BANCO DE DADOS: BASES CONCEITUAIS E PRAGMÁTICAS

Dados Internacionais de Catalogação na Publicação (CIP)
(Jeane Passos de Souza – CRB 8ª/6189)

Leite, Leonardo Alexandre Ferreira
Banco de dados: bases conceituais e pragmáticas / Leonardo Alexandre Ferreira Leite. – São Paulo : Editora Senac São Paulo, 2020. (Série Universitária)

Bibliografia.
e-ISBN 978-65-5536-190-2 (ePub/2020)
e-ISBN 978-65-5536-191-9 (PDF/2020)

1. Banco de Dados (Ciência da Computação) I. Título II. Série.

20-1160t CDD – 005.74
COM018000

Índice para catálogo sistemático
1. Banco de dados 005.74

BANCO DE DADOS: BASES CONCEITUAIS E PRAGMÁTICAS

Leonardo Alexandre Ferreira Leite

Administração Regional do Senac no Estado de São Paulo

Presidente do Conselho Regional
Abram Szajman

Diretor do Departamento Regional
Luiz Francisco de A. Salgado

Superintendente Universitário e de Desenvolvimento
Luiz Carlos Dourado

Editora Senac São Paulo

Conselho Editorial
Luiz Francisco de A. Salgado
Luiz Carlos Dourado
Darcio Sayad Maia
Lucila Mara Sbrana Sciotti
Jeane Passos de Souza

Gerente/Publisher
Jeane Passos de Souza (jpassos@sp.senac.br)

Coordenação Editorial/Prospecção
Luís Américo Tousi Botelho (luis.tbotelho@sp.senac.br)
Dolores Crisci Manzano (dolores.cmanzano@sp.senac.br)

Administrativo
grupoedsadministrativo@sp.senac.br

Comercial
comercial@editorasenacsp.com.br

Acompanhamento Pedagógico
Mônica Rodrigues dos Santos

Designer Educacional
Diogo Maxwell Santos Felizardo

Revisão Técnica
Alexandre Lopes Machado

Preparação e Revisão de Texto
Janaina Lira

Projeto Gráfico
Alexandre Lemes da Silva
Emília Corrêa Abreu

Capa
Antonio Carlos De Angelis

Editoração Eletrônica
Cristiane Marinho de Souza

Ilustrações
Cristiane Marinho de Souza

Imagens
Adobe Stock

E-pub
Ricardo Diana

Editora Senac São Paulo
Rua 24 de Maio, 208 – 3º andar
Centro – CEP 01041-000 – São Paulo – SP
Caixa Postal 1120 – CEP 01032-970 – São Paulo – SP
Tel. (11) 2187-4450 – Fax (11) 2187-4486
E-mail: editora@sp.senac.br
Home page: http://www.livrariasenac.com.br

Sumário

Capítulo 1

Sistema de gerenciamento de banco de dados (SGBD)

Um sistema de gerenciamento de banco de dados (SGBD) é um software projetado para auxiliar a aplicação na manutenção e utilização de grandes conjuntos de dados (RAMAKRISHNAN; GEHRKE, 2011). Quando falamos de manutenção, estamos nos referindo principalmente à escrita (inserção, alteração e deleção); já a utilização está relacionada principalmente à leitura (buscas).

Imagine que você é um desenvolvedor de software e está escrevendo uma aplicação para a Justiça Eleitoral. Nessa aplicação é preciso que você armazene os dados de todos os candidatos a vereador do Brasil. Algum usuário (funcionário da Justiça Eleitoral) terá que fornecer esses dados de alguma forma, talvez digitando entradas para o sistema para

que ele ingira e guarde esses dados. Mas é preciso garantir que, caso o sistema seja desligado e religado, os dados ainda estejam lá. Também é preciso se preocupar com formas eficientes (rápidas) de recuperar a informação (ninguém gosta de esperar um sistema lento apresentar algo na tela).

Você pode escrever seu próprio código para armazenar todas essas informações em arquivos textos (txt), como também pode criar algoritmos para recuperar a informação quando necessário, pesquisando dados pelos arquivos. Nesse cenário, há muitas decisões a serem tomadas. Como estruturar esses arquivos? Um arquivo por candidato ou um arquivo com todos os candidatos? O que fazer se dois usuários tentarem alterar os dados de um candidato ao mesmo tempo? Como impedir o acesso não autorizado a esses dados? Bom, na verdade isso tudo não é nada fácil se você for fazer tudo do zero. E é por isso que desenvolvedores de software contam com o auxílio de um SGBD.

Algumas das vantagens de utilizar SGBDs são (RAMAKRISHNAN; GEHRKE, 2011):

- As aplicações (e os programadores) não precisam conhecer os detalhes de como os dados são armazenados em disco.
- O SGBD fornece diversas funcionalidades e implementa diversos algoritmos e otimizações para que as buscas possam ser realmente rápidas.
- O SGBD ajuda bastante a proteção da integridade e a segurança dos dados.
- O SGBD fornece meios de acesso concorrente (múltiplos usuários acessando os dados).
- O SGBD auxilia a recuperação de falhas (falhas da aplicação, do próprio SGBD, do sistema operacional e até mesmo do hardware da máquina).

- A utilização de um SGBD reduz o tempo de desenvolvimento de uma aplicação.

Com todas essas vantagens, para a maioria das aplicações, não utilizar um SGBD nem é mais considerado uma opção. Alguns exemplos de SGBDs populares são: MySQL, PostgreSQL, Oracle, SQLite, SQL Server e MongoDB.

Neste livro, aprenderemos a como manipular bancos de dados na prática e, para isso, utilizaremos o MySQL como nosso SGBD. Neste primeiro capítulo, daremos os primeiros passos: executar o MySQL, criar uma tabela e manipular os registros dessa tabela.

1 Dado, informação e conhecimento

Um computador pode armazenar e processar dados, que são representações simbólicas de alguma realidade (ou ficção) subjacente. Por exemplo, ao armazenarmos uma foto, do ponto de vista do computador estamos armazenando um conjunto de dados: são *pixels* dispostos matricialmente de forma que cada *pixel* possui um valor que determina sua cor. Um programa visualizador de fotos processa os dados do arquivo da foto e os transforma em comandos de máquina capazes de exibir a foto em um monitor.

Contudo, é somente quando um humano observa a foto que é possível para ele apreender alguma informação. Os processos listados anteriormente (armazenamento e exibição) ocorrem da mesma maneira para qualquer foto, mas um ser humano vai olhar para cada foto e apreender uma informação diferente dela. Por exemplo, um humano pode, a partir de uma foto em que o ator Rami Malek[1] segura a estatueta do Oscar, obter a informação de que ele foi o vencedor do Oscar. Veja que

1 Vencedor do Oscar de melhor ator em 2019 pela sua atuação no filme *Bohemian Rhapsody* (2018).

essa informação depende de associações a conceitos externos aos dados e de outras informações já sabidas pelo observador, que deve saber o que é o Oscar e ser capaz de reconhecer tanto o ator quanto a estatueta do prêmio.

Por fim, do ponto de vista de áreas técnicas, o conhecimento diz respeito à associação que uma pessoa faz entre conceitos, dados e informações, de um lado, e sua própria vivência pessoal, do outro (SETZER; SILVA, 2005).

IMPORTANTE

Mesmo que você leia as palavras contidas neste livro (dados) e obtenha informações relevantes sobre bancos de dados, você não poderá se dizer um conhecedor de bancos de dados sem que tenha praticado, ou seja, sem que tenha criado e manipulado um banco de dados. Então não se esqueça, o banco é de dados. Informações e conhecimento pertencem aos humanos!

2 Tabela, registro e campo

No projeto de um SGBD, há dois grandes tipos de decisão a tomar: decisões no nível físico e no nível conceitual dos dados, este último também chamado de nível lógico (RAMAKRISHNAN; GEHRKE, 2011). O nível físico diz respeito a como os dados são estruturados, ou seja, como os arquivos são gravados em disco. Já o nível conceitual diz respeito à visão que o usuário do SGBD (a aplicação e seu desenvolvedor) terá sobre os dados.

Ao longo deste livro, não nos preocuparemos com o nível físico dos dados. Quanto ao nível conceitual, focaremos a modelagem relacional, na qual um banco de dados é organizado em tabelas bidimensionais que se relacionam entre si. Os SGBDs que adotam esse modelo são

chamados de SGBDs relacionais, sendo o tipo mais popular de SGBD. No último capítulo, estudaremos brevemente alguns outros tipos de modelagens possíveis no nível conceitual dos dados.

Em um banco de dados relacional, os dados são organizados em tabelas. Uma tabela armazena uma coleção de ocorrências para determinado tipo de entidade, ao passo que um banco de dados reúne tabelas pertinentes a determinado assunto. Por exemplo, um banco de dados para a gerência de recursos humanos de uma empresa poderá ter tabelas para funcionários, departamentos, salários, etc.

Cada ocorrência de um tipo de entidade (por exemplo, um funcionário) é um registro na tabela correspondente. Também podemos chamar os registros por linhas. Assim, uma tabela de funcionários possui uma linha para cada funcionário da empresa.

Em um banco relacional, cada tipo de entidade define um conjunto uniforme de atributos possíveis para as ocorrências daquele tipo de entidade. Dito de outra forma: cada tabela define um conjunto de colunas, e cada registro possui um valor para cada coluna.

Na tabela 1, a seguir, é apresentado um exemplo no qual cada linha corresponde a um registro de município e cada coluna representa uma propriedade para a qual todo município deve ter um valor (nome, UF, população e área).

Tabela 1 – Tabela de municípios

nome	uf	populacao	area
São Paulo	SP	12 252 023	1 521
Belo Horizonte	MG	2 512 070	331
Sorocaba	SP	679 378	450
Guaratinguetá	SP	121 798	752
Rancharia	SP	29 707	1 587
Altamira	PA	114 594	159 533

Fonte: adaptado de Brasil (2017).

Para definir uma coluna de uma tabela devemos nomear a coluna e descrever o tipo da coluna (texto ou número, por exemplo). Caso o valor a ser armazenado seja uma grandeza física (área), infelizmente não é possível associar explicitamente a unidade da grandeza (como o metro quadrado – m²). Uma solução por vezes usada é acrescentar a unidade no nome da coluna *(area_m2)*. Outra possibilidade é deixar essa informação apenas na documentação do sistema.

IMPORTANTE

Repare também que a tabela 1 (municípios) utiliza acentuação para os valores (ex.: "Guaratinguetá"), mas não utiliza acentuação para os nomes das colunas ("area").

Em geral é considerada uma boa prática a não utilização de acentos em nomes de colunas e tabelas de um banco de dados, assim como em variáveis de um programa. Essa prática pode evitar alguns aborrecimentos. Também é costume utilizar apenas letras minúsculas em nomes de colunas e utilizar o *underscore* ("_") como separador de palavras.

Nomes de colunas e tabelas serão utilizados pela aplicação que consome o banco de dados, e não pelo usuário final. Já os valores armazenados (por exemplo, "São Paulo") serão lidos pelo usuário final e devem estar adequados às suas necessidades.

3 SGBD

Desenvolvedores de software utilizam um sistema de gerenciamento de banco de dados (SGBD) para criar bancos de dados. Um SGBD é utilizado para o gerenciamento de uma ou mais bases de dados. Um exemplo desse sistema é o MySQL; um exemplo de base de dados seria a base de candidatos a vereador ou até mesmo a base de dados de vereadores eleitos de determinado município.

PARA SABER MAIS

A rigor, um banco de dados seria uma base de dados. Por exemplo: "precisamos construir o banco de dados com as informações dos candidatos a vereador". No entanto, na linguagem coloquial, os profissionais também empregam o termo "banco de dados" para se referir ao SGBD, por exemplo: "qual banco de dados vamos utilizar em nosso próximo projeto? O MySQL ou o PostgreSQL?". Fique atento, pois esse coloquialismo poderá ser encontrado neste material dado o seu uso tão comum.

Geralmente o desenvolvedor – ou um papel mais específico, como o administrador de bancos de dados (DBA) – escreve comandos que o SGBD entenda para a construção de um banco de dados. Usando a linguagem do SGBD, o desenvolvedor define as tabelas que existirão e os campos de cada tabela.

Uma vez criada a estrutura do banco de dados (definição de tabelas e campos), a aplicação desenvolvida interage com o banco para manipular (incluir, alterar, excluir e consultar) os registros. Via de regra, o usuário final interage com a aplicação, e não diretamente com o banco de dados.

IMPORTANTE

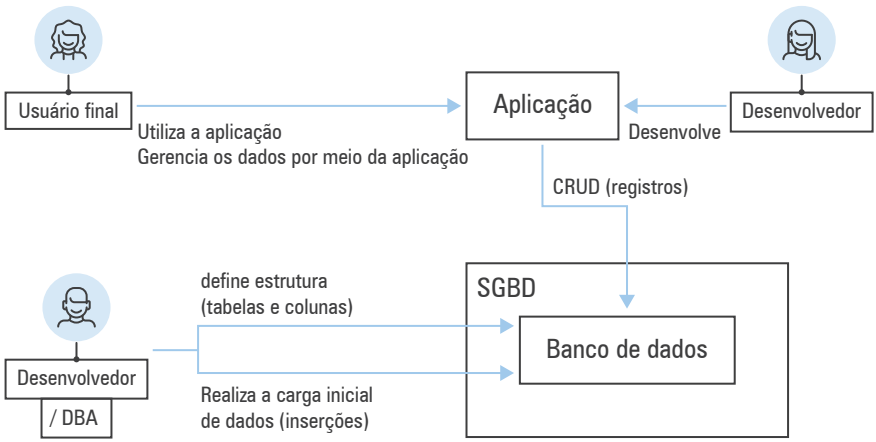
As operações típicas de manipulação de dados são inserir, alterar, excluir e consultar. Essas operações são referidas em conjunto como CRUD (*create, retrieve, update, delete*).

Algumas aplicações bem simples se resumem basicamente a cadastros que "inserem, alteram, excluem e consultam" registros. Assim, uma "aplicação CRUD" (ou mesmo "um CRUDzinho") é aquela aplicação que não faz muito mais do que um cadastro básico, não possuindo regras de negócio elaboradas.

Tipicamente é o usuário final que, por meio da aplicação, define os registros de uma tabela (por exemplo, um gestor de recursos humanos de uma empresa utiliza o sistema para cadastrar os funcionários). Porém, alguns dados já devem vir prontos para o usuário. O gestor de recursos humanos do exemplo anterior pode ter que informar qual o município de nascimento de um funcionário, mas não seria correto pedir aos gestores de recursos humanos que cadastrem todos os municípios brasileiros no sistema. Nesse caso, o próprio desenvolvedor realiza uma carga inicial de dados em uma tabela. Um exemplo de tabela que merece uma carga inicial é justamente a tabela de municípios.

A relação entre desenvolvedor, aplicação e banco de dados pode ser visualizada na figura 1.

Figura 1 – Modelo de utilização típica de um banco de dados



Neste livro, utilizaremos como SGBD o MySQL. Para instalar o MySQL, siga as instruções oficiais disponibilizadas para o seu sistema operacional. Para acessarmos o banco de dados, utilizaremos instruções da linha de comando. Para isso, você deve utilizar o terminal de seu sistema operacional.

PARA SABER MAIS

Se você nunca utilizou a linha de comando, vale a pena conferir algum tutorial na internet. Existem vários. Dois exemplos são a explicação de Schneider (2013) e o tutorial do grupo Django Girls.

Para se conectar ao banco, execute o comando a seguir:

```
$ mysql -u root -p
```

O prompt vai lhe pedir para digitar a senha de root do MySQL. Essa senha deve ter sido configurada durante o processo de instalação (possivelmente por meio do utilitário *mysql_secure_installation*). Se você estiver em um ambiente Linux, pode ser necessário também acrescentar o *sudo* no início do comando (*$sudo mysql -u root -p*).

Como acabamos de instalar o MySQL, vamos executar alguns procedimentos de praxe: criar um novo usuário e um novo banco de dados:

```
> CREATE USER 'aluno'@'localhost' IDENTIFIED BY 'senha';
> GRANT ALL PRIVILEGES ON * . * TO 'aluno'@'localhost';
```

Saia do prompt do MySQL (com o atalho *ctrl + d* ou com o comando *exit*) e se reconecte como o usuário aluno:

```
$ mysql -u aluno -p
```

Agora a senha será "senha". Finalmente, criando um novo banco de dados chamado "brasil":

```
> CREATE DATABASE brasil;
> USE brasil;
```

Para ver as tabelas existentes (no caso, nenhuma):

```
> show tables;
```

NA PRÁTICA

O símbolo "$" representa o prompt do sistema operacional, indicando que o comando deve ser executado no contexto do shell do sistema operacional (Linux, por exemplo).

Já o símbolo ">" representa o prompt do MySQL, indicando que o comando deve ser executado quando você já estiver conectado ao MySQL.

Criando uma tabela:

```
> CREATE TABLE municipios (
       nome text,
       uf varchar(2)
);
```

Inserindo alguns registros:

```
> INSERT INTO municipios (nome, uf) VALUES ('Sorocaba',
'SP'), ('São Paulo', 'SP'), ('Osasco', 'SP'),
('Guaratinguetá', 'SP'), ('Altamira', 'PA'), ('Palmas',
'GO');
```

Para visualizar o estado corrente da tabela:

```
> table municipios;
+---------------+------+
| nome          |  uf  |
+---------------+------+
| Sorocaba      |  SP  |
| São Paulo     |  SP  |
| Osasco        |  SP  |
| Guaratinguetá |  SP  |
| Altamira      |  PA  |
| Palmas        |  GO  |
+---------------+------+
```

Como espero que tenha percebido, temos um erro: Palmas é no Tocantins, e não em Goiás. Vamos corrigir isso:

```
> UPDATE municipios SET uf = 'TO' WHERE nome = 'Palmas';
```

Por fim, vamos também deletar um registro:

```
> DELETE FROM municipios WHERE nome = 'Osasco';
```

Verificando de novo o estado da tabela:

```
> table municipios
+---------------+------+
| nome          |  uf  |
+---------------+------+
| Sorocaba      |  SP  |
| São Paulo     |  SP  |
| Guaratinguetá |  SP  |
| Altamira      |  PA  |
| Palmas        |  TO  |
+---------------+------+
```

Executamos aqui alguns comandos básicos do MySQL para que você já tenha uma ideia de como ocorre a manipulação de banco de dados, mas em outros capítulos vamos estudar em detalhes como realizar esses comandos.

Além de interagir com o banco de dados pela linha de comando, também é possível utilizar ferramentas gráficas, por exemplo, o Adminer.

Considerações finais

Neste capítulo, abordamos a estrutura básica de um banco de dados (tabelas, registros e campos) e como um SGBD é empregado na prática em um contexto com desenvolvedores, usuários e aplicação. Demos também o primeiro passo para que você possa criar o seu próprio banco de dados.

Referências

BRASIL. Instituto Brasileiro de Geografia e Estatística (IBGE). Conheça cidades e estados do Brasil. **IBGE**, 2017. Disponível em: https://cidades.ibge.gov.br/. Acesso em: 9 mar. 2020.

DJANGO GIRLS. Introdução à linha de comando. **Tutorial Django Girls**, [*s. d.*]. Disponível em: https://tutorial.djangogirls.org/pt/intro_to_command_line/. Acesso em: 9 mar. 2020.

RAMAKRISHNAN, Raghu; GEHRKE, Johannes. **Sistemas de gerenciamento de banco de dados**. 3. ed. Porto Alegre: AMGH, 2011.

SCHNEIDER, Bruno. **Linha de comando**. 2013. Disponível em: http://professores.dcc.ufla.br/~bruno/aulas/linha-comando.html. Acesso em: 9 mar. 2020.

SETZER, Valdemar W.; SILVA, Flavio Soares Corrêa da. **Banco de dados**: aprenda o que são, melhore seu conhecimento, construa os seus. São Paulo: Blucher, 2005.

Capítulo 2

Projeto lógico

Neste capítulo, daremos os primeiros passos para que você seja capaz de projetar o modelo lógico de um banco de dados na forma de diagramas. Entenderemos melhor em que nível de abstração essa atividade se insere e aprenderemos que há desafios na escolha da quantidade de detalhes que deve ser fornecida em um diagrama.

Encerraremos o capítulo abordando a criação de índices, um assunto extremamente relevante para a criação de bancos de dados aptos a responder rapidamente a consultas.

1 Níveis de abstração

Uma das capacidades mais importantes para um desenvolvedor de sistemas é a de abstração. Isso significa ser capaz de isolar o problema em diferentes níveis de detalhe e considerar apenas os aspectos envolvidos que importam para a resolução do problema.

O mundo real é cheio de detalhes. Para criar um sistema de gerenciamento de matrículas escolares temos como entidade fundamental o aluno, mas esse aluno que existe no sistema está longe do aluno real, pois desconsidera muitos detalhes do aluno real (pouco importa para o sistema, por exemplo, o tipo de música preferido dele). Dizemos, assim, que a entidade *aluno* presente no sistema é uma abstração do aluno real; podemos dizer ainda que a entidade *aluno* é um modelo do aluno real. Dessa forma, modelar significa definir o nível de detalhe pertinente a um certo problema.

Mas a máquina também é cheia de detalhes. Para que o banco de dados funcione, há muito em jogo: arquivos, estruturas de dados, otimizações, o sistema operacional, processadores, transistores, etc. Todos esses detalhes da máquina real, via de regra, também não importam na hora de modelar um banco de dados. Graças a abstrações fornecidas por outros sistemas, todos esses detalhes já estão escondidos do usuário do banco de dados.

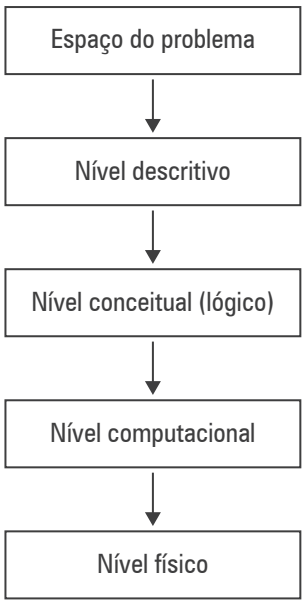
Assim, percebemos que um problema possui diferentes níveis de abstração. O nível mais alto é o que os autores costumam chamar de "mundo real". Vamos evitar esse termo, pois também podemos trabalhar com ficções (por exemplo: ao projetar uma tabela para armazenar as poções mágicas conquistadas por um elfo em um jogo de RPG,[1] pouco sentido

1 Em um *video game* de RPG o jogador controla um personagem que se move por um mundo virtual fictício, interagindo com seres de raças fantasiosas, como elfos, anões e orcs. Conforme o jogador vence desafios, ele vai adquirindo mais força e itens especiais, como novas armas, magias e poções. As poções podem ser

faz falar em "mundo real"). Chamaremos, então, o nível mais alto de abstração de "espaço do problema". Já o nível de abstração mais baixo é o nível físico, que diz respeito à máquina. No contexto de bancos de dados, o nível físico é tudo aquilo que o SGBD esconde de seu usuário, inclusive como os dados são organizados nos arquivos do sistema operacional.

Mas há ainda outros níveis de abstração (SETZER; SILVA, 2005), conforme apresentados na figura 1.

Figura 1 – Níveis de abstração existentes no contexto da utilização de bancos de dados



Um modelo descritivo é gerado pela análise de requisitos, sendo sua principal característica a informalidade. Trata-se da primeira tentativa de um humano abordar um problema e sistematizar os aspectos do problema que interessam para a sua resolução.

utilizadas para, por exemplo, recobrar as forças após uma batalha.

Já os dois próximos níveis, conceitual e computacional, são formais e precisos. São também os que mais interessam para este livro. No contexto dos bancos relacionais, ambos os níveis estão ligados à abstração de tabela, com linhas e colunas. Contudo, o nível computacional é dependente de detalhes da tecnologia e deve dialogar diretamente com os recursos e definições providos por um SGBD em particular, como o MySQL. Já o nível conceitual é mais abstrato. Um mesmo modelo conceitual para bancos relacionais deve ser igualmente aplicável a qualquer SGBD relacional, como o PostgreSQL ou o SQL Server. Uma forma de definir uma tabela no nível conceitual é por meio de um diagrama; já no nível computacional, devemos utilizar o comando *CREATE TABLE* da SQL.

Outra distinção útil entre os modelos conceitual e computacional é o tratamento da eficiência. No modelo conceitual, faz pouco sentido se preocupar com eficiência, pois esse é um problema da máquina. Cabe ao modelo computacional a descrição de objetos do banco de dados que existem para melhorar a eficiência do sistema, como é o caso dos índices que estudaremos ainda neste capítulo.

Por fim, os níveis computacional e físico não se confundem. O nível computacional está ligado à experiência do usuário do banco de dados: tudo o que o usuário comanda e observa ao utilizar o SGBD está no nível computacional. Já o nível físico é justamente aquilo que está escondido do usuário: toda a complexidade que o SGBD gerencia e esconde do usuário para lhe facilitar a vida.

2 Entidades, atributos e esquema

Um modelo conceitual procura descrever um banco de dados de forma abstrata, independentemente de tecnologia. Para isso, é comum que o projeto conceitual seja expresso por meio de um diagrama entidade-relacionamento (ou "diagrama ER"). Nesse caso, o modelo conceitual pode ser chamado de um modelo entidade-relacionamento. A abordagem entidade-relacionamento é a técnica de modelagem de dados mais difundida

para a descrição abstrata de bancos de dados (HEUSER, 2004). Assim, enquanto no modelo computacional falamos em tabelas, no modelo entidade-relacionamento falamos em entidades. A figura 2, por exemplo, ilustra um diagrama ER com duas entidades de um banco de dados de candidatos às eleições. Esse exemplo expressa que nesse banco de dados serão cadastrados tanto candidatos quanto partidos políticos.

Figura 2 – Exemplo de diagrama entidade-relacionamento com duas entidades

A caixa "candidato" representa a entidade *candidato*, que, por sua vez, representa o conjunto de candidatos a serem armazenados no banco de dados. Para nos referirmos a um candidato em particular aplicamos o termo "ocorrência" ou "instância". Em livros sobre bancos de dados também é comum a utilização do termo "relações" para designar o conjunto de ocorrências de uma entidade.

IMPORTANTE

O principal objetivo de um diagrama é a comunicação (FOWLER, 2004). Essa comunicação pode ser entre os membros de um time desenvolvendo um sistema ou entre os membros do time no presente e os membros do time no futuro (sejam novas pessoas ou as mesmas pessoas do passado). Pode ser, ainda, entre equipes diferentes (por exemplo, desenvolvedores e administradores do banco de dados).

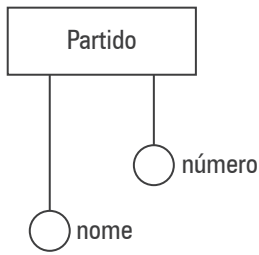
O uso do diagrama entre profissionais ajuda a moldar uma visão comum sobre o sistema (EVANS, 2003).

Caso, no futuro, uma nova equipe tenha que cuidar do sistema, os diagramas serão relevantes como uma primeira fonte de informação sobre o sistema.

Mas mesmo que as pessoas do time continuem as mesmas, pode ser que elas passem a trabalhar em novos projetos e depois de um tempo voltem ao sistema original. Nesse caso, é comum se esquecerem da modelagem do sistema, e diagramas de alto nível ajudam a recuperar a memória rapidamente.

Uma entidade também pode possuir atributos, o que equivale às colunas das tabelas. Assim, um atributo associa informações a ocorrências de entidades. É comum a omissão dos atributos nos diagramas ER para não poluir o diagrama em demasia. Quando representados, os atributos seguem a sintaxe apresentada na figura 3.

Figura 3 – Exemplo de entidade com atributos



PARA SABER MAIS

Acabamos de dizer que um diagrama ER pode ou não exibir os atributos das entidades. Mas como saber quando devemos exibir os atributos e quando devemos omiti-los? É preciso ter em mente que o principal objetivo de qualquer diagrama é a comunicação. Uma comunicação efetiva significa exibir as coisas importantes e esconder as menos importantes. Mas o que é importante ou não dependerá de cada contexto. Vale conferir a discussão sobre o assunto feita por Martin Fowler em seu artigo "Is design dead?" (FOWLER, 2004).

Como se pode deduzir, um modelo entidade-relacionamento também modela relacionamentos entre entidades. Esse será o assunto do próximo capítulo.

A representação completa de um modelo de bancos de dados (entidades, atributos, relacionamentos, etc.) é chamada de esquema. Um esquema conceitual é representado por um diagrama ER ou um conjunto

de diagramas ER. Como já discutimos, é aceitável que o esquema conceitual possa omitir alguns detalhes, como os atributos das entidades. Já o esquema do modelo computacional é a definição precisa das tabelas da base de dados, normalmente feita em linguagem SQL. Como o nível computacional é dependente de tecnologia, pode ser que o esquema definido em SQL tenha ligeiras variações de sintaxe conforme o SGBD. Um exemplo de recurso de SGBDs que deve estar presente no esquema computacional, mas que geralmente é omitido no modelo conceitual, são só índices, que serão apresentados a seguir.

3 Índices

Imagine um banco de dados que registre os livros da maior biblioteca do mundo.[2] Considere agora que é preciso fornecer ao público um serviço de consulta *on-line* para que os usuários busquem livros a partir de seus títulos. Em uma primeira abordagem ingênua, para que o sistema encontre o livro desejado, é preciso que o SGBD percorra todos os registros da tabela de livros. Por mais que computadores sejam rápidos, percorrer milhões de itens pode levar alguns segundos, o que é um tempo muito alto para que o usuário fique esperando. A solução para esse problema são os índices, que são estruturas que tornam as consultas mais rápidas.

Para entender os índices vamos considerar uma versão analógica do problema: sua estante de livros pessoal. Se os livros estiverem posicionados aleatoriamente, de qualquer maneira, provavelmente você vai demorar um pouco para achar determinado livro. Uma base de dados sem índices é como uma estante desorganizada (figura 4). Para achar um livro, temos que ir procurando aleatoriamente ou ir olhando um por um.

2 A biblioteca do Congresso dos EUA é a maior do mundo, possuindo cerca de 155 milhões de itens (MOIÓLI, 2018).

Figura 4 – Estante com livros desordenados

Já uma tabela com índice é como uma estante com livros organizados em ordem alfabética (figura 5). Para achar um livro você pode dar "saltos" em sua busca, estimando posições onde um determinado livro estará. Por exemplo, para procurar pelo livro *Viés de confirmação*, é possível já partir para a região final da prateleira. Um índice possibilita a busca eficiente nas linhas da tabela a partir de um valor da coluna (SETZER; SILVA, 2005). Em termos técnicos, a organização em ordem alfabética corresponde a utilização de estruturas de dados especiais para organizar os dados de modo a acelerar as buscas.

Figura 5 – Estante com livros ordenados

No MySQL, índices podem ser criados com o seguinte comando:

```
CREATE INDEX nome_do_indice ON nome_da_tabela
(nome_da_coluna(comprimento_do_texto));
```

Note que para criar um índice para uma coluna textual é preciso definir quantos caracteres iniciais do texto o índice considerará.

Exemplo:

```
CREATE INDEX index_nome_cliente ON cliente (nome(10));
```

Mas por que o SGBD já não cria índices automaticamente para todas as colunas? Isso ocorre porque a utilização de índices também apresenta desvantagens. Assim como no caso da estante, a desvantagem é o principal motivo pelo qual as pessoas deixam suas estantes desorganizadas: gasta-se tempo guardando um livro na ordem certa. Mesmo para um computador, é mais rápido simplesmente inserir um novo registro no fim da tabela do que reorganizar a estrutura de índice a cada inserção de dados. Além disso, a utilização de índices aumenta o tamanho (em bytes) da base de dados (o índice é uma estrutura à parte que deve ser armazenada).

Ou seja, a criação de muitos índices pode tornar a inserção de dados lenta. Mas a ausência de índices pode tornar a consulta de dados lenta. Também devemos considerar que precisamos de um índice para cada campo pelo qual queremos fazer as buscas. Dessa maneira, se quisermos buscar por autores e por editoras, precisaremos de dois índices a mais. E a cada índice novo, mais lenta se torna a inserção de novos registros. Esse equilíbrio entre ausência e presença de índices deve ser observado cuidadosamente pelos projetistas de bancos de dados.

PARA SABER MAIS

Utilizamos o exemplo de busca de livros por autor pela questão didática de compararmos com a busca física em uma prateleira de livros. Mas esse exemplo apresenta um problema. É que para que o índice seja efetivo, teríamos que buscar pelo título exato do livro. Porém, nesse caso, o usuário gostaria de procurar apenas "Deuses do México" e encontrar o livro "Deuses do México Indígena". Para resolver esse problema, o MySQL fornece o recurso de índices *full text*. Uma breve e amigável introdução a esse recurso é feita por Teteo (2019) no artigo "The magic known as MySQL FULLTEXT index". É comum também a utilização do *Elastic Search* como SGBD auxiliar para a realização de buscas *full text*.

Um exemplo melhor para um índice não *full text* seria a busca de veículos por placa. Nesse novo caso, é razoável exigir do usuário que ele forneça o valor exato correspondente à placa de determinado veículo (por exemplo, AHA0A01).

Considerações finais

Neste capítulo, demos importantes passos para a construção de modelos lógicos de bancos de dados. Porém, focamos as "entidades" da abordagem entidade-relacionamento. No próximo capítulo seguiremos estudando em detalhes a representação de relacionamentos entre entidades em um modelo ER.

Referências

EVANS, Eric. Communication and the use of language. *In:* EVANS, Eric. **Domain-driven design tackling complexity in the heart of software**. [*S. l.*]: Addison-Wesley Professional, 2003.

FOWLER, Martin. UML and XP. **Is Design Dead?**, 2004. Disponível em: https://martinfowler.com/articles/designDead.html. Acesso em: 11 mar. 2020.

HEUSER, Carlos A. **Projeto de banco de dados**. Porto Alegre: Sagra Luzzatto, 2004.

MOIÓLI, Julia. Qual a maior biblioteca do mundo? **Revista Super Interessante**, 2018. Disponível em: https://super.abril.com.br/mundo-estranho/qual-a-maior-biblioteca-do-mundo/. Acesso em: 21 fev. 2020.

SETZER, Valdemar W.; SILVA, Flavio Soares Corrêa da. **Banco de dados**: aprenda o que são, melhore seu conhecimento, construa os seus. São Paulo: Blucher, 2005.

TETEO, Leonardi. **The magic known as MySQL FULLTEXT index**. 2019. Disponível em: https://dev.to/leoat12/the-magic-known-as-mysql-fulltext-index-3npj. Acesso em: 11 mar. 2020.

Capítulo 3

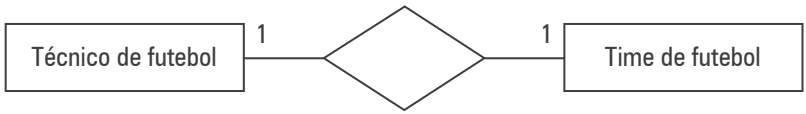
Modelo entidade--relacionamento

Neste capítulo, focaremos a modelagem dos relacionamentos entre entidades de um modelo entidade-relacionamento (modelo ER). Um relacionamento é uma associação entre duas entidades, que podem ter semânticas diversas, mas tipicamente sugerem alguma relação de pertencimento. Exemplos: a entidade *aluno* pode se relacionar com a entidade *turma*, pois o aluno pertence à turma; a entidade *dono* se relaciona à entidade *cachorro*, pois o cachorro pertence ao dono; a entidade *vereador* se relaciona à entidade *partido*, pois o vereador pertence ao partido.

1 Cardinalidade

A cardinalidade de um relacionamento define quantas ocorrências de uma entidade podem estar associadas a determinada ocorrência do relacionamento (HEUSER, 2004). No contexto do futebol profissional, temos o exemplo da figura 1, que apresenta um relacionamento que associa um técnico a um time de futebol. Nesse caso, dizemos que as entidades *técnico* e *time* possuem cardinalidade 1 no relacionamento, pois um time só pode ter um técnico, e uma pessoa, no contexto profissional, só pode ser técnico de um único time. Note que o losango é a notação para relacionamentos em diagramas entidade-relacionamento.[1] Uma associação pode ou não ser nomeada; se for, o losango passa a ostentar um rótulo, como na figura 2, mais adiante.

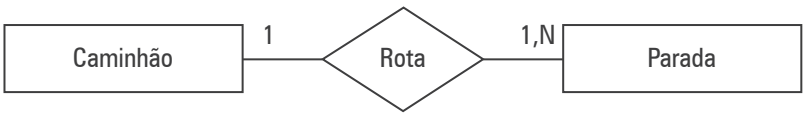
Figura 1 – Exemplo de relacionamento com cardinalidades de grau 1



Mas a cardinalidade de uma entidade pode variar para um mesmo relacionamento. Considere a rota como um relacionamento entre um caminhão de entregas e um conjunto de paradas. Nesse caso, uma rota associa um caminhão a um número indefinido de paradas. Uma rota pode ter dez paradas, enquanto outra pode ter quinze, podendo até ocorrer que uma rota tenha apenas uma parada. Por outro lado, não faz sentido que uma rota tenha zero parada. Nesse caso, dizemos que a cardinalidade das paradas na rota é *1,N*, em que 1 é a cardinalidade mínima e N é a cardinalidade máxima, representando um número natural maior que 1. Esse exemplo é ilustrado na figura 2.

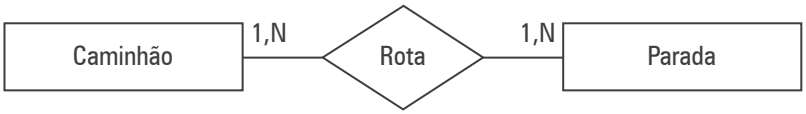
1 O diagrama entidade-relacionamento (diagrama ER) é usado para representar o modelo conceitual do banco de dados, conforme apresentado no capítulo anterior.

Figura 2 – Exemplo de relacionamento com cardinalidade 1,N



No exemplo da figura 2, o diagrama expressa que uma parada estará na rota de somente um caminhão. Ou seja, essa modelagem prevê que dois caminhões não farão entregas em uma mesma parada. Mas podemos flexibilizar o modelo para que uma mesma parada receba entrega de diversos caminhões. Nesse caso, a cardinalidade de caminhão no relacionamento passa a ser *1,N*, como consta na figura 3.

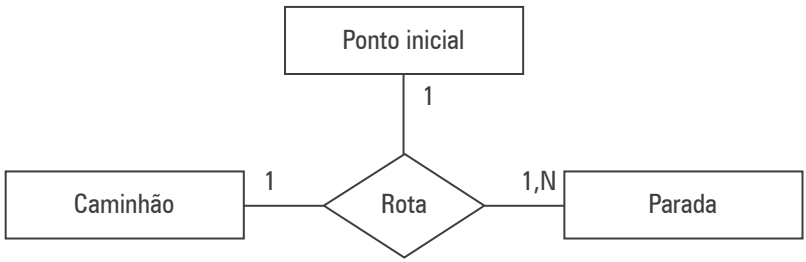
Figura 3– Exemplo de relacionamento N para N



2 Grau de relacionamento – relacionamentos binários, ternários e n-ários

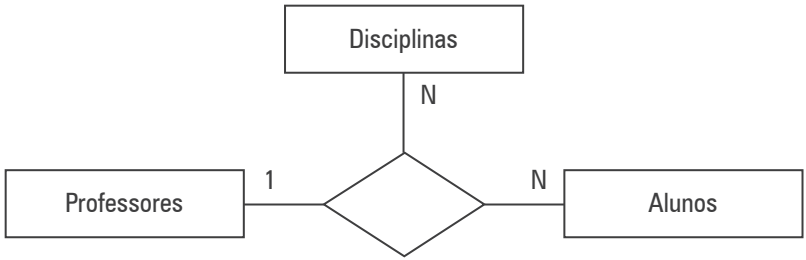
O grau de relacionamento entre entidades de um modelo ER corresponde à quantidade de entidades envolvidas no relacionamento (SETZER; SILVA, 2005). O mais comum é que os relacionamentos tenham grau 2, ou seja, que envolvam duas entidades, como nos exemplos fornecidos na introdução. Chamamos esses casos de relacionamentos binários, mas também é possível que um relacionamento envolva, por exemplo, três entidades, como ilustrado na figura 4.

Figura 4 – Exemplo de relacionamento ternário



No caso de relacionamentos ternários é preciso ter mais atenção ao significado das cardinalidades. O entendimento pode ficar mais claro pela figura 5: i) um professor pode dar aula a um aluno em várias disciplinas; ii) para um professor ministrando uma disciplina, teremos vários alunos; e iii) um aluno em uma certa disciplina terá aula com somente um professor.

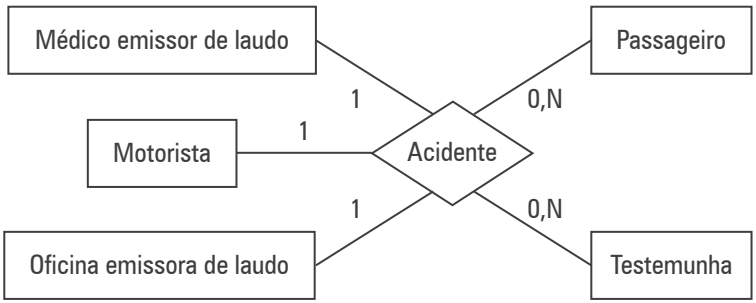
Figura 5 – Outro exemplo de relacionamento ternário



Fonte: adaptado de Setzer e Silva, (2005, p. 52).

Embora não seja usual, um relacionamento pode ter até mesmo mais que três entidades participantes. São os chamados relacionamentos n-ários. Considere o sistema de uma companhia de seguros. Para esse sistema, um acidente pode ser um relacionamento envolvendo diversas entidades: motorista, passageiros, testemunhas, advogado, médico emissor de laudo e oficina emissora de laudo. Esse exemplo é apresentado na figura 6.

Figura 6 – Exemplo de relacionamento n-ário



O motivo pelo qual esse tipo de relacionamento é raro é porque normalmente, como no exemplo, ele é tão central para o sistema que poderia ser promovido a uma entidade.

3 Chave primária e chave estrangeira

Uma chave primária é uma coluna ou uma combinação de colunas de forma que os valores da chave primária para determinado registro identifiquem univocamente esse registro, diferenciando-o dos demais registros (HEUSER, 2004). Por exemplo, em uma base de dados em que registros correspondam a pessoas adultas do Brasil, o CPF pode ser usado como chave primária. A ideia é que duas pessoas nunca terão o mesmo CPF. Quando a chave primária é composta por duas colunas, dizemos que se trata de uma chave composta.

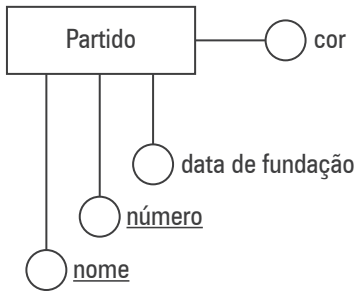
NA PRÁTICA

É preciso muito cuidado ao escolher uma chave primária. Por vezes, a hipótese inicial de trabalho acaba não se sustentando. Por exemplo, somente a partir de 2015 é que todos os brasileiros passaram a nascer com CPF. Pode ser que o sistema venha a ter que registrar, de forma não prevista inicialmente, alguém nascido em 2014 que ainda não tenha CPF. Pode ser que, também inesperadamente, o sistema venha a ter que lidar com estrangeiros sem CPF. Por isso, na prática, pode ser mais con-

veniente utilizar como chave primária um número (ID) artificial gerado automaticamente pelo sistema.

Em um diagrama ER, os atributos de uma entidade correspondente à sua chave primária podem ser destacados com um sublinhado (RAMAKRISHNAN; GEHRKE, 2011), como mostra a figura 7. No exemplo, tanto o número quanto o nome do partido político são considerados como parte da chave primária, uma vez que a modelagem considera que ao longo do tempo nomes e números de partidos são reaproveitados por novos partidos.

Figura 7 – Exemplo de entidade com chave primária explicitada no diagrama ER



Já uma chave estrangeira é o que implementa um relacionamento entre diferentes registros. Tomemos como exemplo o cadastro de candidatos a vereador. É preciso registrar em que município cada candidato habita. Para isso, considere que o sistema já possui uma tabela de municípios, como a tabela 1 a seguir.

Tabela 1 – Tabela de municípios

id	nome	uf
1278	Maceió	AL
3245	Porto Alegre	RS

Em vez de repetir as informações (nome e uf) de um município nos registros de todos os candidatos desse município, esse relacionamento pode ser feito apenas pelo ID do município, como exemplificado na tabela 2.

Tabela 2 – Tabela de candidatos

id	nome	id_municipio	email
1	João	1278	joao@gmail.com
2	Cristina	3245	cris@protonmail.com
3	Maria	1278	maria@gmail.com

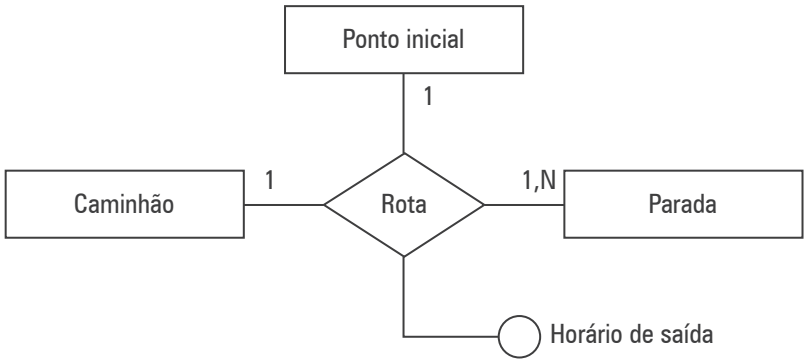
Nesse exemplo, a coluna *id* é a chave primária da tabela de municípios. Já a coluna *id_municipio* na tabela de candidatos é uma chave estrangeira que aponta para a tabela de municípios.

O SGBD costuma garantir certas integridades em relação ao uso de chaves. Ou seja, ele próprio impedirá a criação de dois registros com a mesma chave primária ou a criação de um registro com uma chave estrangeira cujo valor não exista como chave primária na tabela relacionada (por exemplo, não seria possível criar um candidato com um ID de município inválido).

4 Atributos de relacionamento

Retomemos o exemplo da modelagem de rotas de caminhões. Cada rota, além do ponto inicial e das paradas, poderia ter também um horário de saída (supondo uma viagem diária). Esse horário de saída é um atributo do relacionamento e é diagramado conforme apresentado na figura 8.

Figura 8 – Exemplo de relacionamento com atributo



NA PRÁTICA

Então por que não assumir que a rota é simplesmente uma entidade? Nesse caso, a rota seria uma entidade com seus atributos (por exemplo: horário de saída, horário previsto de chegada, etc.) e alguns relacionamentos binários (um relacionamento 1:1 com o ponto inicial e outro relacionamento 1:N com as paradas), pois nada impede que isso aconteça. Uma modelagem rica, bem refinada, costuma mesmo produzir diversos atributos nos relacionamentos, o que é uma força que tende a produzir novas entidades, uma vez que essas entidades se tornam conceitos claramente distintos e relevantes para o negócio. Ou seja, na prática, provavelmente relacionamentos com atributos e relacionamentos ternários serão promovidos a entidades.

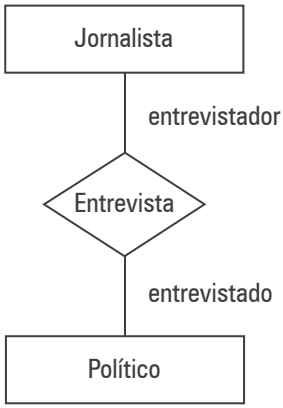
Segundo Ramakrishnan e Gehrke (2011), diagramas de classe da UML[2] podem ser utilizados para representar modelos ER, mesmo sem possuir um tipo de elemento "relacionamento": no caso, tanto entidades quanto relacionamentos acabam sendo modelados como classes da UML, gerando inclusive diagramas mais compactos. Essa possibilidade evidencia como não há prejuízo em modelar relacionamentos como entidades em diagramas ER.

2 A Unified Modeling Language (UML) é uma família de notações gráficas que ajuda a descrição e o projeto de sistemas de software (FOWLER, 2005). Na UML, há diversos tipos de diagramas para descrever o software sob diferentes óticas (estrutura, tempo de execução, implantação, etc.).

5 Relacionamentos recursivos condicionais

O papel (*role,* em inglês) é um rótulo que esclarece a participação da entidade no relacionamento. A figura 9 mostra um exemplo de relacionamento com papéis para um banco de dados que armazena dados sobre entrevistas com políticos: o político possui o papel de entrevistado, enquanto o jornalista possui o papel de entrevistador.

Figura 9 – Exemplo de relacionamento com papéis

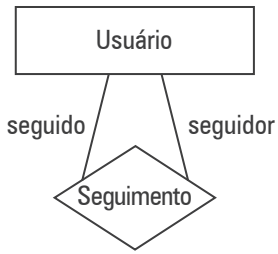


Um relacionamento recursivo é aquele no qual a mesma entidade participa mais de uma vez, com papéis diferentes, em um mesmo relacionamento (ELMASRI; NAVATHE, 2018). Relacionamentos entre entidades diferentes não costumam precisar da especificação de papéis. Mas em relacionamentos recursivos, a definição é essencial para que o relacionamento seja entendível. Os relacionamentos recursivos também são chamados de autorrelacionamentos.

Na figura 10 é apresentado um exemplo de relacionamento recursivo: em uma rede social seguidores podem seguir uns aos outros (Karina pode seguir Felipe, sendo que, nesse caso, Felipe pode ou não

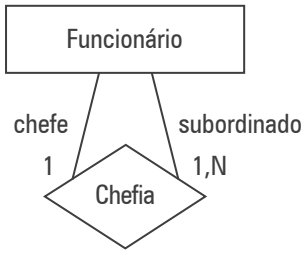
seguir Karina); temos aí uma associação entre duas ocorrências da mesma entidade (usuário). Cada um dos usuários do relacionamento cumpre um papel diferente: o de seguidor ou o de seguido.

Figura 10 – Exemplo de relacionamento recursivo



Dado um relacionamento entre as tabelas A e B, um relacionamento condicional é aquele em que cada ocorrência de A deve estar ligada a uma ocorrência de B, mas nem toda ocorrência de B precisa estar ligada a uma ocorrência de A (EBERLY, 1997). Um exemplo de relacionamento recursivo condicional, exibido na figura 11, seria a relação de chefia: todo funcionário tem como chefe outro funcionário, mas nem todo funcionário é chefe de algum outro funcionário.

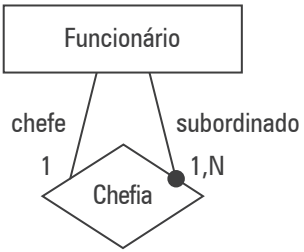
Figura 11 – Exemplo de relacionamento recursivo condicional



Repare nas cardinalidades do relacionamento de chefia da figura 11, em toda relação de chefia há sempre um chefe e um ou mais chefiados. Isso significa que todo funcionário deve possuir apenas um único chefe. Significa também que ninguém pode ser chefe sem ter nenhum subordinado. Note como, infelizmente, não fica evidente pelo diagrama que

nem todo funcionário é um chefe. Supondo que todo funcionário deva ter um chefe, poderíamos resolver o problema com a notação de restrição de integridade de totalidade (SETZER; SILVA, 2005), ou seja, com a bolinha preta na figura 12 indicando que todo funcionário participa de um relacionamento de chefia como subordinado. Aí a ausência da bolinha preta na aresta "chefe" indicaria que nem todo funcionário participa de uma relação de chefia enquanto chefe.

Figura 12 – Exemplo de uso de restrição de integridade de totalidade



Contudo, geralmente teremos pelo menos um funcionário da empresa, o presidente, que não terá um chefe. Esse é então mais um exemplo das limitações da modelagem ER e de que alguns conceitos da modelagem só serão plenamente expressos nas regras de negócio da aplicação.

Considerações finais

Os capítulos 2 e 3 abordaram os elementos essenciais da modelagem conceitual de entidades e relacionamentos. Mas após um projeto conceitual bem elaborado, é hora de pôr os conceitos em prática no SGBD. No próximo capítulo, vamos nos voltar à SQL, a linguagem que usamos para interagir com o SGBD para a definição de tabelas (mapeando o esquema conceitual) e a manutenção dos registros (consultas, inserções, alterações e deleções).

Referências

EBERLY, Wayne. CPSC 333: more about relationships. Notas de aula. **Universidade de Calgary**, 1997. Disponível em: http://pages.cpsc.ucalgary.ca/~eberly/Courses/CPSC333/Lectures/ERD/more_about_relationships.html#conditional. Acesso em: 24 fev. 2020.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de banco de dados**. 7. ed. São Paulo: Pearson, 2018.

FOWLER, Martin. **UML essencial**: um breve guia para a linguagem-padrão de modelagem de objetos. 3. ed. São Paulo: Bookman, 2005.

HEUSER, Carlos Alberto. **Projeto de banco de dados**. Porto Alegre: Sagra Luzzatto, 2004.

RAMAKRISHNAN, Raghu; GEHRKE, Johannes. **Sistemas de gerenciamento de banco de dados**. 3. ed. Porto Alegre: AMGH, 2011.

SETZER, Valdemar W.; SILVA, Flavio Soares Corrêa da. **Banco de dados**: aprenda o que são, melhore seu conhecimento, construa os seus. São Paulo: Blucher, 2005.

Capítulo 4

Introdução à SQL

Finalmente chegamos na parte mais importante deste livro. Para operar um SGBD relacional é preciso intimidade com a linguagem SQL (Structured Query Language, ou linguagem de consulta estruturada). Com a SQL, inventada pela IBM na década de 1970 (RAMAKRISHNAN; GEHRKE, 2011), podemos executar comandos para criar, alterar e remover tabelas dos bancos de dados (além de índices). Podemos consultar, incluir, alterar e excluir registros das tabelas. As consultas são, aliás, um dos maiores atrativos da SQL. A sintaxe para consultas é relativamente simples de aprender, mas ainda assim bem expressiva, de modo que muitas consultas complexas podem ser realizadas com apenas um comando. A SQL possibilita ainda o controle de transações para que diferentes comandos sejam executados de forma atômica em uma única operação, como veremos mais adiante neste capítulo.

1 Data Manipulation Language (DML)

A linguagem de manipulação de dados é o *subset* da SQL que nos possibilita criar, alterar e excluir registros em tabelas de um banco de dados (RAMAKRISHNAN; GEHRKE, 2011).

Considere a existência de uma tabela de municípios com as colunas *nome* e *uf*. Para inserir um novo município nessa tabela, podemos executar o comando:

```
INSERT INTO municipio (nome, uf) VALUES ('São Paulo', 'SP');
```

A sintaxe geral do comando *INSERT* é:

```
INSERT INTO <nome_tabela> (<nomes_colunas>) VALUES
(<valores>);
```

IMPORTANTE

Fique atento: em comandos SQL, nomes de colunas não vão entre aspas simples, já os valores, sim. Em geral, qualquer valor que esteja sendo inventado na hora, se for texto, deve ter aspas. Já uma referência a algo que já existe no banco (como uma coluna) não vai entre aspas. Observação: valores numéricos não vão entre aspas.

Para alterar um ou mais registros, utilizamos o comando *UPDATE*. Supondo que alguém tenha executado o seguinte comando:

```
INSERT INTO municipio (nome, uf) VALUES ('Ozasco', 'SP');
```

Temos uma situação incorreta, pois "Osasco" não se escreve com "z". Para consertar isso:

```
UPDATE municipio SET nome = 'Osasco' WHERE nome = 'Ozasco';
```

A sintaxe geral do comando *UPDATE* é:

```
UPDATE <nome_tabela> SET <nome_coluna> = <valor> WHERE
<condição>;
```

Já a deleção é muito parecida com a alteração. Se quisermos excluir o município de Osasco, podemos utilizar o seguinte comando:

```
DELETE FROM municipio WHERE nome = 'Osasco';
```

IMPORTANTE

Se você fizer um *DELETE* sem a cláusula *WHERE,* vai deletar todos os registros da tabela. O mesmo vale para o *UPDATE*: sem o *WHERE* o comando vai alterar todos os registros da tabela. Então muito cuidado!

2 Data Definition Language (DDL)

A linguagem de definição de dados é o *subset* da SQL que dá suporte à criação, exclusão e alteração das definições de estruturas das tabelas de um banco de dados (RAMAKRISHNAN; GEHRKE, 2011).

Para criar a tabela de municípios com a SQL, podemos executar:

```
CREATE TABLE municipio (
       nome text,
       uf varchar(2)
);
```

Note que, além do nome de cada coluna, especificamos também o tipo da coluna. O tipo *varchar(2)* indica que o conteúdo da coluna *uf* só pode ter dois caracteres. Alguns outros tipos existentes no MySQL são: *date, datetime, decimal, int*, entre outros.

Se depois quisermos acrescentar novas colunas, podemos fazer da seguinte forma:

```
ALTER TABLE municipio ADD COLUMN populacao int;
ALTER TABLE municipio ADD COLUMN area int;
```

Se quiséssemos criar a tabela com uma chave primária com autoincremento, poderíamos fazer da seguinte forma:

```
CREATE TABLE municipio (
id int auto_increment primary key,
nome text,
uf varchar(2)
);
```

Nesse caso, ao inserir um registro na tabela não precisamos especificar o ID:

```
> table municipio;
+----+------------+------+
| id | nome       | uf   |
+----+------------+------+
|  1 | São Paulo  | SP   |
|  2 | Osasco     | SP   |
+----+------------+------+
> INSERT INTO municipio (nome, uf) VALUES ('Diadema',
'SP');
> table municipio;
+----+------------+------+
| id | nome       | uf   |
+----+------------+------+
|  1 | São Paulo  | SP   |
|  2 | Osasco     | SP   |
|  3 | Diadema    | SP   |
+----+------------+------+
```

Note que mesmo sem especificarmos o valor do id no comando *INSERT*, o registro de Diadema ganhou o id 3.

E veja o que acontece se tentarmos inserir um ID repetido:

```
> INSERT INTO municipio (id, nome, uf) VALUES (2,
'Ozasco', 'SP');
ERROR 1062 (23000): Duplicate entry '2' for key
'municipio.PRIMARY'
```

O MySQL não permite que isso ocorra. Para criar uma tabela de pessoas com uma chave estrangeira para a tabela de municípios, indicando a que município pertence a pessoa, pode-se utilizar o seguinte comando:

```
CREATE TABLE pessoa (
id int auto_increment primary key,
nome text,
```

```
municipio_id int,
INDEX municipio_idx (municipio_id),
FOREIGN KEY (municipio_id)
REFERENCES municipio(id)
);
```

Dessa forma, na tabela *pessoa*, a coluna *municipio_id* conterá o id do município da pessoa. Repare que, além da restrição de chave estrangeira, foi criado também um índice, pois é uma boa prática que todos os campos que sejam chaves estrangeiras tenham um índice. No caso de chaves primárias, o próprio MySQL já se encarrega de criar automaticamente os índices.

Veja o que acontece se tentarmos inserir uma pessoa pertencente a um município inexistente:

```
> INSERT INTO pessoa(nome, municipio_id) VALUES ('Nick
Fury', 999);
    ERROR 1452 (23000): Cannot add or update a child
row: a foreign key constraint fails (`temp`.`pessoa`,
CONSTRAINT `pessoa_ibfk_1` FOREIGN KEY (`municipio_id`)
REFERENCES `municipio` (`id`))
```

O MySQL não permite que isso ocorra.

3 Data Query Language (DQL)

Os recursos para consultas da SQL são muito ricos, cobri-los de forma completa está fora do nosso escopo. Faremos aqui apenas uma breve introdução.

As consultas na SQL são feitas com o comando *SELECT*, sendo a estrutura-base desse comando a seguinte:

```
SELECT [DISTINCT] <colunas ou funções agregadoras>
FROM <tabelas>
WHERE <condições>
GROUP BY <colunas>
ORDER BY <colunas> [DESC]
LIMIT n
```

Vamos explorar essas opções com a base de dados sobre os homicídios ocorridos nos municípios brasileiros no ano de 2017 (GAZETA DO POVO, 2019).

Para filtrarmos sobre quais registros queremos obter informações, utilizamos a cláusula *WHERE*. Para saber a taxa de homicídios de Manaus em 2017, por exemplo, podemos utilizar o comando:

```
> SELECT * FROM homicidios WHERE municipio = "Manaus";
uf          municipio   taxa        homicidios  população
----------  ----------  ----------  ----------  ----------
AM          Manaus      55.9        1187        2130264
```

No comando acima, o asterisco ("*") se refere a "todas as colunas", enquanto "homicidios" se refere à tabela onde os dados estão. Já a condição (*municipio = "Manaus"*) compara os dados de uma coluna com um valor literal (*"Manaus"*). A condição pode ser composta para que saibamos de uma vez a situação de dois municípios:

```
> SELECT * FROM homicidios WHERE municipio = "Manaus" or
municipio = "Altamira";
uf          municipio   taxa        homicidios  população
----------  ----------  ----------  ----------  ----------
PA          Altamira    133.7       149         111435
AM          Manaus      55.9        1187        2130264
```

Além do *or*, poderíamos utilizar também os operadores lógicos *and*, *not* e *in*. Aliás, o comando anterior poderia ser reescrito, com o mesmo efeito, usando o *in*:

```
> SELECT * FROM homicidios WHERE municipio in (“Manaus”,
“Altamira”);
```

Para apresentar apenas algumas colunas, podemos trocar o “*” pelo nome das colunas desejadas:

```
> SELECT municipio, uf, homicidios FROM homicidios WHERE
municipio = “Manaus” or municipio = “Altamira”;
municipio   uf          homicidios
----------  ----------  ----------
Altamira    PA          149
Manaus      AM          1187
```

Um recurso muito útil é a realização de agrupamentos de registros. Para cada grupo, podemos então aplicar uma função agregadora, como o *count*, que conta quantos registros há em cada grupo, ou o *sum*, que soma os valores de uma coluna dentro de cada grupo. Como exemplo, vamos ver o total de homicídios em cada UF (ou seja, vamos realizar um agrupamento por UF e para cada um deles ver o valor da soma dos homicídios de cada município):

```
> SELECT uf, sum(homicidios) FROM homicidios GROUP BY uf;
uf          sum(homicidios)
----------  ---------------
AC          324
AL          755
AM          1204
AP          319
BA          3912
…
```

Note no exemplo anterior o uso da cláusula *GROUP BY*. Nesse caso é interessante também ordenar os registros; para isso, use a cláusula *ORDER BY*:

```
> SELECT uf, sum(homicidios) as soma_homicidios FROM
homicidios GROUP BY uf ORDER BY soma_homicidios DESC;
uf          soma_homicidios
----------  ---------------
RJ          5426
BA          3912
SP          3456
CE          3243
PA          2865
PE          2654
RS          2206
MG          2076
GO          1756
PR          1506
...
```

O *ORDER BY* pode ser seguido pelo termo *DESC* para que a ordenação seja em ordem decrescente. Se o *DESC* for omitido, a ordenação será em ordem crescente. Note também como a coluna *sum(homicidios)* ganhou o apelido de *soma_homicidios*. Esse recurso de apelidar a coluna é útil para referenciá-la mais adiante no comando (em nosso caso, logo após o *ORDER BY*). Agora, se quisermos ver apenas os cinco primeiros registros da lista, podemos utilizar a cláusula *LIMIT*:

```
> SELECT uf, sum(homicidios) as soma_homicidios FROM
homicidios GROUP BY uf ORDER BY soma_homicidios DESC LIMIT 5;
uf          soma_homicidios
----------  ---------------
RJ          5426
BA          3912
SP          3456
CE          3243
PA          2865
```

Com o *ORDER BY* e o *LIMIT* podemos também ver os cinco municípios mais violentos do Brasil, de acordo com essa tabela:

```
> SELECT municipio, uf, taxa FROM homicidios ORDER BY taxa
DESC LIMIT 5;
municipio                uf          taxa
-----------------------  ----------  ----------
Lauro de Freitas         BA          99
Camaçari                 BA          98.1
Caucaia                  CE          96.6
Nossa Senhora do Socorro SE          96.3
Cabo de Santo Agostinho  PE          94
```

Ou então só os cinco mais violentos do estado de São Paulo:

```
> SELECT municipio, taxa FROM homicidios WHERE uf = 'SP'
ORDER BY taxa DESC LIMIT 5;
municipio             taxa
--------------------  ----------
Franca                9.1
Birigui               8.9
Santa Bárbara d'Oeste 8.5
Sertãozinho           8.5
Catanduva             8.4
```

PARA SABER MAIS

Em um sistema completo, a SQL não basta por si só. Em geral, é preciso combinar as capacidades da principal linguagem de programação do sistema com a SQL para obter resultados úteis para o usuário (SETZER; SILVA, 2005). Um exemplo dessa ação combinada é a paginação, na qual dados de uma tabela são exibidos aos poucos ao usuário, pois exibir todos de uma vez poderia ser uma operação muito pesada e mesmo não agradável do ponto de vista da experiência do usuário. Do lado da SQL, a paginação pode ser realizada com o apoio da cláusula *LIMIT*.

Já com o *count* podemos ver quantos municípios há em cada UF nesta tabela:

```
> SELECT uf, count(*) FROM homicidios GROUP BY uf;
uf          count(*)
----------  ----------
AC          1
AL          2
AM          2
AP          2
BA          17
CE          9
…
```

Podemos ver também o total de registros na tabela:

```
select count(*) from homicidios;
count(*)
----------
310
```

Por fim, a cláusula *DISTINCT* é útil para revelar todos os elementos de certo domínio, por exemplo, a lista de todas as UFs existentes em nossa tabela:

```
> SELECT DISTINCT uf from homicidios;
uf
----------
CE
PA
RN
BA
RJ
RS
SE
...
```

PARA SABER MAIS

Essa foi apenas uma degustação da SQL enquanto linguagem de consulta, mas muito mais pode ser feito. Um recurso extra bem importante é o *join*, que possibilita a consulta concomitante em dados de mais de uma tabela com um único comando.

Outro conceito muito útil é o de subconsultas, em que temos um *SELECT* dentro de outro *SELECT*. Isso também pode ser feito com o uso de *VIEWS*, tabelas virtuais que não existem fisicamente, mas que facilitam as consultas.

Há vários lugares na internet que você pode utilizar para se aprofundar sobre a SQL enquanto linguagem de consulta.

4 Data Transaction Language (DTL)

Um importante recurso do banco de dados é o uso de transações. Com transações podemos definir sequências de comandos que devem ser executadas atomicamente, ou seja, ou todos os comandos são efetivados ou nenhum é. A SQL também fornece recursos para que o usuário do banco de dados possa explicitamente controlar aspectos de como uma transação deve ser executada (RAMAKRISHNAN; GEHRKE, 2011).

Considere a seguinte sequência de comandos que visa transferir R$ 123,00 de uma conta bancária (id = 1324) para outra conta (id = 7634):

```
UPDATE conta_corrente SET saldo = saldo - 123 WHERE id =
1324;
UPDATE conta_corrente SET saldo = saldo + 123 WHERE num =
7634;
```

No exemplo acima, o segundo comando tem um erro: ele está usando *num* no lugar de *id*. Nesse caso teremos um erro na execução do segundo comando. Mas o primeiro comando já terá sido executado,

o que significa que o dinheiro terá sumido e ido para lugar nenhum. Inadmissível!

Para evitar que isso aconteça basta utilizar os comandos *BEGIN* e *COMMIT* para envolver os comandos em uma transação:

```
BEGIN;
UPDATE conta_corrente SET saldo = saldo - 123 WHERE id =
1324;
UPDATE conta_corrente SET saldo = saldo + 123 WHERE num =
7634;
COMMIT;
```

Nesse caso, quando houver um erro na segunda linha, a transação será abortada, e todos os efeitos dos comandos anteriores desaparecerão. Ou seja, o dinheiro continuará na conta 1324.

Considerações finais

Neste capítulo, vimos como a SQL pode ser utilizada para criar tabelas, manipular os registros de dados, consultar os dados e, por fim, controlar transações. Vale lembrar também que o aprendizado deste capítulo vale não somente para o MySQL, mas também para praticamente todos os outros SGBDs relacionais (PostgreSQL, SQLite, etc.).

Referências

GAZETA DO POVO. **Atlas da Violência 2019 por municípios**. 2019. Disponível em: https://infograficos.gazetadopovo.com.br/seguranca-publica/atlas-da-violencia-2019-por-municipios/. Acesso em: fev. 2019.

RAMAKRISHNAN, Raghu; GEHRKE, Johannes. **Sistemas de gerenciamento de banco de dados**. 3. ed. Porto Alegre: AMGH, 2011.

SETZER, Valdemar W.; SILVA, Flavio Soares Corrêa da. **Banco de dados**: aprenda o que são, melhore seu conhecimento, construa os seus. São Paulo: Blucher, 2005.

Capítulo 5

Normalização

Um bom projeto de banco de dados deve balancear certas propriedades. Por um lado, um projeto com dados redundantes pode levar a um maior consumo de disco e anomalias durante a manutenção dos dados. Por outro lado, a eliminação completa de redundância poderá penalizar o desempenho da solução. Neste capítulo, entenderemos o que significa redundância nesse contexto e como combatê-la por meio do processo de normalização e suas formas normais. Também discutiremos situações nas quais vale a pena manter a redundância.

Pelo fato de que o processo de normalização é apoiado sobre um formalismo matemático, neste capítulo será feito maior uso da nomenclatura teórica, a saber: relações para denominar tabelas, tuplas para denominar linhas e atributos para denominar colunas (DATE, 2004).

1 Conceitos

Uma base de dados sem redundância é aquela que não armazena mais informação do que o necessário. Toda vez que colocamos na base de dados algum dado que se retirado não geraria perda de informação, estamos adicionando redundância à base. Um exemplo de redundância é armazenar o mesmo dado em duas tabelas. Outro exemplo é quando o valor de certa coluna pode ser derivado a partir de cálculos envolvendo outras colunas.

Um esquema com redundância leva ao consumo de espaço em disco maior que o estritamente necessário, o que implica diretamente custos financeiros (embora, em muitos casos, esse custo possa ser desprezível). Além disso, a redundância provoca anomalias durante a manutenção dos dados. Se a informação está em dois lugares, as operações de inserção, atualização e exclusão não podem esquecer de sempre considerar todas as ocorrências dos valores a serem manipulados. No caso de colunas que são função de cálculos envolvendo outras colunas, as anomalias envolvem também a inserção ou a atualização de valores que não respeitem à regra de cálculo definida. Mais ainda, se todos os registros de uma tabela forem excluídos, pode-se perder a relação de cálculo entre as colunas.

Por todos esses problemas, os projetistas de bancos de dados procuram, via de regra, evitar a redundância, sendo a normalização o processo que busca reduzir a redundância em um projeto de banco de dados.

No entanto, por vezes, não é preciso normalizar uma base de dados ao máximo possível. Um banco totalmente normalizado, por exemplo, não deveria conter valores calculados (recuperáveis a partir do valor de outras colunas). Mas pode ser interessante manter essas colunas por questão de desempenho. Considere como exemplo uma tabela que representa ruas em um mapa. Para isso, essa tabela possui as colunas *p1x* e *p1y,* representando o par coordenado (X, Y) de um dos extremos

da rua, e as colunas *p2x* e *p2y*, representando o outro extremo da rua. Se estamos interessados no comprimento dessas ruas, podemos armazenar a distância entre os extremos da rua em uma coluna *comprimento*. Se quisermos fazer, por exemplo, uma consulta pesquisando ruas com comprimentos menores que um quilômetro, a ausência da coluna *comprimento* implicará a necessidade de percorrer toda a tabela e calcular o comprimento de cada rua[1] existente na tabela, o que pode ser muito lento. Por outro lado, com a presença da coluna *comprimento*, podemos até mesmo criar um índice para essa coluna a fim de melhorar o tempo de resposta de nossa consulta de exemplo.

Antes de partir para o processo de normalização, apoiado pelas definições das formas normais, veremos ainda mais alguns conceitos (ELMASRI; NAVATHE, 2018; RAMAKRISHNAN; GEHRKE, 2011) necessários para compreender a próxima seção.

Uma superchave é um conjunto de colunas da tabela que, no conjunto, nunca se repete entre os registros. Dito de outra forma, para uma superchave *{c1, c2, c3}*,[2] se existe um registro com os valores c1 = 1, c2 = 2 e c3 = 3, então certamente não haverá nenhum outro registro da tabela no qual c1 = 1, c2 = 2 e c3 = 3.

Uma chave é uma superchave mínima. Considere a chave *{c1, c2}*: se removermos *c1* ou *c2* da chave, então o conjunto resultante não é mais uma chave. Já com superchaves pode acontecer que uma coluna seja removida e o conjunto resultante ainda seja uma superchave.

Um atributo primário diz respeito a uma coluna que pertence a alguma chave da tabela. Já um atributo não primário corresponde a uma coluna que não participa de nenhuma das chaves da tabela.

1 Lembre-se, conforme se deduz pelo teorema de Pitágoras, a distância entre dois pontos P1 e P2, sendo P1 = (p1x, p1y) e P2 = (p2x, p2y) é $\sqrt{((p1x - p2x)^2 + (p1y - p2y)^2)}$.

2 Obs.: neste e em outros capítulos, as chaves ("{" e "}") são usadas no contexto da notação de conjuntos.

Uma dependência funcional X → Y significa que os valores de um conjunto de colunas X determinam os valores de um conjunto de colunas Y. No exemplo da tabela que armazena ruas da cidade, o comprimento da rua é funcionalmente dependente das coordenadas dos extremos da rua.

Uma dependência funcional trivial X → Y é quando todas as colunas de Y estão também em X. Normalmente não consideraremos esse tipo de dependência nas análises das formas normais.

Uma dependência funcional X → Y é total se para qualquer coluna c em X, X-{c} não determina Y, ou seja, em uma dependência total, todas as colunas de X são necessárias para que a dependência funcione.

2 Formas normais

A forma normal de uma tabela refere-se ao grau de normalização de uma tabela (ELMASRI; NAVATHE, 2018). Ou seja, quanto mais alta for a forma normal de uma tabela, mais normalizada a tabela estará. Temos assim a primeira forma normal (1FN), a segunda forma normal (2FN) e a terceira forma normal (3FN), de modo que se uma tabela está na 3FN, ela está também na 2FN, e se ela está na 2FN, está também na 1FN.

Uma vantagem de trabalhar com o conceito de formas normais é possibilitar que o projetista de banco de dados possa analisar a forma normal de certa tabela e, a partir disso, estar ciente dos problemas que podem surgir nesse esquema e dos problemas que seguramente não surgiram em dada forma normal (RAMAKRISHNAN; GEHRKE, 2011).

2.1 A primeira forma normal (1FN)

Uma tabela está na 1FN se cada coluna possui apenas valores atômicos, isto é, não possui listas ou conjuntos contidos em uma coluna.

A tabela 1 mostra um exemplo de violação da primeira forma normal: em uma única coluna armazenamos os nomes de todos os jogadores de um time de futebol.

Tabela 1 – Tabela de partidas de futebol, com violação da 1FN

TIME1	TIME2	DATA	ESCALACAOTIME1	ESCALACAOTIME2	PLACAR
Corinthians	Juventude	2003-09-28	Rubinho, Coelho, Marquinhos, Anderson, Moreno...	Márcio Angonese, Neto Gaúcho, Índio, Marcão, Hugo...	1 × 6
Brasil	Alemanha	2014-07-08	Júlio César, David Luiz, Fernandinho, Marcelo, Hulk...	Manuel Neuer, Benedikt Howedes, Mats Hummels, Sami Khedira...	1 × 7

Fonte: adaptado de O Gol (2003) e Fifa (2014).

IMPORTANTE

Em sistemas computacionais, em geral, datas são representadas no formato aaaa-mm-dd. Ou seja, ano-mês-dia, sendo o ano com quatro dígitos, o mês com dois dígitos e o dia com dois dígitos. Na SQL é possível utilizar esse formato para escrever condições no formato *coluna_do_tipo_data > '2020-04-01'*.

Uma motivação interessante para a adoção desse formato é que ele é bem conveniente para ambientes que não possuem um tipo especial para datas (como é o caso do SQLite, por exemplo), já que a ordem lexicográfica ("alfabética") dos valores coincide com a ordem cronológica das datas. Isso significa que mesmo que uma coluna seja do tipo texto, a comparação *coluna_do_tipo_texto_contendo_data > '2020-04-01'* funcionará.

Para resolver o problema da tabela 1 devemos decompor a tabela original em duas: uma tabela para a partida, com as colunas *{Time1, Time2, Data, Placar}*, e outra para a escalação. Chamamos de decomposição esse processo de quebrar uma tabela em outras duas ou mais.

No exemplo, a chave primária (aquilo que identifica univocamente uma partida) é o conjunto *{Time1, Time2, Data}* (dois times não se enfrentam duas vezes na mesma data), então a tabela de escalação poderia ter as seguintes colunas: *{Time1, Time2, Data, Time, NomeJogador}*, em que *Time* indica se é o time 1 ou o time 2. Outra opção, com menos chance de erro, seria ter duas tabelas: *EscalaçãoTime1* e *EscalaçãoTime2*, sendo que as duas teriam a mesma estrutura: *{Time1, Time2, Data, NomeJogador}*, com uma linha para cada jogador.

Alguém poderia também contestar que o placar contém duas informações (a quantidade de gols do time 1 e a quantidade de gols do time 2). Essa consideração é subjetiva e depende de se alguma aplicação fará uso dessas informações de forma separada (totalizando todos os gols feitos por um time em todas as suas partidas, por exemplo). Se a objeção for aceita, para que a tabela esteja na 1FN deve-se então decompor a coluna *Placar* nas colunas *QuantidadeGolsTime1* e *QuantidadeGolsTime2*.

O resultado final da normalização da tabela 1 na 1FN são as tabelas 2, 3 e 4.

Tabela 2 – Tabela de partidas de futebol, respeitando a 1FN

TIME1	TIME2	DATA	QUANTIDADEGOLSTIME1	QUANTIDADEGOLSTIME2
Corinthians	Juventude	2003-09-28	1	6
Brasil	Alemanha	2014-07-08	1	7

Tabela 3 – Tabela de escalação "time 1" associada à tabela 2

TIME1	TIME2	DATA	NOMEJOGADOR
Corinthians	Juventude	2003-09-28	Rubinho
Corinthians	Juventude	2003-09-28	Coelho

(cont.)

TIME1	TIME2	DATA	NOMEJOGADOR
Corinthians	Juventude	2003-09-28	Marquinhos
...			
Brasil	Alemanha	2014-07-08	Júlio César
Brasil	Alemanha	2014-07-08	David Luiz
Brasil	Alemanha	2014-07-08	Fernandinho
...			

Tabela 4 – Tabela de escalação "time 2" associada à tabela 2

TIME1	TIME2	DATA	NOMEJOGADOR
Corinthians	Juventude	2003-09-28	Márcio Angonese
Corinthians	Juventude	2003-09-28	Neto Gaúcho
Corinthians	Juventude	2003-09-28	Índio
...			
Brasil	Alemanha	2014-07-08	Manuel Neuer
Brasil	Alemanha	2014-07-08	Benedikt Howedes
Brasil	Alemanha	2014-07-08	Mats Hummels
...			

2.2 A segunda forma normal (2FN)

Iniciemos com o caso de tabelas que possuem uma única chave primária, inclusive as que possuem a chave composta por múltiplas colunas. Nesse caso, uma tabela está na 2FN se ela está na 1FN e se todos os seus atributos não primários possuem dependência funcional total

da chave primária da tabela. Dito de outra forma, a 2FN não admite que uma coluna (que não faça parte da chave) seja determinada por apenas parte da chave primária. Por consequência, esse problema não atinge as tabelas cuja chave primária é constituída de uma única coluna.

Há também tabelas nas quais duas (ou mais) colunas independentes podem assumir o papel de chave primária. Nesse caso, a definição de pertencimento à 2FN é ligeiramente estendida, exigindo que todo atributo não primário possua dependência funcional total de todas as chaves possíveis.

Vamos a um exemplo de violação da 2FN. Considere a tabela 5, uma tabela de músicas. Como diferentes artistas podem ter músicas com o mesmo nome (exemplo na tabela: a música "Time"), a chave primária não é somente o nome da música, mas sim a tupla *{nome, artista}*; ou seja, o que identifica unicamente uma música é o nome da música com o nome do artista da música. Temos também nessa tabela uma coluna identificando o gênero musical. No entanto, pelo menos no caso dessa tabela, o gênero se refere na verdade ao gênero do artista, e não da música. Dito de outra forma, o valor da coluna *artista* determina funcionalmente o valor da coluna *gênero*. Sendo a coluna *artista* apenas parte da chave primária, temos que essa dependência funcional viola a 2FN (temos parte da chave primária determinando o valor de uma outra coluna não primária).

Tabela 5 – Tabela de músicas, com violação da 2FN

NOME	ARTISTA	GÊNERO
Nothing to say	Angra	Power metal
Time	Angra	Power metal
Time	Pink Floyd	Rock progressivo
Chega de saudade	João Gilberto	Bossa nova

(cont.)

NOME	ARTISTA	GÊNERO
Bim bom	João Gilberto	Bossa nova
Roots bloody roots	Sepultura	Thrash metal
Ratamahatta	Sepultura	Thrash metal
Assum Preto	Luiz Gonzaga	Forró
Tuatha de Danann	Tuatha de Danann	Folk metal

Para normalizar a tabela 5 na 2FN, devemos criar uma nova tabela de artistas (tabela 6) e excluir a coluna *gênero* da tabela de músicas (tabela 7). Dessa forma, nenhuma das tabelas geradas nesse processo de decomposição estará violando a 2FN. Repare que, diferentemente do que ocorre na tabela de músicas, os artistas não se repetem na tabela de artistas.

Tabela 6 – Tabela de artistas

NOME	GÊNERO
Angra	Power metal
Pink Floyd	Rock progressivo
João Gilberto	Bossa nova
Sepultura	Thrash metal
Luiz Gonzaga	Forró
Tuatha de Danann	Folk metal

Tabela 7 – Tabela de músicas, respeitando a 2FN

NOME	NOME_ARTISTA
Nothing to say	Angra
Time	Angra

(cont.)

NOME	NOME_ARTISTA
Time	Pink Floyd
Chega de saudade	João Gilberto
Bim bom	João Gilberto
Roots bloody roots	Sepultura
Ratamahatta	Sepultura
Assum Preto	Luiz Gonzaga
Tuatha de Danann	Tuatha de Danann

2.3 A terceira forma normal (3FN)

Uma tabela está na 3FN se estiver na 2FN e, se para toda dependência funcional não trivial X → A da tabela, uma das seguintes afirmações é verdadeira: a) *X* é uma superchave da tabela ou b) *A* é parte de uma chave da tabela.

O caso típico de violação da 3FN é quando uma coluna é determinada por outra que não seja chave. Considere novamente a tabela que armazena as ruas de uma cidade e que admita a existência de ruas paralelas iniciando e terminando nos mesmos pontos; dessa forma, as coordenadas dos extremos da rua (p1x, p1y, p2x, p2y) não podem ser consideradas chaves primárias. Nesse exemplo, a existência da coluna *distância* viola a 3FN, pois a coluna *distância* é determinada pelas colunas não primárias *{p1x, p1y, p2x, p2y}* (se a coluna *distância* fosse removida, não estaríamos perdendo informação).

2.4 A forma normal de Boyce-Codd (BCNF)

A BCNF é a forma normal mais desejável do ponto de vista da eliminação da redundância. Uma tabela está na BCNF se todo atributo não

primário respeita a condição de não ser funcionalmente determinado por outro atributo que não seja chave. Dito de outra forma mais geral, se a tabela está na BCNF e há uma dependência funcional não trivial X → Y, então X é uma superchave da tabela. Note que, embora a definição da BCNF seja aparentemente um recorte da definição da 3FN, a BCNF é mais restritiva: toda tabela na BCNF está na 3FN.

Como consequência, o mesmo exemplo dado anteriormente para a violação da 3FN viola também a BCNF. Um exemplo, tomado de Ramakrishnan e Gehrke (2011), de tabela que está na 3FN, mas não na BCNF é o seguinte: considere uma tabela de reservas de barcos com as colunas *{marinheiro, barco, data, cartão de crédito}* (tabela 8), sendo a chave primária composta por *{marinheiro, barco, data}*. Nesse exemplo, cada marinheiro utiliza sempre o mesmo cartão de crédito, o que faz com que haja uma dependência funcional do marinheiro para o cartão e do cartão para o marinheiro. Nesse caso, a tupla *{barco, data, cartão de crédito}* também é uma chave primária válida. A questão aqui é a relação cartão → marinheiro: o lado esquerdo não é uma chave, então a tabela não respeita a BCNF; mas o lado direito é um atributo primário, o que faz com que a tabela esteja na 3FN (respeita a condição *b* da 3FN). Perceba que nesse exemplo há uma redundância: se todo marinheiro usa apenas um único cartão de crédito, a relação entre marinheiro e cartão de crédito poderia ser extraída para uma nova tabela.

Tabela 8 – Tabela de reserva de barcos, respeitando a 3FN, mas não a BCNF

MARINHEIRO	BARCO	DATA	CARTAO DE CREDITO
João Gomes	Rainha do mar	2020-03-13	4324763598471254
Roberto da Silva	Princesa das águas	2020-03-13	4224773588471214
Wesley Figueiredo	Rainha do mar	2020-03-14	4727763798471258
Carlos Monteiro	Embaixatriz fluvial	2020-03-14	4364763568471444
João Gomes	Sereia	2020-03-15	4324763598471254

Apresentemos agora um breve quadro comparativo entre as formas normais:

Quadro 1 – Resumo sobre as formas normais

FORMA NORMAL	CONDIÇÃO PARA SATISFAÇÃO
1FN	Todos os valores são atômicos, isto é, o banco de dados não possui listas ou conjuntos contidos em uma coluna.
2FN	Todo atributo não primário deve ser funcionalmente dependente de todas as chaves possíveis da tabela.
3FN	Para toda dependência funcional não trivial $X \rightarrow A$, então vale: a) *X* é uma superchave ou b) *A* é parte de uma chave.
BCNF	Para toda dependência funcional não trivial $X \rightarrow A$, então *X* é uma superchave.

2.5 Propriedades da 3FN e da BCNF

O processo de decomposição de uma tabela possui as seguintes propriedades binárias:

1. Junção sem perda: possibilita recuperar a tabela decomposta a partir das tabelas geradas pela decomposição.

2. Preservação de dependência: considerando uma restrição existente na tabela original, é possível garantir a mesma restrição por meio de restrições em cada uma das tabelas geradas pela decomposição. Ou seja, para garantir a restrição original não é preciso juntar os dados das tabelas geradas.

Uma decomposição pode ou não apresentar cada uma das propriedades listadas acima. Mas as duas propriedades são desejáveis. Embora a BCNF elimine mais redundância que a 3FN, ela garante apenas a propriedade 1, enquanto a 3FN garante ambas as propriedades. Essa é a principal motivação para a aplicação da 3FN. Caso uma tabela

seja decomposta sem respeitar a propriedade 2, a restrição deverá ser garantida pela lógica de negócio da aplicação, não podendo ser garantida pelas restrições do SGBD.

NA PRÁTICA

O SGBD fornece várias restrições (*constraints*) para garantir a integridade dos dados. Restrições de unicidade e chaves estrangeiras são exemplos. Outro exemplo é o uso de enumerações (MYSQL, 2020), que restringem os possíveis valores de uma coluna. Contudo, do ponto de vista da lógica de negócio, sempre existirão restrições lógicas que o banco não poderá garantir. Em um sistema de emissão de documentos, o valor do estado do processo deve seguir uma sequência com possibilidades predefinidas, por exemplo, após o documento estar *Emitido*, não é possível que o processo volte para o estado *Requisitado*. Portanto, a aplicação sempre deverá ter a responsabilidade pela semântica dos dados manipulados.

Por outro lado, a utilização de restrições do SGBD possui suas vantagens: caso mais de uma aplicação acesse o banco, não é preciso replicar a regra de restrição em ambas as aplicações. Além disso, a restrição no SGBD é mais adequada para lidar com questões de acesso concorrente; isso fica bem evidente na restrição de unicidade: caso ela seja deixada para a aplicação, duas aplicações atualizando o banco ao mesmo tempo poderiam inserir o mesmo valor em registros diferentes, assim violando a restrição. Se a restrição de unicidade estiver definida no banco de dados, ela nunca será quebrada.

Considerações finais

A atividade de modelagem entidade-relacionamento é altamente subjetiva. Como julgar a qualidade de uma modelagem? Neste capítulo, aprendemos uma forma objetiva para tal: podemos avaliar a modelagem de um banco de dados pelo grau de normalização de suas tabelas, usando como referência as formas normais. Cada tabela não normalizada é

uma oportunidade de melhoria do modelo – mas uma oportunidade que deve ser julgada levando em conta outras forças, como o desempenho final do sistema, uma vez que algumas decomposições podem aumentar o tempo de resposta para determinadas consultas.

Por fim, embora já existam definições para a quarta, quinta e até mesmo a sexta forma normal, o estudo das formas normais apresentadas neste capítulo já deve ajudar a construir uma intuição para detectar modelos potencialmente não normalizados, identificando assim possíveis pontos de melhoria no sistema.

Referências

DATE, C. J. **Introdução a sistemas de bancos de dados**. Rio de Janeiro: Campus, 2004.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de banco de dados**. 7. ed. São Paulo: Pearson, 2018.

FIFA. **Brazil *vs* Germany match report**. 2014. Disponível em: https://resources.fifa.com/mm/document/tournament/competition/02/40/19/83/eng_61_0708_bra-ger_fulltime_neutral.pdf. Acesso em: 8 abr. 2020.

MYSQL. **11.3.5 the ENUM type**. 2020. Disponível em: https://dev.mysql.com/doc/refman/8.0/en/enum.html. Acesso em: 8 abr. 2020.

O GOL. **Ficha técnica Juventude *vs* Corinthians**. 2003. Disponível em: https://www.ogol.com.br/jogo.php?id=502589ht. Acesso em: 8 abr. 2020.

RAMAKRISHNAN, Raghu; GEHRKE, Johannes. **Sistemas de gerenciamento de banco de dados**. 3. ed. Porto Alegre: AMGH, 2011.

Capítulo 6

Introdução à álgebra relacional

A SQL, a linguagem de consulta utilizada em bancos relacionais, é fortemente baseada em dois formalismos que definem operações sobre conjuntos matemáticos: a álgebra relacional e o cálculo relacional (ELMASRI; NAVATHE, 2010). Neste capítulo, estudaremos como os conceitos da álgebra relacional e do cálculo relacional se aplicam de forma prática à utilização da SQL.

Por estarmos falando em formalismos matemáticos, este capítulo fará maior uso da nomenclatura teórica, a saber: relações para denominar tabelas, tuplas para denominar linhas e atributos para denominar colunas.

1 Operações relacionais unárias

A primeira operação que veremos da álgebra relacional é a *SELEÇÃO*, que possibilita a seleção de um conjunto de tuplas a partir de uma relação original e de uma condição imposta sobre as tuplas. Trata-se de uma operação unária por ser aplicada a somente uma relação. Na SQL, corresponde à utilização do comado *SELECT* em combinação com a cláusula *WHERE*.

Já a *PROJEÇÃO* é uma operação relacional unária para selecionar os atributos de interesse de certa relação. Na SQL, trata-se da listagem de campos que vem logo após a palavra *SELECT*.

Uma distinção dessas operações teóricas (*SELEÇÃO* e *PROJEÇÃO*) para o uso prático da SQL é que as operações teóricas não admitem tuplas repetidas na relação resultante da operação. Isso porque na teoria matemática, um conjunto nunca possui elementos repetidos. Logo, *SELEÇÃO* e *PROJEÇÃO* teóricas correspondem na prática à utilização do *SELECT DISTINCT* da SQL.

Mas o mais importante de entender aqui é que o resultado de uma operação relacional é uma relação tanto quanto a relação de entrada da operação, essa relação resultante pode ser utilizada em outras operações relacionais. É aí que, na prática, surgem os chamados *subselects*, que consistem na utilização do comando *SELECT* da SQL aninhado a outro comando *SELECT*. Considere como exemplo uma tabela de restaurantes com as colunas *nome* e *bairro*. Com um comando da SQL podemos sumarizar quantos restaurantes há em cada bairro:

```
SELECT bairro, count(*) FROM restaurantes GROUP BY bairro;
```

Se quisermos identificar os bairros carentes de restaurantes (talvez para decidir onde abrir um novo restaurante), podemos verificar quais

bairros possuem menos de dez restaurantes. Uma forma de fazer isso é utilizar a relação obtida pelo *SELECT* anterior e fazer uma *SELEÇÃO* sobre essa relação, especificando a condição de que a quantidade de restaurantes deve ser menor que 10. Na prática, temos então o seguinte comando:

```
SELECT bairro FROM (SELECT bairro, count(*) as qtd_
restaurantes FROM restaurantes GROUP BY bairro) AS tabela_
bairros WHERE qtd_restaurantes < 10;
```

Nesse exemplo, o que vai entre parênteses resulta em uma relação, que é chamada de *tabela_bairros*. A partir dessa nova relação, *tabela_bairros*, fazemos um *SELECT* impondo a condição *qtd_restaurantes < 10*.

Outra operação unária existente e que acabamos de utilizar é a *REBATIZAR*, utilizada para nomear relações resultantes ou renomear colunas. A renomeação de colunas é particularmente útil para rebatizar colunas resultantes de funções de agregação, como o *count(*)* que é utilizado no exemplo. Na SQL, a operação *REBATIZAR* é realizada com a palavra-chave *AS* ("como", em inglês).

Contudo, por vezes encadear um *SELECT* dentro do outro pode exigir certo esforço cognitivo – talvez mais para quem vá ler a consulta no futuro do que para quem está escrevendo a consulta. Uma alternativa prática é criar uma *view* ("visão", em inglês) no banco de dados a partir do *SELECT* mais interno. A *view* funciona como se fosse uma tabela, mas sem que os dados da *view* estejam armazenados em disco. O exemplo de *subselect* apresentado no código anterior pode ser refeito da seguinte forma:

```
CREATE VIEW bairros AS SELECT bairro, count(*) as qtd_
restaurantes FROM restaurantes GROUP BY bairro;
SELECT bairro FROM bairros WHERE qtd_restaurantes < 10;
```

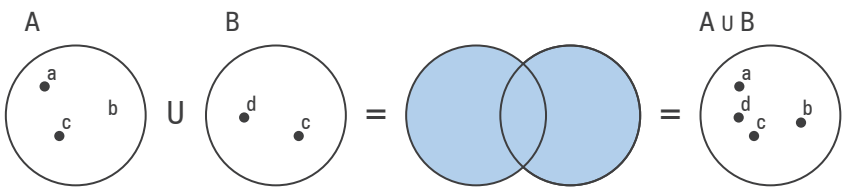
Nesse novo exemplo, primeiro criamos a *view* (*CREATE VIEW*). Depois realizamos uma consulta na *view* (*SELECT … FROM bairros*), sendo que a view corresponde ao *SELECT* interno do exemplo do *subselect*. Para selecionarmos apenas os bairros com menos de dez restaurantes, assim como feito no exemplo do *subselect*, colocamos também a restrição na consulta da *view* (*WHERE qtd_restaurantes < 10*).

2 Operações da álgebra relacional a partir da teoria dos conjuntos

Vamos agora tratar de operações correspondentes a operações matemáticas sobre conjuntos: *UNIÃO*, *INTERSECÇÃO* e *SUBTRAÇÃO*. Essas são operações binárias, pois precisam de dois conjuntos de entrada para produzir um conjunto de saída.

A operação de *UNIÃO* tem como entrada um conjunto A e um conjunto B, produzindo como saída um conjunto C contendo todos os elementos de A e todos os elementos de B. No exemplo de união da figura 1, um conjunto *{a, b, c}* unido do conjunto *{c, d}* resulta no conjunto *{a, b, c, d}*, que é composto por todos os elementos dos dois conjuntos de entrada, mas sem repetir o elemento *c*.

Figura 1 – Como funciona a operação de UNIÃO sobre conjuntos



Considere que uma base de dados de gestão fabril, em vez de possuir uma única tabela de funcionários, possui tabelas separadas para gestores e para operadores. Isso pode ocorrer pelo fato de que cada tipo

de funcionário tem atributos específicos; em nosso exemplo, um operador está associado a um tipo de máquina que ele opera. Se precisarmos de uma listagem com os nomes de todos os funcionários, tanto gerentes quanto operadores, isso pode ser resolvido com o operador *UNION*.

Considere primeiro a saída em separado do *SELECT* em cada tabela:

```
> SELECT nome AS nome_gerente FROM gerentes;
+---------------------+
| nome_gerente        |
+---------------------+
| João da Silva       |
| Henrique de Almeida |
+---------------------+
> SELECT nome AS nome_operador FROM operadores;
+----------------+
| nome_operador  |
+----------------+
| Mateus Lopes   |
| João da Silva  |
+----------------+
```

Vamos agora unir os dois resultados para termos a relação de nomes de todos os funcionários:

```
> SELECT nome AS nome_funcionario FROM gerentes
  UNION
  SELECT nome AS nome_funcionario FROM operadores;
+---------------------+
| nome_funcionario    |
+---------------------+
| João da Silva       |
| Henrique de Almeida |
| Mateus Lopes        |
+---------------------+
```

Repare que havia um gerente e um operador com o mesmo nome: João da Silva. Nesse caso, o operador *UNION* da SQL se comportou conforme a operação teórica *UNIÃO*: a relação resultante não possui elementos repetidos. Nesse caso, isso pode não ser o desejável. Para resolver isso, podemos utilizar o *UNION ALL* (W3SCHOOLS, [*s. d.*]):

```
> SELECT nome AS nome_funcionario FROM gerentes
  UNION ALL
  SELECT nome AS nome_funcionario FROM operadores;
+--------------------+
| nome_funcionario   |
+--------------------+
| João da Silva      |
| Henrique de Almeida |
| Mateus Lopes       |
| João da Silva      |
+--------------------+
```

Perceba também que duas relações só podem ser unidas se tiverem a mesma quantidade de colunas. Tentemos desrespeitar essa regra, e o MySQL nos retornará um erro:

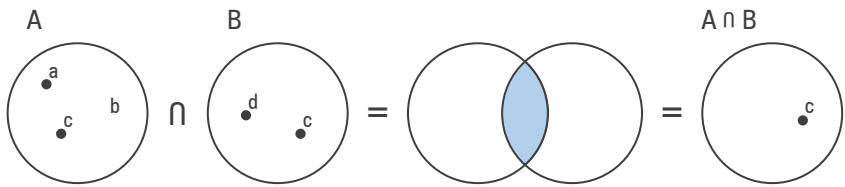
```
> SELECT nome FROM gerentes
  UNION
  SELECT nome, maquina from operadores;
ERROR 1222 (21000): The used SELECT statements have a
different number of columns
```

Contudo, o MySQL tolera a união de relações com a mesma quantidade de colunas, mesmo que não haja correspondência entre os nomes e até entre os tipos das colunas. No exemplo a seguir, vamos unir a relação de nomes de gerentes com os salários dos operadores. Nesse caso, os nomes das colunas são diferentes (*nome* e *salario*) e os tipos também são diferentes (*text* e *float*). Na prática, não faz muito sentido, mas funciona:

```
> SELECT nome FROM gerentes
    UNION
    SELECT salario FROM operadores;
+---------------------+
| nome                |
+---------------------+
| João da Silva       |
| Henrique de Almeida |
| 2000                |
| 2500                |
+---------------------+
```

A operação *INTERSECÇÃO* tem como resultado os elementos em comum das relações de entrada da operação. No exemplo da figura 2, a intersecção entre um conjunto *{a, b, c}* e um conjunto *{c, d}* resulta no conjunto *{c}*, uma vez que *c* é o único elemento presente em ambos os conjuntos de entrada.

Figura 2 – Como funciona a operação de INTERSECÇÃO sobre conjuntos



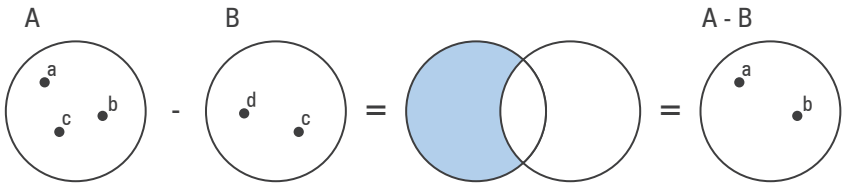
Na SQL, pode-se utilizar o operador *INTERSECT* para realizar uma operação de *INTERSECÇÃO*, com a mesma estrutura usada para o operador *UNION*:

```
relacao1 INTERSECT relacao2
```

A operação *SUBTRAÇÃO* tem como resultado os elementos do primeiro conjunto que não estão no segundo conjunto. No exemplo da figura 3, a diferença entre o conjunto *{a, b, c}* e o conjunto *{c, d}* é o

conjunto *{a, b}*: o elemento *c* foi removido do resultado, pois está em ambos os conjuntos de entrada. Note que, diferentemente da *UNIÃO* e da *INTERSECÇÃO*, a ordem das entradas da *SUBTRAÇÃO* faz diferença no resultado.

Figura 3 – Como funciona a operação de SUBTRAÇÃO sobre conjuntos



Na SQL, pode-se utilizar o operador *EXCEPT* para realizar uma operação de *SUBTRAÇÃO* com a mesma estrutura usada para o operador *UNION*:

```
relacao1 EXCEPT relacao2
```

Temos também a operação teórica de *DIVISÃO*. No exemplo de Elmasri e Navathe (2010), a *DIVISÃO* é utilizada para recuperar os empregados que trabalham em todos os projetos em que John Smith (um empregado em particular) trabalha. Considere a notação *c(R)* representando o conjunto dos atributos da relação *R* e *c(t)* representando o conjunto dos atributos de uma tupla. Na divisão *T = R ÷ S*, uma tupla *t* (com *c(t) = c(R) - c(S)*) aparece no resultado *T* se os valores em *t* aparecem em *R* em combinação com todas as tuplas de *S*. Retomando o exemplo, sendo *R* a relação que associa empregados a projetos e *S* a relação de projetos de John Smith, teremos como resultado todos os empregados de *R* que fizerem par com todos os projetos em que John Smith trabalha. Esse exemplo é mostrado nas tabelas 1, 2 e 3.

Tabela 1 – Relação R, possuindo as colunas ESSN e NRP, associando em cada linha um funcionário (ESSN) a um projeto (NRP)

ESSN	NRP
123456789	1
123456789	2
666884444	3
453453453	1
453453453	2
333445555	2
333445555	3
333445555	10
333445555	20
999887777	30
999887777	10
987987987	10
987987987	30
987654321	30
987654321	20
888665555	20

Fonte: adaptado de Elmasri e Navathe (2010, p. 117).

Tabela 2 – Relação S (projetos do John Smith), possuindo apenas uma coluna NRP com duas linhas, com os seguintes valores: 1 e 2

NRP
1
2

Fonte: adaptado de Elmasri e Navathe (2010, p. 117).

Tabela 3 – Relação R ÷ S, possuindo apenas a coluna SSN e apenas duas linhas, mostrando os empregados (retirados de R.ESSN) que em R apareceram associados tanto ao NRP 1 quanto ao NRP 2

SSN
123456789
453453453

Fonte: adaptado de Elmasri e Navathe (2010, p. 117).

A SQL não fornece um operador de divisão, mas podemos reproduzir o exemplo das tabelas 1, 2 e 3 com a seguinte sequência de comandos SQL (os comandos *TABLE* são colocados para ajudar a evidenciar o que está sendo feito):

```
> CREATE VIEW ssn_nrps_agregados AS SELECT essn, JSON_
ARRAYAGG(nrp) FROM ssn_nrps GROUP BY essn;
> TABLE ssn_nrps_agregados;
+-----------+----------------+
| essn      | nrps           |
+-----------+----------------+
| 123456789 | [1, 2]         |
| 333445555 | [2, 3, 10, 20] |
| 453453453 | [1, 2]         |
| 666884444 | [3]            |
| 888665555 | [20]           |
| 987654321 | [30, 20]       |
| 987987987 | [10, 30]       |
| 999887777 | [30, 10]       |
+-----------+----------------+
> CREATE VIEW smith_nrps_agregados AS SELECT JSON_
ARRAYAGG(nrp) AS nrps FROM smith_nrps;
> TABLE smith_nrps_agregados;
+--------+
| nrps   |
+--------+
| [1, 2] |
+--------+
> SELECT essn FROM ssn_nrps_agregados WHERE nrps = (SELECT
nrps FROM smith_nrps_agregados);
```

```
+-----------+
| essn      |
+-----------+
| 123456789 |
| 453453453 |
+-----------+
```

PARA SABER MAIS

A função agregadora *JSON_ARRAYAGG* concatena os valores de uma coluna em uma lista JSON (Java Script Object Notation). Essa lista é uma sintaxe para a troca de dados estruturados entre todas as linguagens de programação (ECMA INTERNATIONAL, 2017). No JSON, pares de chave-valor são separados por :, sendo que esses pares são agrupados em um conjunto delimitado pelos símbolos *{"* e *"}*. Além disso, os caracteres *["* e *"]* delimitam uma lista de grupos de chave-valor ou mesmo de valores atômicos.

Um exemplo de dado estruturado no formato JSON:

```
{ "produto" :
  { "tipo": "livro",
   "titulo" : "Seven Databases in Seven Weeks",
   "ano" : 2012,
   "editora" : { "nome" : "Pragmatic Bookshelf",
                 "endereco" : "Dallas, Texas" }
  },
  "palavras-chaves" : ["bancos de dados", "TI",
"programação", "NoSQL" ]
}
```

3 Operações relacionais de junção

Vamos agora a uma das partes mais importantes deste livro. Veremos como, em uma consulta só, associamos dados localizados

em diferentes tabelas que, em geral, estão relacionadas por uma chave estrangeira.

Considere a existência de duas tabelas: *pessoas* e *cachorros*, sendo que há uma associação entre essas tabelas para identificar que pessoa é dona de qual cachorro. Considere que essas tabelas possuem os seguintes esquemas e os seguintes valores (e repare que no exemplo há uma pessoa sem cachorro e um cachorro sem dono):

```
CREATE TABLE pessoas (
    id int,
    nome text);
CREATE TABLE cachorros (
    id int,
    nome text,
    id_dono int);
> TABLE pessoas;
+------+------------+
| id   | nome       |
+------+------------+
|    1 | João       |
|    2 | Maria      |
|    3 | Sigismundo |
|    4 | Magnólia   |
+------+------------+

> TABLE cachorros;
+------+-----------+---------+
| id   | nome      | id_dono |
+------+-----------+---------+
|   10 | Rex       |       1 |
|   11 | Brasinha  |       2 |
|   12 | Floquinho |       3 |
|   13 | Bidu      |       3 |
|   14 | Auau      |    NULL |
+------+-----------+---------+
```

Vamos agora introduzir mais uma operação binária relacional, que é o *PRODUTO CARTESIANO*, que combina todas as linhas de uma tabela

com todas as linhas de uma outra tabela. Na SQL, o produto cartesiano de duas tabelas é obtido colocando o nome das duas tabelas no comando *SELECT*, conforme o exemplo a seguir:

```
> SELECT * FROM pessoas, cachorros;
+------+------------+------+-----------+---------+
| id   | nome       | id   | nome      | id_dono |
+------+------------+------+-----------+---------+
|    4 | Magnólia   |   10 | Rex       |       1 |
|    3 | Sigismundo |   10 | Rex       |       1 |
|    2 | Maria      |   10 | Rex       |       1 |
|    1 | João       |   10 | Rex       |       1 |
|    4 | Magnólia   |   11 | Brasinha  |       2 |
|    3 | Sigismundo |   11 | Brasinha  |       2 |
|    2 | Maria      |   11 | Brasinha  |       2 |
|    1 | João       |   11 | Brasinha  |       2 |
|    4 | Magnólia   |   12 | Floquinho |       3 |
|    3 | Sigismundo |   12 | Floquinho |       3 |
|    2 | Maria      |   12 | Floquinho |       3 |
|    1 | João       |   12 | Floquinho |       3 |
|    4 | Magnólia   |   13 | Bidu      |       3 |
|    3 | Sigismundo |   13 | Bidu      |       3 |
|    2 | Maria      |   13 | Bidu      |       3 |
|    1 | João       |   13 | Bidu      |       3 |
|    4 | Magnólia   |   14 | Auau      |    NULL |
|    3 | Sigismundo |   14 | Auau      |    NULL |
|    2 | Maria      |   14 | Auau      |    NULL |
|    1 | João       |   14 | Auau      |    NULL |
+------+------------+------+-----------+---------+
```

O resultado acima não é de utilidade prática. Mas veja que podemos filtrar as linhas nas quais o ID do dono do cachorro corresponda ao ID da pessoa. Teremos, então, em uma única relação as pessoas e seus respectivos cachorros:

```
> SELECT pessoas.nome AS dono, cachorros.nome AS cachorro
FROM pessoas, cachorros WHERE cachorros.id_dono = pessoas.
id;
+------------+-----------+
| dono       | cachorro  |
+------------+-----------+
| João       | Rex       |
| Maria      | Brasinha  |
| Sigismundo | Floquinho |
| Sigismundo | Bidu      |
+------------+-----------+
```

Outra forma mais explícita de realizar a junção é utilizando a cláusula *JOIN*:

```
> SELECT pessoas.nome AS dono, cachorros.nome AS cachorro
FROM pessoas JOIN cachorros ON cachorros.id_dono =
pessoas.id;
+------------+-----------+
| dono       | cachorro  |
+------------+-----------+
| João       | Rex       |
| Maria      | Brasinha  |
| Sigismundo | Floquinho |
| Sigismundo | Bidu      |
+------------+-----------+
```

Veja, o resultado foi o mesmo, então escolher um ou outro é uma questão de estilo. A desvantagem de utilizar o produto cartesiano filtrado é que a importante condição de relação entre as tabelas pode se perder em meio a tantas outras condições (*pessoas.idade < 18*, *cachorros.raca = 'Poodle'*, etc.). Por outro lado, a sintaxe da *JOIN* pode ser um pouco mais difícil de lembrar na hora de escrever a consulta.

Mas a cláusula *JOIN* possui ainda algumas vantagens, que são formas mais expressivas de tratar os elementos sem associação (no

exemplo: pessoas sem cachorro e cachorros sem dono). São essas formas: *INNER JOIN*, *LEFT JOIN*, *RIGHT JOIN* e *FULL OUTER JOIN* (resumidas no quadro 1).

Quadro 1 – Descrição do efeito causado por cada tipo de junção

TIPO DA JUNÇÃO	EFEITO NA OPERAÇÃO "A JOIN B"
INNER JOIN	Exibe as junções entre os elementos de A e B apenas para os elementos com correspondência (elemento de A possui correspondência com elemento de B e elemento de B possui correspondência com elemento de A).
LEFT JOIN	Exibe as junções entre os elementos de A e B para os elementos com correspondência (elemento de A possui correspondência com elemento de B e elemento de B possui correspondência com elemento de A). Exibe também os elementos de A sem correspondência com os elementos de B.
RIGHT JOIN	Exibe as junções entre os elementos de A e B para os elementos com correspondência (elemento de A possui correspondência com elemento de B e elemento de B possui correspondência com elemento de A). Exibe também os elementos de B sem correspondência com os elementos de A.
FULL OUTER JOIN	Exibe as junções entre os elementos de A e B para os elementos com correspondência (elemento de A possui correspondência com elemento de B e elemento de B possui correspondência com elemento de A). Exibe também os elementos de A sem correspondência com os elementos de B e os os elementos de B sem correspondência com os elementos de A.

O *INNER JOIN* é o *JOIN* padrão que já realizamos: exibe apenas os elementos com correspondência. O *LEFT JOIN* exibe a mais os elementos da primeira tabela sem correspondência (pessoas sem cachorro). O *RIGHT JOIN* exibe a mais os elementos da segunda tabela sem correspondência (cachorros sem dono). Por fim, o *FULL OUTER JOIN* mostra todos os elementos. Vamos aos exemplos:

```
> SELECT pessoas.nome AS dono, cachorros.nome AS cachorro
FROM pessoas LEFT JOIN cachorros ON cachorros.id_dono =
pessoas.id;
```

```
+------------+-----------+
| dono       | cachorro  |
+------------+-----------+
| João       | Rex       |
| Maria      | Brasinha  |
| Sigismundo | Floquinho |
| Sigismundo | Bidu      |
| Magnólia   | NULL      |
+------------+-----------+
> SELECT pessoas.nome AS dono, cachorros.nome AS cachorro
FROM pessoas RIGHT JOIN cachorros ON cachorros.id_dono =
pessoas.id;
+------------+-----------+
| dono       | cachorro  |
+------------+-----------+
| João       | Rex       |
| Maria      | Brasinha  |
| Sigismundo | Floquinho |
| Sigismundo | Bidu      |
| NULL       | Auau      |
+------------+-----------+
```

Diferentemente de outros SGBDs, o MySQL não disponibiliza o *FULL OUTER JOIN*. Mas se necessário, podemos fazer a união do *LEFT JOIN* com o *RIGHT JOIN:*

```
> SELECT pessoas.nome AS dono, cachorros.nome AS cachorro
FROM pessoas LEFT JOIN cachorros ON cachorros.id_dono =
pessoas.id
  UNION
  SELECT pessoas.nome AS dono, cachorros.nome AS cachorro
FROM pessoas RIGHT JOIN cachorros ON cachorros.id_dono =
pessoas.id;
+------------+-----------+
| dono       | cachorro  |
+------------+-----------+
| João       | Rex       |
| Maria      | Brasinha  |
| Sigismundo | Floquinho |
```

```
| Sigismundo | Bidu      |
| Magnólia   | NULL      |
| NULL       | Auau      |
+------------+-----------+
```

4 O cálculo relacional

Nas seções anteriores estudamos sobre como a SQL se embasa na álgebra relacional. Mas outro pilar teórico da SQL é o cálculo relacional, que é outra linguagem formal de consulta para o modelo relacional (ELMASRI; NAVATHE, 2010). O poder expressivo do cálculo relacional é equivalente ao da álgebra relacional, mas o cálculo relacional possui um estilo declarativo, ou seja, expressamos apenas o que será recuperado e não como será recuperado. Na álgebra relacional é diferente, pois nela pode-se especificar uma sequência de operações algébricas para chegar a um resultado, sendo que a implementação dessa busca feita no SGBD executará a sequência correspondente de operações definidas na consulta.

Duas variantes do cálculo relacional foram desenvolvidas em paralelo: o cálculo relacional de tuplas e o cálculo relacional de domínio. Enquanto o cálculo de tuplas se tornou um dos pilares da SQL, o cálculo de domínio serviu de base para a interface gráfica do banco de dados DB2. Contudo, as duas variantes são muito similares, e, para os propósitos desta seção, essas diferenças mínimas não importam.

A forma básica de uma expressão do cálculo relacional é a seguinte:

$$\{t_1.A, t_1.B, t_2.C, t_2.D, \ldots \mid COND(t_1, t_2, \ldots)\}$$

Ela denota um conjunto resultante (*{" e "}*) formado por valores de colunas (A, B, C, D,...) de tuplas (t_1, t_2,...) que respeitem uma certa condição (COND(t_1, t_2,...). Trata-se novamente, na prática, da estrutura *SELECT <colunas> FROM <tabelas> WHERE <condição>*.

Mas o que vamos aproveitar para destacar, com base no cálculo relacional e com efeitos práticos, é a estrutura da condição. Uma condição possui minimamente um átomo, que é constituído pela comparação entre duas colunas ou entre uma coluna e um valor literal. Exemplos de átomos são: *pais.exportacoes > pais.importacoes* (comparando uma coluna com outra coluna) e *pessoa.idade < 18* (comparando uma coluna com um valor literal). A comparação em um átomo é feita com os operadores de comparação *>*, *<*, *>=*, *<=*, *= e !=*.

Uma condição pode também ser escrita com fórmulas que combinam átomos (ou outras fórmulas) por meio dos operadores lógicos *AND*, *OR* e *NOT*. Por exemplo, para obter os funcionários do sexo masculino que já podem se aposentar, pode-se aplicar a seguinte condição sobre a tabela de funcionários: *idade > 65 AND sexo = 'M'*; nesse exemplo, o *AND* está ligando dois átomos. Já para obter todos os funcionários e funcionárias que já podem se aposentar, podemos utilizar o operador lógico *OR* para unir duas fórmulas (uma fórmula para os homens e outra para as mulheres): *idade > 65 AND sexo = 'M' OR idade > 60 AND sexo = 'F'*. Note que a expressão anterior funciona porque, em praticamente qualquer linguagem de programação, o *OR* é avaliado somente depois do *AND* (enquanto o *AND* é avaliado só depois do *NOT*). Mas para não exigir demais do colega que lerá essa consulta no futuro, é gentil fazer uso de parênteses para maior clareza: *(idade > 65 AND sexo = 'M') OR (idade > 60 AND sexo = 'F')*.

Considerações finais

Este capítulo introduziu vários recursos avançados da SQL. Embora alguns sejam mais raramente aplicados na prática (*UNION*, *INTERSECT* e *EXCEPT*), você pode se diferenciar sabendo aplicá-los nos momentos em que for conveniente. Por outro lado, a utilização de junções é fundamental para o uso de bancos relacionais: é o que distingue os SGBDs relacionais dos outros tipos de SGBDs. Outros recursos avançados e

relevantes que vimos foram os *subselects* e a utilização de *views*. Por fim, aproveitamos a ocasião para refinar conceitualmente o poder de expressividade das condições da cláusula *WHERE*.

Referências

ECMA INTERNATIONAL. **The JSON data interchange syntax**: standard ECMA-404. 2. ed. 2017. Disponível em: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf. Acesso em: 28 abr. 2020.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de banco de dados**. 4. ed. São Paulo: Pearson, 2010.

W3SCHOOLS. **SQL UNION Operator**. [*S. d.*]. Disponível em: https://www.w3schools.com/sql/sql_union.asp. Acesso em: 28 abr. 2020.

Capítulo 7

Administração

A utilização de um SGBD envolve muito mais que o uso quotidiano de criação de tabelas e operações de manutenção dos dados (CRUD). É preciso garantir a segurança do banco de dados, impedindo que um usuário mal-intencionado escreva, altere ou mesmo leia dados aos quais não possui direitos. É preciso garantir o desempenho e a disponibilidade: os dados devem ser rapidamente recuperados para quem precisa deles. Garantir isso requer uma série de atividades administrativas, que geralmente são desempenhadas pelo administrador do banco de dados. Neste capítulo, estudaremos algumas dessas tarefas, assim como o papel exercido por esse administrador.

1 Administração do banco de dados

O uso comum que um desenvolvedor faz do banco de dados é a criação de tabelas e a manutenção dos dados (CRUD). Porém, há muito mais com que se preocupar em se tratando de bancos de dados:

- Para garantir a disponibilidade, é possível configurar uma réplica da base de dados.
- Para diminuir o tempo de resposta de consultas em tabelas muito grandes, é possível particionar uma tabela dessas em várias tabelas menores.
- É possível realizar uma série de otimizações de desempenho de acordo com os padrões de uso do banco de dados, o que é conhecido também como *tuning*.
- É desejável manter o banco de dados atualizado com a versão mais recente de seu SGBD, pois cada nova versão pode corrigir defeitos anteriormente existentes, melhorar o desempenho do SGBD ou mesmo fornecer novas possibilidades de uso.
- É preciso monitorar o que está acontecendo no ambiente de produção (HUMBLE; FARLEY, 2014). Para isso, é preciso coletar dados, disponibilizar painéis e configurar alertas. Assim, os responsáveis podem agir rapidamente em caso de incidentes. A monitoração do consumo de recursos também possibilita o planejamento do crescimento da infraestrutura.
- Controlar a autenticação e a autorização de usuários para garantir que não haverá acesso indevido aos dados.
- Realizar cópias de segurança periodicamente para que os dados possam ser restaurados em caso de pane no sistema.

Todas essas atividades não estão diretamente ligadas ao desenvolvimento da aplicação em si, são tarefas administrativas do banco de dados.

Mesmo que ambientes modernos de nuvem facilitem bastante, ou mesmo automatizem a execução das tarefas administrativas, é preciso ter algum responsável pelo banco de dados que garantirá que essas atividades sejam executadas. Como essas tarefas requerem um tipo de conhecimento e habilidades específicas, é comum em grandes empresas a existência de um papel dedicado a essas atividades: o do administrador de bancos de dados, também conhecido como DBA (*database administrator*).

Enquanto a criação de tabelas e a manutenção dos dados (CRUD), realizados por desenvolvedores de software, são atividades bem padronizadas entre os SGBDs relacionais, as tarefas administrativas, embora comuns a todos eles, possuem formas bem mais específicas de serem realizadas, variando de um SGBD para outro. Dessa forma, é comum que os DBAs sejam especializados no SGBD. Assim, por exemplo, é comum a existência de "DBAs Oracle", isto é, DBAs especializados no SGBD Oracle. Um DBA deve, portanto, ter profunda intimidade com seu SGBD de escolha. No caso do MySQL, é preciso que o DBA se aprofunde na leitura do manual do MySQL (MYSQL, 2020), que referenciaremos ao longo deste capítulo.

Segundo Ramakrishnan e Gehrke (2011), as principais responsabilidades de um DBA são:

- projeto dos esquemas conceituais e computacionais do banco de dados;
- segurança e autorização;
- garantia da disponibilidade e recuperação de dados (*backup*); e
- otimização do banco de dados para assegurar o desempenho (*tuning*).

Dessas atribuições, a mais controversa é a definição do esquema do banco de dados. Em algumas organizações isso é feito pelos

desenvolvedores. Em outras, pelo DBA. Já em algumas, é feito pelos desenvolvedores com a aprovação do DBA, o que acaba potencializando demoras na entrega de novas funcionalidades do sistema. Para que essas demoras não ocorram, independentemente de quem seja responsável pela evolução do esquema, Ambler e Sadalage (2006) advogam que a gerência do banco de dados seja feita com técnicas evolucionárias ágeis – isto é, que admitem que o banco de dados deve ser evoluído aos poucos – como: refatoração, modelagem evolucionária, testes de regressão, gerenciamento de configuração dos artefatos do banco de dados e instâncias dedicadas para desenvolvedores.

Humble e Farley (2014) dão um passo adiante: defendem que a evolução do esquema de banco de dados deve ser gerenciada por *scripts* automatizados como parte do processo de integração contínua do sistema. Para entender a importância de todo esse controle, é preciso lembrar que, além do banco de dados de produção, existem outras instâncias correspondentes a serem controladas, por exemplo, o banco existente na estação de trabalho do desenvolvedor e o banco utilizado no ambiente de validação. A evolução do esquema desses bancos deve ser coordenada: após certa validação, uma alteração já aplicada no ambiente de validação deve ser aplicada no banco de produção. Ou seja, é preciso garantir que a mesma mudança seja aplicada em todos os ambientes, mas ao mesmo tempo é importante controlar para que cada mudança seja aplicada em seu devido momento. Um exemplo de ferramenta para a execução desse controle é a Flyway. Além disso, é importante garantir a harmonização da versão do SGBD em todos os ambientes: uma certa funcionalidade que tenha sido testada com a versão 8 do MySQL pode não funcionar adequadamente se o MySQL de produção estiver na versão 5, por exemplo.

2 Segurança

São propriedades básicas da segurança de sistemas a confidencialidade, a autenticação, a integridade, a disponibilidade e o controle de

acesso (KUROSE; ROSS, 2006). Sem essas garantias básicas de segurança o sistema fica comprometido: o sistema pode ficar inacessível ou os dados podem ser adulterados ou roubados. Qualquer uma dessas situações pode ocasionar danos irreparáveis às finanças e à imagem da empresa dona do sistema.

Quadro 1 – Propriedades básicas da segurança de sistemas

CONFIDENCIALIDADE	Garante que só pessoas autorizadas terão acesso aos dados.
AUTENTICAÇÃO	Garante a identidade de quem está utilizando o sistema.
INTEGRIDADE	Garante que os dados não sejam alterados de maneira indevida, seja de forma acidental, seja de forma maliciosa.
DISPONIBILIDADE	Garante que agentes maliciosos não consigam derrubar o sistema, o que impediria o acesso de seus usuários legítimos.
CONTROLE DE ACESSO	Também chamado de autorização, garante que somente os usuários que tiverem os direitos de acesso apropriados possam realizar determinadas operações no sistema.

Fonte: adaptado de Kurose e Ross (2006).

No contexto de banco de dados, essas propriedades de segurança são implementadas principalmente pelos processos de autenticação e autorização, que garantem que apenas os devidos usuários possam escrever e ler o conteúdo do banco de dados. Com esses mecanismos, é possível definir o mínimo de privilégios necessários para que cada ator do sistema realize as tarefas que deve executar, aplicando assim o que chamamos de princípio do menor privilégio (FOX; PATTERSON, 2015). A disponibilidade pode se valer também de um processo robusto de armazenamento e restauração de cópias de segurança. Esses assuntos serão os temas das próximas seções. Outro mecanismo que pode auxiliar a integridade dos dados é a auditoria das alterações realizadas ou mesmo a criptografia dos dados armazenados.

No entanto, a segurança não pode depender somente da configuração do SGBD. A segurança do sistema como um todo é como uma parede de tijolos: basta um buraco em uma grande muralha para que o invasor entre. Outra metáfora popular é a da corrente: assim como para partir uma corrente basta partir seu elo mais fraco, basta que um invasor explore o ponto mais vulnerável do sistema para conseguir algum acesso. Por isso, é preciso defesa em profundidade: assim como o SGBD autentica e autoriza seus usuários, a aplicação também deve autenticar, autorizar e mesmo auditar seus operadores. As configurações da aplicação e do SGBD devem evitar descuidos, por exemplo, a escrita de senhas em arquivos de log, pois um invasor que consiga acesso ao sistema operacional dos servidores, mesmo sem acesso ao SGBD ou à aplicação, terá acesso a esses logs.

Um tipo comum de ataque que procura obter acesso indevido ao banco de dados por meio de uma vulnerabilidade na aplicação é o *SQL Injection*: o invasor envia em um formulário do sistema alguns dados que ele espera que sejam intercalados diretamente em um comando de consulta SQL executado pelo sistema (FOX; PATTERSON, 2015). Como exemplo, considere um aplicativo para aluguel de carros. Em uma das telas, esse aplicativo fornece um formulário para que o usuário visualize e selecione a marca/modelo do carro desejado. Se o nome da marca/modelo for capturado no formulário, a execução da busca pode ser realizada com a seguinte consulta SQL:

```
SELECT * FROM veiculo WHERE marca-modelo = ':marca-modelo'
```

Aqui, *:marca-modelo* indica o parâmetro passado na busca. Veja que se a aplicação não tomar os devidos cuidados de sanitizar suas entradas, é possível que o agente malicioso forneça como entrada a *string x'; DROP TABLE veiculo;*, o que poderia remover todos os dados da principal

tabela da aplicação caso a aplicação venha a executar ambos os comandos resultantes dessa interpolação:

```
SELECT * FROM veiculo WHERE marca-modelo = 'x'; DROP TABLE
veiculo;
```

Isso mostra que a segurança deve ser papel de todos os envolvidos no processo de produção de software.

Para saber mais sobre segurança no contexto específico do MySQL, consulte o capítulo 6 "Security" do manual (MYSQL, 2020).

3 Logins

O MySQL possibilita a criação de contas para que clientes possam se conectar ao servidor e acessar os dados gerenciados por ele. A principal função do sistema de privilégios do MySQL é autenticar um usuário que se conecta a partir de certo *host*[1] e associá-lo a privilégios sobre a base de dados, tais como *SELECT*, *INSERT*, *UPDATE* e *DELETE* (MYSQL, 2020).

Uma instalação típica do MySQL se inicia com um usuário administrativo denominado *root*. Utilizando o usuário *root*, o administrador do banco de dados pode criar as bases de dados e os usuários que acessarão a base, bem como definir os privilégios de cada usuário sobre cada base existente.

No MySQL, a sintaxe do nome completo do usuário é:

1 Um *host* (ou "hospedeiro", em português) é qualquer equipamento (físico ou virtual) que esteja ligado a uma rede de computadores (KUROSE; ROSS, 2006). No contexto desta obra, geralmente trata-se dos servidores onde estão hospedados os bancos de dados e as aplicações.

```
'nome_usuario'@'nome_host'
```

Nesse comando, o *nome_usuario* designa uma *string* qualquer, enquanto *nome_host* designa um IP, uma faixa de IPs ou um nome de *host* (que seja convertível para um IP pelo sistema de DNS[2]). Dessa forma, a identidade de um usuário é determinada pelo nome de usuário e do *host* a partir do qual o usuário se conecta ao banco. Um exemplo de criação de usuário:

```
CREATE USER 'aplicacao_user'@'198.51.100.0/255.255.255.0';
```

Este comando utiliza a notação de máscara de rede para criar um usuário *aplicacao_user* que pode acessar o sistema a partir de uma certa faixa de IPs: 198.51.100.1 – 198.51.100.255. Mas atenção: quando um usuário vai se conectar a um banco de dados, esse usuário informa o nome do usuário (a parte *nome_usuario*). O *nome_host* não é de escolha do usuário, é definido pela máquina e pela rede a partir de onde o usuário está se conectando ao banco. Caso se queira permitir o acesso somente a partir do próprio servidor do banco de dados, podemos utilizar o valor *localhost* para o *nome_host*. Podemos também utilizar o coringa % no *nome_host* para designar que o usuário pode se conectar a partir de qualquer *host*.

PARA SABER MAIS

Cada endereço IP tem um comprimento de 4 bytes (32 bits). Esses endereços são escritos em uma notação tal que cada byte do endereço é

2 Domain Name System (DNS) é um protocolo de redes que traduz nomes fáceis de entender (por exemplo: sp.senac.br) em um endereço de rede IP (por exemplo: 187.51.127.23) (KUROSE; ROSS, 2006).

escrito em sua forma decimal e separado dos outros bytes do endereço por um ponto. Por exemplo, no IP 193.32.216.9, o 193 é o número decimal equivalente ao primeiro byte do endereço. *Hosts* em uma sub-rede são interconectados sem o intermédio de um roteador IP (podem ser interligados por *hubs* Ethernet). O endereçamento IP designa um endereço a essa sub-rede, como 223.1.1.0/24, no qual a notação /24, a máscara de rede, indica que os 24 bits (3 bytes) mais à esquerda do conjunto de 32 bits definem o endereço da sub-rede, enquanto o byte restante designa o *host* dentro dessa sub-rede (KUROSE; ROSS, 2006).

Pelo conteúdo apresentado até agora, já é perceptível que, como um profissional de infraestrutura, o DBA deve possuir um conhecimento razoável sobre redes de computadores, ponto no qual tipicamente os desenvolvedores de software possuem menos conhecimento.

Tendo sido criado o usuário, o cliente agora pode se conectar ao banco. Existem várias formas de fazer isso. Aplicações geralmente utilizam bibliotecas da linguagem de programação correspondente para se conectar (exemplo: JDBC para o Java). Para um humano, é possível utilizar o seguinte comando:

```
mysql -u aplicacao_user -p aplicacao_db_name
```

No comando, o que vem após o *-u* indica o nome do usuário, o *-p* indica que a senha será fornecida em seguida via teclado e o último termo se refere ao nome do banco de dados a ser gerenciado.

IMPORTANTE

Muita atenção para não confundir os usuários do banco de dados com os usuários da aplicação que acessa esse banco. Tomemos como exemplo uma aplicação de caixa de supermercado. Essa aplicação possui um banco de dados e operadores que trabalham no caixa utilizando

a aplicação. Geralmente uma aplicação tem muitos usuários, no exemplo, todos os trabalhadores que operam os caixas de supermercados. Já o banco de dados tem como seu principal usuário a própria aplicação: é a aplicação do supermercado que vai se conectar ao banco de dados, e não os trabalhadores do caixa. Fora esse usuário principal, é de esperar que um banco de dados possua muito poucos outros usuários, como o usuário do DBA e talvez um usuário diferente para algum processo de carga.

O DBA deve também definir uma senha para cada usuário, o que pode ser feito com a cláusula *IDENTIFIED BY* no momento da criação do usuário ou logo após, como segue nos exemplos:

```
CREATE USER 'app_user'@'localhost' IDENTIFIED BY 'senha';
ALTER USER 'app_user'@'localhost' IDENTIFIED BY 'nova_
senha';
```

O DBA pode também definir os seguintes limites para cada usuário: *MAX_QUERIES_PER_HOUR*, *MAX_UPDATES_PER_HOUR*, *MAX_CONNECTIONS_PER_HOUR* e *MAX_USER_CONNECTIONS*. Esses limites ajudam a garantir a saúde e a disponibilidade da base de dados.

O parâmetro *MAX_USER_CONNECTIONS*, por exemplo, define a quantidade máxima de conexões simultâneas permitidas para um usuário e é muito importante em ambientes de nuvem, nos quais uma aplicação escala horizontalmente criando réplicas de si mesma. Nesse caso, cada instância da aplicação vai multiplicar a quantidade de conexões com o banco de dados por um fator. A limitação imposta por esse parâmetro impede que um mau dimensionamento na quantidade de réplicas da aplicação (ou na quantidade de conexões por réplica) indisponibilize o banco para a aplicação. O seguinte comando provê um exemplo de criação de usuário com essas limitações de acesso:

```
CREATE USER 'myapp'@'%' IDENTIFIED BY 'senha'
 WITH MAX_QUERIES_PER_HOUR 200
      MAX_UPDATES_PER_HOUR 100
      MAX_CONNECTIONS_PER_HOUR 50
      MAX_USER_CONNECTIONS 30;
```

4 Permissões

O sistema de privilégios do MySQL garante que todos os usuários possam executar apenas as operações a eles permitidas. Para cada requisição disparada por uma conexão, o servidor determina a identidade do cliente (nome e *host*) e confere se essa identidade está autorizada a executar a operação requisitada (MYSQL, 2020).

A forma geral do comando utilizado pelo administrador para conceder privilégios é:

```
GRANT <permissões>
  ON <banco>.<tabela>
  TO <usuário>;
```

Dessa forma, para conceder acesso total para a aplicação fazer o que quiser com o banco, temos:

```
GRANT ALL
  ON aplicacao_db.*
  TO 'app_user'@'%';
```

Para criar um superusuário com plenos poderes sobre qualquer banco:

```
GRANT ALL
  ON *.*
  TO 'admin'@'%';
```

Mas podemos também ser mais seletivos e conceder apenas alguns privilégios em uma determinada tabela:

```
GRANT SELECT, INSERT
  ON aplicacao_db.modelos_automoveis
  TO 'carga_user'@'%';
```

Este último exemplo fornece permissão para que um processo de carga periódica escreva apenas em determinada tabela. Assim, evita-se o risco de que, por acidente, esse processo altere indevidamente algum outro dado do sistema.

Para conferir os privilégios de determinado usuário:

```
SHOW GRANTS FOR <usuário>;
```

Já para remover privilégios, pode-se utilizar o comando *REVOKE*. Exemplo:

```
REVOKE CREATE,DROP
  ON expenses.*
  FROM 'custom'@'server10';
```

Esse comando revoga os privilégios de criação e remoção de tabelas para o usuário *custom* conectado a partir do *host* server10.

No MySQL, os privilégios podem também ser administrados por meio de *roles* (papéis). Um *role* é uma coleção de privilégios que pode ser atribuída a usuários (MYSQL, 2020). Segue um exemplo de criação de *role*:

```
CREATE ROLE 'app_developer';
GRANT ALL ON app_db.* TO 'app_developer';
```

Na sequência, deve-se atribuir o *role* para algum usuário, como segue no exemplo:

```
GRANT 'app_developer' TO 'dev1'@'localhost';
SET DEFAULT ROLE ALL TO 'dev1'@'localhost';
```

A primeira linha atribui o *role* para o usuário. Já a segunda faz com que os *roles* do usuário fiquem ativos logo que ele inicie uma sessão com o banco.

O comando *SHOW GRANTS*, além dos privilégios diretamente atribuídos ao usuário, vai listar também os *roles* do usuário.

PARA SABER MAIS

Uma instância de MySQL é capaz de gerenciar vários bancos de dados. Contudo, com a arquitetura dominante de microsserviços (NEWMAN, 2015), o comum é que cada serviço tenha sua base de dados e que cada base de dados esteja em uma instância diferente do MySQL, isto é, cada banco em um servidor diferente. Uma grande vantagem de deixar bases de dados diferentes em servidores diferentes (assim como de dividir uma aplicação em microsserviços) é a degradação suave (BREWER, 2001): qualquer problema em um dos servidores afetará apenas uma base de dados.

Além disso, é esperado que cada base de dados não tenha muitos usuários. No mínimo terá um usuário administrativo e o usuário da aplicação. No contexto de microsserviços, há um caso especial para um usuário a mais: as bases de dados para relatórios. Essas bases normalmente são construídas a partir da composição de várias outras. Então há um processo especial que produz a base de relatórios e a aplicação que consome a base para disponibilizar os dados consolidados ao usuário. Nesse contexto, faz sentido que o processo de carga tenha um usuário com permissão de escrita, ao passo que a aplicação de relatórios tenha um usuário com permissão somente de leitura.

Para saber mais sobre a criação de usuários e o gerenciamento de permissões no MySQL, consulte o capítulo 6.2 "Access Control and Account Management" do manual (MYSQL, 2020).

5 *Backups*

Backups são cópias de segurança de uma base de dados e são importantes para que o DBA possa restaurar os dados em caso de problemas, como panes no sistema, falhas no hardware ou usuários que tenham deletado os dados por acidente (MYSQL, 2020).

O MySQL oferece uma variedade de estratégias para *backup,* vamos conhecer as principais opções.

Um *backup* pode ser físico ou lógico. *Backups* físicos copiam os arquivos e diretórios do SGBD tal como se encontram no sistema operacional, possibilitando uma rápida restauração dos dados. Já o *backup* lógico gera arquivos de texto com comandos SQL contendo as instruções para a criação das tabelas e a inserções dos dados, sendo uma opção mais flexível para a inserção dos dados em outras instâncias do MySQL e até mesmo em outros SGBDs.

Um *backup* pode ser *on-line* ou *off-line*. Um *backup on-line* é executado enquanto o banco está sendo usado por seus usuários. Assim, essa

opção não indisponibiliza o serviço, contudo requer cuidado para que se garanta a integridade dos dados copiados.[3] Já o *backup off-line* é executado com o banco de dados parado, o que significa que, por um lado, a aplicação se torna indisponível aos usuários, mas, por outro, não apresenta riscos quanto à integridade da cópia de segurança.

Um *backup* pode ser local ou remoto. Um *backup* local é executado no mesmo servidor onde o banco de dados está, enquanto o *backup* remoto é feito em um servidor diferente.

Um *backup* pode ser completo ou incremental. Cada vez que um *backup* completo é executado, toda a base de dados é copiada. Já o *backup* incremental é capaz de copiar apenas as alterações realizadas a partir de certo instante no tempo.

Uma estratégia completa de *backup* precisa ainda definir o agendamento (periodicidade) do *backup* e decidir se a cópia deve ser compactada (para que ocupe menos espaço em disco) e criptografada (para impedir acesso indevido). Os processos de agendamento, compactação e criptografia de *backups* não são fornecidos pelo MySQL – o DBA deve utilizar recursos do sistema operacional ou ferramentas complementares.

IMPORTANTE

Além de planejar e aplicar uma estratégia de *backup*, é muito importante treinar o processo de restauração de *backup*. Se um DBA não realizar esse exercício, será incapaz de garantir que, em caso de um raro evento de pane, ele conseguirá restaurar os dados. Será mais incapaz ainda de saber em quanto tempo ele conseguirá restaurar os dados.

3 Imagine o seguinte cenário: o processo de *backup* copiou primeiro a tabela de fornecedores. Posteriormente vai copiar a tabela de produtos – só que a essa altura já podem ter surgido produtos de fornecedores recém-cadastrados que não foram copiados. Esse cenário pode apresentar um *backup* inconsistente, contendo produtos sem os devidos fornecedores na cópia de segurança.

Uma forma simples de produzir um *backup* é utilizando o comando *mysqldump*, um utilitário que produz *backups* lógicos na forma de um conjunto de comandos SQL que podem ser executados para reproduzir a base de dados original. O arquivo de *backup* gerado é também chamado de *dump*, podendo também ser gerado em formato CSV ou XML. A forma mais simples de utilização do *mysqldump* é a seguinte:

```
mysqldump [opções de conexão] <nome do banco de dados>
```

Nesse comando, as opções de conexão são as mesmas utilizadas no comando *mysql* para se conectar ao banco (*-u* e *-p*).

Para saber mais sobre *backups* no MySQL, consulte o capítulo 7 "Backup and Recovery" do manual (MYSQL, 2020).

Considerações finais

Ao longo desta obra, abordamos o uso básico do SGBD que todo desenvolvedor deve dominar. Já neste capítulo, abordamos alguns temas mais especializados, cujo entendimento em profundidade cabe ao DBA, isto é, o administrador do banco de dados.

Além de um conhecimento profundo sobre seu SGBD de escolha, o DBA deve possuir conhecimentos razoáveis sobre redes e sistemas operacionais. Esse tipo de conhecimento, em geral, não é amplamente dominado por desenvolvedores de software. Dessa forma, este capítulo pode lhe servir também de reflexão sobre que tipo de profissional você prefere ser no contexto da produção de software: um desenvolvedor ou um especialista em infraestrutura (como o DBA).

Procure conhecer as oportunidades de mercado para cada especialidade e, se possível, até mesmo frequentar eventos voltados a esses

diferentes perfis, de forma a lhe apoiar em suas decisões de carreira. Por fim, caso se torne um DBA, não deixe de estudar em profundidade a documentação oficial de seu SGBD de escolha.

Referências

AMBLER, Scott W.; SADALAGE, Pramod J. **Refactoring databases**: evolutionary database design. [*S. l.*]: Addison-Wesley Professional, 2006.

BREWER, Eric A. Lessons from giant-scale services. **IEEE Computer**, v. 5, n. 4, p. 46-55, 2001.

FOX, Armando; PATTERSON, David. Segurança: protegendo os dados do cliente no seu aplicativo. *In*: FOX, Armando; PATTERSON, David. **Construindo software como serviço**: uma abordagem ágil usando computação em nuvem. Tradução: Daniel Cordeiro, Fabio Kon. [*S. l.*]: Strawberry Canyon, 2015.

HUMBLE, Jez; FARLEY, David. **Entrega contínua**: como entregar software de forma rápida e confiável. São Paulo: Bookman, 2014.

KUROSE, James F.; ROSS, Keith W. **Redes de computadores e a internet**: uma abordagem top-down. 3. ed. São Paulo: Pearson, 2006.

MYSQL. **MySQL 8.0 reference manual**. 2020. Disponível em: https://dev.mysql.com/doc/refman/8.0/en/. Acesso em: abr. 2020.

NEWMAN, Sam. **Building microservices**: designing fine-grained systems. [*S. l.*]: O'Reilly Media, 2015.

RAMAKRISHNAN, Raghu; GEHRKE, Johannes. **Sistemas de gerenciamento de banco de dados**. 3. ed. Porto Alegre: AMGH, 2011.

Capítulo 8

Tipos de bancos de dados e modelo conceitual

Apesar de o mercado ofertar diversas opções de SGBDs, ao longo deste livro focamos a utilização do MySQL. Mas por que há tantos outros SGBDs diferentes disponíveis? Será que é preciso aprender os outros bancos também?

Ao escrever um código que armazene e recupere informações em arquivos, há diversas decisões a serem tomadas pelos projetistas. É natural que diferentes projetistas tenham diferentes ideias e opiniões sobre como projetar detalhes de um SGBD. Além disso, como ficará mais evidente ao longo deste capítulo, na seção sobre o NoSQL, alguns bancos de dados são projetados para serem utilizados em cenários específicos.

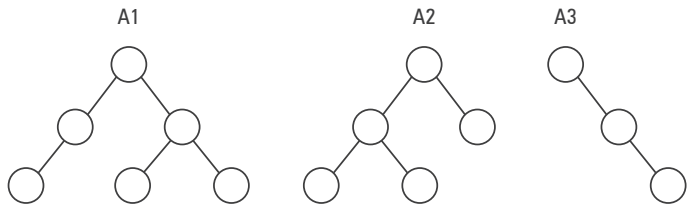
Muitos desses outros SGBDs (PostgreSQL, Oracle, SQL Server, SQLite) são bem parecidos com o MySQL, pois aderem ao modelo relacional dos dados. Após o nosso estudo do MySQL, você não deverá ter problemas para se adaptar a esses outros SGBDs, caso necessário. Mas há também outros tipos de SGBDs que se diferenciam principalmente por oferecer outras opções de modelo conceitual (lógico) ao seu usuário.

Neste capítulo, veremos como os SGBDs podem ser classificados quanto à visão conceitual fornecida ao usuário.

1 Hierárquico

Os bancos de dados hierárquicos fornecem uma visão lógica dos dados em formato de árvores (DATE, 2004). Árvores são estruturas de dados compostas de nós e arestas, ligando os nós de forma que a árvore começa com um nó raiz. Cada nó pode ter vários filhos, mas cada nó só pode ter um pai, e pode ser que um nó não tenha filho – a esse tipo de nó chamamos de folha.

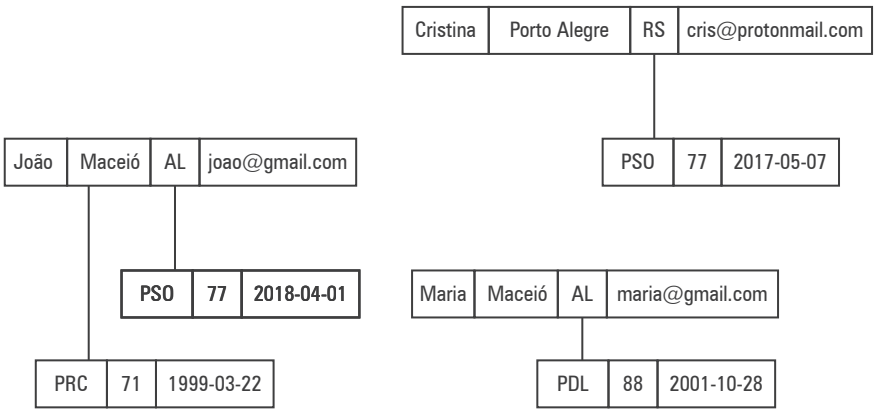
Figura 1 – Exemplos de três árvores (A1, A2, A3)



Vamos a um exemplo: considere que estamos desenvolvendo uma aplicação para a Justiça Eleitoral cadastrar os candidatos às eleições municipais. As primeiras informações importantes a serem armazenadas são os dados do candidato em si. Nesse caso, consideremos nome, endereço (a Justiça precisa dessa informação, pois um candidato não pode se candidatar para vereador fora do município de sua residência) e contato (para a Justiça enviar comunicados). Outra informação importante é também a qual partido o candidato está filiado. O partido também tem suas próprias informações, minimamente a sigla e o número. Um banco de dados hierárquico amarra essa relação candidato–partido por meio de uma interligação entre os registros de candidatos e de partidos. Podemos ter também dados que fazem parte da relação candidato–partido. Em nosso exemplo, é importante para a Justiça saber quando o candidato se

filiou ao partido (não se pode trocar de partido às vésperas das eleições). Então o nosso banco ficaria conforme a figura 2.

Figura 2 – Exemplo de modelo hierárquico contendo entidades superiores, entidades dependentes e dados de relacionamento entre superiores e dependentes



No linguajar de bancos hierárquicos, dizemos que os candidatos são entidades superiores e que os partidos são entidades dependentes. Note pela figura 2 que *(i)* um superior pode ter vários dependentes; em nosso exemplo isso acontece porque um candidato já foi filiado a outro partido no passado; *(ii)* a árvore de cada candidato (a entidade central do nosso problema) deve ser independente (desconexa) das outras árvores; *(iii)* em um banco de dados hierárquico, as propriedades da interligação ficam armazenadas no registro dependente.

O modelo hierárquico traz alguns problemas. O mais evidente no nosso exemplo é a repetição de dados das ocorrências das entidades dependentes – nesse caso, os partidos. Isso quer dizer que para atualizar os dados de um partido (se ele mudar de nome), teremos que fazer isso em vários registros. E o pior, se esquecermos de algum, teremos dados inconsistentes. Há também algumas "anomalias" a mais: não dá para cadastrar um novo partido sem candidatos, então se excluirmos o único candidato de um partido, perdemos todos os dados daquele partido.

PARA SABER MAIS

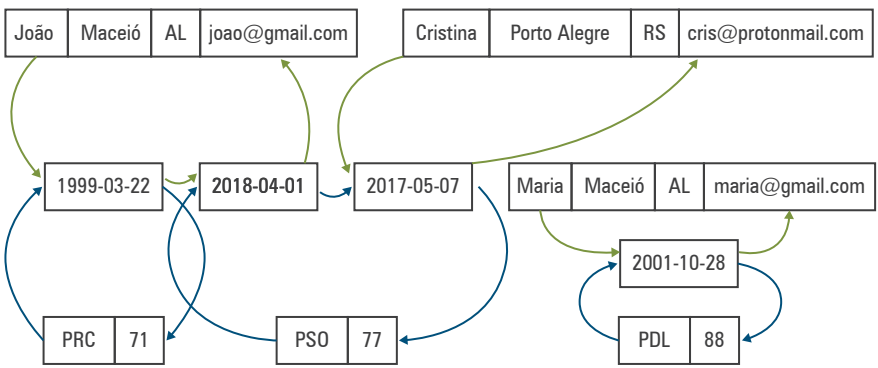
Alguns exemplos de bancos de dados hierárquicos são:

- Information Management System (IMS) da IBM
- Mark IV
- System 2000
- Time-Shared Data Management (TDMS)

2 Rede

O modelo de um banco de dados em rede é muito similar ao hierárquico, mas não há a restrição de que um registro deve possuir apenas um registro superior (DATE, 2004). Isso evita a grande repetição de dados que ocorria no modelo hierárquico e outras anomalias. As interligações agora têm seus próprios registros. Nosso banco de candidatos teria agora a seguinte modelagem:

Figura 3 – Exemplo de modelo em rede



Note que a modelagem das interligações não é trivial de entender. Funciona assim: cada interligação faz parte de dois ciclos; um ciclo

para a entidade superior (candidato) e outro para a entidade dependente (partido). Então para saber qual a data de filiação de um candidato CX a um partido PY, é preciso percorrer o ciclo do candidato CX até encontrar a interligação que está também no ciclo do partido PX. Embora alguns desses detalhes possam ser escondidos por comandos mais convenientes fornecidos pelo SGBD, considera-se que a modelagem em rede resulta em modelos de excessiva complexidade.

Pelas desvantagens aqui apresentadas e outros motivos, os modelos hierárquicos e em rede praticamente caíram em desuso, pelo menos para o desenvolvimento de novos sistemas.

PARA SABER MAIS

Alguns exemplos de bancos de dados em rede são:

- DMS 1100
- IDMS
- Total
- Dbomp
- Integrated Data Store (IDS)

3 Relacional

A modelagem relacional, conforme já estudado ao longo deste livro, fornece uma visão tabelada dos dados, sendo que o relacionamento entre tabelas distintas (feito por chaves estrangeiras) é uma forma de evitar a repetição de dados e garantir a consistência. Por consistência, neste contexto, queremos garantir que o nome da cidade "Guaratinguetá" não será escrito como "Guaratingueta" para alguns candidatos e "Guaratingüetá" para outros, por exemplo.

A modelagem relacional é a de maior sucesso entre todos os tipos de modelos conceituais. Os SGBDs relacionais são também uma das tecnologias mais perenes no mundo da computação. O sucesso foi tanto que, nos anos 1980 e 1990, as arquiteturas eram totalmente centradas no banco, chegando a incluir regras de negócio no banco de dados, o que já não é mais visto como uma boa prática (MARTIN, 2012). Um grande trunfo dos bancos relacionais é o oferecimento da linguagem SQL, por ser extremamente poderosa, flexível e simples de usar.

PARA SABER MAIS

Exemplos de SGBDs relacionais são:

- MySQL
- PostgreSQL
- Oracle
- SQLite
- SQL Server

Embora não seja uma propriedade intrínseca ao modelo lógico relacional, os SGBDs relacionais também são reconhecidos por fornecerem transações ACID (RAMAKRISHNAN; GEHRKE, 2011). Vários clientes podem utilizar o banco ao mesmo tempo, e cada cliente pode abrir várias sequências de comandos em paralelo. Cada sequência de comando que um cliente mantém com o banco é chamada de transação; ACID diz respeito às propriedades que o SGBD garante a cada transação:

- Atomicidade: se uma transação realiza várias alterações no banco, em caso de alguma pane no sistema, o próprio SGBD assegura que todas as alterações serão feitas ou nenhuma alteração será feita. Não há transações que "ficam pelo meio do caminho".

- Consistência:[1] uma transação iniciada em um estado consistente deve terminar em um estado consistente. Aqui, consistência significa o respeito às restrições impostas no esquema, como unicidades e relações entre chaves estrangeiras e chaves primárias.
- Isolamento: uma transação não será influenciada por alterações realizadas por outras transações concorrentes ainda em andamento. Ou seja, as transações são isoladas umas das outras.
- Durabilidade: após o cliente concluir uma transação, ele tem a garantia de que aqueles dados salvos no banco serão persistidos e não sumirão mais do disco (a menos que haja alguma falha de hardware ou ação maliciosa).

IMPORTANTE

No MySQL, uma transação é delimitada pelos comandos *BEGIN* e *COMMIT*. Em uma sessão S, após o comando *BEGIN*, é como se o banco de dados tivesse sido congelado para aquela sessão S. Nada do que for alterado em outras transações (mesmo após o *COMMIT* dessas outras transações) será perceptível à sessão S enquanto ela não executar o *COMMIT*. Da mesma forma, quando uma sessão S inicia uma transação com o *BEGIN*, as alterações já realizadas por outras transações em andamento (que ainda não executaram o *COMMIT*) também não são visíveis à sessão S.

4 Objeto-relacional

A orientação a objetos (OO) é um estilo de programação no qual a execução de um programa é constituída de objetos que possuem

1 Note que o uso da palavra "consistente" não é nada consistente. Só neste capítulo temos pelo menos três significados diferentes para essa palavra. Então fique atento.

estruturas de dados e que encapsulam (escondem) essas estruturas por meio de operações publicamente definidas. Assim, para acessar as operações uns dos outros, os objetos trocam mensagens entre si. É comum também que os objetos tenham relações entre si (ex.: um aluno cursando várias disciplinas).

Com o ganho de popularidade da OO nos anos 1990, surgiu a ideia de criar SGBDs orientados a objetos. Isso porque há uma certa complexidade em mapear objetos para tabelas relacionais. O propósito era então facilitar esse mapeamento. Ou seja, a ideia central do SGBD de objetos é apoiar a aplicação a persistir e recuperar seus objetos (ELMASRI; NAVATHE, 2018). Outra característica é introduzir no banco de dados conceitos típicos das linguagens orientadas a objetos, como a utilização de operações. Por exemplo, poderíamos fazer uma busca de alunos ordenando esses alunos pela média global deles, sendo a média global uma operação definida para a entidade *aluno* e que realiza um cálculo baseado nas notas do aluno em suas disciplinas já cursadas.

PARA SABER MAIS

Exemplos comerciais de SGBDs orientados a objetos são:

- GemStone/Pal
- Ontos
- Objectivity
- Versant
- ObjectStore
- Ardent
- Poet

Contudo, tamanha era (e é) a popularidade dos bancos relacionais, que os SGBDs orientados a objetos não alcançaram sucesso significativo. Com isso, os SGBDs relacionais passaram a dar suporte a novas

funcionalidades que fornecessem as vantagens dos SGBDs de objetos. Para isso, a linguagem SQL foi estendida com novas capacidades, como a utilização de operações definidas para a entidade. Essa abordagem é conhecida como "objeto-relacional".

No fim das contas, a prática mais comum de mercado hoje em dia é realizar o mapeamento entre os objetos da aplicação e as tabelas do SGBD relacional. Esse mapeamento exige o uso de *frameworks* de mapeamento, os chamados ORMs (Object-relational mapping). Um exemplo bem popular de *framework* ORM é o Hibernate para a linguagem Java. Junto com o Hibernate, o programador pode também utilizar a JPQL (Java Persistence Query Language), uma linguagem de consulta de dados muito similar à SQL, mas que possibilita ao desenvolvedor escrever suas consultas em termos dos objetos da aplicação, e não das tabelas subjacentes.

O uso de um ORM facilita a persistência dos objetos em banco, evitando que o desenvolvedor escreva muito código trivial (*boilerplate code*) em SQL, porém ao preço de ter que aprender a lidar com uma nova camada de abstração nada trivial. Além disso, para todas as operações, o ORM realiza gerações automáticas de código SQL. Esse código SQL gerado não é necessariamente o mais otimizado possível. Então, em certos cenários, para a obtenção de um melhor desempenho nas consultas pode ser necessário abrir mão do ORM.

5 Dimensional

Há um perfil de usuários (executivos, gerentes, analistas) que precisa acessar os dados para embasar tomadas de decisões. Essas consultas são arbitrárias e devem ser respondidas assim que preciso. Como o desenvolvimento de novos relatórios na aplicação pode ser muito moroso, é interessante oferecer acesso direto aos dados para esses usuários. Mas o acesso direto a bancos relacionais apresenta algumas dificuldades: 1) o espalhamento dos dados por várias bases (às vezes em

diferentes SGBDs) dificulta o cruzamento de dados; e 2) dados normalizados espalhados por várias tabelas dificultam a criação de consultas SQL, pois exigem muitos *joins*.

Para esse cenário, criou-se a ideia de *data warehouse* (depósito de dados), um conjunto de dados que agrega em um só lugar dados vindos de fontes heterogêneas (ELMASRI; NAVATHE, 2018). O *data warehouse* é focado na leitura massiva para apoio a decisões. É comum também o armazenamento de séries históricas para a detecção de tendências. Por esse foco histórico, geralmente um *data warehouse* não está atualizado com os dados mais recentes. A "carga" no *data warehouse* pode ocorrer periodicamente (diariamente, por exemplo).

Tipicamente, o *data warehouse* armazena os dados utilizando a modelagem dimensional. Isso quer dizer que uma tabela pode ter uma terceira dimensão. Assim sendo, se temos uma tabela de vendas de produtos (linhas) por regiões (colunas), podemos acrescentar o trimestre da venda como a terceira dimensão. As ferramentas de *data warehouse* oferecem uma operação chamada pivoteamento, pela qual é possível alterar quem é a terceira dimensão (por exemplo: podemos analisar vendas nas regiões por tempo e ter os produtos como terceira dimensão). Isso dá grande poder ao analista para alterar a perspectiva de análise.

Na modelagem dimensional temos dois tipos de tabela: dimensões e fatos. Uma tabela de fatos armazena os dados observados (exemplo: quantidade de vendas), com ponteiros para tabelas de dimensões, que, por sua vez, indicam a grandeza dos dados registrados (exemplo: a quantidade de vendas registradas se refere a determinada variante de produto, a uma microrregião e a um mês). Essa organização, chamada de esquema estrela, possibilita o uso de outras operações além do pivoteamento. O analista pode também executar as operações de *roll-up* para facilmente agregar os dados (exemplo: vendas por tipos de produto, macrorregiões e ano) e *drill-down* para facilmente detalhar os dados (exemplo: vendas por variantes do produto, microrregiões e mês).

PARA SABER MAIS

Um exemplo de ferramenta que possibilita a análise de dados dimensionais é o Pentaho.

6 NoSQL

Durante muito tempo (anos 1980, 1990 e 2000), os SGBDs relacionais têm sido a solução de armazenamento de dados padrão. Mas na década de 2010, a adoção de diferentes estratégias para diferentes cenários e necessidades tomou força. Essas diferentes soluções foram agrupadas sob o termo NoSQL. Nesta seção, veremos alguns casos de uso para bancos de dados NoSQL.

6.1 *Schema* flexível

Imagine que você está desenvolvendo um sistema para lojas de quinquilharias. A principal entidade a ser armazenada é o produto, mas há muitos tipos e cada produto tem características diversas. Por exemplo, mesas e violões podem ter o atributo *cor*, enquanto livros podem ter o atributo *autor*, assim como bicicletas podem ter o atributo *quantidade de marchas*. Nesse cenário, é inviável tentar prever todos os tipos de produtos possíveis e seus respectivos campos. Tentar criar uma única tabela *produto* com os mais diversos atributos pode levar a uma situação em que, para cada registro, a maioria das colunas vai ficar vazia. Fazer uma tabela auxiliar de características de produtos (com colunas *id_produto*, *nome_caracteristica* e *valor_caracteristica*) seria uma opção, porém uma opção difícil se precisarmos de aninhamento.[2]

2 Aninhamento significa que temos uma coisa dentro da outra. Por exemplo, uma característica do livro pode ser o autor; mas o autor, por sua vez, pode ter outras características, como nome e data de nascimento.

Nessas situações, um *schema* flexível é conveniente. Para isso, podemos utilizar bancos de dados orientados a documentos, como o MongoDB (REDMOND; WILSON, 2012). Em nosso exemplo, cada produto seria um documento, que é um arquivo JSON, como a seguir:

```
{ "produto" :
  { "tipo": "livro",
   "titulo" : "Seven Databases in Seven Weeks",
   "ano" : 2012,
   "editora" : { "nome" : "Pragmatic Bookshelf",
                 "endereco" : "Dallas, Texas" }
  }
}
```

6.2 Alto desempenho

Em vários cenários da computação é importante a utilização de *caches*, que consistem na reutilização de resultados previamente calculados para diminuir o tempo de resposta do sistema. Por exemplo: uma vez que um supercomputador realiza o cálculo da previsão do tempo para determinado dia e local, não é conveniente repetir o cálculo para cada usuário que queira saber a previsão do tempo para aquele dia e local. Isso porque esse cálculo é muito caro. Nesse caso, guardamos o resultado em um *cache*, de forma que o resultado da previsão seja facilmente acessível quando fornecidos o dia e o local de interesse.

No uso de *cache*, tipicamente a informação é acessada por uma simples chave (dia e local em nosso exemplo). Não é esperada a realização de consultas complexas e imprevistas (por exemplo, em que dia da semana que vem vai fazer mais de 30°C e não vai chover?). Nesses casos, toda a capacidade de buscas flexíveis da SQL não é necessária; desse modo, passa a ser conveniente o uso de SGBDs do tipo chave-valor. A ideia é que você fornece uma chave (dia e local) para o SGBD, e ele lhe devolve um valor (previsão).

O Redis é um exemplo de SGBD NoSQL do tipo chave-valor (REDMOND; WILSON, 2012). Como a busca possui uma estrutura bem simples (a busca é sempre simplesmente pela chave), isso possibilita que uma consulta no Redis seja muito mais rápida do que em um banco relacional, que, por sua vez, deve dar suporte a consultas complexas possibilitadas pela SQL. O que calha bem para nosso caso, pois velocidade é um atributo relevante de um *cache*.

Outro aspecto de *caches* é que, na maioria das vezes, a durabilidade (o D do ACID) não é um requisito crítico: se perdermos o resultado do cálculo, ainda podemos recalculá-lo. Dessa forma, o Redis fornece várias opções de ajuste para o controle da durabilidade, proporcionando uma velocidade ainda maior. Isso acontece porque o SGBD pode lhe retornar um "sucesso" para uma operação de escrita sem mesmo ter certeza de que o valor será devidamente persistido.

Dependendo da situação, podemos afrouxar a durabilidade ao extremo e guardar o dado apenas na memória RAM. Nesse caso, a leitura e a escrita são realmente muito rápidas, porém no banco não vai caber muita coisa (a RAM costuma ter um tamanho modesto), e se houver algum problema, podemos perder todos os dados. Contudo, esses problemas não são críticos para uma solução de *cache*, por exemplo. Um exemplo de banco em memória é o Memcached.

6.3 Alta disponibilidade

Em muitas aplicações, a disponibilidade é um requisito muito importante. Uma queda em um site de vendas bem na *Black Friday* pode custar milhões de reais à empresa que vende produtos por ele. A indisponibilidade de sistemas embarcados que controlem veículos, como trens e aviões, pode até mesmo apresentar risco a vidas humanas.

Uma técnica comum para garantir a disponibilidade dos dados é a replicação: mantém-se uma cópia da base de dados em um outro servidor.

Se houver qualquer problema com a base principal, a aplicação aponta para a réplica. Essas réplicas também podem ser usadas para diminuir o tempo de resposta nas leituras: usuários no Brasil acessam uma réplica dos dados em São Paulo, enquanto usuários no Europa acessam uma réplica localizada em Paris, por exemplo. Bancos de chaves-valor, como o Riak, são também bem convenientes para a manutenção de réplicas de bases de dados (REDMOND; WILSON, 2012). Também é possível fazer réplicas com bancos relacionais, como o PostgreSQL.

PARA SABER MAIS

Escalabilidade horizontal é a ideia de que aumentamos a quantidade de réplicas de uma aplicação para que ela suporte uma carga crescente de requisições. Porém se o banco possui uma única réplica, chegará uma hora em que não adiantará mais replicar a aplicação, pois o banco se tornará o gargalo. Por isso, escalar o banco (fazer réplicas) pode ser uma necessidade. Mas escalar um banco de dados é bem mais complicado que escalar uma aplicação, por causa do problema da sincronização dos dados entre as réplicas.

Contudo, em bases relacionais que fornecem as propriedades ACID, o uso de réplicas aumenta o tempo de escrita. O SGBD precisa ter certeza de que o dado foi devidamente persistido em todas as réplicas antes de retornar sucesso à aplicação. Além disso, problemas de rede acontecem, e pode ser que o SGBD não consiga temporariamente acessar uma das réplicas. Nesse caso, para manter a consistência, o SGBD se negará a escrever qualquer dado enquanto alguma réplica estiver indisponível. Esse cenário prejudica a disponibilidade da aplicação em favor da consistência dos dados.

PARA SABER MAIS

O algoritmo clássico para garantir consistência entre sistemas distribuídos é o Two-Phase Commit (2PC). Mas dada a existência de falhas de comunicação na rede, percebe-se que é muito difícil que uma implementação de 2PC funcione para todos os modos de falha possíveis.

Procure conhecer o 2PC e depois tente responder: o que ocorre quando uma falha na rede impede que um participante receba a ordem de *commit* do coordenador? O que você faria para contornar essa situação?

Dependendo da situação, seria possível fazer o seguinte: se uma réplica B está indisponível, o SGBD pode escrever o dado na réplica A ativa e replicar o dado para B quando conseguir se reconectar com B. Mas note, quando B voltar à disponibilidade, pode ser que usuários façam leituras em B antes que a replicação aconteça. Pode ser também que B não esteja visível para A, mas que esteja visível para outros usuários que continuam efetuando operações em B; nessa situação, o estado de B vai divergindo do estado de A. Temos nesse cenário um afrouxamento da consistência para aumentar a disponibilidade.

Essa técnica de replicação tardia é também conhecida como "consistência em momento indeterminado" (*eventual consistency*) e é fornecida por alguns bancos NoSQL, como o Cassandra (DIANA, 2013).

PARA SABER MAIS

Partições na rede ocorrem quando, por exemplo, um sistema A não consegue acessar o sistema B. O teorema CAP diz que quando essas partições ocorrem, devemos escolher entre consistência ou disponibilidade (*availability*) (PRITCHETT, 2008).

Para privilegiar a disponibilidade em detrimento da consistência, devemos trocar as propriedades ACID pelas BASE (*basically available, soft state, eventually consistent*). O princípio fundamental por trás do BASE consiste em tolerar falhas parciais, de parte do sistema, sem que isso leve a falhas totais do sistema.

6.4 Modelagem com grafos

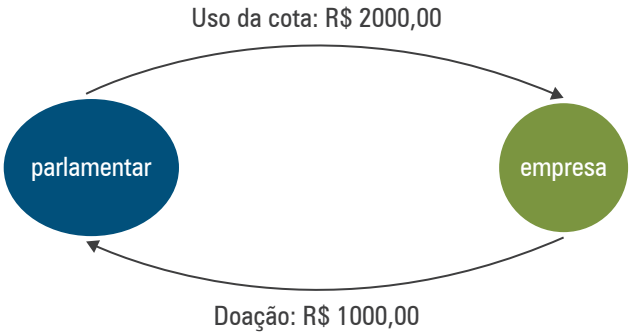
Grafos são estruturas de dados com nós ligados por arestas. São similares às árvores, já comentadas, mas sem a restrição de que um nó deve possuir apenas um pai. Há um grande corpo de conhecimento matemático sobre as propriedades de grafos e algoritmos que são executados sobre eles. Esse interesse surge porque muitos problemas do mundo real podem ser convenientemente modelados como grafos.

Exemplo 1: *nós* são cidades e as *arestas* indicam os caminhos existentes entre as cidades. Cada *aresta* possui um peso que indica a distância entre duas cidades. Existem algoritmos bem estudados para resolver problemas como o caminho mais curto entre duas certas cidades.

Exemplo 2: redes sociais podem ser modeladas como grafos. Usuários são os *nós* e as *arestas* indicam amizades entre os usuários. Existem algoritmos de "centralidade" que fornecem a importância de certo *nó* em termos de suas conexões.

Exemplo 3: a análise de grafos também tem sido utilizada para detectar padrões de fraudes (WEBBER, 2017a). Em geral, indícios de fraudes são sinalizados por certos padrões no grafo, tipicamente envolvendo ciclos, os chamados *fraud rings* (anéis de fraude). Um exemplo extremamente simples de detecção de indício de fraude com grafos seria a detecção do seguinte padrão: um candidato recebeu doação de determinada empresa; após eleito, o deputado gastou dinheiro de sua cota parlamentar com essa mesma empresa (figura 4).

Figura 4 – Exemplo de um pequeno anel de fraude em uma modelagem em grafo



Um SGBD orientado a grafos é o Neo4J (REDMOND; WILSON, 2012), que já foi utilizado até mesmo por equipes de jornalistas investigando casos de corrupção, como o caso *Panama Papers* (WEBBER, 2017b). O Neo4J fornece uma linguagem de consulta estilo SQL, mas voltada para buscas em grafos. Por exemplo, a detecção das ocorrências do padrão da figura 4 poderia ser feita com a consulta:

```
MATCH ring = (c:Parlamentar)-[:COTA]->(e:Empresa)-[:DOACAO]-
>(c:Parlamentar) RETURN ring
```

Considerações finais

Conhecemos neste capítulo diversos tipos de modelagem conceituais oferecidos por diversos SGBDs. A abordagem relacional é indiscutivelmente a mais popular entre desenvolvedores de sistemas. Embora algumas modelagens apresentadas dificilmente sejam utilizadas para novas aplicações, como a hierárquica e a rede, o profissional pode se deparar com essas modelagens na manutenção de sistemas legados. Além disso, outras abordagens são bastante usadas para situações específicas,

como a modelagem dimensional utilizada nos *data warehouses*. As abordagens de bancos NoSQL vêm também ganhando cada vez mais popularidade.

Por fim, tenha em mente que aprender a utilizar novos SGBDs faz parte da vida profissional e não deve ser um grande problema. O grande desafio é ter o embasamento conceitual e a experiência necessária para ser capaz de determinar qual o melhor SGBD ou tipo de modelagem conceitual a ser empregado para certo problema. Esteja preparado!

Referências

DATE, Christopher J. **Introdução a sistemas de bancos de dados**. Rio de Janeiro: Campus, 2004.

DIANA, Mauricio José de Oliveira de. **Desempenho de sistemas com dados georreplicados com consistência em momento indeterminado e na linha do tempo**. Dissertação (Mestrado em Ciência da Computação) – Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2013.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de banco de dados**. 7. ed. São Paulo: Pearson, 2018.

MARTIN, Robert C. The clean architecture. **The Clean Code Blog**, 2012. Disponível em: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html. Acesso em: 26 maio 2020.

PRITCHETT, Dan. Base: an Acid alternative. **ACM Queue**, v. 6 n. 3, p. 48-55, 2008.

RAMAKRISHNAN, Raghu; GEHRKE, Johannes. **Sistemas de gerenciamento de banco de dados**. 3. ed. Porto Alegre: AMGH, 2011.

REDMOND, Eric; WILSON, Jim R. **Seven databases in seven weeks**: a guide to modern databases and the NoSQL movement. [*S. l.*]: Pragmatic Bookshelf, 2012.

WEBBER, Jim. **Billions and billions**: using large-scale graph databases to detect financial fraud in real time. Palestra na Qcon, São Paulo, 2017a. Disponível em:

https://www.infoq.com/br/presentations/using-large-scale-graph-databases-
-to-detect-financial-fraud-in-real-time/. Acesso em: 26 maio 2020.

WEBBER, Jim. **The Panama Papers, graphs and data science**: unravelling the shady world of offshore finance one data structure at a time. Palestra na Qcon, São Paulo, 2017b. Disponível em: https://www.infoq.com/br/presentations/the-
-panama-papers-graphs-and-data-science/. Acesso em: 26 maio 2020.

Sobre o autor

Leonardo Alexandre Ferreira Leite é desenvolvedor de software no Serviço Federal de Processamento de Dados (Serpro) e mestre em ciência da computação pela Universidade de São Paulo (USP). Tem interesse no ensino de programação, já tendo realizado cursos livres sobre bancos de dados para profissionais da área de humanas, como jornalistas e gestores públicos. O conteúdo desses cursos está acessível em seu blog *Entrevistando Dados*.