_Comprada	Vlr_Promocional
	Vlr_Pago	

**Figura 6.4: Esquema de um banco de dados para exemplificar o uso das visões**

Fonte: Elaborada pelo autor

**Exemplo 1** – Visão baseada em uma única tabela base, mostrada na Figura 6.5 a seguir.

```
Create view ProdutoPromocao as  
select Cod_Produto, Nom_Produto, Vlr_Venda, Vlr_promocional  
from Produto  
where Vlr_Promocional < Vlr_Venda
```

**Figura 6.4: Exemplo 1 de criação de uma visão**

Fonte: Elaborada pelo autor

**Exemplo 2** – Visão baseada em uma única tabela que utiliza expressão aritmética, mostrada na Figura 6.6 a seguir.

```
Create view TotalItem as  
select Nro_Nf, Nro_Item, (Qtd_Comprada * Vlr_Pago) Vlr_Item  
from ItemNF
```

**Figura 6.6: Exemplo 2 de criação de uma visão**

Fonte: Elaborada pelo autor

**Exemplo 3** – Visão baseada em uma junção de uma tabela base e uma visão utilizando uma função agregada, mostrada na Figura 6.7 a seguir.

```
Create view TotalNota as  
Select Nro_Nf, Dat_Emissao, sum(Vlr_Item) Vlr_Nf  
from NotaFiscal A inner join TotalItem B  
on B.Nro_Nf = A.Nro_Nf  
group by Nro_Nf, Dat_Emissao;
```

**Figura 6.7: Exemplo 3 de criação de uma visão**

Fonte: Elaborada pelo autor

## 6.3 Esquema (*Schema*)

Um *schema* na SQL permite agrupar tabelas, visões, entre outros elementos do banco de dados que têm características de uso comuns. Para exemplificar, considere um banco de dados corporativo onde existem centenas ou até milhares de tabelas para atender às diversas áreas de negócio da empresa (Vendas, Compras, Contabilidade, etc.). Podemos subdividir o sistema

coorporativo em uma série de subsistemas específicos, um para cada área de negócio da empresa. Para armazenar os dados em cada subsistema, poderão existir tabelas específicas do subsistema e tabelas gerais que podem ser acessadas por mais de um subsistema. Ou seja, é como se o banco de dados fosse um conjunto de vários subsistemas que trabalham com tabelas específicas e compartilhadas. O conceito de *schema* possibilita representarmos os diferentes subconjuntos atribuindo um nome a cada um deles. Isso facilita tratar de forma agrupada dos elementos que compõem os bancos de dados. Enfim, o *schema* é um agrupamento lógico de objetos de banco de dados relacionados. Um *schema* SQL é identificado por um nome e um identificador de autorização para indicar o usuário que o criou. Cada usuário tem um esquema padrão que é o seu próprio nome de *login*. Um usuário só pode criar objetos em seu próprio esquema, a não ser que tenha privilégios adequados.

Para o MySQL, *schema* e *database* são sinônimos. Entretanto, os demais SGBDs relacionais, tais como, DB2, Informix, etc., tratam esses conceitos como coisas distintas. Um banco de dados pode possuir grupos de objetos com esquemas diferentes.

## 6.4 Restrição de unicidade

A restrição de unicidade serve para que um determinado atributo ou conjunto de atributos não possuam valor repetido em uma determinada tabela. A materialização dessa restrição é feita através da criação de um índice único com os atributos envolvidos na tabela na qual se deseja criar a restrição.

Um exemplo típico para a necessidade de criação desse tipo de restrição pode ser facilmente entendido a seguir. Vamos supor que estamos modelando um cadastro nacional de veículos. Ao colocarmos o chassi como sendo a chave primária da tabela Veículo, estamos restringindo, para não haver possibilidade de se cadastrar um número de chassi repetido. Entretanto, o mesmo tipo de restrição seria interessante para o número da placa e o número do Renavam. Como só pode existir uma única chave primária por tabela, podemos então criar um índice único para a placa e outro para o Renavam. Vejamos a sintaxe do comando mostrada na Figura 6.8 a seguir.

```
Create unique index <nome do indice> on <nome da tabela> (<lista de atributo(s)>)
```

**Figura 6.8: Sintaxe para criação de índice**

Fonte: Elaborada pelo autor

Comandos para criar os índices únicos adicionais em Veiculo são mostrados na Figura 6.9 a seguir.

```
Create unique index i_Placa on Veiculo (cod_Placa);  
Create unique index i_renavam on Veiculo (cod_renavam);
```

**Figura 6.9: Exemplo de comando para criação de índice**

Fonte: Elaborada pelo autor

Um índice pode ser apagado através do comando mostrado na Figura 6.10 a seguir.

```
Drop index <nome do índice>
```

**Figura 6.10: Sintaxe do comando para excluir um índice**

Fonte: Elaborada pelo autor

Os índices podem ser criados em qualquer outro atributo de uma tabela com objetivo de melhorar o tempo de acesso na recuperação de dados. Normalmente, somente deve-se criar índices adicionais nos casos em que houver problemas de desempenho na execução dos comandos *select*.



## 6.5 Uniões (*Union*)

O conceito de *UNION* está associado ao de união de conjunto da teoria matemática de conjuntos. A cláusula *union* é utilizada para unir as respostas de dois ou mais comandos *select*. As condições necessárias para que esses comandos possam ser unidos é que tenham como resposta a mesma quantidade de colunas e que cada coluna de uma mesma posição dos diversos *result sets* produzidos por cada comando *select* sejam do mesmo tipo. A união resultante pode ser de duas formas:

- *UNION ALL* – mostra todas as tuplas;
- *UNION | UNION DISTINCT* – mostra apenas as tuplas distintas.

Para exemplificar, vejamos o seguinte exemplo. Vamos supor que temos duas tabelas, Aluno e Professor, conforme mostrado na Figura 6.11. Deseja-se listar em ordem alfabética o nome de todos os alunos e professores existentes na comunidade escolar.

Aluno	Professor
#Cod_Aluno	#Cod_Professor
Nom_Aluno	Nom_Professor
*Cod_Curso	Nro_Telefone

**Figura 6.11: Representação das tabelas Aluno e Professor**

Fonte: Elaborada pelo autor

O comando utilizado é mostrado na Figura 6.12 a seguir.

```
Select Nom_Aluno Nome
From Aluno
Union
Select Nom_Professor
From professor
Order by 1;
```

**Figura 6.11: Comando que utiliza a união de conjuntos**

Fonte: Elaborada pelo autor

## Resumo



Assista ao vídeoAula6.avi |  
disponível no ambiente virtual de  
ensino-aprendizagem (AVEA).

Nesta aula são apresentados os principais conceitos sobre restrições declarativas e procedurais, que têm como objetivo dotar os dados armazenados em banco de dados de um maior poder semântico de expressão do “minimundo” que eles representam. Nesse sentido são apresentados os conceitos de visões, *schema* e criação de índice único que corroboram na busca desse objetivo.

## Atividades de aprendizagem

1. Explique o que é e para que serve o catálogo de um banco de dados.
2. Explique o que são as restrições declarativas.
3. Explique o que são as restrições procedurais.
4. Explique é o *schema*.
5. Explique o que é e para que servem as visões.
6. Qual é o objetivo de se criarem índices únicos?
7. Faça o roteiro de prática 3 disponibilizado no arquivo RoteiroBD03.pdf.

Poste as atividades no AVEA.

# Aula 7 – Estudo de caso – Fórmula I

## Objetivo

Apresentar um estudo de caso prático.

Nesta aula iremos utilizar os conceitos vistos nos capítulos anteriores, dando um sentido prático a eles. Iremos trabalhar com um estudo de caso. O problema proposto será trabalhado integralmente, com solução e comentários. Serão apresentadas as possíveis soluções para montagem do DER discutindo alternativas. Ao final, várias requisições de consulta propostas em forma de perguntas serão resolvidas utilizando a SQL.

### 7.1 Problema proposto

Vamos utilizar nesta aula, como base da discussão, um problema proposto que é um caso real existente no dia a dia de um projetista desenvolvedor. Como a ênfase deste texto foca a construção de um modelo conceitual para desenvolvimento de sistemas, iremos pular alguns temas decorrentes do processo de gerenciamento de um projeto de desenvolvimento de *software*, tais como: estudo de viabilidade; definição de escopo; além de considerações sobre prazos e custos. Portanto, iremos trabalhar um problema proposto a partir de um recorte de uma situação do mundo real. Tomando como base o problema, vamos apresentar uma proposta de solução para ele.

Normalmente, na prática, as informações para entendimento do problema são obtidas através de entrevistas com o grupo de usuários solicitantes. Cabe ao analista de sistema registrar essas informações de alguma forma e, a partir delas, produzir um modelo conceitual inicial, que será validado, em seguida refinado, até ser aprovado. Com base no modelo conceitual aprovado pelos usuários, amplia-se o nível de detalhamento, produzindo um modelo de implementação. Em uma sequência natural do desenvolvimento de um sistema, viriam os próximos passos: desenvolvimento, testes, implantação entre outros, os quais fogem ao escopo deste texto.

No problema proposto vamos considerar que as informações serão obtidas a partir de um texto.

**Problema proposto (Fórmula I)** – Suponha que você foi convidado pela Federação Internacional de Automobilismo, para apresentar um projeto que deve atender aos requisitos de controlar historicamente todos os campeonatos mundiais de Fórmula I realizados. O objetivo é disponibilizar um *site* de consulta de páginas dinâmica pela internet. Sabe-se que uma equipe possui até dois pilotos competindo em uma prova. Uma equipe é originária de um único país. A cada campeonato são realizados vários *Grand Prix* (GP) em vários países. O GP pode ser realizado em circuitos diferentes de um mesmo país, podendo também variar a cada campeonato. A pontuação é obtida pelo piloto em cada prova que ele participa, desde que termine a prova em uma posição pontuada, de acordo com as regras de cada campeonato. A mesma quantidade de pontos conseguida pelos pilotos é contabilizada para a equipe na qual ele disputou a prova. Os organizadores desejam gerar relatórios com os resultados de cada campeonato, classificar qual é a pontuação obtida por cada uma das equipes considerando todas as participações dos pilotos.

Tomando como base o problema apresentado, vamos elaborar um modelo de entidade e relacionamento, passo a passo, considerando todas as etapas do raciocínio envolvidas no processo. Em seguida, a partir do modelo físico das tabelas, vamos propor várias possibilidades de consultas, as quais serão resolvidas utilizando sentenças SQL.

O primeiro passo para entender o problema é identificar quais são as entidades que aparecem nele. Consideramos como entidade um agrupamento de dados correlatos que deverão ser armazenados no banco de dados com o propósito de gerar as consultas demandadas pela página dinâmica *web*. Portanto, ao reler o texto que apresenta o problema, podemos identificar inicialmente as seguintes entidades:

1. Equipe
2. Piloto
3. País
4. Campeonato
5. Regras do Campeonato
6. Resultado/pontuação por prova
7. *GrandPrix* ou Prova
8. Autódromo.

No problema proposto não há referência a respeito dos autódromos. Usou-se a expressão GP ou prova. Mas, se formos analisar, autódromo e prova são coisas distintas. Um autódromo é uma entidade física e possui atributos próprios como país, data de inauguração, tamanho do circuito em quilômetros, etc. Uma prova é a realização de uma corrida em um autódromo cujos atributos podem ser data e hora da realização da prova, número de voltas previstas para a corrida, etc. É importante observar que mesmo quando estamos tratando de um levantamento de dados real, as informações que nos são passadas podem vir de forma incompleta ou até mesmo errada. Cabe ao analista entender e documentar cada conceito apresentado pelo usuário, a fim de evitar erros de ambiguidade e completeza das informações coletadas.

Ao desenharmos o DER, e a partir dos relacionamentos muitos para muitos que surgirem, novas entidades deverão ser criadas. Outro dado importante é que durante o desenvolvimento da solução aparecerão algumas entidades e atributos que não estão explícitas no texto do problema. Na prática, isso normalmente também pode ocorrer, pois a partir de uma análise mais detalhada do problema poderão surgir dados importantes que devem ser armazenados e que foram esquecidos durante a fase de levantamento de dados. Cabe ao analista incluir essas informações no projeto e apresentá-las aos usuários solicitantes a fim de validar ou não a necessidade da informação.

Identificadas as entidades, mesmo que *grosso modo*, agora vamos identificar quais são os relacionamentos entre elas, e quais atributos serão necessários para registrar informações. Vamos começar pelas entidades equipes e pilotos:

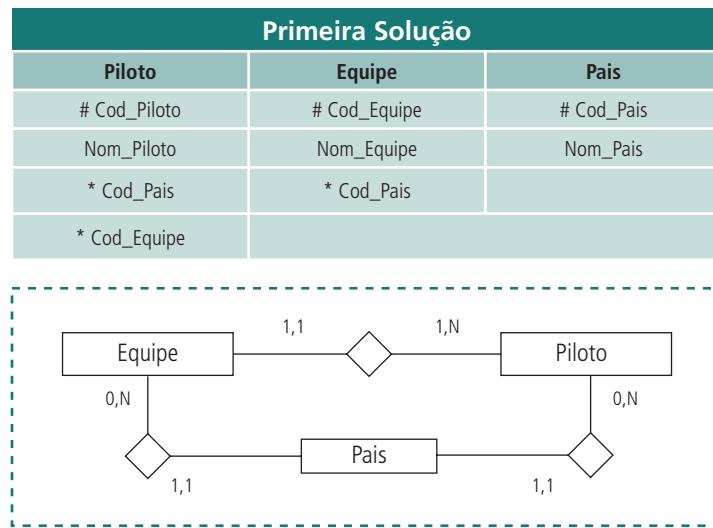
- uma equipe pode ter até dois pilotos;
- uma equipe é de um único país;
- um piloto tem uma única nacionalidade.

Uma equipe pode ter até dois pilotos. Portanto, “N”. Um piloto pode ao longo de sua carreira trabalhar em mais de uma equipe. Temos a princípio que o relacionamento entre piloto e equipe é muitos para muitos. Entretanto, um piloto num determinado período só pode estar associado a uma única equipe. Será necessário guardar a informação histórica de quais foram as equipes de um piloto? Ou, será que podemos obter essa informação quando olharmos para os resultados obtidos pelo piloto em cada prova? Responder a essas perguntas é a parte mais interessante. Ela envolve uma grande capacidade de abstração a fim de estudar as possíveis soluções para o problema que contemplem as respostas demandadas. Então você deve estar se per-

guntando: por que soluções, no plural? Isto é para mostrar que modelagem de dados não é uma ciência totalmente exata, na qual a resposta tem de ser única. Possivelmente, para a grande maioria dos problemas a serem modelados, haverá mais de uma solução. Entre as possíveis soluções, cada uma delas pode apresentar vantagens em relação às demais, no ponto de vista de simplicidade *versus* complexidade, flexibilidade *versus* rigidez, maior *versus* menor rapidez de acesso aos dados, entre outros fatores.

## 7.2 Possíveis soluções para as entidades Piloto, Equipe e País

A Figura 7.1 apresenta a primeira possível solução, considerando as entidades inicialmente discutidas.

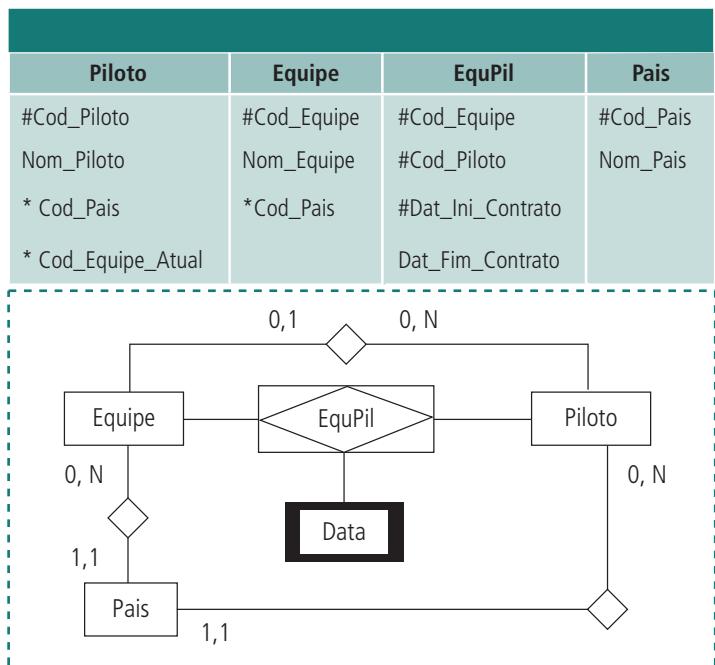


**Figura 7.1: Primeira proposta de solução**

Fonte: Elaborada pelo autor

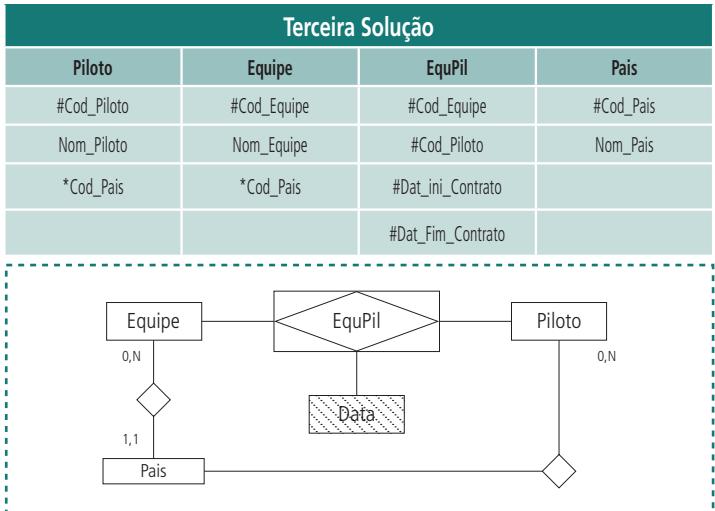
Na solução proposta pela Figura 7.1 é guardada apenas a informação de qual é a equipe atual do piloto. O modelo pode ser lido da seguinte maneira: uma equipe pode ter de “1” a “N” piloto(s); uma equipe é de “1” e somente “1” país; um país pode ter de “0” a “N” equipes; um piloto tem “1” e somente “1” nacionalidade; um país pode ter de “0” a “N” piloto(s). A entidade País se relaciona com as entidades Equipe e Piloto. Portanto, pode haver países em que só existam pilotos, só equipes ou ambos. Como um piloto tem uma nacionalidade, a entidade Piloto recebe como chave estrangeira a chave primária da entidade País. Na entidade Piloto existe um atributo que aponta para o país de nascimento do piloto. O mesmo se dá com a equipe que tem como chave estrangeira a chave primária de país, a fim de identificar a nacionalidade da equipe.

As Figuras 7.2 e 7.3 apresentam uma segunda e uma terceira possível solução, respectivamente.



**Figura 7.2: Segunda proposta de solução**

Fonte: Elaborada pelo autor



**Figura 7.3: Terceira proposta de solução**

Fonte: Elaborada pelo autor

Nas duas últimas soluções foi criada a entidade EquPil, que relaciona equipes e pilotos, possibilitando manter uma informação histórica das trocas dos contratos entre os pilotos e as equipes e registrando a quais equipes um piloto já pertenceu e quais pilotos já fizeram parte de uma equipe. Como se trata de uma entidade que registra dados históricos, fomos obrigados a transformar o relacionamento binário em ternário, inserindo uma entidade virtual temporal.

Isso ocorre porque se uma equipe contratar um piloto em um determinado momento, e depois de um prazo esse piloto mudar de equipe e em seguida retornar a essa mesma equipe, se mantido o relacionamento como binário, a chave primária seria duplicada. Dessa forma, não há como armazenar todas as informações das trocas. Se a chave primária fosse composta apenas pelos atributos Cod\_Equipe e Cod\_Piloto, não seria possível registrar mais de um contrato entre um piloto e uma equipe ao longo do tempo, pois, não saberíamos onde colocar informações das datas de início e fim de cada um dos contratos, uma vez que é possível existir na tabela apenas uma linha.

A diferença entre as duas últimas soluções está na forma de como é possível identificar qual é a última equipe na qual um piloto trabalhou. Na solução da Figura 7.2 foi definido na entidade Piloto o atributo Cod\_Equipe\_Atual, que associa o piloto com a última equipe na qual ele trabalha ou trabalhou. Dessa forma fica mais fácil recuperar a informação da equipe atual do piloto, porque não é necessário ler a tabela com os dados históricos. Quando um piloto troca de equipe, ao sair deve ser atualizado o atributo Dat\_Fim\_Contrato. Ao entrar na nova equipe, deve-se criar uma linha em EquPil, sendo que, Dat\_Fim\_Contrato permanecerá igual a nulo, enquanto que, na entidade Piloto o atributo Cod\_Equipe\_Atual deve ser atualizado. Cabe destacar a cardinalidade do relacionamento que diz respeito à equipe atual de um piloto, apresentada na Figura 7.2. Uma equipe pode ter de '0' a 'N' pilotos e um piloto pode estar atualmente ligado de '0' a 'N' equipe. Ou seja, um piloto pode não estar ligado a nenhuma equipe atualmente; portanto, o atributo Cod\_Equipe\_Atual tem de aceitar valores nulos.

Já na solução da Figura 7.3, para descobrir a equipe atual de um piloto é necessário fazer uma junção entre as tabelas Piloto e EquPil onde o atributo Dat\_Ini\_Contrato for igual à maior data ou, senão, onde Dat\_Fim\_Contrato for igual a nulo. É importante observar que a regra semântica que nos diz qual é a equipe atual do piloto não tem como ser implementada no banco de dados. Pode não haver nenhuma ou haver mais de uma linha com o atributo Dat\_Fim\_Contrato igual a nulo. No caso de não haver nenhuma, significa que o piloto não tem contrato atualmente na Fórmula I. Havendo mais de uma linha com Dat\_Fim\_Contrato nulas indica que existe um erro semântico no banco de dados. Um piloto não pode ao mesmo tempo trabalhar para mais de uma equipe. Nesse caso, a integridade deverá ser garantida por uma regra da aplicação codificada nos programas que atualizam essa tabela. Isso caracteriza uma falha de aderência entre a regra de negócio e a semântica dos dados implementados no banco de dados. Esse tipo de situação, sempre

que possível, deve ser evitado porque uma atualização no banco de dados por fora da aplicação poderá fazer com que a aplicação passe a se comportar de forma errônea. Quando transferimos a semântica dos dados para os programas de aplicação que atualizam as tabelas, estamos deixando de usar os recursos do banco de dados, fazendo com que ele deixe de ser o principal guardião das informações. Sendo assim, estando as regras na aplicação, em uma eventual manutenção manual via SQL ou através da integração com outros sistemas aplicativos que atualizam a tabela, não teremos garantia de que essas mesmas regras serão cumpridas.

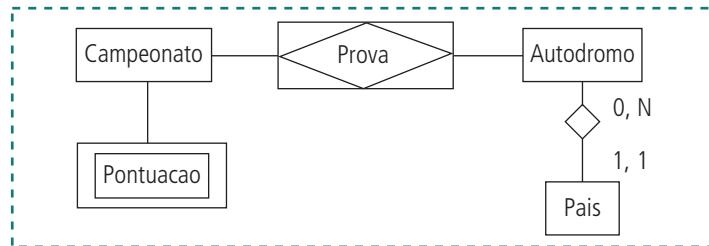
### 7.3 Possíveis soluções para as entidades Campeonato, Autódromo, Prova e Pontuação

Vamos pensar agora em novos relacionamentos que foram identificados. Por exemplo:

- um campeonato tem várias provas/GP;
- uma prova é realizada em um autódromo;
- a pontuação pode variar a cada campeonato;
- as provas podem mudar de autódromo dependendo do campeonato.

Como um campeonato tem várias provas/GP e uma prova é realizada em um autódromo, podemos entender que prova e GP são a mesma entidade e significam a realização de uma corrida em um autódromo. O relacionamento entre um campeonato e um autódromo é exatamente a ocorrência de uma corrida ou uma prova. Quando obtemos a informação de que a pontuação pode variar a cada campeonato, concluímos que as regras de pontuação não são fixas ao longo do tempo. A pontuação muda, mas não necessariamente a cada campeonato. Portanto, podemos ter mais de uma solução aqui também. A primeira considera que a pontuação depende de campeonato; portanto, deve ser cadastrada todas as vezes que for iniciado um novo campeonato, conforme mostrado na Figura 7.4 a seguir.

Primeira solução			
Campeonato	Autodromo	Prova	Pontuação
# Ano_Campeonato	#Cod_Autodromo	#Ano_Campeonato	#Ano_Campeonato
Nom_Campeonato	Nom_Autodromo	#Cod_Autodromo	#Nro_Pos_chegada
	*Cod_Pais	Dat_Realizacao	Qtd_Pontos
	Qtd_Km	Nro_Voltas	
	Qtd_Publico_Pag	Hor_Melhor_Volta	

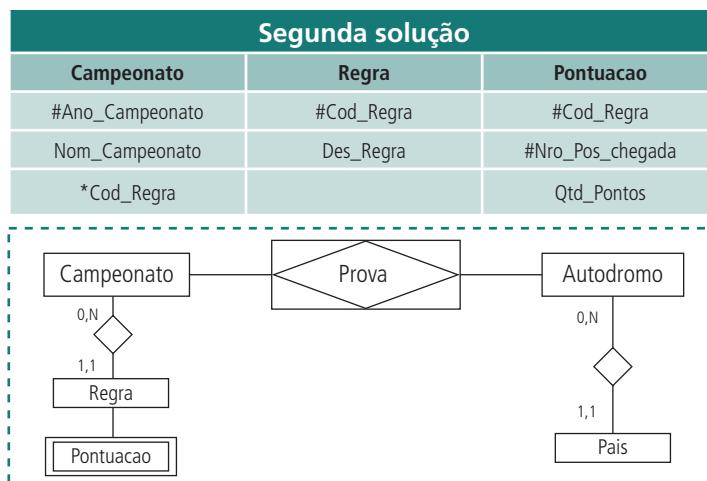


**Figura 7.4: Primeira proposta de solução**

Fonte: Elaborada pelo autor

Conforme podemos ver na Figura 7.4, o DER expressa que um campeonato é composto de várias provas que são realizadas em vários autódromos; um autódromo pode ser utilizado para realizar várias provas ao longo dos campeonatos; um autódromo está em “1” e somente “1” país; um país pode ter de nenhum até vários autódromos; as regras de pontuação pertencem a um campeonato específico.

Uma outra possível solução é criar um padrão de pontuação que pode ser relacionado a um ou mais campeonatos. Isso evita ter de cadastrar as regras ano a ano, conforme mostrado na Figura 7.5. Nesse caso ganhamos em reusabilidade das regras para mais de um campeonato.



**Figura 7.5: Segunda proposta de solução**

Fonte: Elaborada pelo autor

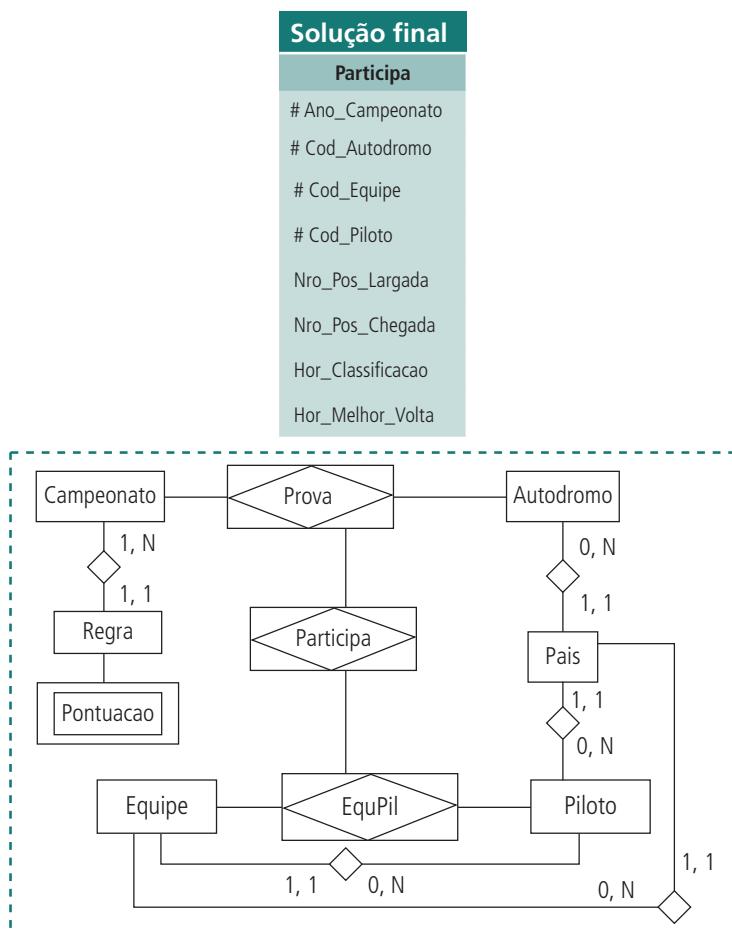
Conforme podemos ver na Figura 7.5, foi acrescentada a entidade Regra. Ela é responsável por agrupar todas as informações de pontuação. Uma vez montadas as regras, basta associar qual será a regra de cada campeonato. Isso permite a reutilização de uma mesma regra para vários campeonatos.

## 7.4 Solução final

Para finalizar, vamos pensar agora em um novo relacionamento que registra informações dos resultados:

- os pontos do piloto dependem do resultado de chegada em uma prova.

O que é um resultado? Um resultado são todas as posições de chegada dos pilotos/equipes em uma determinada prova realizada em um autódromo durante uma prova de um campeonato. Portanto, surge uma nova entidade a partir dos relacionamentos das entidades envolvidas. A esse relacionamento daremos o nome de Participa. Após criarmos a entidade Participa, podemos ver a proposta de solução final conforme mostrado na Figura 7.6 a seguir.



**Figura 7.6: Proposta de solução final**

Fonte: Elaborada pelo autor

Definido o modelo, o analista poderá usar várias formas de documentá-lo, como por exemplo, criar um dicionário de dados com os seguintes itens:

- relação das entidades, apresentado na Tabela 7.1;
- relação dos atributos, apresentado na Tabela 7.2;
- descrição dos eventos, apresentado na Tabela 7.3.

**Tabela 7.1: Relação das entidades**

Nro	Nome	Descrição
1	Autodromo	Pista onde são executadas as corridas.
2	Campeonato	Competição anual da Fórmula I.
3	Equipe	Empresas participantes fornecedoras de chassi e motor.
4	EquPil	Relacionamento entre equipe e piloto.
5	País	Países que participam da Fórmula I, através de equipes, pilotos ou autódromos.
6	Participa	Relacionamento que envolve a participação de uma equipe/piloto em uma prova do campeonato.
7	Piloto	Pilotos que participam ou que participaram das provas de Fórmula I.
8	Pontuação	Informações dos pontos computados de acordo com a posição de chegada.
9	Prova	Relacionamento entre campeonato e os autódromos que irão realizar provas durante um campeonato.
10	Regra	Conjunto de posição de chegada e os respectivos pontos marcados.

Fonte: Elaborada pelo autor

**Tabela 7.2: Relação dos atributos**

Nro	Nome	Descrição	Tipo
<b>Autodromo</b>			
1.1	Cod_Autodromo	Codificação para identificação única de um autódromo.	Char(3)
1.2	Nom_Autodromo	Nome oficial do autódromo.	Varchar(50)
1.3	Cod_Pais	Referência do país de localização do autódromo.	Char(3)
1.4	Qtd_Km	Tamanho da pista em metros.	Float
1.5	Qtd_Publico_Pag	Capacidade de espectadores pagantes do autódromo.	Inteiro
<b>Campeonato</b>			
2.1	Ano_Campeonato	Ano de referência da realização do campeonato.	Inteiro
2.2	Nom_Campeonato	Nome do campeonato. Ex.: "25º Campeonato Mundial de Fórmula I".	Varchar(50)
2.3	Cod_Regra	Referência à regra de pontuação do campeonato.	Integer

Relação dos atributos			
Nro	Nome	Descrição	Tipo
<b>Equipe</b>			
3.1	Cod_Equipe	Codificação única para equipe.	Char(3)
3.2	Nom_Equipe	Nome oficial da equipe.	Varchar(30)
3.3	Cod_Pais	Referência do país da equipe.	Char(3)
<b>EquPil</b>			
4.1	Cod_Equipe	Referência da equipe.	Char(3)
4.2	Cod_Piloto	Referência do piloto.	Char(3)
4.3	Dat_Ini_Contrato	Data inicial em que o piloto assinou o contrato com a equipe.	Date
4.4	Dat_Fim_Contrato	Data de encerramento do contrato com a equipe. Obs.: para o contrato vigente o campo tem seu conteúdo nulo.	Date
<b>País</b>			
5.1	Cod_Pais	Codificação única de país.	Char(3)
5.2	Nom_Pais	Nome do país.	Varchar(30)
<b>Participa</b>			
6.1	Ano_Campeonato	Referência a um campeonato.	Inteiro
6.2	Cod_Autodromo	Referência a um autódromo.	Char(3)
6.3	Cod_Equipe	Referência a uma equipe.	Char(3)
6.4	Cod_Piloto	Referência a um piloto.	Char(3)
6.5	Nro_Pos_Largada	Número da posição no <i>grid</i> de largada.	Inteiro
6.6	Nro_Pos_Chegada	Número da posição de chegada. Obs.: nulo se não completar a prova.	Inteiro
6.7	Hor_Classificação	Tempo em minutos / segundo / milésimos da volta de classificação.	Time
6.8	Hor_Melhor_Volta	Tempo em minutos / segundo / milésimos da melhor volta durante a prova.	Time
<b>Piloto</b>			
7.1	Cod_Piloto	Codificação única que identifica o piloto.	Char(3)
7.2	Nom_Piloto	Nome oficial do piloto	Varchar(30)
7.3	Cod_Pais	Referência para código do país.	Char(3)
7.4	Cod_Equipe_Atual	Referência para equipe atual com que o piloto tem contrato.	Char(3)
<b>Pontuação</b>			
8.1	Cod_Regra	Referência ao código da regra.	Integer
8.2	Nro_Pos_Chegada	Posição de chegada.	Integer
8.3	Qtd_Pontos	Quantidade de pontos correspondente à posição de chegada.	Integer

Relação dos atributos			
Nro	Nome	Descrição	Tipo
<b>Prova</b>			
9.1	Ano_Campeonato	Referência ao código do campeonato.	Integer
9.2	Cod_Autodromo	Referência ao código do autódromo.	Char(3)
9.3	Dat_Realizacao	Data em que prova ocorreu.	Datetime
9.4	Nro_Voltas	Número máximo de voltas para completar a prova.	Inteiro
9.5	Hor_Melhor_Volta	Tempo da melhor volta durante a corrida.	Time
<b>Regra</b>			
10.1	Cod_Regra	Codificação única que identifica uma regra de pontuação.	Integer
10.2	Des_Regra	Descrição da regra.	Varchar(256)

Fonte: Elaborada pelo autor

**Tabela 7.3: Relação dos processos**

Nro	Descrição
<b>1</b>	Cadastrar tabelas independentes: País; Regra/Pontuação.
<b>2</b>	Cadastrar tabelas com dependência simples: Piloto, Autódromo.
<b>3</b>	Cadastrar tabelas com dependência múltipla: Equipe/EquPil; Campeonato/Prova.
<b>4</b>	Cadastrar resultados Participa.
<b>5</b>	Executar consultas web.

Fonte: Elaborada pelo autor

Os comandos necessários para a criação do banco de dados são mostrados na Tabela 7.4 a seguir.

**Tabela 7.4: Script de criação do banco de dados****Script de criação do banco de dados: Formula I**

```
Create table autodromo (
    Cod_Autodromo char(3) not null,
    Nom_Autodromo char(50) not null,
    Cod_Pais char(3) not null,
    Qtd_Km float default null,
    Qtd_Publico_Pag int default null,
    Primary key (cod_Autodromo)
);

Create table campeonato (
    Ano_Campeonato smallint not null,
    Nom_Campeonato varchar(60) not null,
    Cod_Regra smallint not null,
    Primary key (Ano_Campeonato)
);

Create table equipe (
    Cod_Equipe char(3) not null,
    Nom_Equipe varchar(30) not null,
    Cod_Pais char(3) not null,
    Primary key (Cod_Equipe)
);

Create table equipil (
    Cod_Equipe char(3) not null,
    Cod_Piloto char(3) not null,
    Dat_Ini_Contrato date not null,
    Dat_Fim_Contrato date,
    Primary key (Cod_Equipe,
        Cod_Piloto)
);

Create table pais (
    Cod_Pais char(3) not null,
    Nom_Pais varchar(30) not null,
    primary key (Cod_Pais)
);

Create table regra (
    Cod_Regra smallint not null,
    Des_Regra varchar(256) not null,
    Primary key (Cod_Regra)
);

Create table participa (
    Ano_Campeonato smallint not null,
    Cod_Autodromo char(3) not null,
    Cod_Equipe char(3) not null,
    Cod_Piloto char(3) not null,
    Nro_Pos_Largada smallint not null,
    Nro_Pos_Chegada smallint default null,
    Hor_Classificacao time default null,
    Hor_Melhor_Volta time default null,
    Primary key (Ano_Campeonato,
        Cod_Autodromo,
        Cod_Equipe,
        Cod_Piloto)
);

Create table piloto (
    Cod_Piloto char(3) not null,
    Nom_Piloto varchar(30) not null,
    Cod_Pais char(3) not null,
    Cod_Equipe_Atual char(3),
    Primary key (Cod_Piloto)
);

Create table pontuacao (
    Cod_Regra smallint not null,
    Nro_Pos_Chegada smallint not null,
    QtdPontos smallint not null,
    Primary key (Cod_Regra,
        Nro_Pos_Chegada)
);

Create table prova (
    Ano_Campeonato smallint not null,
    Cod_Autodromo char(3) not null,
    Dat_Realizacao date not null,
    Nro_Voltas smallint,
    Hor_Melhor_Volta int default null,
    Primary key (Ano_Campeonato,
        Cod_Autodromo)
);
```

Fonte: Elaborada pelo autor

Definido o modelo de dados, podemos analisar, a partir das informações armazenadas no banco de dados, quais seriam as possíveis consultas a serem disponibilizadas no *site web*. O que iremos fazer agora é apresentar perguntas e o comando SQL capaz de extrair as informações no banco de dados que possa respondê-las.

1. Quais campeonatos houve?

```
Select Ano_Campeonato, Nom_Campeonato  
from Campeonato  
order by 1;
```

Comentário: Está selecionando dois atributos da tabela campeonato, ordenando de forma ascendente pela primeira coluna da lista de atributos Ano\_Campeonato.

O resultado é apresentado na Figura 7.7 a seguir.

			Ano_Campeonato	Nom_Campeonato
			2009	Campeonato Formula I - 2009
			2010	Campeonato Formula I - 2010

**Figura 7.7: Resposta da consulta da pergunta 1**

Fonte: Elaborada pelo autor

2. Quais foram os países (em ordem alfabética) que já realizaram provas de Fórmula I?

```
select Nom_Pais  
from pais  
where Cod_Pais in (select distinct Cod_Pais from Autodromo  
where  
Cod_Autodromo in (select distinct Cod_Autodromo  
From Prova))  
Order by 1
```

Comentário: Esta consulta possui uma *query* e duas *subqueries*. A execução é feita de dentro para fora. Primeiro resolve a *subquery* mais interna. O seu resultado é passado como filtro para a *subquery* menos interna cujo resultado é finalmente passado para a *query* principal. A primeira seleciona todos os códigos de autódromos onde ocorreram provas de Fórmula I sem repetição. A segunda relaciona de forma distinta todos os códigos de países onde esses autódromos estão situados. Finalmente acessa a tabela (Pais) a fim de traduzir o código pelo nome do país e ordenar o resultado.

O resultado é apresentado na Figura 7.8 a seguir.

			Nom_Pais
			Brasil
			Inglaterra

**Figura 7.8: Resposta da consulta da pergunta 2**

Fonte: Elaborada pelo autor

3. Quais foram os pilotos que pontuaram e quantos pontos fizeram no campeonato de 2010 em ordem decrescente de pontuação?

```
select B.Nom_Piloto, Sum(D.Qtd_Pontos) TotPontos  
from participa A join piloto B  
on B.Cod_Piloto = A.Cod_Piloto  
join campeonato C  
on C.Ano_Campeonato = A.Ano_Campeonato  
join pontuacao D  
on D.Cod_Regra = C.Cod_Regra  
and D.Nro_Pos_Chegada = A.Nro_Pos_Chegada  
where A.Ano_Campeonato = "2010"  
group by 1  
order by 2 desc,1
```

Comentário: A tabela que servirá como ponto de partida para sabermos o resultado da participação de cada piloto é a Participa. Portanto, ela será a tabela *master*, a primeira a ser relacionada na cláusula *from*. A partir da tabela Participa serão filtrados todos os resultados do ano de 2010. Para saber o nome do piloto, temos de traduzir o código do piloto existente na tabela Participa fazendo a junção com a tabela Piloto. Para obtermos a pontuação, precisamos saber qual é a regra do campeonato. Isto é feito através da junção da tabela Participa com a tabela Campeonato. De posse da regra, podemos descobrir quantos pontos o piloto fez em função de sua participação. Para tal, é necessário fazer uma junção da regra do campeonato com a posição de chegada. A pontuação deve ser somada pelos resultados da participação em todas as provas do campeonato. A função agregada *sum* foi utilizada na lista de atributos. Dessa forma, aqueles atributos que não fazem parte da agregação devem ser listados na cláusula *group by*. Finalmente o resultado é ordenado em ordem decrescente de pontuação pelo segundo atributo da lista e em ordem ascendente para o primeiro atributo da lista.

O resultado é apresentado na Figura 7.9 a seguir.

Nom_Piloto	TotPontos
Felipe Massa	30
Fernando Alonso	23
Rubens Barrichello	1

**Figura 7.9: Resposta da consulta da pergunta 3**

Fonte: Elaborada pelo autor

4. Quais foram os pilotos que pontuaram e quantos pontos fizeram no campeonato de 2010 em ordem decrescente de pontuação?

```
select B.Nom_Piloto, Sum(D.Qtd_Pontos) TotPontos  
from participa A join piloto B  
on B.Cod_Piloto = A.Cod_Piloto  
join campeonato C  
on C.Ano_Campeonato = A.Ano_Campeonato  
join pontuacao D  
on D.Cod_Regra = C.Cod_Regra  
and D.Nro_Pos_Chegada = A.Nro_Pos_Chegada  
where A.Ano_Campeonato = "2010"  
group by 1  
order by 2 desc,1
```

Comentário: Esta consulta é semelhante à anterior, exceto que neste caso deseja-se obter os resultados incluindo os pilotos que não marcaram pontos. Quando usamos a cláusula *join* significa dizer que a consulta deve selecionar apenas as linhas em que haja junção. Um piloto que chega numa posição não pontuada não consegue fazer junção com a tabela de pontuação, pois a linha não existe. Portanto, para tornar possível trazer também aqueles pilotos que não pontuaram na junção com a tabela Pontuação, a cláusula *join* foi alterada para *left outer join*. Isto permite que as linhas que não conseguirem a junção com a pontuação sejam selecionadas mesmo assim. Entretanto, a soma obtida do valor para esses pilotos será igual a *null*. Para que o resultado mostrado seja igual a zero, foi utilizada a função *ifnull*.

O resultado é apresentado na Figura 7.10 a seguir.

Nom_Piloto	TotPontos
Felipe Massa	30
Fernando Alonso	23
Rubens Barrichello	1

**Figura 7.10: Resposta da consulta da pergunta 4**

Fonte: Elaborada pelo autor

5. Quais os nomes das equipes e o total de pontos de cada uma delas que já pontuaram em pelo menos em uma prova de Fórmula 1?

```
select B.Nom_Equipe, Sum(D.Qtd_Pontos) TotPontos
from participa A join equipe B
on B.Cod_Equipe = A.Cod_Equipe
join campeonato C
on C.Ano_Campeonato = A.Ano_Campeonato
left outer join pontuacao D
on D.Cod_Regra = C.Cod_Regra
and D.Nro_Pos_Chegada = A.Nro_Pos_Chegada
group by 1
Order by 2 desc,1
```

Comentário: A partir da tabela Participa faz-se uma junção com a tabela Equipe a fim de traduzir o nome da equipe. Para obtermos a pontuação, precisamos saber qual é a regra do campeonato. Isto é feito através da junção da tabela Participa com a tabela Campeonato. De posse da regra, podemos descobrir quantos pontos a equipe fez em função de sua participação. Para tal, é necessário fazer uma junção da regra do campeonato com a posição de chegada. Como a pontuação deve ser somada pelos resultados em todos os campeonatos, a função agregada *sum* foi utilizada na lista de atributos. Dessa forma aqueles atributos que não fazem parte da agregação devem ser listados na cláusula *group by*. Note que, mesmo o resultado da prova sendo atribuído aos pilotos, como o atributo de agregação é o nome da equipe, isto faz com que o soma dos pontos mostrados seja por equipe. Finalmente o resultado é ordenado em ordem decrescente de pontuação.

O resultado é apresentado na Figura 7.11 a seguir.

Nom_Piloto	TotPontos
Ferrari	92
Willians-Cosworth	1

**Figura 7.11: Resposta da consulta da pergunta 5**

Fonte: Elaborada pelo autor

6. Quais foram os pilotos que já tiveram contrato com mais de uma equipe ao longo de sua carreira?

```
select Nom_Piloto
  from equipil A inner join piloto B
    on B.Cod_Piloto = A.Cod_Piloto
   group by 1
  having count(*) > 1
 order by 1

select Nom_Piloto
  from equipil inner join piloto
    using (Cod_Piloto)
   group by 1
  having count(*) > 1
 order by 1

select Nom_Piloto
  from equipil A, piloto B
 where B.Cod_Piloto = A.Cod_Piloto
   group by 1
  having count(*) > 1
 order by 1
```

Comentário: Neste caso a tabela *master* será a EquPil. Faz-se uma junção com Piloto para buscar o nome. Os nomes são agrupados em função do *group by* que trabalha em conjunto com a cláusula *having*, que verifica quantas são as múltiplas ocorrências de nome de piloto só mostrando aquelas que forem maiores que um.

O resultado é apresentado na Figura 7.12 a seguir.

Nom_Piloto
Rubens Barrichello

**Figura 7.12: Resposta da consulta da pergunta 6**

Fonte: Elaborada pelo autor

Foram mostradas três formas sintaticamente diferentes, mas que produzem o mesmo resultado. A primeira, utilizando a cláusula *on* para informar a junção. A segunda, utilizando a cláusula *using*, pode ser usada quando os atributos das tabelas envolvidas na junção tiverem o mesmo nome. Finalmente, uma terceira forma, em que a junção foi definida na cláusula *where*; esta última considero menos adequada, pois mistura dentro de uma única cláusula as condições de junção com as condições de filtro.

## Resumo

Nesta aula o objetivo foi de apresentar um estudo de caso de um problema real em que o processo de solução foi apresentado passo a passo. Dessa forma, deu-se uma visão real da elaboração mental necessária à execução, desde a modelagem dos dados do problema até a apresentação das respostas produzidas pelo sistema implementado.

## Atividades de aprendizagem

1. Identifique e relacione quais são os atributos que são chave primária em cada uma das tabelas.
2. Identifique e relacione quais são os atributos que são chave estrangeira em cada uma das tabelas.
3. Qual é o objetivo de usar as junções nos comandos *select*?
4. Faça o roteiro de prática 4 disponibilizado no arquivo RoteiroBD04.pdf



Assista ao vídeoAula7.avi  
disponível no ambiente virtual de ensino-aprendizagem (AVEA).

Poste as atividades no AVEA.



# Aula 8 – Integridade referencial – estudo de caso – Fórmula I

## Objetivo

Entender como funciona na prática a integridade de dados referencial.

Esta aula apresenta conceitos de integridade de dados referencial. Juntamente com a teoria, será apresentado um estudo de caso prático – Fórmula I. Esta aula dá sequência ao estudo de caso da Fórmula I apresentado na Aula 7.

### 8.1 Integridade referencial

Nós já vimos a importância da correta definição da chave primária, a qual garante a unicidade das linhas em uma tabela. Foi discutida também a definição de chave estrangeira como sendo os atributos oriundos da chave primária de uma tabela que foram trazidos para a tabela na qual a chave estrangeira está sendo implementada, a fim de servir como referência para junção entre as tabelas. A integridade referencial é a forma de implementar no banco de dados a chave estrangeira representada através dos relacionamentos do DER.

A integridade referencial tem implicação direta com os comandos *insert*, *update* e *delete*, restringindo ou estendendo as tuplas e tabelas envolvidas nos comandos SQL. A integridade referencial trata as relações de dependência entre tabelas que têm uma chave estrangeira fazendo referência à tabela na qual ela é chave primária. A implementação física da integridade referencial no banco de dados pode ser feita através de comandos DDL (*Data Dictionary Language*), ou através de programas aplicativos do SGBD que permitem a manipulação da estrutura das tabelas dentro do banco de dados.

A integridade referencial pode ser definida de três maneiras:

**Cascade** – Na tentativa de deletar ou atualizar a chave de uma tabela que possui tabelas referenciadas em cascata, fará o comando propagar para todas as tabelas referenciadas. Normalmente utilizada quando existe uma dependência de existência entre a tabela (dependente) na qual se está implementando a chave estrangeira e a tabela que está sendo referenciada pela sua chave primária completa.

**Restrict** – Na tentativa de deletar ou atualizar a chave de uma tabela que é referenciada por qualquer tabela de forma restrita; isso irá provocar uma violação da integridade referencial. Portanto, o SGBD irá retornar uma mensagem de erro na tentativa de execução do comando. Normalmente utilizada nas definições de chaves estrangeiras com integridade total (atributos não aceitam nulos) e cardinalidade um para muitos.

**Set Null** – Na tentativa de deletar ou atualizar a chave de uma tabela que é referenciada por outras tabelas com *set null*, faz com que ao executar o comando propague uma atualização de nulos para todas as tabelas a fim de não violar a integridade referencial. Nesse caso todas as referências de chave estrangeira têm de ser parciais. Ou seja, têm de aceitar valores nulos na tabela onde são chave estrangeira.

A sintaxe para criação da integridade referencial entre duas tabelas é mostrada na Figura 8.1 a seguir.

```
Alter table tabela1 add constraint foreign key (cod_atributo_fk)
    References tabela2 (cod_atributo_pk) [on delete cascade | set null | restrict]
```

**Figura 8.1: Referência de sintaxe para implementação da integridade referencial**

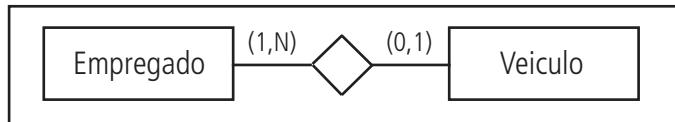
Fonte: Elaborada pelo autor

Para entender melhor esse conceito, vamos analisar detalhadamente cada caso em que a restrição de integridade deve ser implementada. Cabe ressaltar que a integridade referencial está representada no DER como sendo as ligações existentes entre as tabelas. Portanto, ao olhar o DER, devemos criar um comando para cada ligação existente no desenho.

### 1º Caso – Relacionamento 0:N (integridade parcial)

Vamos considerar o exemplo mostrado na Figura 8.2. Uma distribuidora de bebidas possui diversos empregados, sendo que aqueles que trabalham com vendas externas utilizam um único veículo específico fornecido pela empresa. Os demais empregados não utilizam veículos da empresa. Cada veículo da empresa pode ser usado por de um a N empregados, pois o veículo pode ser usado por outros vendedores desde que em turno diferente.

Portanto, nesse exemplo existe uma integridade referencial parcial, pois a chave primária de Veiculo que vai para o Empregado como chave estrangeira tem de aceitar nulo porque existem empregados que não têm acesso a nenhum veículo.



Empregado	Veiculo
# Cod_Empregado	#Cod_Placa
Nom_Empregado	Des_veiculo
*Cod_Placa	

**Figura 8.2: Relacionamento 0:N – integridade referencial parcial**

Fonte: Elaborada pelo autor

O comando para representar esse relacionamento usará *set null* para o caso de vender um veículo ou o Empregado deixar de ser vendedor externo. Veja o comando na Figura 8.3 a seguir.

```
Alter table empregado add constraint foreign key (cod_placa)
References veiculo (cod_placa) set null;
```

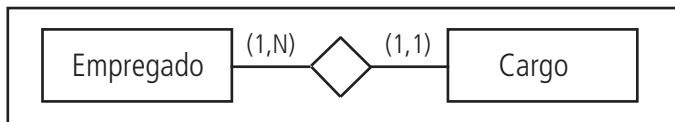
**Figura 8.3: Comando para criação da integridade referencial**

Fonte: Elaborada pelo autor

## 2º Caso – Relacionamento 1:N (integridade total)

Vamos considerar o exemplo mostrado na Figura 8.4. Em empresa deseja-se cadastrar todos os empregados e os cargos que eles ocupam. Sabe-se que todo empregado só pode ter um cargo e que num cargo pode haver N empregados.

Portanto, nesse exemplo existe uma integridade referencial total, pois a chave primária de Cargo que vai para o Empregado como chave estrangeira não pode aceitar nulos porque não existe empregado sem cargo.



Empregado	Cargo
# Cod_Empregado	#Cod_Cargo
Nom_Empregado	Des_Cargo
*Cod_Cargo	

**Figura 8.4: Relacionamento 1:N – integridade referencial total**

Fonte: Elaborada pelo autor

O comando para representar esse relacionamento usará *restrict* para não permitir que se exclua um cargo ocupado por pelo menos um Empregado. Vejamos.

```
Alter table empregado add constraint foreign key (cod_cargo)
References veiculo (cod_cargo);
```

**Figura 8.5: Comando integridade referencial total**

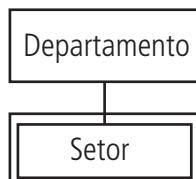
Fonte: Elaborada pelo autor

Obs.: A cláusula *Restrict* é *default*.

### 3º Caso – Relacionamento 1:N (com dependência de existência)

Vamos considerar o exemplo mostrado na Figura 8.6. Em uma empresa deseja-se cadastrar sua hierarquia funcional que é composta por departamentos e setor. Sabe-se que um departamento possui vários setores e que um setor é exclusivo de um único departamento.

Portanto, nesse exemplo existe uma integridade referencial com dependência de existência, pois se um departamento deixar de existir, todos os setores ligados a ele também deixarão de existir.



Departamento	Setor
# Cod_Departamento	#Cod_Departamento
Nom_Departamento	#Cod_Setor
Nom_Setor	

**Figura 8.6: Relacionamento 1:N – integridade referencial com dependência de existência**

Fonte: Elaborada pelo autor

O comando para representar esse relacionamento usará o *cascade* para possibilitar propagar a deleção de forma automática em Setor quando ocorrer uma deleção em Departamento. Veja o comando mostrado na Figura 8.7 a seguir.

```
Alter table setor add constraint foreign key (cod_departamento)
References Departamento (cod_departamento)on delete cascade;
```

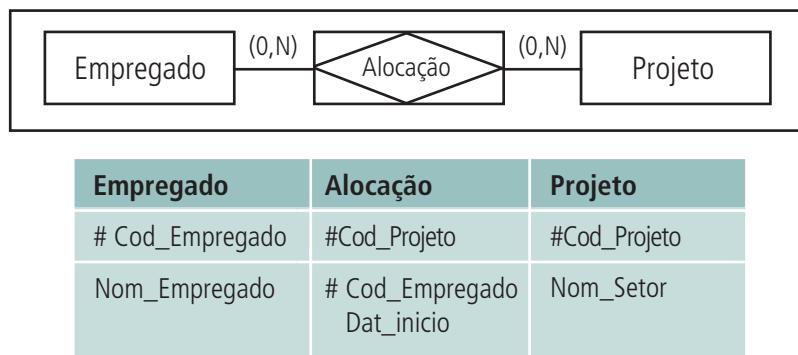
**Figura 8.7: Comando para implementar a integridade referencial**

Fonte: Elaborada pelo autor

#### 4º Caso – Relacionamento N:N

Vamos considerar o exemplo mostrado na Figura 8.8. Em empresa deseja-se cadastrar seus projetos, seus empregados e quais empregados trabalham em um projeto. Sabe-se que em um projeto vários empregados podem trabalhar e que um empregado pode trabalhar em vários projetos.

Portanto, nesse exemplo, como temos um relacionamento N:N, uma entidade Alocação será criada para registrar esse fato. A integridade referencial ocorre duas vezes: uma de Alocação para Empregado e outra de Alocação para Projeto.



**Figura 8.8: Relacionamento N:N**

Fonte: Elaborada pelo autor

O comando para representar esse relacionamento usará o *cascade* entre Alocacao/Projeto; isso significa deletarmos de um projeto todos os relacionamentos de empregados, que serão automaticamente apagados. Já para Alocacao/Empregado foi utilizada a condição default *restrict*. Nesse caso, não se pode apagar um Empregado que estiver vinculado a pelo menos um Projeto. Veja o comando mostrado na Figura 8.9 a seguir.

```
Alter table alocacao add constraint foreign key (cod_projeto)
References projeto (cod_projeto) on delete cascade;
```

```
Alter table alocacao add constraint foreign key (cod_empregado)
References projeto (cod_empregado);
```

**Figura 8.9: Comando para implementar o relacionamento N:N**

Fonte: Elaborada pelo autor

Para exemplificar ainda mais esse conceito, vamos retornar ao problema proposto na Aula 7, o da Fórmula I. A Figura 7.6 apresentou um DER de uma solução final do problema. Entretanto, o *script* de criação das tabelas mostrado definiu apenas as tabelas e suas respectivas chaves primárias. É como se o DER tivesse apenas as entidades. Para implementar no banco de dados as informações sobre os relacionamentos, é necessário definir as referências existentes entre as entidades. Uma das formas de fazer isso é alterando a tabela e adicionando uma restrição de chave estrangeira. A chave estrangeira adicionada registra que esses atributos existentes na tabela significam uma referência para outra tabela. A seguir serão mostrados os comandos que implementam esses relacionamentos.

Vamos começar apresentando os relacionamentos com dependência de existência. Aqueles em que a chave primária pai é parte inicial da chave primária da tabela filha. Nesse caso é importante informar ao banco de dados que, ao remover um registro da tabela pai, deve-se automaticamente propagar a deleção para os registros dependentes na tabela filha. Isso é feito utilizando a cláusula *on delete cascade*. Veja o exemplo mostrado na Figura 8.10 a seguir.

```
01     alter table pontuacao add constraint foreign key (cod_regra) references regra (cod_regra) on delete cascade;
```

**Figura 8.10: Comando para implementar o *delete cascade***

Fonte: Elaborada pelo autor

A seguir iremos implementar as restrições de integridade referencial nas tabelas que possuem uma chave estrangeira vinda de um relacionamento de um para muitos. Essas tabelas recebem a chave primária da tabela referenciada como atributo. Nesses casos temos duas opções: não declarar nada, pois o *default* é *restrict*; ou, nos casos da integridade ser parcial, atribuir a cláusula *set null*, que permite deletar o registro na tabela e atualizar com nulo a referência na tabela onde a restrição está declarada. Veja os exemplos mostrados na Figura 8.11 a seguir.

```
02     alter table campeonato add constraint  
          foreign key (cod_regra) references regra (cod_regra);  
  
03     alter table autodromo add constraint  
          foreign key (cod_pais) references pais (cod_pais);  
  
04     alter table piloto add constraint  
          foreign key (cod_pais) references pais (cod_pais);  
  
05     alter table equipe add constraint  
          foreign key (cod_pais) references pais (cod_pais);  
  
06     alter table equipe add constraint  
          foreign key (cod_piloto) references piloto (cod_piloto);
```

**Figura 8.11: Comando para implementar o *delete restrict***

Fonte: Elaborada pelo autor

Finalmente, iremos implementar os relacionamentos muitos para muitos. Nesses casos, a tabela é criada a partir da composição da chave primária das tabelas que fazem parte do relacionamento. Quanto à integridade referencial a ser implementada no banco de dados, também temos duas opções: não declarar nada, pois o *default* é *restrict*; ou, declarar *on delete cascade* nos casos que ao remover registros de uma relação que compõem o relacionamento quisermos excluir automaticamente o relacionamento. Veja os exemplos a seguir.

```
07 alter table prova add constraint  
    foreign key (ano_campeonato) references campeonato (ano_campeonato)  
    on delete cascade;  
  
08 alter table prova add constraint  
    foreign key (cod_autodromo) references autodromo (cod_autodromo);  
  
09 alter table equipil add constraint  
    foreign key (cod_equipe) references equipe (cod_equipe) on delete cascade;  
  
10 alter table equipil add constraint  
    foreign key (cod_piloto) references piloto (cod_piloto) on delete cascade;  
  
11 alter table participa add constraint  
    foreign key (ano_campeonato, cod_autodromo)  
    references prova (ano_campeonato, cód_autodromo);  
  
12 alter table participa add constraint  
    foreign key (cod_equipe, cod_piloto)  
    references equipil (cód_equipe, cod_piloto);
```

**Figura 8.12: Comando para implementar o *delete cascade***

Fonte: Elaborada pelo autor

Apresentadas todas as possíveis restrições de integridade, note que foram 12 comandos. Se verificarmos a Figura 7.6 percebe-se que existem doze linhas de ligação entre as tabelas apresentadas no modelo. Isso não é uma coincidência, ocorre porque o DER é uma representação exata do banco de dados e vice-versa.

É importante observar que algumas variações dos comandos apresentados são possíveis, embora isso altere o resultado dos comandos de deleção executados nas tabelas em relação à restrição ou propagação deles. Vamos analisar da forma como está. Ao executarmos o comando mostrado na Figura 8.13, qual será o resultado?

```
delete from campeonato  
where ano_campeonato = 2008;
```

**Figura 8.13: Comando para deletar campeonato**

Fonte: Elaborada pelo autor

Para exemplificar o poder semântico da implementação dessas restrições, observe a integridade referencial (07). Ela informa ao banco que ao deletar Campeonato deve-se propagar a deleção das provas do campeonato na tabela Prova. Entretanto, ao observarmos a integridade referencial (11), ela informa ao banco para não deletar Prova se houver registrada alguma participação na tabela Participa. Portanto, nesse caso só será possível apagar a linha de Campeonato, inclusive propagando a deleção das provas do campeonato em Prova, se ainda não tiver sido informado nenhum resultado de quaisquer das provas do campeonato.

## Resumo

Esta aula complementa o estudo de caso da aula anterior, acrescentando na prática os conceitos de integridade referencial. Ou seja, são apresentadas formas de informar ao SGBD como que os dados das tabelas interagem uns com os outros e, dessa forma, possibilitar que o banco de dados não permita que os dados se tornem inconsistentes por causa de atualizações indevidas.

## Atividades de aprendizagem

1. Explique como funcionam na prática a utilização dos comandos *on delete restrict, cascade e set null?*
2. Faça o roteiro de prática 5 disponibilizado no arquivo RoteiroBD05.pdf e poste as atividades no AVEA.

## Referências

CÓDIGO DE BARRAS. In: WIKIPÉDIA, a enciclopédia livre. Flórida: Wikimedia Foundation, 2012. Disponível em: <[http://pt.wikipedia.org/w/index.php?title=C%C3%B3digo\\_de\\_barras&oldid=30040399](http://pt.wikipedia.org/w/index.php?title=C%C3%B3digo_de_barras&oldid=30040399)>. Acesso em: 18 out. 2011.

DATE, C. J. **Bancos de dados: introdução aos sistemas de bancos de dados.** Tradução de Hélio Auro Golveia. 8. ed. Rio de Janeiro: Campus, 2004.

ELMASRI, Ramez; NAVATHE, S.B. **Fundamentals of database system.** 3. ed. Vancouver: Addison-Wesley, 2000.

ELMASRI, Ramez; NAVATHE, S.B. **Sistema de banco de dados.** 4. ed. São Paulo: Pearson, 2005.

KORTH, Henry F; SILBERSCHATZ A. **Sistemas de banco de dados.** São Paulo: McGraw-Hill, 1989.

SOUZA, Artur; LOUREIRO, Jorge. **Modelo Relacional Normalização Diagramas E-R e Tabelas Originadas.** Disponível em:

<[http://www.estv.ipv.pt/paginaspessoais/steven/Disciplinas/II2/Bibliografia/Sebenta/seb\\_cap5\\_1.pdf](http://www.estv.ipv.pt/paginaspessoais/steven/Disciplinas/II2/Bibliografia/Sebenta/seb_cap5_1.pdf)> Acesso em: 10 jul. 2012

## **Currículo do professor-autor**



Edson Marchetti da Silva – Professor responsável pela elaboração do material da disciplina Banco de Dados do Curso de PGTI. Atua como professor no CEFET-MG desde 2007, lecionando nos cursos Técnico em Informática, Técnico em Eletromecânica, na graduação em Engenharia Mecatrônica e na Pós-Graduação *lato sensu*. Ministra disciplinas relacionadas a Banco de Dados, Projeto de Sistemas, Lógica e Programação de Computadores. Já trabalhou como professor na Universidade Federal de Ouro Preto no Curso de Graduação em Sistemas de Informação, entre os anos de 2005 e 2007. Na iniciativa privada, atuou por 23 anos em diversos cargos, desde programador de computadores, analista de sistemas, administrador de banco de dados e gerente de informática. É graduado em Tecnologia de Processamento de Dados (1991), em Ciência da Computação (2003) e mestre em Administração (2006) pela Universidade FUMEC. É especialista em Engenharia de Software, pelo Instituto de Educação Continuada da Universidade Católica de Minas Gerais (2001) e doutorando em Ciências da Informação pela Universidade Federal de Minas Gerais desde 2010. Atualmente, atua com professor nos cursos Técnico em Informática, e como coordenador adjunto do Curso de pós-graduação *lato sensu* em Banco de Dados, além de atuar no ensino de educação a distância (EAD).



ISBN 978-85-99872-26-0



9 788599 872260 >