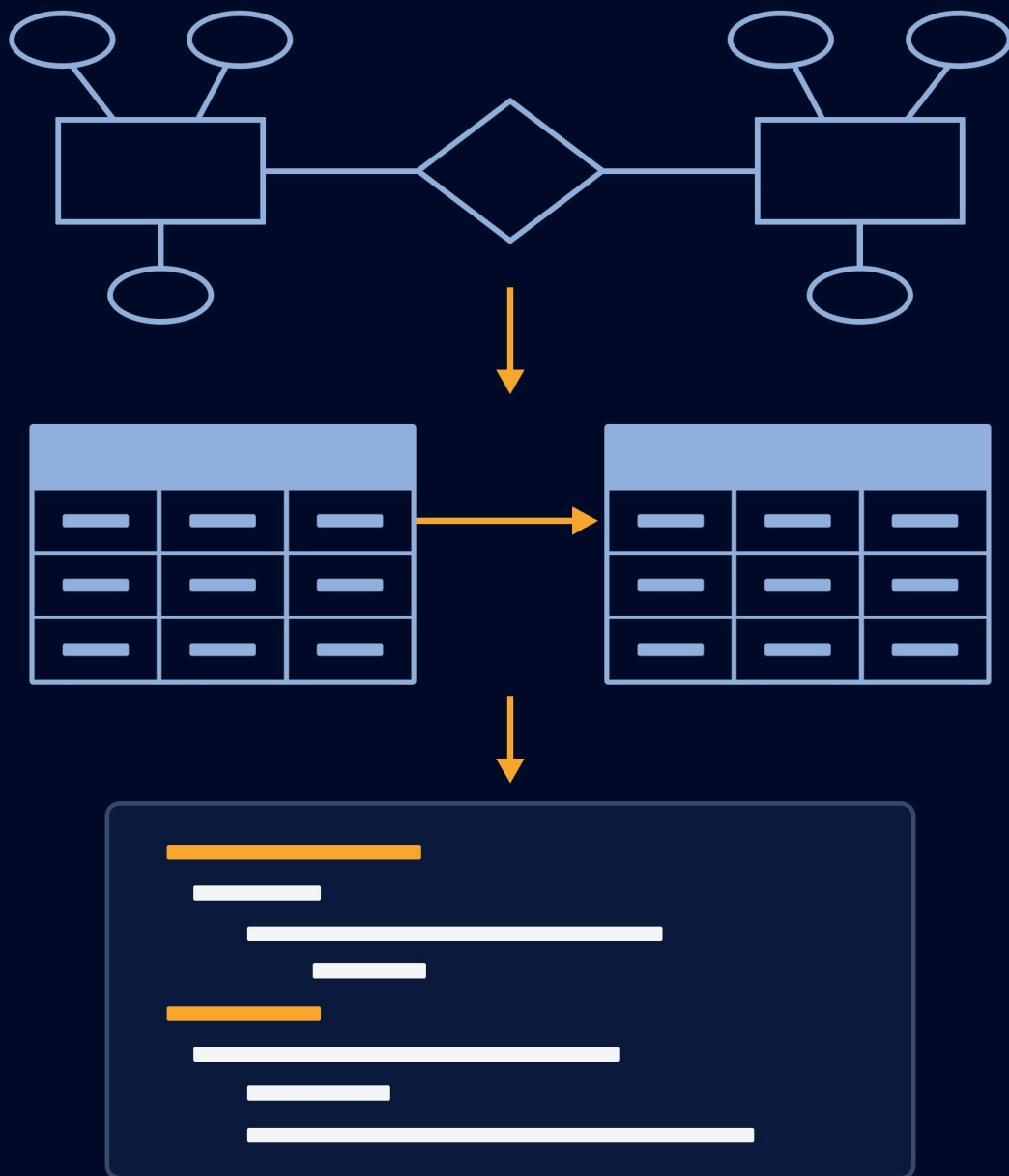


Banco de dados

Modelagem de dados

Do modelo conceitual ao SQL



2ª edição

Fábio J L Ferreira

Banco de dados — Modelagem de dados

Do modelo conceitual ao SQL

2ª edição

Fábio J L Ferreira

Site: <https://fabiojanio.com>

E-mail: contato@fabiojanio.com

Instagram: [@fabiojaniolima](https://www.instagram.com/fabiojaniolima)

LinkedIn: [linkedin.com/in/fabiojanio](https://www.linkedin.com/in/fabiojanio)

Licença

© 2026 Fábio J L Ferreira

Esta obra está licenciada sob a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0). O texto completo da licença está em <https://creativecommons.org/licenses/by-nc-sa/4.0/>.

Em resumo:

Você pode:

- copiar, distribuir e exibir esta obra;
- criar obras derivadas dela.

Com as seguintes condições:

- **Atribuição.** Dê crédito ao autor, Fábio J L Ferreira, indique o site dele (<https://fabiojanio.com>), inclua um link para a licença e informe se fez alterações.
- **Uso não comercial.** Você não pode usar a obra para fins comerciais. Os direitos de comercialização são exclusivos do autor, e qualquer uso comercial depende de autorização formal dele.
- **Mesma licença.** Se você alterar, transformar ou criar outra obra a partir desta, só poderá distribuí-la sob a mesma licença.

Este resumo não substitui a licença: em caso de divergência, vale o texto completo.

Por que uma 2ª edição

A primeira edição deste livro foi muito bem recebida. E, com a boa acolhida, vieram os pedidos: muitos leitores queriam que o trabalho continuasse. Pedido de leitor a gente não ignora.

Só que eu não queria apenas retocar o que já existia. Preferi escrever uma obra nova, que reunisse um conhecimento mais amplo, mais objetivo e mais relevante, em um material que valesse ainda mais para você.

Existe também um motivo pessoal. Sou apaixonado por tecnologia e, para mim, nunca bastou saber como uma coisa é feita: eu preciso entender como ela funciona. Decorar que uma tabela “deve” ter chave primária é fácil. Entender por que ela precisa de uma, e o que dá errado sem ela, é o que separa quem repete receitas de quem sabe modelar.

É isso que tento entregar nestas páginas. Em cada capítulo, você vai encontrar o como e, logo ao lado, o porquê. Se, ao final da leitura, você olhar para um banco de dados e enxergar as decisões que estão por trás dele, este livro terá cumprido o seu papel.

Sumário

Introdução	7
A quem este livro serve	7
O que este livro é, e o que não é	7
Como ler este livro	8
1 Banco de dados: que problema ele resolve	9
1.1 Dados e informação	9
1.2 Guardar dados em arquivos: onde isso quebra	10
1.3 Banco de dados e SGBD	11
1.4 O modelo relacional em poucas palavras	13
1.5 Os três níveis de um modelo: conceitual, lógico e físico	15
1.6 Resumo	16
2 Modelo conceitual: entidades e relacionamentos	18
2.1 Regras de negócio: a matéria-prima do modelo	19
2.2 O modelo entidade-relacionamento e o seu diagrama	20
2.3 Entidade	21
2.4 Atributo	22
2.5 Identificador	24
2.6 Relacionamento	25
2.7 Cardinalidade	27
2.8 Relacionamento com atributos e entidade associativa	29
2.9 Generalização e especialização	31
2.10 O DER do estudo de caso	33
2.11 Como avaliar um DER	36
2.12 Resumo	38
3 Modelo relacional: do diagrama às tabelas	39
3.1 Tabela, linha e coluna	39
3.2 Chaves: candidata, primária e estrangeira	40
3.3 Integridade de entidade e integridade referencial	42
3.4 NULL: o valor ausente	44
3.5 Do DER às tabelas: entidades, atributos e identificadores	45
3.6 Do DER às tabelas: relacionamentos	46
3.7 Do DER às tabelas: generalização e especialização	49
3.8 O estudo de caso em tabelas	51
3.9 Resumo	55

4 Normalização	56
4.1 Redundância e anomalias	56
4.2 Dependência funcional	58
4.3 Primeira forma normal (1FN)	60
4.4 Segunda forma normal (2FN)	61
4.5 Terceira forma normal (3FN)	63
4.6 Até onde normalizar	65
4.7 Resumo	67
5 SQL: do modelo ao banco	68
5.1 O padrão SQL	69
5.2 Um mapa da linguagem	69
5.3 Tipos de dados	70
5.4 Criando tabelas	72
5.5 Restrições	74
5.6 Ações referenciais	78
5.7 Alterando e removendo estruturas	81
5.8 O estudo de caso em SQL	83
5.9 Resumo	88
6 SQL: o modelo em uso	89
6.1 Inserindo linhas	90
6.2 Consultando	94
6.3 Juntando tabelas	97
6.4 Resumindo dados	101
6.5 Alterando e excluindo linhas	103
6.6 Transações	104
6.7 Dados derivados e visões	107
6.8 Resumo	109
Conclusão	110
O que você sabe fazer agora	110
Por onde continuar	111
Glossário	113

Introdução

Antes de desenhar a primeira tabela, vale combinar algumas coisas: para quem este livro foi escrito, o que você vai encontrar nele (e o que não vai) e como aproveitar melhor a leitura. É disso que esta introdução trata.

A quem este livro serve

Escrevi este livro para quem está começando a modelar bancos de dados relacionais e quer ir além da receita pronta. O conteúdo vai do básico ao intermediário e não exige que você saiba SQL: o que for preciso da linguagem aparece no momento em que fizer falta. Se você já trabalha com banco de dados, mas nunca entendeu direito por que as tabelas foram desenhadas daquele jeito, este livro também é para você.

O caminho que vamos percorrer é o mesmo de quem modela um banco na vida real. Primeiro, entendemos o problema a partir das regras de negócio. Depois, desenhamos o modelo conceitual, ainda sem pensar em tabelas. Em seguida, transformamos esse desenho em tabelas, conferimos o resultado com a normalização e, por fim, escrevemos tudo em SQL e colocamos o modelo para responder às perguntas do negócio.

Ao terminar, você vai saber percorrer esse caminho do começo ao fim e, principalmente, justificar cada decisão que tomar. Se alguém perguntar por que uma informação ficou em uma tabela e não em outra, ou por que um dado pode ficar em branco, você vai ter a resposta, e ela vai começar por uma regra de negócio. Esse é o refrão do livro: a modelagem depende da regra de negócio.

O que este livro é, e o que não é

Este é um livro sobre modelagem de dados em bancos relacionais, e é um livro teórico. Para acompanhar a leitura, você não precisa instalar programa, configurar servidor nem executar comando algum: tudo o que importa está no texto, nas figuras e nos exemplos.

Os exemplos de SQL seguem o padrão SQL, o conjunto oficial de regras da linguagem, mantido por organizações internacionais de padronização, e não por uma empresa. Para escrever cada comando de forma concreta, usei o PostgreSQL como referência de sintaxe: todo exemplo do livro está escrito em uma forma que o PostgreSQL aceita. Mas este não é um livro de PostgreSQL. Quando ele se afasta do padrão, ou oferece um atalho que você certamente vai encontrar por aí, o livro avisa em um bloco próprio. Também não prometo que todo exemplo funcione igual em qualquer banco: nenhum sistema de banco de dados declara seguir por completo nem mesmo o núcleo do padrão, e as diferenças existem.

Algumas coisas ficaram de fora de propósito. Instalação, ferramentas e administração do banco, como a criação de usuários e permissões, não entram porque nada disso é modelagem. Índices e ajuste de desempenho também não: são decisões que dependem do sistema de banco de dados escolhido, e não do desenho do modelo. Na normalização, o livro vai até a terceira forma normal e apenas cita as formas seguintes.

Como ler este livro

Todos os exemplos do livro vêm do mesmo lugar: uma escola de cursos livres de tecnologia, com turmas presenciais e on-line. Ela não tem nome; no texto, é simplesmente “a escola”. Escolhi esse cenário porque você conhece as regras dele sem precisar de explicação (afinal, todo mundo já foi aluno) e porque ele oferece situações para tudo o que o livro precisa mostrar. As regras de negócio da escola aparecem quando começarmos a modelar e acompanham você até o fim. As figuras e os comandos SQL usam os mesmos cursos, as mesmas turmas e os mesmos alunos: o dado que aparece em uma figura é o mesmo que aparece no SQL.

Quando um termo técnico aparece pela primeira vez, ele vem em **negrito** e é definido na mesma frase. Esses termos estão reunidos no glossário, no fim do livro, cada um com a indicação da seção que o explica. Nomes de tabelas e colunas, comandos e valores, ou seja, tudo o que você digitaria ou veria no banco, aparecem em fonte de largura fixa, `assim`. De vez em quando, mostro de propósito um comando errado, para você ver o banco recusá-lo. Esses comandos começam com a linha - - **ERRO ESPERADO**, seguida do motivo.

Ao longo do texto, alguns trechos aparecem destacados em blocos, cada um com um papel:

- **Atenção:** um risco real, como perder dados ou cometer um erro frequente, e a forma de evitá-lo.
- **Importante:** uma ideia da qual o restante do capítulo depende.
- **Padrão SQL × PostgreSQL:** um ponto em que o PostgreSQL difere do padrão, vai além dele ou oferece um atalho que você vai encontrar em outros materiais.
- **Cenário:** uma situação do mundo real que ancora o conceito que acabou de ser apresentado.

Combinado? Então vamos começar pelo começo: afinal, que problema um banco de dados resolve?

1 Banco de dados: que problema ele resolve

“Os futuros usuários de grandes bancos de dados precisam ser protegidos de ter de saber como os dados estão organizados na máquina (a representação interna).”

Edgar F. Codd (tradução livre)

Na Introdução, deixei uma pergunta no ar: afinal, que problema um banco de dados resolve? Para responder, vamos voltar a um tempo em que a escola ainda não tinha banco de dados. Os dados existiam, claro: a secretaria anotava alunos, cursos e turmas numa planilha, o financeiro controlava as cobranças em outra, e cada setor cuidava da sua. Funcionava. Até o dia em que deixou de funcionar.

Neste capítulo, vamos ver onde esse arranjo quebra e o que um banco de dados, junto com o software que o gerencia, faz para resolver isso. No caminho, você ganha o vocabulário mínimo para o resto do livro: o que é o modelo relacional e em quais níveis um modelo de dados é desenhado. Ainda não vamos modelar nada. Primeiro, precisamos entender por que vale a pena.

O que será visto neste capítulo:

- [Dados e informação](#)
- [Guardar dados em arquivos: onde isso quebra](#)
- [Banco de dados e SGBD](#)
- [O modelo relacional em poucas palavras](#)
- [Os três níveis de um modelo: conceitual, lógico e físico](#)
- [Resumo](#)

1.1 Dados e informação

Antes de falar de banco de dados, vale separar duas palavras que, no dia a dia, usamos como se fossem a mesma coisa: dado e informação. A diferença entre elas explica por que alguém se dá ao trabalho de organizar dados com tanto cuidado, e é por essa diferença que começamos.

Um **dado** é uma sequência de símbolos que pode ser guardada e processada por uma máquina sem que a máquina precise entender o que esses símbolos querem dizer. Já a **informação** é o que um ou mais dados passam a significar para alguém que conhece o contexto deles. Uso aqui a distinção proposta por Valdemar W. Setzer no artigo “Dado, informação, conhecimento e competência”, de 1999. O critério de Setzer é justamente este: o dado pode ser tratado por uma máquina; o significado depende de quem interpreta o dado.

Pense no valor 8.5. Sozinho, ele é só um dado: o computador guarda, compara e soma esse número sem fazer ideia de que se trata. Agora junte outros dados ao 8.5: esse número é a nota final da Ana Lima no curso Modelagem de Dados, e a escola emite certificado para quem termina o curso com nota a partir de 7. Para a secretária que lê essa combinação, o que era dado virou informação: a Ana foi aprovada e tem direito ao certificado. Notou a diferença? O número continua o mesmo. O que mudou foi o contexto em que ele aparece.

É aqui que entra quem modela um banco de dados. A informação depende de o dado estar no lugar certo, ligado aos outros dados certos e sem contradição. Se a nota é registrada sem o aluno e o curso a que ela pertence, o dado existe, mas não informa nada. Se a mesma nota aparece em dois lugares, com valores diferentes, aí já era: não dá para saber em qual dos dois confiar. Organizar dados, portanto, é garantir que eles continuem produzindo informação confiável.

E é justamente isso que fica difícil quando os dados se espalham por vários arquivos, como vamos ver a seguir.

1.2 Guardar dados em arquivos: onde isso quebra

Guardar dados em arquivos, como planilhas e documentos de texto, é o jeito mais simples de começar, e muita organização começa assim. O problema não aparece no primeiro dia. Ele aparece quando os dados crescem, quando mais de um setor passa a depender deles e quando mais de uma pessoa precisa alterá-los ao mesmo tempo. Nesta seção, vamos ver três problemas que surgem nesse momento, porque são eles que o restante do livro resolve.

Cenário

Antes de ter um banco de dados, a escola guardava tudo em planilhas. A secretaria mantinha uma, com os alunos, os cursos e as turmas; o financeiro mantinha outra, com os dados de quem precisava pagar. Quando a Ana Lima se matriculou no curso Modelagem de Dados, cada setor anotou o e-mail dela na própria planilha. Meses depois, a Ana trocou de e-mail e avisou a secretaria, que corrigiu a planilha. Ninguém avisou o financeiro.

Veja como ficou a linha da Ana em cada planilha. Na da secretaria:

Nome	E-mail	Curso
Ana Lima	ana.lima@example.com	Modelagem de Dados

E na do financeiro:

Nome	E-mail	Valor
Ana Lima	analima@example.com	560,00

O e-mail da Ana está guardado em dois lugares. Esse é o primeiro problema: **redundância**, o mesmo dado guardado em mais de um lugar. Não confunda com a redundância planejada, como as cópias que um sistema mantém para continuar funcionando quando um servidor falha: nesse caso, as cópias existem de propósito, e o próprio sistema se encarrega de levar cada alteração a todas elas. Aqui, falamos de cópias que ninguém controla. Por si só, essa redundância já custa trabalho, porque toda alteração precisa ser repetida em todas as cópias. O estrago de verdade vem quando alguém esquece uma delas, como aconteceu aqui.

As duas planilhas passaram a discordar, e esse é o segundo problema: **inconsistência**, a situação em que cópias do mesmo dado guardam valores diferentes. Qual dos dois e-mails está certo? Olhando só para os arquivos, não há como saber. O financeiro vai mandar a cobrança para um endereço que a Ana não usa mais, e ninguém vai perceber até o pagamento atrasar.

O terceiro problema aparece quando duas pessoas alteram o mesmo arquivo ao mesmo tempo. Suponha que a planilha da secretaria fique numa pasta compartilhada e que duas atendentes abram o arquivo na mesma manhã. A primeira corrige o e-mail da Ana. A segunda cadastra um aluno novo, o Bruno Costa. O programa que elas usam carrega o arquivo inteiro ao abri-lo e o grava inteiro ao salvar, então cada uma trabalha na sua própria cópia. A primeira salva. Minutos depois, a segunda salva também, e a cópia dela, que tem o Bruno, mas ainda tem o e-mail antigo da Ana, é gravada por cima da que a primeira salvou. A correção feita pela primeira atendente sumiu, sem mensagem de erro e sem ninguém perceber.

Repare que o problema não é o arquivo estar aberto por duas pessoas. Dois programas conseguem, sim, abrir o mesmo arquivo e gravar nele ao mesmo tempo. O problema é que ninguém coordena essas gravações: nada no arquivo sabe que há duas alterações em andamento, nem decide o que fazer quando uma passa por cima da outra.

Redundância, inconsistência e gravações simultâneas sem coordenação têm uma causa em comum: os dados ficam soltos, e cada pessoa ou programa que mexe neles precisa cuidar das regras por conta própria. A saída é colocar no meio do caminho alguém que conheça os dados e as regras e coordene quem os acessa. Esse alguém é o assunto da próxima seção.

1.3 Banco de dados e SGBD

Banco de dados e SGBD são duas coisas diferentes, que no dia a dia se confundem o tempo todo. Separar as duas agora mostra quem, afinal, resolve os problemas da seção anterior, e evita confusão no resto do livro.

Um **banco de dados** (*database*) é uma coleção de dados relacionados entre si, organizada para atender a um propósito. No nosso caso, são os dados de que a escola precisa para funcionar: cursos, turmas, alunos, matrículas. Já o **SGBD**, sistema gerenciador de banco de dados (*Database Management System*), é o software que cria, guarda e controla essa coleção. Os programas da escola, como o sistema da secretaria e o do financeiro, não mexem nos dados diretamente: fazem pedidos ao SGBD, e é ele quem lê, grava e decide se cada pedido pode ser atendido.

Importante

Banco de dados é a coleção de dados; SGBD é o software que a gerencia. Quando alguém diz que “o banco recusou o cadastro”, quem recusou foi o SGBD, cumprindo uma regra declarada no banco. Ao longo do livro, em frases como essa, “o banco” quer dizer o SGBD.

Com um banco de dados, a secretaria e o financeiro passam a consultar o mesmo lugar, e o e-mail da Ana pode ficar guardado uma vez só. Mas calma aí: o banco não elimina a redundância sozinho. Se o modelo prever o e-mail da Ana em dois lugares diferentes do banco, o SGBD vai guardar as duas cópias sem reclamar, e a inconsistência volta pela porta dos fundos. Quem elimina a redundância é a modelagem, e é para isso que este livro existe.

O que o SGBD assume para si são quatro tarefas que, com arquivos, ficavam por conta de cada programa.

A primeira é a **integridade** (*integrity*), a garantia de que os dados obedecem às regras declaradas no banco. Com arquivos, cada programa precisa conferir as regras por conta própria, e basta um deles esquecer para o dado errado entrar. Com um SGBD, a regra é declarada uma vez, no próprio banco, e o SGBD a confere em toda gravação, venha de qual programa vier. Se a escola declarar que não pode haver duas pessoas com o mesmo CPF, o SGBD recusa o segundo cadastro com um CPF já usado, não importa se o pedido veio da secretaria, do financeiro ou de um programa que ainda nem existe.

A segunda é a **concorrência** (*concurrency*), o acesso de várias pessoas ou programas aos mesmos dados ao mesmo tempo. O SGBD coordena esse acesso: cada alteração atinge só os dados que ela muda, e, quando duas alterações disputam o mesmo dado, o SGBD faz uma esperar a outra ou, conforme o caso, recusa uma delas com um aviso, em vez de deixar uma apagar a outra em silêncio. Lembra-se das duas atendentes? Com um SGBD, a correção do e-mail da Ana e o cadastro do Bruno são duas alterações separadas, cada uma no seu dado, e as duas ficam gravadas.

A terceira é a **transação** (*transaction*), um conjunto de operações que o SGBD executa por inteiro ou não executa. Pense na transferência de um aluno para outra turma do mesmo curso: ele sai de uma turma e entra na outra. Se a primeira parte acontece e a segunda falha, por uma queda de energia, por exemplo, o aluno fica sem turma nenhuma. Agrupadas numa transação, as duas operações acontecem juntas ou nenhuma acontece: se algo falha no meio do caminho, o SGBD desfaz o que já tinha feito. Com arquivos, quem garante isso é o programa, se o programador tiver se lembrado disso. As transações voltam, com os comandos que as controlam, no [capítulo 6](#).

A quarta é uma linguagem única. Com arquivos, cada programa precisa saber em que formato os dados estão gravados: em que coluna da planilha fica o e-mail, como a data foi escrita, onde termina um registro e começa o outro. Se o formato muda, todo programa que lê o arquivo precisa mudar junto. Com um SGBD, os programas pedem os dados numa linguagem feita para isso, sem saber como os dados estão organizados lá dentro. Nos bancos relacionais, essa linguagem é o **SQL**, que começamos a escrever no [capítulo 5](#). É a ideia da epígrafe deste capítulo: quem usa os dados não precisa saber como eles estão guardados na máquina.

Falei em “bancos relacionais” sem explicar o que isso quer dizer. É o que vem a seguir.

1.4 O modelo relacional em poucas palavras

Todo SGBD representa os dados de um certo jeito, e esse jeito define como você vai descrever e consultar os dados da escola. Este livro trabalha com o **modelo relacional** (*relational model*), a forma de representar dados proposta por Edgar F. Codd em 1970, em que todos os dados ficam em tabelas. Aqui vai só o bastante para você se situar; o [capítulo 3](#) aprofunda cada termo.

No modelo relacional, uma **tabela** (*table*) reúne os dados de um mesmo assunto. Cada **linha** (*row*) da tabela guarda um registro, como um curso ou uma pessoa, e cada **coluna** (*column*) guarda o mesmo tipo de dado para todos os registros, como o título ou o preço. Veja a tabela curso da escola:

id	título	carga_horaria	preco
1	Lógica de Programação	40	480.00
2	Python Essencial	60	720.00
3	Modelagem de Dados	40	560.00

Cada linha é um curso do catálogo, e cada coluna é um dado sobre ele: o título, a carga horária, medida em horas, e o preço. A segunda linha diz, por exemplo, que o curso Python Essencial tem carga horária de 60 horas e preço de 720.00.

E a coluna `id`? Ela existe para que cada linha possa ser identificada sem ambiguidade: o valor 2 aponta para um curso, e só para ele. Uma **chave** (*key*) é uma coluna, ou um conjunto de colunas, com um papel especial na tabela: identificar cada linha, como faz a `id`, ou apontar para uma linha de outra tabela. É por meio das chaves que as tabelas se ligam. A tabela das turmas, por exemplo, vai guardar em uma coluna o `id` do curso de cada turma. Se o título também não se repete na escola, por que não usar o título para identificar o curso? Boa pergunta, e ela tem resposta: é a escolha entre chave natural e chave artificial, que o [capítulo 3](#) faz com calma.

Por que o livro usa o modelo relacional? Por dois motivos, ligados a tudo o que vimos até aqui. O primeiro é que a estrutura das tabelas é declarada antes dos dados: o SGBD sabe de antemão quais colunas cada tabela tem e a quais regras elas precisam obedecer, e por isso consegue cobrar essas regras em toda gravação, como na integridade da seção anterior. O segundo é que as tabelas se ligam pelos valores das chaves, e não por um caminho fixo gravado junto dos dados. Por isso, uma consulta pode combinar quaisquer tabelas que tenham valores em comum, inclusive para responder a perguntas que ninguém previu quando o banco foi desenhado.

Isso não quer dizer que o modelo relacional sirva para tudo. Existem bancos de outros tipos, que muita gente agrupa sob o nome **NoSQL**, uma etiqueta para os bancos que não seguem o modelo relacional. Em vez de um catálogo deles, vale conhecer o critério de escolha. Martin Fowler observa que boa parte desses bancos, como os bancos de documentos, guarda junto, num bloco só, o que costuma ser lido junto: tudo sobre uma turma, com o curso, o instrutor e os alunos, num único documento. Isso funciona muito bem enquanto o acesso segue esse bloco, e complica quando alguém precisa olhar os mesmos dados por outro ângulo, como na pergunta “em quais turmas a Ana está matriculada?”. Outro tipo, o banco de grafos, guarda cada ligação junto dos dados que ela liga. Ele serve quando as perguntas consistem em percorrer ligações em cadeia, sem saber de antemão quantos passos o caminho vai ter; no modelo relacional, cada passo desse caminho é mais uma combinação de tabelas.

Na [Figura 1.1](#) temos as mesmas turmas guardadas de três jeitos. O painel de cima mostra o modelo relacional: a tabela `turma` guarda na coluna `curso_id` o `id` do curso de cada turma, e a seta representa essa ligação. A turma 10, por exemplo, é do curso 2, Python Essencial. O painel do meio mostra o banco de documentos, em que cada turma é um documento que reúne o curso, o instrutor e os alunos. Para ler uma turma, basta um documento; já a pergunta sobre a Ana atravessa os documentos, porque o nome dela aparece repetido, uma vez em cada um. O painel de baixo mostra o banco de grafos, em que a Ana aparece num único círculo; as duas ligações “cursa” que saem desse círculo levam direto às turmas 10 e 11.

A escola é o oposto das situações que favorecem os bancos de documentos e de grafos: dados com estrutura estável, muitas ligações entre eles, regras que precisam valer para todos os programas e perguntas que mudam com o tempo. É o terreno do modelo relacional.

Relacional Tabelas ligadas por chaves

curso

id	título
1	Lógica de Programação
2	Python Essencial
3	Modelagem de Dados

turma

id	curso_id
10	2
11	3

NoSQL

Documentos

Guarda junto o que é lido junto

```
{
  "turma": 10,
  "curso": "Python Essencial",
  "instrutor": "Marta",
  "alunos": ["Ana", "Bruno"]
}
```

```
{
  "turma": 11,
  "curso": "Modelagem de Dados",
  "instrutor": "Paulo",
  "alunos": ["Ana", "Carla"]
}
```

Grafos

Percorre ligações em cadeia

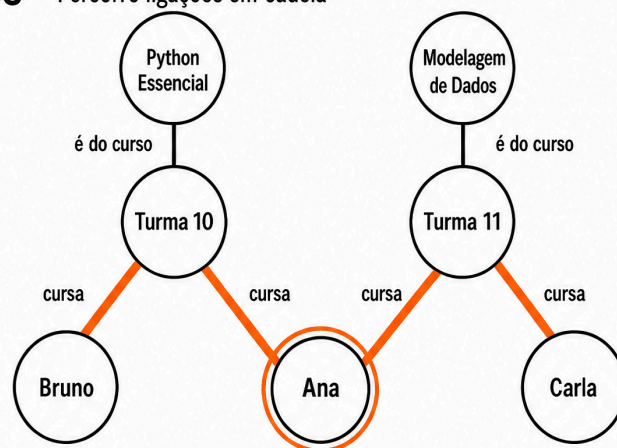


Figura 1.1: Os mesmos dados da escola guardados em um banco relacional, em um banco de documentos e em um banco de grafos.

1.5 Os três níveis de um modelo: conceitual, lógico e físico

Um **modelo de dados** (*data model*) é a descrição dos dados que um banco guarda e de como eles se organizam. O mesmo banco pode ser descrito com mais ou menos detalhes de implementação, e é daí que vêm os três níveis desta seção. Eles também servem de mapa: cada nível corresponde a uma parte do livro, e saber em qual delas você está ajuda a entender por que cada capítulo se preocupa com umas coisas e ignora outras.

O **modelo conceitual** (*conceptual model*) descreve os dados do negócio sem depender de nenhum SGBD: quais coisas a escola precisa registrar e como elas se ligam. Quem lê um modelo conceitual não precisa saber nada de tabelas; basta conhecer o negócio. É o nível do [capítulo 2](#), em que vamos desenhar o modelo da escola num diagrama.

O **modelo lógico** (*logical model*) descreve os mesmos dados do jeito que o tipo de SGBD escolhido os representa. Como escolhemos um SGBD relacional, o modelo lógico da escola é feito de tabelas, colunas e chaves. Ele é construído no [capítulo 3](#), conferido no [capítulo 4](#) e escrito em SQL no [capítulo 5](#) e no [capítulo 6](#).

O **modelo físico** (*physical model*) trata de como um SGBD específico grava e encontra os dados: estruturas de acesso, como os índices, e outros parâmetros que afetam o desempenho. O modelo físico não muda quais dados o banco guarda nem as regras que o banco cobra. Esse nível fica fora do livro, pelo motivo que dei na Introdução: são decisões que dependem do SGBD escolhido, e não do desenho do modelo.

Veja um mesmo fato da escola nos três níveis: toda turma é de um curso. No modelo conceitual, isso aparece como uma ligação entre **Turma** e **Curso**. No modelo lógico, a tabela `turma` ganha uma coluna `curso_id`, que guarda o `id` do curso de cada turma. No modelo físico, entrariam as decisões sobre como o SGBD encontra depressa as turmas de um curso. A regra de negócio é a mesma nos três; o que muda é o quanto de detalhe de implementação cada nível carrega.

Nível	O que descreve	Onde fica no livro
Conceitual	os dados do negócio, sem depender de SGBD	capítulo 2
Lógico	as tabelas, as colunas e as chaves	do capítulo 3 ao capítulo 6
Físico	como o SGBD grava e encontra os dados	fora do livro

Uma última observação, para não confundir as coisas: os três níveis são níveis de modelagem, ou seja, três descrições do mesmo banco, feitas em momentos diferentes do projeto. Eles não são partes do SGBD nem etapas do funcionamento dele.

1.6 Resumo

Começamos separando dado de informação: o dado é a sequência de símbolos que a máquina guarda e processa; a informação é o que esse dado significa para quem conhece o contexto em que ele aparece. Vimos que a informação só é confiável quando os dados estão organizados e sem contradição.

Depois, voltamos ao tempo das planilhas da escola e encontramos três problemas: a redundância, o mesmo dado guardado em mais de um lugar; a inconsistência, que aparece quando essas cópias passam a discordar; e as gravações simultâneas sem coordenação, em que a alteração de uma pessoa apaga a de outra sem ninguém perceber.

Em seguida, separamos o banco de dados, que é a coleção de dados, do SGBD, que é o software que a gerencia. Vimos que o SGBD cobra as regras declaradas no banco, coordena o acesso simultâneo, executa cada transação por inteiro ou não a executa e oferece uma linguagem única para todos os programas. Vimos também que juntar os dados num lugar só não elimina a redundância: quem faz isso é a modelagem.

Conhecemos o modelo relacional, em que os dados ficam em tabelas, com linhas, colunas e chaves, e os motivos de ele ser o modelo deste livro, além das situações em que outros tipos de banco servem melhor. Por fim, vimos os três níveis de um modelo de dados, o conceitual, o lógico e o físico, e onde cada um aparece no livro.

Hora de uma pausa para o café. Na volta, começamos a modelar a escola, pelo nível conceitual.

2 Modelo conceitual: entidades e relacionamentos

“O modelo entidade-relacionamento adota a visão mais natural de que o mundo real é formado por entidades e relacionamentos.”

Peter Chen (tradução livre)

No capítulo anterior, vimos por que a escola precisa de um banco de dados e que a redundância só desaparece se o modelo for bem desenhado. Agora é hora de desenhar. Mas por onde começar? A tentação é sair criando tabelas, e é aí que muita modelagem dá errado: antes de decidir como guardar os dados, precisamos entender quais dados a escola guarda e como eles se ligam.

É disso que trata o modelo conceitual, o primeiro dos três níveis de modelo. Neste capítulo, vamos conhecer as regras de negócio da escola e transformá-las, uma a uma, num diagrama que qualquer pessoa da escola consegue ler, mesmo sem saber o que é uma tabela. As tabelas ficam para o próximo capítulo.

O que será visto neste capítulo:

- Regras de negócio: a matéria-prima do modelo
- O modelo entidade-relacionamento e o seu diagrama
- Entidade
- Atributo
- Identificador
- Relacionamento
- Cardinalidade
- Relacionamento com atributos e entidade associativa
- Generalização e especialização
- O DER do estudo de caso
- Como avaliar um DER
- Resumo

2.1 Regras de negócio: a matéria-prima do modelo

Todo modelo de dados começa por uma pergunta simples: como o negócio funciona? A resposta vem na forma de regras de negócio, e é delas que sai cada decisão deste livro. Por isso, antes de desenhar qualquer coisa, vamos conhecer as regras da escola.

Uma **regra de negócio** (*business rule*) é uma afirmação sobre o funcionamento da organização que diz quais dados ela guarda ou quais limites esses dados precisam respeitar. “Toda turma tem exatamente um instrutor” é uma regra de negócio: ela diz que a escola registra o instrutor de cada turma e que não existe turma sem instrutor nem turma com dois.

As regras não nascem na cabeça de quem modela. Elas vêm de quem opera o negócio: da secretária que matricula os alunos, do coordenador que monta as turmas, do financeiro que cobra o valor de cada matrícula. Vêm também dos documentos que o negócio já usa, como a ficha de matrícula e o certificado. O trabalho de quem modela é ouvir, ler e escrever cada regra de um jeito que não deixe dúvida.

Para servir à modelagem, uma regra precisa dizer as quantidades envolvidas. “O aluno se matricula em turmas” não basta: em quantas turmas? Pode ser em nenhuma? Pode ser em mais de uma? Além disso, a regra não pode admitir duas leituras. Quando admite, alguém escolhe uma delas sem perguntar, e o banco passa a cobrar uma regra que ninguém aprovou.

A pergunta “pode ser em nenhuma?” não está aí por acaso: esbarrei nela ao preparar este capítulo. A primeira versão da regra das matrículas dizia que um aluno podia se matricular em várias turmas, mas não dizia se ele podia ficar sem nenhuma matrícula. Parece detalhe, e não é: a resposta decide se o banco aceita ou recusa o cadastro de um aluno que ainda não se matriculou em turma nenhuma. Quem responde é o negócio, e não quem modela: a escola cadastra a pessoa como aluna antes de fechar a matrícula. A regra 8, logo abaixo, ganhou uma frase que registra essa resposta.

Atenção

Lacuna preenchida por suposição vira regra que ninguém aprovou, e o erro só aparece quando o banco já está em uso, recusando um dado válido ou aceitando um inválido. Diante de uma regra incompleta, pergunte a quem conhece o negócio e registre a resposta na própria regra.

Estas são as regras da escola. Elas acompanham o livro inteiro, e cada decisão dos próximos capítulos vai remeter a uma delas pelo número.

1. **Cursos.** A escola mantém um catálogo de cursos e guarda o título, a ementa, a carga horária e o preço atual de cada um. Não há dois cursos com o mesmo título.
2. **Pré-requisitos.** Um curso pode exigir outros cursos como pré-requisito, e um mesmo curso pode ser pré-requisito de vários.
3. **Turmas.** Cada turma é de um único curso e tem data de início, data de término e número de vagas. A escola dá a cada turma um código, que não se repete. Um curso pode estar no catálogo sem ter turma.

4. **Modalidade.** Toda turma é presencial ou on-line, nunca as duas. A escola guarda a sala de cada turma presencial e o endereço da sala virtual de cada turma on-line.
5. **Pessoas.** A escola guarda o nome, o CPF, o e-mail, a data de nascimento e um ou mais telefones de cada pessoa. Não há duas pessoas com o mesmo CPF.
6. **Papéis.** Toda pessoa cadastrada é aluna, instrutora ou as duas coisas. A escola guarda a escolaridade de cada aluno, assim como a data de admissão e o valor da hora-aula de cada instrutor.
7. **Instrutor da turma.** Toda turma tem exatamente um instrutor. Um instrutor pode ter várias turmas ou nenhuma.
8. **Matrículas.** Um aluno pode se matricular em várias turmas, e uma turma recebe vários alunos, até o limite de vagas. A escola cadastra a pessoa como aluna antes de fechar a matrícula, de modo que pode haver aluno ainda sem matrícula. Ninguém se matricula duas vezes na mesma turma. A escola guarda a data, o valor cobrado e a nota final de cada matrícula. O valor cobrado é o preço do curso naquele dia. A nota final vai de 0 a 10 e só é lançada ao fim da turma.
9. **Certificados.** A matrícula com nota final a partir de 7 dá direito a um certificado, com código de verificação único e data de emissão. Cada matrícula gera no máximo um certificado, e todo certificado é de exatamente uma matrícula.
10. **Exclusões.** Quando uma pessoa é excluída, os telefones dela saem junto. Turma com matrículas não pode ser excluída, e curso com turmas também não.
11. **Transferência.** Um aluno pode ser transferido para outra turma do mesmo curso: ele sai de uma turma e entra na outra, as duas coisas ou nenhuma.

Repare que as regras não são todas do mesmo tipo. Umas dizem o que a escola guarda, como a regra 1; outras dizem como as coisas se ligam, como a regra 7; e as regras 10 e 11 tratam do que acontece quando os dados são excluídos ou mudam. Neste capítulo, interessam as que dizem o que se guarda e como as coisas se ligam. As regras 10 e 11 voltam no [capítulo 5](#) e no [capítulo 6](#).

Com as regras em mãos, falta uma forma de desenhá-las. É o que vem a seguir.

2.2 O modelo entidade-relacionamento e o seu diagrama

Regras escritas em português descrevem bem o negócio, mas não mostram de relance como as coisas se ligam. Para isso, a modelagem conceitual usa uma forma de descrição própria, com poucos elementos, e um desenho que a representa. Esta seção apresenta essa forma de descrição e esse desenho. Mostra também a notação do livro, ou seja, o conjunto de símbolos com que o desenho é feito.

O **Modelo Entidade-Relacionamento (MER)** (*Entity-Relationship Model*) é uma forma de descrever os dados de um negócio por meio de entidades, dos atributos delas e dos relacionamentos entre elas. O MER foi proposto por Peter Chen em 1976, no artigo de onde saiu a epígrafe deste capítulo. O **Diagrama Entidade-Relacionamento (DER)** (*Entity-Relationship Diagram*) é o desenho que representa um MER. As duas siglas andam juntas: o MER é a descrição; o DER, o desenho dessa descrição.

Não confunda o MER com o modelo relacional, do capítulo anterior. Os nomes se parecem, mas o MER pertence ao nível conceitual e não fala em tabelas, enquanto o modelo relacional, de Codd, é o que representa os dados em tabelas. E o nome certo é “entidade-relacionamento”, e não “entidade relacional”, que mistura os nomes dos dois modelos.

O DER não tem um desenho único. Ao longo dos anos, surgiram várias notações, que representam as mesmas ideias com símbolos diferentes. Este livro usa a notação de Chen, com as convenções de Carlos Alberto Heuser, autor de *Projeto de Banco de Dados*, livro de referência da literatura brasileira sobre o assunto. Escolhi essa notação por um motivo prático: nela, entidade e relacionamento têm símbolos de formatos diferentes, e cada atributo aparece ligado ao elemento que descreve. Nada no diagrama lembra uma tabela, o que ajuda a manter a atenção no negócio, e não na implementação.

Os símbolos são poucos:

Elemento	Símbolo	Onde aparece
entidade	retângulo	seção 2.3
atributo	círculo vazio, ligado por uma linha	seção 2.4
identificador	círculo preenchido	seção 2.5
relacionamento	losango	seção 2.6
cardinalidade	par (mínima,máxima) junto à linha	seção 2.7
entidade associativa	retângulo em volta do losango	seção 2.8
generalização e especialização	triângulo	seção 2.9

Não precisa decorar a tabela: cada símbolo volta, com uma figura, na seção que o explica. Começamos pelo mais básico deles, o retângulo.

2.3 Entidade

Hora de começar a modelar. A primeira pergunta de quem modela é: sobre o que a escola precisa guardar dados? Cursos, turmas, pessoas, certificados: cada resposta é uma candidata a entidade. Vale caprichar nesta etapa, porque o resto do modelo se apoia nela. Os atributos descrevem entidades, os relacionamentos ligam entidades, e uma entidade mal escolhida arrasta junto tudo o que depende dela.

Uma **entidade** (*entity*) é um conjunto de coisas do negócio, concretas ou abstratas, sobre as quais se guardam dados. Cada elemento desse conjunto é uma **ocorrência** (*instance*) da entidade. Na escola, **Curso** é uma entidade: o catálogo tem vários cursos, e a escola guarda os mesmos dados de cada um. Lógica de Programação, Python Essencial e Modelagem de Dados são três ocorrências de **Curso**. Da mesma forma, **Pessoa** é uma entidade, e a Ana Lima e o Bruno Costa são ocorrências dela. No DER, a entidade é um retângulo com o nome dentro. As ocorrências não aparecem no diagrama, que descreve o conjunto, e não cada elemento.

Repare que uma entidade não precisa ser algo que se possa tocar. Uma pessoa é concreta; um curso, não. Ele é uma proposta de ensino, com ementa, carga horária e preço. O que faz de **Curso** uma entidade é a escola guardar dados sobre cada curso, e não o fato de um curso ser concreto ou abstrato.

Agora, um ponto que derruba muita modelagem: curso e turma são entidades diferentes. Python Essencial é um curso, com ementa, carga horária e preço. A turma 10 é uma das vezes em que esse curso é oferecido, com datas, vagas e um instrutor próprio. Se tratássemos curso e turma como uma coisa só, a ementa e o preço do curso teriam de ser repetidos em cada turma, e estaríamos de volta à redundância do capítulo anterior: no dia em que o preço mudasse, alguém esqueceria uma das cópias. A regra 3 já dá a pista: um curso pode estar no catálogo sem ter turma, e isso só é possível se o curso existir por conta própria.

Como reconhecer uma entidade? Um bom teste é perguntar se o negócio precisa guardar os mesmos dados sobre várias coisas do mesmo tipo. Cursos, turmas e pessoas passam no teste. A própria escola, não: ela é uma só, e o banco inteiro é sobre ela. Por isso, no nosso modelo, “a escola” não é uma entidade. Seria, se o banco atendesse a várias escolas; mais uma vez, quem decide é a regra de negócio.

Uma última combinação de vocabulário: entidade é um termo do nível conceitual. Quando o modelo virar tabelas, no [capítulo 3](#), as entidades vão dar origem a tabelas, mas entidade e tabela não são a mesma coisa, e o livro não usa uma palavra pela outra.

Uma entidade, sozinha, é só um nome num retângulo. O que a escola guarda sobre cada curso e cada pessoa é o assunto da próxima seção.

2.4 Atributo

Saber que a escola guarda dados sobre pessoas ainda não diz quais dados. É aí que entram os atributos, e escolher cada um é uma decisão de modelagem: o que fica de fora não pode ser consultado depois, e o que entra sem necessidade dá trabalho para manter.

Um **atributo** (*attribute*) é um dado que se guarda sobre cada ocorrência de uma entidade. Pela regra 5, a escola guarda o nome, o CPF, o e-mail, a data de nascimento e um ou mais telefones de cada pessoa. Cada um desses dados é um atributo de **Pessoa**.

Na [Figura 2.1](#) temos a entidade **Pessoa** com os seus atributos. Cada atributo é um pequeno círculo ligado ao retângulo por uma linha, com o nome ao lado. O círculo preenchido do CPF marca um atributo especial, o identificador, que é o assunto da próxima seção. E o (1,n) depois do telefone? Essa anotação vem da regra 5 e merece uma explicação.

Nem todo atributo tem exatamente um valor para cada ocorrência. A **cardinalidade de atributo** é a quantidade mínima e a máxima de valores que o atributo pode ter para cada ocorrência, e se escreve como um par entre parênteses: o primeiro número é o mínimo; o segundo, o máximo. Quando o par é (1,1), ou seja, um valor e só um, o par fica implícito e não aparece no diagrama. É o caso do nome, do CPF, do e-mail e da data de nascimento.

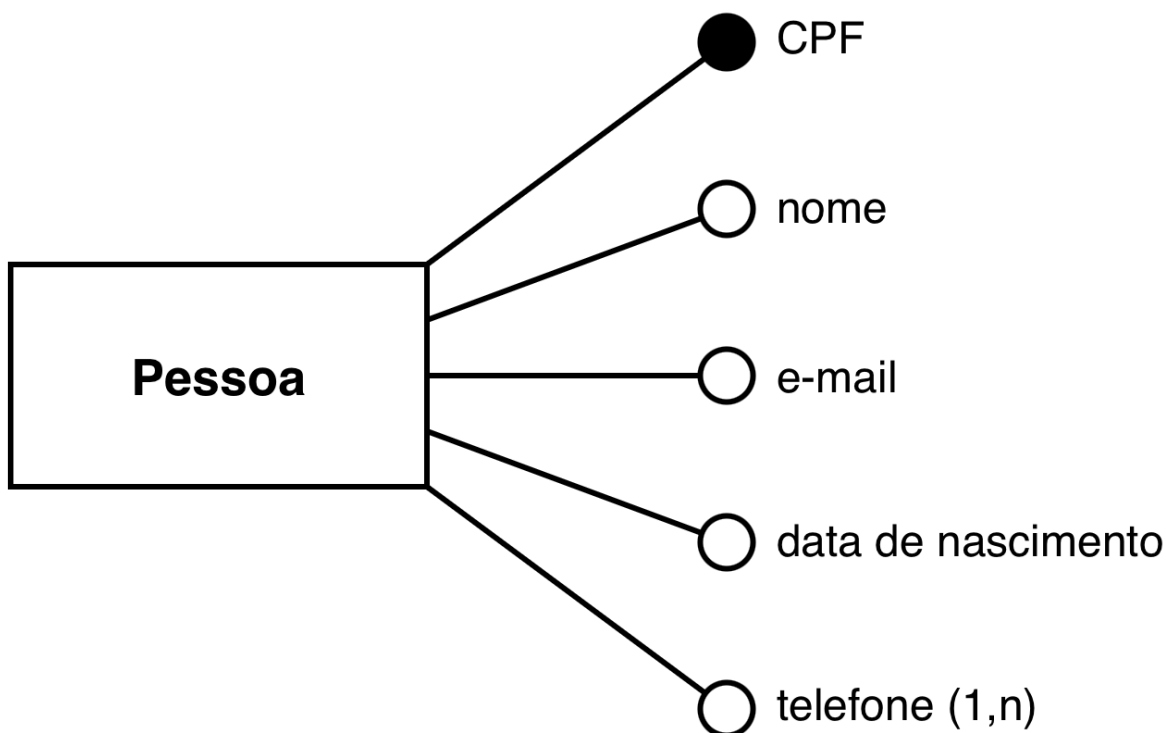


Figura 2.1: A entidade Pessoa e os seus atributos.

O telefone é diferente. A regra 5 diz que a escola guarda um ou mais telefones de cada pessoa: no mínimo um, sem limite máximo, e daí o (1,n). Um atributo que pode ter mais de um valor para a mesma ocorrência é um **atributo multivalorado** (*multivalued attribute*). Se a regra previsse um único telefone por pessoa, o atributo seria (1,1), como os outros, e o diagrama mudaria. É a regra de negócio decidindo o desenho.

O caso oposto é o do atributo que pode ficar sem valor. Pela regra 8, a nota final da matrícula só é lançada ao fim da turma: enquanto o curso acontece, a nota não existe. A cardinalidade dela é (0,1), ou seja, no mínimo nenhum valor e no máximo um. Um atributo cuja cardinalidade mínima é zero é um **atributo opcional**. A nota volta, com o (0,1), na [seção 2.8](#). Tornar um atributo opcional também é uma decisão de modelagem, com consequências que o [capítulo 3](#) mostra ao tratar do valor ausente.

Há ainda uma decisão escondida na lista da regra 5: a escola guarda a data de nascimento, e não a idade. Por quê? Porque a idade muda sozinha, todo ano, sem que ninguém altere nada no banco. Um atributo “idade” estaria certo no dia do cadastro e errado a partir do aniversário seguinte. A data de nascimento não muda, e a idade pode ser calculada a partir dela a qualquer momento. Onde fazer esse cálculo é assunto do [capítulo 6](#).

Entre os atributos de **Pessoa**, o CPF tem um papel diferente dos demais, e é por ele que começa a próxima seção.

2.5 Identificador

Com as entidades e os atributos definidos, surge uma pergunta prática: se a escola tem centenas de pessoas cadastradas, como saber de qual delas um dado está falando? Se não houver como distinguir uma pessoa das demais, não dá para corrigir o e-mail da pessoa certa nem ligar uma matrícula ao aluno certo. A resposta é o identificador.

Um **identificador** (*identifier*) é um atributo, ou um conjunto de atributos e relacionamentos, cujo valor distingue uma ocorrência da entidade de todas as outras. Os relacionamentos aparecem na próxima seção; por enquanto, fiquemos com os atributos. No DER, o atributo que identifica tem o círculo preenchido, como o CPF na [Figura 2.1](#).

Repare na diferença de papel. Todo atributo descreve a ocorrência; o identificador, além de descrever, distingue. O nome descreve a Ana, mas não a distingue: nada impede que a escola tenha duas pessoas chamadas Ana Lima. O CPF distingue, e quem garante isso é a regra 5: não há duas pessoas com o mesmo CPF.

É sempre uma regra de negócio que autoriza um identificador, porque só o negócio pode garantir que um valor não se repete. Os identificadores da escola são estes:

Entidade	Identificador	Regra
Curso	título	1
Turma	código	3
Pessoa	CPF	5
Certificado	código de verificação	9

O título só serve de identificador porque a regra 1 garante que dois cursos nunca têm o mesmo título. Se a escola um dia aceitar dois cursos com o mesmo título, digamos uma versão nova e uma antiga, o título deixa de identificar, e o modelo precisa de outro identificador.

Talvez você esteja se perguntando sobre a coluna `id` da tabela `curso`, que apareceu no capítulo anterior. Por que essa coluna não é o identificador? Porque ela não existe no negócio: na escola, ninguém fala em “curso 2”, e sim em Python Essencial. A coluna `id` pertence ao modelo lógico. Criar ou não uma coluna como essa é uma decisão que se toma ao transformar o modelo conceitual em tabelas, e o [capítulo 3](#) explica como tomá-la. No modelo conceitual, o identificador sai das regras. O código da turma é outra história: a escola usa os números 10 e 11 no dia a dia para falar das turmas, e foi por isso que a regra 3 incluiu o código. Como é um dado do negócio, o código já aparece no modelo conceitual.

Às vezes, um atributo sozinho não basta, e o identificador precisa juntar mais de um elemento. É o **identificador composto** (*composite identifier*), formado por mais de um atributo ou relacionamento. A escola tem um caso assim, o da matrícula, que vamos identificar na [seção 2.8](#). Por ora, vale a regra que evita o erro mais comum: cada parte de um identificador composto precisa ser indispensável. Se o CPF já distingue uma pessoa de todas as outras, juntar o nome a ele não forma um identificador composto; forma apenas um identificador com uma parte que não acrescenta nada. Para conferir, retire uma parte de cada vez: se o que restar ainda distinguir as ocorrências, a parte retirada era dispensável e sai do identificador. Um identificador que passa nesse teste é mínimo.

Até aqui, cada entidade está isolada no seu retângulo. Mas a escola não é um conjunto de coisas soltas: turmas são de cursos, alunos se matriculam em turmas. É hora de ligar os retângulos.

2.6 Relacionamento

As entidades da escola não vivem isoladas: toda turma é de um curso, todo certificado é de uma matrícula. Essas ligações também são dados, porque a escola precisa saber, por exemplo, de qual curso é a turma 10. No modelo conceitual, elas viram relacionamentos.

Um **relacionamento** (*relationship*) é uma associação entre ocorrências de entidades. No DER, ele é um losango com um nome, ligado por linhas às entidades que associa.

Na [Figura 2.2](#) temos dois relacionamentos. O primeiro, **Oferta**, liga **Curso** a **Turma** e registra a regra 3: cada turma é de um curso. Uma ocorrência de **Oferta** é um par formado por um curso e uma turma, como Python Essencial e a turma 10. Por enquanto, o diagrama não diz quantas turmas um curso pode ter nem quantos cursos uma turma pode ter. Isso fica para a próxima seção.

Dei aos relacionamentos nomes de substantivo, como **Oferta**. Nomes de verbo, como “oferece”, também são comuns. O que importa é que o nome diga, a quem lê o diagrama, que fato aquela ligação registra.

O segundo relacionamento da figura é diferente: as duas linhas do losango **Pré-requisito** chegam à mesma entidade. Ele registra a regra 2: um curso pode exigir outros cursos, e um mesmo curso pode ser exigido por vários. Um relacionamento que associa ocorrências da mesma entidade é um **autorrelacionamento** (*recursive relationship*). Nele, cada linha precisa de um rótulo que diga o **papel** (*role*) da ocorrência, ou seja, a função que ela cumpre no relacionamento. Na figura, um curso faz o papel de quem exige, e o outro, o de quem é exigido. Se a escola decidir que Python Essencial exige Lógica de Programação, Python Essencial ocupa o papel “exige”, e Lógica de Programação, o papel “é exigido”. Sem os papéis, não haveria como ler o diagrama: quem exige quem?

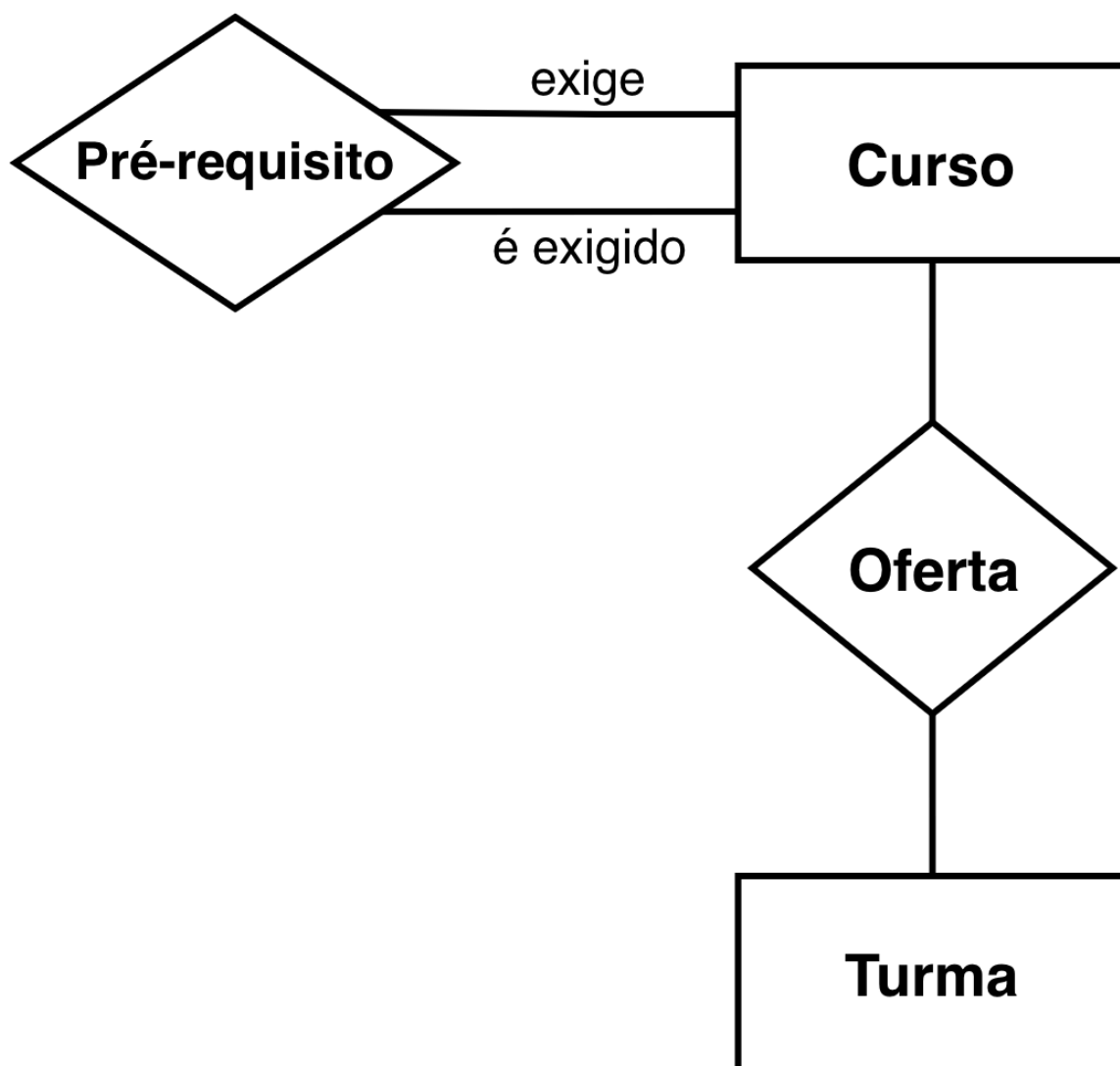


Figura 2.2: Dois relacionamentos da entidade Curso, um deles com a própria entidade.

Uma armadilha comum nesta etapa é guardar uma ligação como se fosse um atributo. Imagine que a turma ganhasse um atributo “nome do instrutor”. Parece prático, mas a escola guarda outros dados do instrutor, como o CPF, o e-mail e o valor da hora-aula, e esses dados teriam de ficar em algum lugar. Se ficassem na turma também, esses dados se repetiriam em cada turma do mesmo instrutor, e voltaríamos à redundância do capítulo anterior. Se ficassem numa entidade própria, o nome guardado na turma viraria uma cópia, que poderia discordar do original. A saída é a da regra 7: o instrutor é uma ocorrência de outra entidade, e a turma se liga a ele por um relacionamento, que chamei de **Docência**. A **Docência** entra no diagrama junto com a entidade dos instrutores, na [seção 2.9](#).

O losango diz que duas entidades se ligam, mas não diz quantas ocorrências de cada lado participam da ligação. Essa informação é a próxima peça.

2.7 Cardinalidade

Saber que turmas se ligam a cursos não basta para modelar. Uma turma pode ser de dois cursos? Um curso pode ficar sem turma? As respostas mudam o modelo e, como o [capítulo 3](#) vai mostrar, mudam também as tabelas. No DER, quem dá essas respostas é a cardinalidade.

A **cardinalidade** (*cardinality*) de uma entidade num relacionamento é a quantidade mínima e a máxima de ocorrências dessa entidade que podem estar associadas, por meio do relacionamento, a uma ocorrência da outra entidade. A cardinalidade é escrita como um par, (mínima,máxima), e os dois números têm papéis diferentes. A **cardinalidade máxima** (*maximum cardinality*) diz se uma ocorrência pode se associar a no máximo uma ocorrência do outro lado (máxima 1) ou a várias (máxima n, de “muitos”). A **cardinalidade mínima** (*minimum cardinality*) diz se a associação é obrigatória (mínima 1) ou opcional (mínima 0).

Na [Figura 2.3](#), a parte de cima mostra as ocorrências: cada curso e cada turma da escola, com uma linha para cada par de **Oferta**. A turma 10 se liga a Python Essencial, e a turma 11, a Modelagem de Dados. Lógica de Programação não se liga a turma nenhuma. A parte de baixo mostra o mesmo relacionamento no DER, agora com as cardinalidades.

Para ler a cardinalidade, parta de uma ocorrência de um lado e conte quantas ocorrências ela alcança do outro. O par fica junto à entidade contada. Partindo de uma turma, quantos cursos ela alcança? Exatamente um, pela regra 3: nem nenhum, nem dois. Por isso o (1,1) fica junto a **Curso**. Partindo de um curso, quantas turmas? Nenhuma ou várias: Lógica de Programação não tem nenhuma, e nada impede que Python Essencial ganhe uma segunda turma. Por isso o (0,n) fica junto a **Turma**. Em frases: toda turma é de um e só um curso; um curso pode ter nenhuma, uma ou várias turmas.

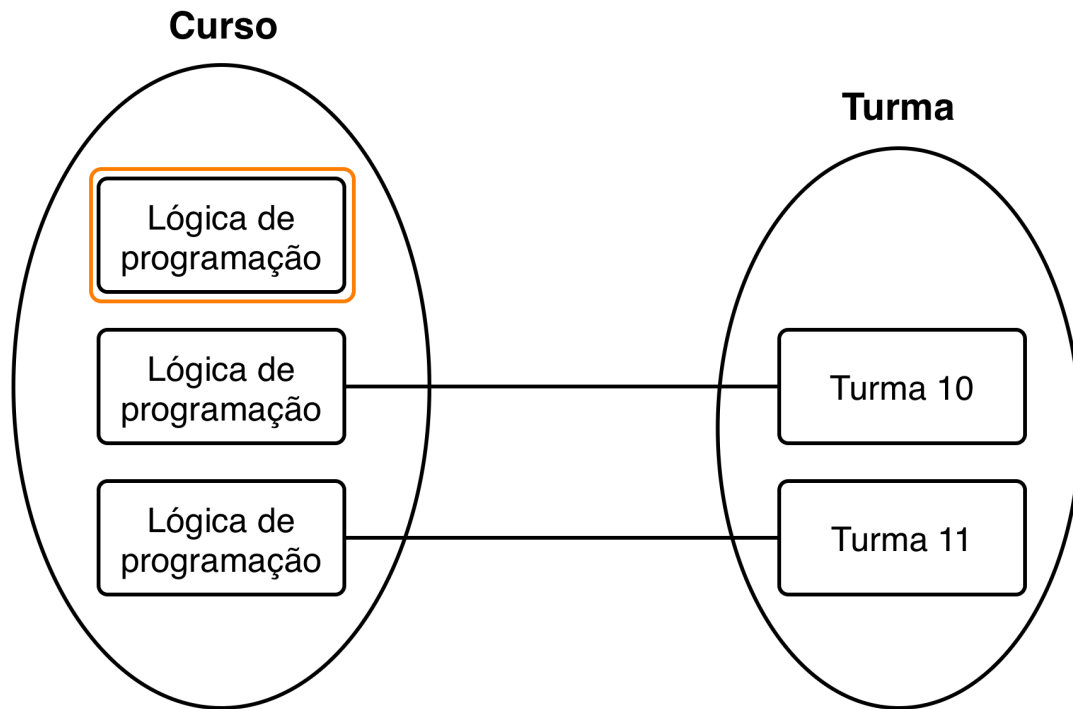
Repare que a parte de cima da figura mostra o que acontece hoje na escola, e o DER mostra o que as regras permitem. Hoje, nenhum curso tem duas turmas, mas a regra admite essa possibilidade, e é a regra que vai para o diagrama.

Atenção

A posição do par é uma convenção, e ler do lado errado inverte o sentido do diagrama: o (1,1) junto a **Curso** passaria a dizer que todo curso tem exatamente uma turma, o que contradiz a regra 3. Neste livro, o par fica sempre junto à entidade contada. Na dúvida, refaça a leitura em frases e confira cada frase com a regra.

A cardinalidade máxima dos dois lados dá nome ao relacionamento. Com máxima 1 de um lado e n do outro, como em **Oferta**, temos um **relacionamento 1:N** (*one-to-many*), que se lê “um para muitos”. Com máxima n dos dois lados, temos um **relacionamento N:N** (*many-to-many*), “muitos para muitos”. Com máxima 1 dos dois lados, temos um **relacionamento 1:1** (*one-to-one*), “um para um”. Essa classificação fala só da máxima: dizer que um relacionamento é 1:N não diz nada sobre obrigatoriedade, que é assunto da mínima e se lê à parte.

Ocorrências



DER

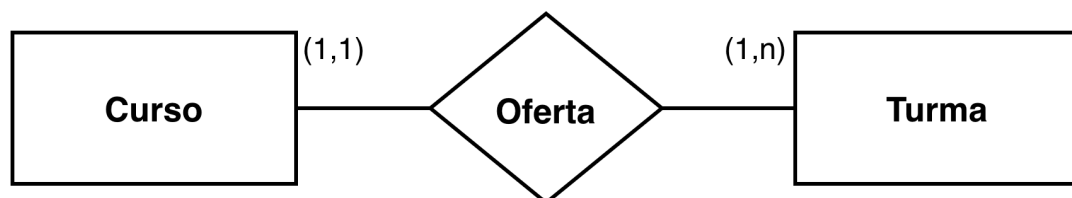


Figura 2.3: As ocorrências de Curso e de Turma e a cardinalidade do relacionamento Oferta.

Cada caso tem a sua regra na escola. Pela regra 7, a **Docência** também é 1:N: toda turma tem exatamente um instrutor, e um instrutor pode ter várias turmas ou nenhuma. O $(1,1)$ fica junto a **Instrutor**, e o $(0,n)$, junto a **Turma**, como na **Oferta**.

Pela regra 8, a matrícula é N:N: um aluno pode se matricular em várias turmas, e uma turma recebe vários alunos. A mínima é 0 dos dois lados. Do lado do aluno, pode haver aluno ainda sem matrícula, porque a escola cadastra a pessoa antes de fechar a matrícula. Do lado da turma, uma turma recém-aberta ainda não tem nenhum aluno. Pela regra 2, o **Pré-requisito** também é N:N, com $(0,n)$ dos dois lados: um curso pode exigir vários cursos e ser exigido por vários, mas também pode não exigir nenhum nem ser exigido por nenhum.

Falta o 1:1, que vem da regra 9: cada matrícula gera no máximo um certificado, e todo certificado é de exatamente uma matrícula. Partindo de uma matrícula, o certificado pode não existir, porque a matrícula sem nota suficiente não gera nenhum; por isso, junto ao certificado, fica (0,1). Partindo de um certificado, a matrícula sempre existe, porque não há certificado sem a matrícula que lhe deu origem; por isso, junto à matrícula, fica (1,1).

Mas aqui o modelo esbarra num problema. Até agora, a matrícula é um relacionamento, um losango entre aluno e turma. Como ligar um certificado a um losango? A resposta está na próxima seção.

2.8 Relacionamento com atributos e entidade associativa

Até aqui, todo dado do modelo pertencia a uma entidade: o título é do curso, o CPF é da pessoa. A regra 8, porém, traz dados que não são de nenhuma entidade, e o modelo precisa de um lugar para eles. Esta seção mostra que lugar é esse e o que fazer quando a ligação entre duas entidades precisa, ela mesma, se ligar a uma terceira.

Pela regra 8, a escola guarda a data, o valor cobrado e a nota final de cada matrícula. De quem é a nota final? Da Ana? Não exatamente: ela está matriculada em duas turmas, a 10 e a 11, e pode ter uma nota em cada uma. Da turma 11? Também não: a Carla está na mesma turma e tem a própria nota. A nota pertence ao par formado pela Ana e pela turma 11, ou seja, à matrícula. Um atributo que descreve a ligação entre ocorrências, e não uma das ocorrências, é um **atributo de relacionamento** (*relationship attribute*). No DER, ele fica ligado ao losango, e não aos retângulos.

O atributo de relacionamento dá lugar à data, ao valor cobrado e à nota, mas não resolve o caso do certificado. Pela regra 9, cada certificado é de uma matrícula, e o MER não permite ligar uma entidade a um relacionamento: as linhas que saem de um losango só chegam a retângulos. A saída é tratar a própria matrícula como uma entidade. Uma **entidade associativa** (*associative entity*) é um relacionamento redefinido para ser tratado como entidade, de modo que possa participar de outros relacionamentos. No DER, ela é o losango do relacionamento com um retângulo desenhado em volta.

Por enquanto, trate **Aluno** como uma entidade qualquer. Aluno e instrutor são pessoas, e a ligação entre eles e **Pessoa** é o assunto da próxima seção.

Na [Figura 2.4](#) temos a matrícula como entidade associativa. O losango continua ligando **Aluno** a **Turma**, com as cardinalidades que vimos na seção anterior, e os três atributos da regra 8 estão ligados ao losango. A nota final traz o (0,1), porque só é lançada ao fim da turma. O retângulo em volta do losango faz da matrícula uma entidade, e é por isso que ela pode se ligar a **Certificado** pelo relacionamento **Emissão**, com o 1:1 da seção anterior: (1,1) junto à matrícula e (0,1) junto ao certificado.

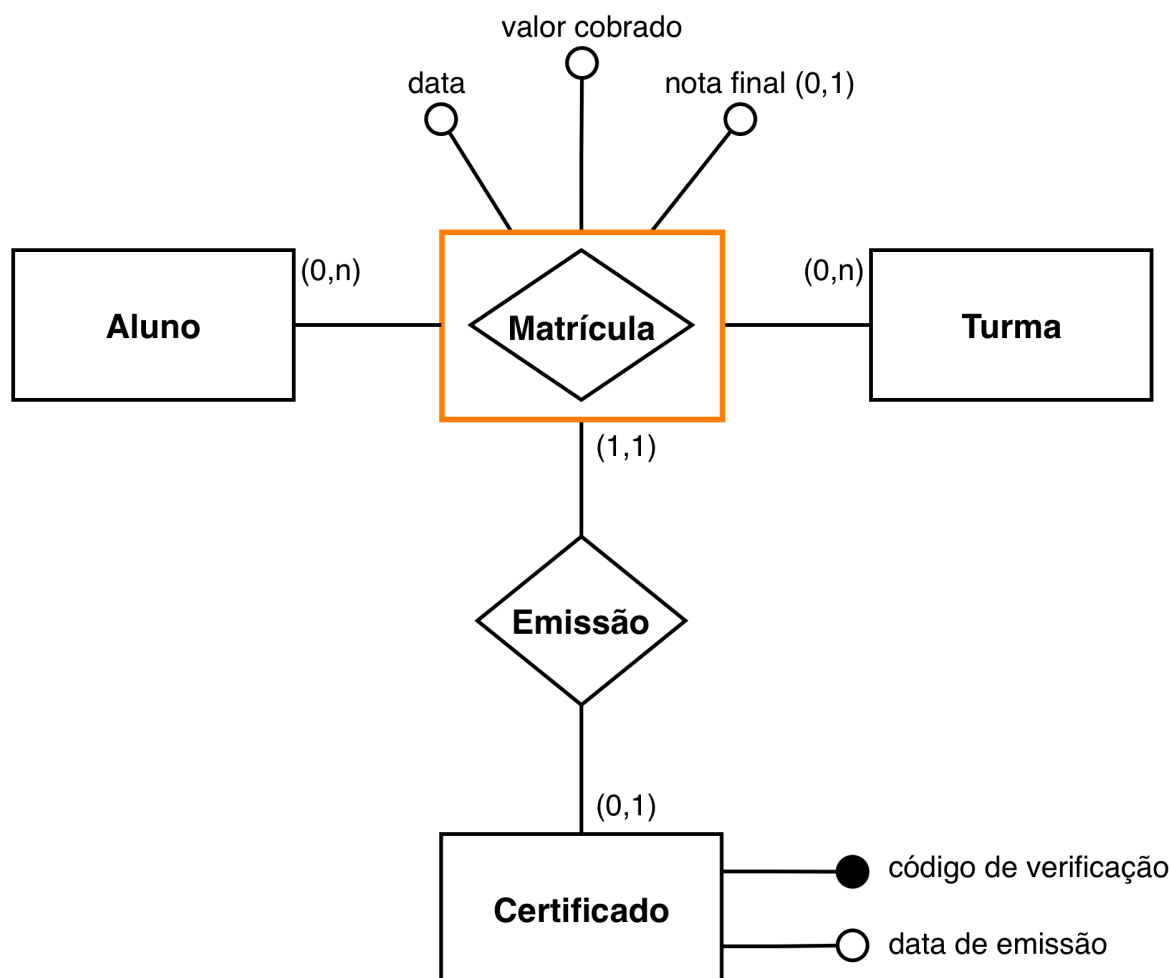


Figura 2.4: A matrícula como entidade associativa, com os seus atributos e a ligação com o certificado.

Como entidade, a matrícula também precisa de um identificador. Que dado distingue uma matrícula de todas as outras? A data, não: vários alunos se matriculam no mesmo dia. A resposta está na regra 8: ninguém se matricula duas vezes na mesma turma. Logo, o aluno e a turma, juntos, distinguem uma matrícula de todas as outras. Esse é o identificador composto que a [seção 2.5](#) deixou para cá, formado não por atributos, e sim pelas ligações com **Aluno** e com **Turma**. E ele é mínimo: só o aluno não basta, porque a Ana tem duas matrículas; só a turma também não, porque a turma 11 tem a Ana e a Carla.

Um último cuidado, que o [capítulo 4](#) retoma: o valor cobrado pode parecer uma cópia do preço do curso, mas não é. O preço da regra 1 é o atual e muda sempre que a escola reajusta os preços do catálogo. O valor cobrado é o preço do dia da matrícula e não muda depois: ninguém refaz a cobrança da Ana porque o curso ficou mais caro. São dois fatos diferentes, e cada um tem o seu lugar no modelo.

A essa altura, você já deve ter notado que alunos e instrutores aparecem no modelo sem que tenhamos dito de onde eles vêm. É hora de resolver essa pendência.

2.9 Generalização e especialização

Alunos e instrutores têm muito em comum: todos têm nome, CPF, e-mail, data de nascimento e telefones. Mas cada grupo tem também dados que o outro não tem, e só os instrutores dão aula. Modelar os dois grupos como entidades separadas repetiria os dados comuns; modelar os dois como uma entidade só misturaria dados que não valem para todos. A generalização e a especialização resolvem esse impasse.

O par **generalização e especialização** (*generalization and specialization*) é o recurso do MER para representar uma entidade genérica, com o que todas as ocorrências têm em comum, e entidades especializadas, com o que só uma parte delas tem. Os dois nomes descrevem a mesma estrutura, vista de lados opostos: do geral para o particular, especializamos; do particular para o geral, generalizamos. Cada ocorrência de uma entidade especializada é também uma ocorrência da genérica e tem, além dos próprios atributos e relacionamentos, os atributos e relacionamentos da genérica. Essa propriedade se chama **herança** (*inheritance*).

Na [Figura 2.5](#), **Pessoa** é a entidade genérica, com os atributos da regra 5, e **Aluno** e **Instrutor** são as especializadas, ligadas a ela pelo triângulo. Cada especializada guarda só o que é dela, pela regra 6: o aluno, a escolaridade; o instrutor, a data de admissão e o valor da hora-aula. O nome e o CPF de um instrutor não aparecem em **Instrutor**, porque ele os herda de **Pessoa**. Pelo mesmo motivo, nenhuma das especializadas tem identificador próprio na figura: cada aluno e cada instrutor é uma pessoa, e o CPF que distingue a pessoa distingue também o aluno e o instrutor.

A figura mostra ainda o outro motivo para especializar, além dos atributos: só **Instrutor** participa da **Docência**, o relacionamento da regra 7 que a [seção 2.6](#) deixou pendente. Se a **Docência** ligasse a turma a **Pessoa**, o diagrama permitiria que qualquer pessoa desse aula, inclusive um aluno que nunca foi instrutor. Ligada a **Instrutor**, ela só admite quem é instrutor. Da mesma forma, a matrícula da seção anterior liga a turma a **Aluno**, e não a qualquer pessoa.

Toda especialização recebe duas classificações, independentes entre si. A primeira diz se toda ocorrência da genérica precisa estar em alguma especializada. Na **especialização total** (*total specialization*), precisa; na **especialização parcial** (*partial specialization*), pode haver ocorrência da genérica que não está em nenhuma especializada. A segunda diz em quantas especializadas uma mesma ocorrência pode estar. Na **especialização exclusiva** (*disjoint specialization*), em uma só; na **especialização compartilhada** (*overlapping specialization*), em mais de uma ao mesmo tempo.

Pela regra 6, toda pessoa cadastrada é aluna, instrutora ou as duas coisas. O “toda pessoa é aluna ou instrutora” torna a especialização de **Pessoa** total: não existe pessoa cadastrada que não seja nem uma coisa nem outra. O “ou as duas coisas” a torna compartilhada. Se a Marta, que é instrutora da turma de Python Essencial, se matricular um dia numa turma de Modelagem de Dados, ela passa a ser instrutora e aluna ao mesmo tempo, sem deixar de ser uma pessoa só, com um CPF só. Na figura, as letras junto ao triângulo registram as duas classificações: c, de compartilhada, e t, de total.

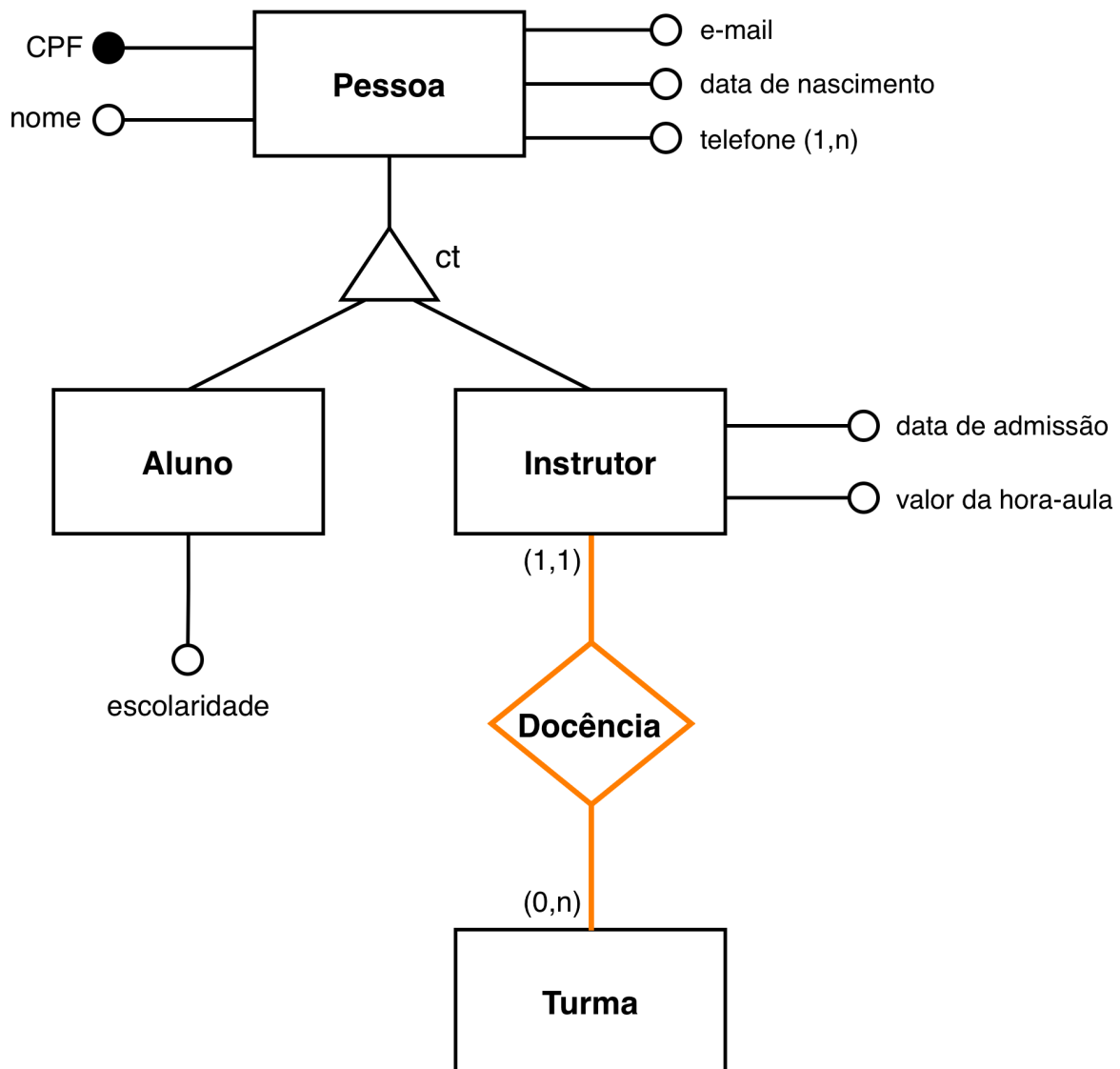


Figura 2.5: A especialização de Pessoa em Aluno e Instrutor, e o relacionamento que só o instrutor tem.

Atenção

“Total” não quer dizer que cada pessoa está em todas as especializadas. Quer dizer que cada pessoa está em pelo menos uma. Quem diz se ela pode estar em mais de uma é a outra classificação, a de exclusiva ou compartilhada. Confundir as duas classificações leva a desenhar uma regra que o negócio não tem.

A regra 4 traz outra combinação. Toda turma é presencial ou on-line, nunca as duas: o “toda turma” torna a especialização total, e o “nunca as duas” a torna exclusiva.

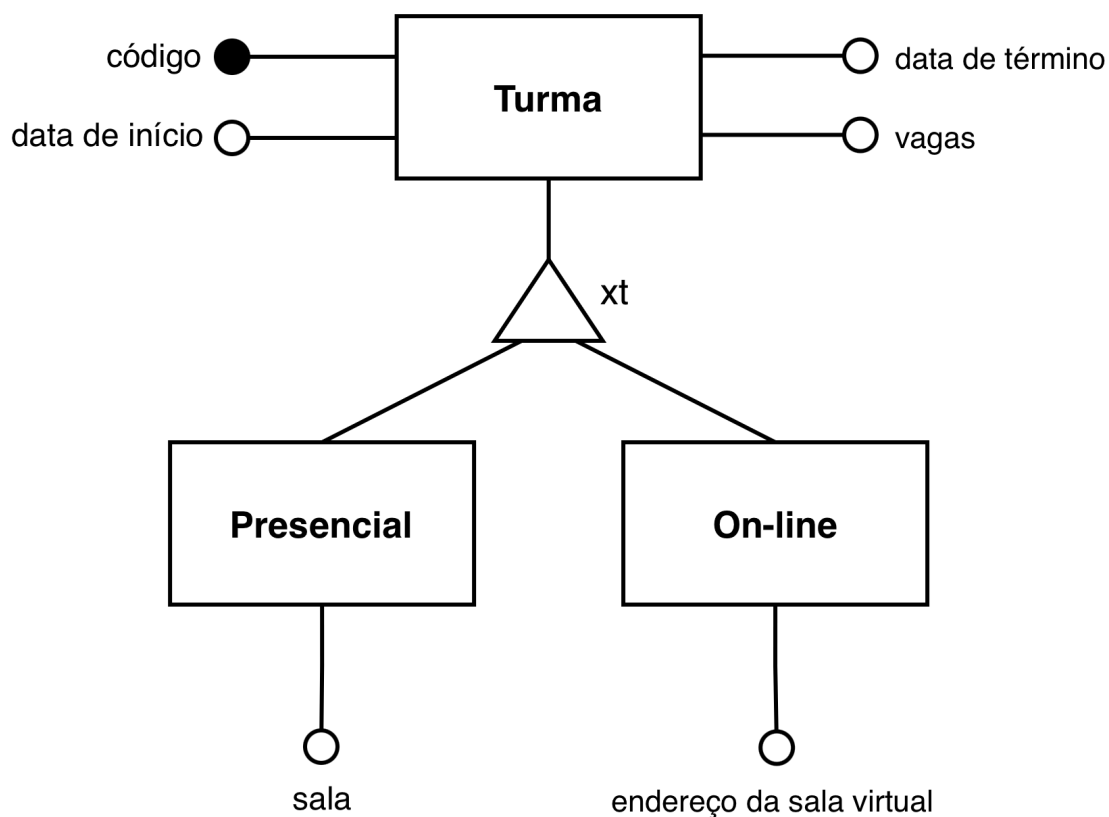


Figura 2.6: A especialização de Turma em Presencial e On-line.

Na [Figura 2.6](#), as letras junto ao triângulo são x, de exclusiva, e t, de total. A turma 11, que é presencial, é uma ocorrência de **Turma** e de **Presencial**, e tem sala. A turma 10, que é on-line, é uma ocorrência de **Turma** e de **On-line**, e tem o endereço da sala virtual. Nenhuma das duas tem os dois dados, e é exatamente isso que a especialização registra: a sala não é um atributo de **Turma** que fica em branco nas turmas on-line, e sim um atributo que só existe nas presenciais.

Nenhuma regra da escola produz uma especialização parcial, mas basta mudar uma regra para ela aparecer. E se a escola passasse a cadastrar também interessados, pessoas que pediram informações e ainda não são alunas nem instrutoras? A regra 6 deixaria de valer como está escrita, e a especialização de **Pessoa** passaria a ser parcial: haveria pessoas fora de **Aluno** e de **Instrutor**. No diagrama, mudaria uma letra; no banco, passaria a ser aceita uma pessoa sem papel nenhum. Mais uma vez, a regra decide o desenho.

Agora temos todas as peças. Falta juntá-las.

2.10 O DER do estudo de caso

Ao longo do capítulo, desenhamos a escola aos pedaços, uma figura para cada conceito. Agora vamos juntar os pedaços num diagrama só, percorrendo as regras na ordem. O caminho serve de roteiro para qualquer modelagem: para cada regra, pergunte o que ela acrescenta ao diagrama.

- Pela regra 1, os cursos viram a entidade **Curso**, com título, ementa, carga horária e preço. O título é o identificador, porque não se repete.
- Pela regra 2, o pré-requisito vira um autorrelacionamento N:N de **Curso**, com os papéis “exige” e “é exigido”: um curso pode exigir vários cursos e ser exigido por vários, mas também pode não exigir nenhum nem ser exigido por nenhum.
- Pela regra 3, as turmas viram a entidade **Turma**, identificada pelo código e com datas e vagas. Ela se liga a **Curso** pela **Oferta**, 1:N: toda turma tem um curso, e um curso pode não ter turma.
- Pela regra 4, **Turma** se especializa em **Presencial**, com a sala, e **On-line**, com o endereço da sala virtual, de forma total e exclusiva.
- Pela regra 5, as pessoas viram a entidade **Pessoa**, identificada pelo CPF, com nome, e-mail, data de nascimento e o telefone multivalorado.
- Pela regra 6, **Pessoa** se especializa em **Aluno**, com a escolaridade, e **Instrutor**, com a data de admissão e o valor da hora-aula, de forma total e compartilhada.
- Pela regra 7, **Instrutor** se liga a **Turma** pela **Docência**, 1:N: toda turma tem um instrutor, e um instrutor pode ter várias turmas ou nenhuma.
- Pela regra 8, **Aluno** se liga a **Turma** pela **Matrícula**, N:N, com data, valor cobrado e nota final opcional: pode haver aluno ainda sem matrícula e turma ainda sem aluno. A matrícula é identificada pelo aluno e pela turma.
- Pela regra 9, os certificados viram a entidade **Certificado**, identificada pelo código de verificação e com a data de emissão. Para que o certificado se ligue à matrícula pela **Emissão**, a matrícula vira entidade associativa. A **Emissão** é 1:1: todo certificado é de uma matrícula, e uma matrícula pode não ter certificado.

As regras 10 e 11 não acrescentam nada ao diagrama: elas tratam do que acontece quando os dados são excluídos ou mudam.

O resultado está em duas figuras, para caber numa tela estreita. Elas mostram as entidades, os relacionamentos e as cardinalidades, mas não os atributos, que já apareceram nas figuras anteriores e estão reunidos na tabela do fim desta seção.

Na [Figura 2.7](#) temos a parte da escola que gira em torno dos cursos: o catálogo com os pré-requisitos, as turmas com as duas modalidades e o instrutor de cada turma. Na [Figura 2.8](#) temos a parte que gira em torno das pessoas: os dois papéis, a matrícula e o certificado. As duas figuras se encontram em **Turma** e em **Instrutor**, que aparecem em ambas. Em cada caso, é a mesma entidade desenhada duas vezes, e não duas entidades com o mesmo nome: o diagrama completo da escola é a junção das duas figuras por esses dois pontos.

Os atributos de cada entidade são estes:

Entidade	Identificador	Demais atributos
Curso	título	ementa, carga horária, preço
Turma	código	data de início, data de término, vagas
Presencial	o de Turma	sala
On-line	o de Turma	endereço da sala virtual

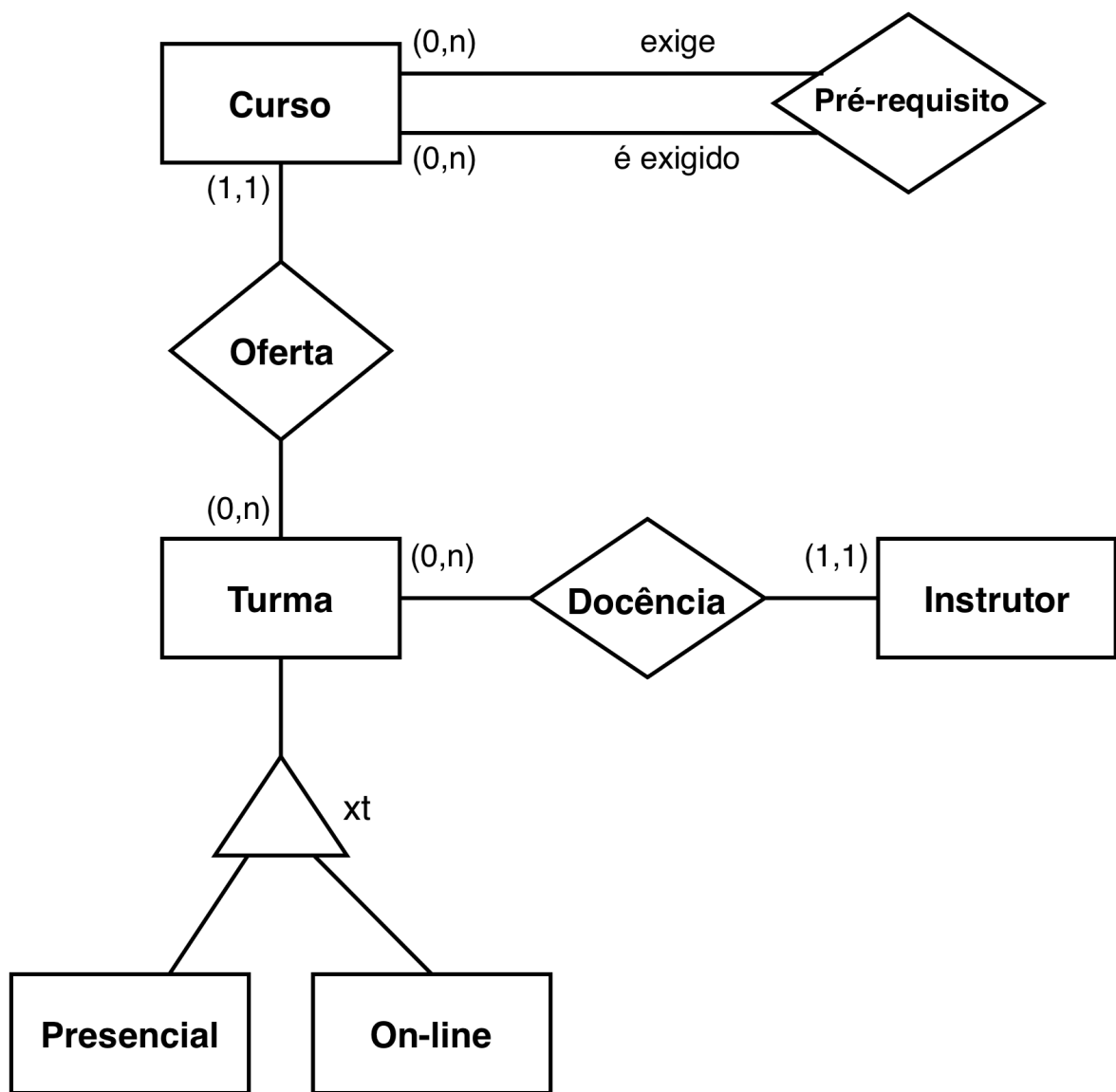


Figura 2.7: O DER da escola, parte 1: cursos, turmas e instrutores.

Entidade	Identificador	Demais atributos
Pessoa	CPF	nome, e-mail, data de nascimento, telefone (1,n)
Aluno	o de Pessoa	escolaridade
Instrutor	o de Pessoa	data de admissão, valor da hora-aula
Matrícula	aluno e turma	data, valor cobrado, nota final (0,1)
Certificado	código de verificação	data de emissão

Pronto: a escola está no papel. Mas como saber se o diagrama está certo?

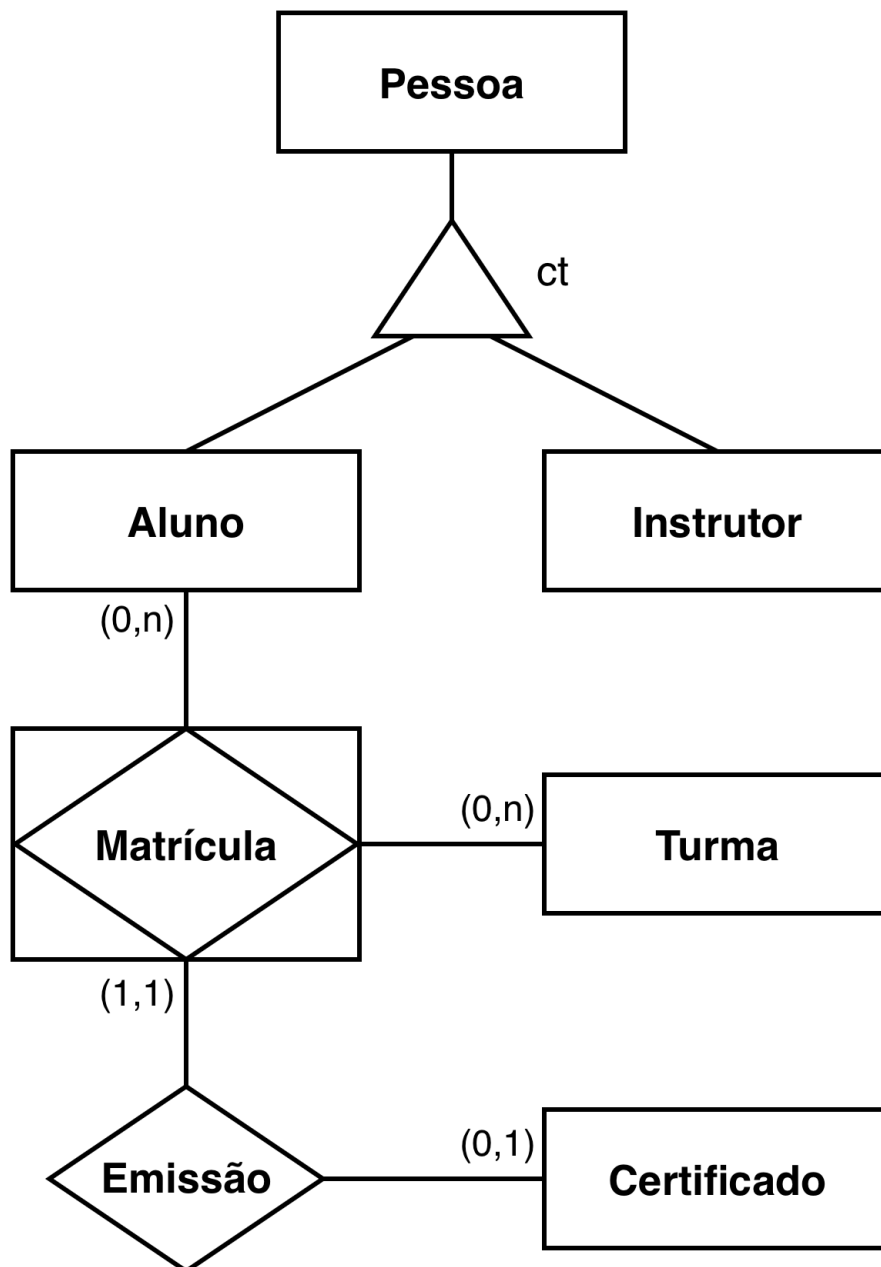


Figura 2.8: O DER da escola, parte 2: pessoas, matrículas e certificados.

2.11 Como avaliar um DER

Um DER pode estar impecável na notação e errado no conteúdo: todos os símbolos no lugar, e uma regra de negócio desenhada ao contrário. Por isso, um diagrama não se julga pela aparência, e sim pelas regras que ele atende. Esta seção mostra como fazer essa conferência e o que fazer com as regras que o diagrama não consegue mostrar.

O método mais direto é ler o diagrama de volta em português, relacionamento por relacionamento, e comparar cada frase com a regra que a originou. Na **Oferta**, a leitura dá: “toda turma é de um e só um curso; um curso pode ter nenhuma, uma ou várias turmas”. Bate com a regra 3. Na **Emissão**: “todo certificado é de uma e só uma matrícula; uma matrícula pode ter um certificado ou nenhum”. Bate com a regra 9. Se uma frase não bater, o erro pode estar no diagrama ou na regra, e só quem conhece o negócio sabe dizer em qual dos dois.

Essa leitura tem uma vantagem: ela pode ser feita com quem não sabe nada de modelagem. A secretária da escola não precisa saber o que é um losango para confirmar se um curso pode ficar sem turma. Mostre o diagrama, leia as frases em voz alta e pergunte se é assim que a escola funciona. É uma conferência barata: corrigir um losango agora custa uma borracha; corrigir um banco em uso significa alterar tabelas que já têm dados e programas que dependem delas.

A leitura também revela que algumas regras não aparecem em lugar nenhum do diagrama. Não é erro: o DER tem um vocabulário limitado, e parte das regras de negócio não cabe nele. Na escola, ficam de fora:

- o limite de vagas da turma (regra 8): o (0,n) diz que uma turma recebe vários alunos, mas não até quantos;
- a nota de 0 a 10 (regra 8): o diagrama diz que a nota existe, e não quais valores ela aceita;
- a nota mínima 7 para o certificado (regra 9): o (0,1) diz que a matrícula pode ter um certificado, mas não em que condição;
- o valor cobrado igual ao preço do curso no dia da matrícula (regra 8): o diagrama mostra os dois atributos, mas não a relação entre eles;
- a transferência só para outra turma do mesmo curso (regra 11): o diagrama mostra as matrículas, mas não as condições para trocar uma pela outra.

Essas regras não se perdem: elas ficam escritas ao lado do diagrama, como parte do modelo. O [capítulo 5](#) mostra quais delas o banco consegue cobrar e de que forma.

Por fim, não existe um único DER correto para um conjunto de regras. Pense no telefone. Neste capítulo, ele é um atributo multivalorado de **Pessoa**, mas daria para modelá-lo também como uma entidade **Telefone**, ligada a **Pessoa** por um relacionamento. Os dois diagramas atendem à regra 5. Qual é o melhor? Depende das regras, de novo. A escola guarda apenas o número de cada telefone e não precisa ligar um telefone a mais nada, então o atributo diz tudo com menos símbolos. Se cada telefone tivesse dados próprios, ou se um mesmo número pudesse pertencer a mais de uma pessoa, como o telefone fixo de uma família, a entidade passaria a fazer mais sentido. Quando dois diagramas atendem às mesmas regras, os dois estão certos. Quando um deles não atende, nenhum símbolo bem desenhado o salva.

2.12 Resumo

Começamos pelas regras de negócio, as afirmações sobre o funcionamento da escola das quais sai cada decisão do modelo, e vimos que uma regra só serve à modelagem se disser as quantidades envolvidas e não admitir duas leituras. Conhecemos o MER, a forma de descrever um negócio por entidades, atributos e relacionamentos, e o DER, o desenho que representa o MER, na notação de Chen com as convenções de Heuser.

Vimos que uma entidade é um conjunto de coisas sobre as quais a escola guarda dados, que cada uma dessas coisas é uma ocorrência e que curso e turma são entidades diferentes. Vimos que os atributos são os dados guardados sobre cada ocorrência e que a cardinalidade de atributo diz quando um atributo é opcional, como a nota final, ou multivalorado, como o telefone. Vimos também que o identificador distingue uma ocorrência das demais, que ele é sempre autorizado por uma regra e que cada parte dele precisa ser indispensável.

Ligamos as entidades por relacionamentos, inclusive de uma entidade com ela mesma, como no pré-requisito, e aprendemos a ler a cardinalidade de cada um: a máxima classifica o relacionamento em 1:1, 1:N ou N:N; a mínima diz se a participação é obrigatória ou opcional. Vimos que dados como a nota final pertencem ao relacionamento, e não às entidades, e que a matrícula virou uma entidade associativa para poder se ligar ao certificado. Com a generalização e a especialização, separamos o que alunos e instrutores têm em comum do que é próprio de cada grupo, e classificamos cada especialização como total ou parcial e como exclusiva ou compartilhada.

Por fim, montamos o DER da escola regra a regra e vimos como avaliá-lo: lendo o diagrama de volta em frases, conferindo essas frases com quem conhece o negócio e registrando por escrito as regras que o diagrama não consegue expressar. Vimos também que dois diagramas diferentes podem estar certos, desde que atendam às mesmas regras.

No próximo capítulo, o diagrama vira tabelas.

3 Modelo relacional: do diagrama às tabelas

“Mostre-me os seus fluxogramas e esconda as suas tabelas, e eu continuarei intrigado. Mostre-me as suas tabelas, e em geral não vou precisar dos seus fluxogramas: eles serão óbvios.”

Frederick P. Brooks Jr. (tradução livre)

No capítulo anterior, desenhamos a escola num diagrama que quem trabalha nela consegue ler, mesmo sem saber o que é uma tabela. Só que o SGBD não lê losangos: o que ele entende são tabelas, colunas e chaves. Neste capítulo, vamos converter o DER no modelo lógico, o segundo dos três níveis de modelo.

A boa notícia é que essa conversão não depende de inspiração. Cada elemento do DER tem uma regra de conversão, e sobram poucas escolhas, que dependem, mais uma vez, das regras de negócio. Antes de aplicar essas regras, porém, precisamos de três peças. A primeira são os tipos de chave, que o capítulo 1 só apresentou de passagem. A segunda é o que o banco passa a cobrar quando as chaves são declaradas. A terceira é a forma de registrar, numa tabela, que um dado está faltando.

O que será visto neste capítulo:

- [Tabela, linha e coluna](#)
- [Chaves: candidata, primária e estrangeira](#)
- [Integridade de entidade e integridade referencial](#)
- [NULL: o valor ausente](#)
- [Do DER às tabelas: entidades, atributos e identificadores](#)
- [Do DER às tabelas: relacionamentos](#)
- [Do DER às tabelas: generalização e especialização](#)
- [O estudo de caso em tabelas](#)
- [Resumo](#)

3.1 Tabela, linha e coluna

No capítulo 1, a tabela curso apareceu com três linhas e quatro colunas. Agora que temos o DER, a pergunta muda: o que, no diagrama, vira tabela, o que vira linha e o que vira coluna? Responder a ela é o primeiro passo da conversão e evita misturar o vocabulário dos dois níveis.

No caso mais simples, o de uma entidade cujos atributos têm um valor só, a correspondência é direta:

Modelo conceitual	Modelo relacional	Exemplo na escola
entidade	tabela	Curso vira a tabela <code>curso</code>
ocorrência	linha	Python Essencial vira a linha de <code>id 2</code>
atributo	coluna	a carga horária vira a coluna <code>carga_horaria</code>
identificador	chave	o título vira uma chave da tabela <code>curso</code>

A correspondência, porém, não é exata. Um atributo multivalorado, como o telefone, não cabe numa coluna. Um relacionamento às vezes vira uma coluna, e às vezes vira uma tabela. Uma especialização admite mais de uma solução. Mais adiante, as seções sobre a conversão tratam de cada um desses casos.

Talvez você já tenha encontrado, em outros livros, a palavra “relação” no lugar de “tabela”. Na teoria do modelo relacional, a tabela se chama **relação** (*relation*), termo que Codd tomou da matemática em 1970. Daí vem o nome do modelo: relacional quer dizer “feito de relações”, ou seja, de tabelas. O nome não tem nada a ver com o relacionamento do DER, embora a semelhança das palavras engane muita gente. Um banco cujas tabelas não se ligassem entre si continuaria sendo relacional.

No artigo em que propôs o modelo, Codd aponta duas propriedades que interessam a quem modela. A primeira é que a ordem das linhas não tem significado. A tabela `curso` não “começa” por Lógica de Programação: se você precisar dos cursos em ordem alfabética, a ordem é pedida na consulta, como mostra o [capítulo 6](#). Por isso, nenhuma informação pode depender da posição de uma linha. Se a escola precisar saber qual matrícula veio primeiro, essa informação vai numa coluna, como a data da matrícula.

A segunda propriedade é que não há duas linhas iguais. Se duas linhas tivessem os mesmos valores em todas as colunas, não haveria como apontar para uma delas sem apontar também para a outra, nem como corrigir uma sem corrigir a outra.

Um lembrete de vocabulário, antes de seguir: entidade, ocorrência e atributo continuam sendo termos do DER, e tabela, linha e coluna são termos do modelo relacional. A tabela `curso` representa a entidade **Curso**, mas não é uma entidade. Quem mistura os dois vocabulários acaba desenhando o DER já pensando em tabelas, o que é justamente o erro que o [capítulo 2](#) ensinou a evitar.

O que impede que duas linhas sejam iguais é a chave, e é por ela que seguimos.

3.2 Chaves: candidata, primária e estrangeira

Como o banco sabe que a linha de `id 2` da tabela `curso` é Python Essencial, e só ela? E como ele sabe de qual curso é a turma 10? As duas respostas passam pelas chaves, que o capítulo 1 apresentou como as colunas que identificam as linhas ou apontam para linhas de outra tabela. Agora precisamos separar os tipos de chave, porque cada um resolve um problema diferente.

Uma **chave candidata** (*candidate key*) é uma coluna, ou um conjunto mínimo de colunas, cujo valor não se repete entre as linhas da tabela. Mínimo quer dizer que nenhuma coluna pode ser retirada do conjunto sem que o valor do que sobra passe a poder se repetir. A definição é a de Codd, num artigo de 1971, e a exigência de ser mínimo é o teste que a [seção 2.5](#) aplicou ao identificador: cada parte precisa ser indispensável. Não por acaso, a chave candidata é o identificador do DER levado para a tabela. Na tabela **pessoa**, o **cpf** é uma chave candidata, e quem a autoriza é a regra 5: não há duas pessoas com o mesmo CPF.

Uma tabela pode ter mais de uma chave candidata. Na tabela **curso**, o **título** não se repete, pela regra 1, e o **id** também não. Das candidatas, uma é escolhida para identificar as linhas da tabela e servir de alvo às referências que vêm de outras tabelas: é a **chave primária** (*primary key*). As candidatas que não foram escolhidas são as **chaves alternativas** (*alternate keys*), e elas continuam sem poder se repetir. Não ser a chave primária não autoriza o título a aparecer duas vezes.

Quando a chave é formada por mais de uma coluna, ela é uma **chave composta** (*composite key*). A matrícula é o caso da escola. Pela regra 8, ninguém se matricula duas vezes na mesma turma, e o identificador composto da [seção 2.8](#) vira o par de colunas **aluno_id** e **turma_id** da tabela **matricula**. A Ana aparece em duas linhas dessa tabela, uma para cada turma, e a turma 11 também aparece em duas, uma para a Ana e outra para a Carla. Já o par formado pela Ana e pela turma 11 aparece uma vez só.

A outra metade do trabalho das chaves é ligar as tabelas. Uma **chave estrangeira** (*foreign key*) é uma coluna, ou um conjunto de colunas, cujos valores apontam para linhas de outra tabela, ou da própria tabela, por meio da chave primária ou de outra chave candidata da tabela apontada. Na tabela **turma**, a coluna **curso_id** é uma chave estrangeira: o valor 2 guardado na linha da turma 10 aponta para a linha de **id** 2 da tabela **curso**. A tabela que guarda a chave estrangeira é a tabela que referencia; a outra é a tabela referenciada.

Falta a escolha que o capítulo 1 deixou pendente. Se o título não se repete, por que a tabela **curso** tem uma coluna **id**? Para responder, precisamos de mais dois termos. Uma **chave natural** (*natural key*) é formada por dados que existem no negócio, como o título do curso ou o CPF da pessoa. Uma **chave artificial** (*surrogate key*) é uma coluna criada só para identificar as linhas, com valores que não significam nada para o negócio, como o **id** da tabela **curso**, cujo valor o próprio banco gera.

O argumento a favor da chave artificial é a estabilidade. Os valores do negócio mudam: nada nas regras impede a escola de renomear o curso Python Essencial, e um CPF digitado errado no cadastro precisa ser corrigido. Se o valor que muda é o da chave primária, ele precisa mudar também em todas as chaves estrangeiras que apontam para ele, em todas as tabelas. Codd já apontava esse problema em 1979: as chaves controladas pelos usuários estão sujeitas a mudar, justamente porque são os usuários que decidem os valores delas. Um **id** sem significado não tem motivo para mudar. Com ele, corrigir o CPF da Ana altera uma coluna de uma linha, e nada mais.

O argumento a favor da chave natural é que ela já existe e já tem significado. A chave artificial acrescenta uma coluna e, sozinha, só impede a repetição do próprio `id`. É aí que mora o erro mais comum dessa escolha.

Atenção

Chave artificial não dispensa a chave candidata natural. Se a tabela `curso` tiver só o `id` como chave, nada impede que Python Essencial seja cadastrado duas vezes, uma com o `id 2` e outra com o `id 4`, e a regra 1 deixa de valer sem que ninguém perceba. Toda chave candidata que vem de uma regra de negócio continua declarada no banco, como chave alternativa, mesmo quando a chave primária é artificial.

Na escola, a decisão ficou assim, tabela por tabela:

Tabela	Chave primária	Motivo
curso	id, artificial	o título pode mudar
pessoa	id, artificial	o CPF pode precisar de correção
turma	id, com o código da escola	o código existe para identificar a turma
matricula	aluno_id e turma_id	os dois valores não têm por que mudar
certificado	id, artificial	o formato do código pode mudar

A turma é o caso em que a chave natural é a melhor escolha. O código não descreve a turma: a escola o criou justamente para identificar cada turma, e é a própria escola que atribui os valores, como o 10 e o 11. Um dado criado com esse propósito tem a estabilidade que se espera de uma chave artificial e ainda tem significado para quem trabalha na escola. Repare que a coluna se chama `id` também na turma. Neste livro, toda chave primária formada por uma única coluna se chama `id`, seja ela natural ou artificial; o que muda é a origem do valor. O `id` de um curso é gerado pelo banco, e o de uma turma é o código que a escola dá a ela.

Na matrícula, a chave composta é formada por duas chaves estrangeiras, que apontam para as chaves primárias de `aluno` e de `turma`, e esses valores não têm motivo para mudar. Uma coluna `id` a mais não traria estabilidade nenhuma, e o par de colunas já cobra a regra 8 sozinho. No certificado, o código de verificação continua como chave alternativa, mas fica fora da chave primária: o formato desse código é decisão da escola, e, se ele mudar um dia, a mudança fica restrita a uma coluna.

Declarar as chaves, porém, faz mais do que documentar o modelo: o banco passa a cobrar regras a partir delas.

3.3 Integridade de entidade e integridade referencial

E se alguém cadastrar uma turma de um curso que não existe? Ou uma matrícula sem dizer quem é o aluno? Depois que as chaves são declaradas, o banco passa a recusar esses dados. Ele faz isso cobrando duas regras que o modelo relacional tem desde Codd e que valem para qualquer tabela, seja da escola, seja de qualquer outro negócio.

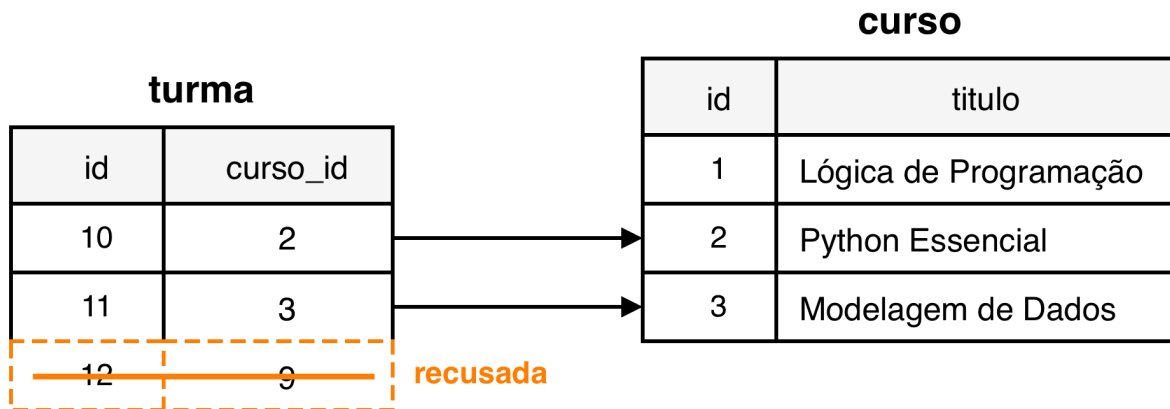


Figura 3.1: As turmas 10 e 11 apontam para cursos que existem. O cadastro da turma 12 é recusado, porque o curso 9 não existe.

A primeira é a **integridade de entidade** (*entity integrity*), a regra pela qual nenhuma parte da chave primária pode ficar sem valor. Codd a enunciou em 1979, e o motivo é direto: a chave primária existe para identificar a linha, e uma linha sem esse valor não pode ser identificada nem referenciada. Na chave composta, a regra vale para cada parte: uma matrícula com a turma e sem o aluno não identifica matrícula nenhuma. Junte a isso a propriedade de toda chave candidata, a de não se repetir, e temos o que o banco cobra de toda chave primária: valor presente e único em cada linha. Apesar do nome, a regra fala de tabelas e chaves. “Entidade” é o termo histórico, e o livro o mantém porque é com ele que você vai encontrar a regra em outros livros.

A segunda é a **integridade referencial** (*referential integrity*), a regra pela qual toda chave estrangeira aponta para uma linha que existe na tabela referenciada ou, quando a ligação é opcional, fica sem valor. É essa regra que impede o cadastro de uma turma de um curso que não existe.

Na [Figura 3.1](#) temos as tabelas `curso` e `turma`, com as setas das chaves estrangeiras. A turma 10 aponta para o curso 2, e a turma 11, para o curso 3. A tentativa de cadastrar uma turma 12 com `curso_id` 9 é recusada, porque não existe curso de `id` 9. Sem a chave estrangeira declarada, o banco aceitaria essa turma, e a pergunta “de qual curso é a turma 12?” ficaria sem resposta: o valor 9 apontaria para lugar nenhum.

A integridade referencial vale também no sentido contrário. Se alguém excluísse o curso Python Essencial, a turma 10 ficaria apontando para uma linha que não existe mais. O que o banco faz nesse caso depende da ação referencial declarada junto com a chave estrangeira. Quando nenhuma ação é declarada, ele recusa a exclusão; outras ações permitem, por exemplo, excluir junto as linhas que apontavam para a linha excluída. A regra 10 da escola pede ações diferentes para casos diferentes: os telefones saem junto com a pessoa, mas o curso com turmas não pode ser excluído. As ações referenciais são assunto do [capítulo 5](#).

E a parte da definição que admite a chave estrangeira sem valor? Ela existe porque nem toda ligação é obrigatória. Uma chave estrangeira sem valor não aponta para linha nenhuma, e o banco não procura uma linha correspondente. Quando a ligação é obrigatória, como a da turma com o curso pela regra 3, a coluna da chave estrangeira é declarada de modo a não aceitar a falta de valor. Mas o que é, exatamente, uma coluna sem valor?

3.4 NULL: o valor ausente

A turma 10 ainda está em andamento, e por isso a Ana e o Bruno ainda não têm nota final nela. O que a coluna `nota_final` guarda nas linhas dessas duas matrículas? Nada. E a forma de registrar esse nada é uma decisão de modelagem, com consequências.

NULL é a marca que indica que uma coluna não tem valor numa linha. O NULL não é zero nem um texto vazio: zero é um valor, e o texto vazio também é. NULL é a falta de valor. Veja a nota final nas quatro matrículas da escola:

aluno_id	turma_id	nota_final
1	10	NULL
2	10	NULL
1	11	8.5
3	11	6.0

As duas primeiras linhas são as matrículas da Ana e do Bruno na turma 10, ainda sem nota. As duas últimas são as matrículas da Ana e da Carla na turma 11, que já terminou.

Na escola, a falta de valor aparece em duas situações diferentes. Na primeira, o valor ainda não existe: a nota da Ana na turma 10 só vai ser lançada ao fim da turma, pela regra 8. Na segunda, o valor não se aplica àquela linha: uma turma on-line não tem sala, e, como a [seção 3.7](#) vai mostrar, a tabela `turma` guarda NULL na coluna `sala` da turma 10. O banco registra as duas situações da mesma forma, e quem lê os dados precisa saber, pelas regras de negócio, qual das duas cada NULL representa.

Quando uma coluna deve aceitar NULL? A resposta sai da cardinalidade de atributo do DER. Um atributo com cardinalidade mínima 1, como o nome da pessoa, vira uma coluna que não aceita NULL: o banco recusa o cadastro da pessoa sem nome. Um atributo com cardinalidade mínima 0, como a nota final, vira uma coluna que aceita NULL. A [seção 3.6](#) aplica o mesmo raciocínio às chaves estrangeiras, com a cardinalidade mínima do relacionamento no lugar da cardinalidade do atributo.

Aceitar NULL numa coluna tem duas consequências. A primeira é que o banco não distingue a falta de valor que a regra admite do simples esquecimento. Se a coluna do nome aceitasse NULL, o banco guardaria sem reclamar um cadastro sem nome, e ninguém saberia se o nome faltou porque não existe ou porque alguém se esqueceu de digitá-lo. Por isso, uma coluna só aceita NULL quando a regra de negócio admite a falta do valor.

A segunda consequência é menos óbvia: NULL não é igual a valor nenhum, nem a outro NULL. Como a nota da Ana na turma 10 ainda não existe, a pergunta “essa nota é maior que 7?” não tem resposta verdadeira nem falsa. Uma consulta que procura as notas maiores que 7 não traz essa matrícula, e uma que procura as notas até 7 também não. Isso muda o resultado das consultas, e o [capítulo 6](#) mostra como lidar com esse efeito.

Atenção

Não use um valor qualquer para representar a falta de valor. Se a escola guardasse 0 nas notas ainda não lançadas, o banco não teria como distinguir a Ana, que ainda não terminou a turma, de um aluno que tirou zero, e a média das notas cairia por causa de notas que não existem. Para dizer que o valor falta, use NULL; para dizer que o valor é zero, use 0.

Com as chaves e o NULL em mãos, temos tudo de que a conversão precisa.

3.5 Do DER às tabelas: entidades, atributos e identificadores

Vamos começar a conversão por **Curso**, a entidade mais simples da escola: ela tem quatro atributos, todos de um valor só, e um deles, o título, é o identificador. Esse exemplo serve para apresentar as três primeiras regras de conversão, que valem para toda entidade do DER.

A primeira regra: cada entidade vira uma tabela, com o nome da entidade no singular, em minúsculas e sem acento. **Curso** vira `curso`, e **Pessoa** vira `pessoa`. A segunda: cada atributo vira uma coluna, com o mesmo cuidado no nome. A carga horária vira `carga_horaria`, e a data de nascimento, `data_nascimento`. Esse formato tem motivo, que o [capítulo 5](#) explica ao criar as tabelas. A terceira: o identificador vira uma chave candidata, e a [seção 3.2](#) já decidiu qual candidata é a chave primária de cada tabela.

Junte a isso a regra da [seção 3.4](#), pela qual a cardinalidade mínima decide se a coluna aceita NULL, e a tabela `curso` fica assim:

Coluna	Vem de	Aceita NULL?
<code>id</code>	chave artificial	não
<code>titulo</code>	identificador, agora chave alternativa	não
<code>ementa</code>	atributo	não
<code>carga_horaria</code>	atributo	não
<code>preco</code>	atributo	não

Nenhuma coluna aceita NULL: o `id` é a chave primária, e todos os atributos de **Curso** são (1,1), porque a regra 1 diz que a escola guarda o título, a ementa, a carga horária e o preço de cada curso.

Pessoa quase segue o mesmo caminho. O nome, o CPF, o e-mail e a data de nascimento viram colunas, e o CPF vira a chave alternativa. O telefone é que não cabe. Pela regra 5, uma pessoa pode ter mais de um telefone, como a Ana, que tem dois, e uma coluna guarda um único valor em cada linha. Criar duas colunas, `telefone_1` e `telefone_2`, também não resolve. A regra não diz que o máximo é dois, e a pessoa com três telefones ficaria sem lugar para o terceiro; enquanto isso, quem tem um telefone só deixaria uma das colunas vazia.

A saída é a quarta regra de conversão: o atributo multivalorado vira uma tabela própria. A tabela `telefone` tem duas colunas. A primeira, `pessoa_id`, é uma chave estrangeira que aponta para a pessoa dona do telefone; a segunda, `numero`, guarda o número. Cada telefone é uma linha, e a pessoa com dois telefones aparece em duas linhas:

<code>pessoa_id</code>	<code>numero</code>
1	1091111111
1	1031111111
2	1092222222

As duas primeiras linhas são os telefones da Ana, e a terceira é o do Bruno; os telefones das demais pessoas seguem o mesmo formato. Repare que os números não têm parênteses nem hífen: a tabela guarda só os dígitos, e a formatação fica para quando o número é exibido. O [capítulo 5](#) explica o motivo, ao tratar dos tipos de dados. A chave primária é o par `pessoa_id` e `numero`, e cada parte dela é indispensável. Só o `pessoa_id` não basta, porque a Ana tem duas linhas. Só o `numero` também não basta, porque nenhuma regra diz que um número não se repete entre pessoas, e, sem uma regra que garanta que o valor não se repete, não há chave.

Agora que os telefones moram numa tabela à parte, alguém precisa dizer o que acontece com eles quando a pessoa é excluída. É isso que a regra 10 diz: os telefones saem junto com a pessoa. O [capítulo 4](#) volta ao caso do telefone por outro caminho, na primeira forma normal.

Uma coisa, porém, se perde na conversão. O (1,n) do telefone diz que toda pessoa tem pelo menos um telefone, mas nada nas duas tabelas impede uma pessoa sem nenhuma linha em `telefone`. A chave estrangeira garante que todo telefone tem uma pessoa; ela não garante que toda pessoa tem um telefone. A exigência de pelo menos um telefone vai para a lista das regras que as tabelas não cobram sozinhas, que a [seção 3.8](#) reúne.

Com as entidades e os atributos convertidos, falta ligar as tabelas.

3.6 Do DER às tabelas: relacionamentos

Pela regra 3, toda turma é de um curso. No DER, essa ligação é o losango **Oferta**; nas tabelas, não existe losango. Onde essa ligação vai parar? A resposta depende da cardinalidade máxima, e é por isso que a [seção 2.7](#) insistiu tanto na máxima.

No relacionamento 1:N, a chave estrangeira fica na tabela do lado n. Na **Oferta**, o lado n é o da turma: um curso pode ter várias turmas, e cada turma tem um curso só. A tabela **turma** ganha a coluna `curso_id`, e cada linha guarda o `id` do curso da turma: a turma 10 guarda 2, e a turma 11, 3. Por que não fazer o contrário, com uma coluna na tabela **curso** apontando para as turmas? Porque um curso pode ter várias turmas, e essa coluna teria de guardar vários valores na mesma linha, justamente o que uma coluna não faz, como vimos com o telefone. Do lado n, cada linha aponta para uma única linha do outro lado, e uma coluna dá conta.

A cardinalidade mínima decide se a chave estrangeira aceita NULL. Partindo de uma turma, a cardinalidade junto a **Curso** é (1,1): toda turma tem um curso, e por isso `curso_id` não aceita NULL. Partindo de um curso, a cardinalidade junto a **Turma** é (0,n), e essa mínima 0 não pede nada às tabelas: um curso sem turma, como Lógica de Programação, é só um curso para o qual nenhuma linha de turma aponta.

A **Docência** segue o mesmo caminho. Pela regra 7, toda turma tem exatamente um instrutor, e a tabela **turma** ganha a coluna `instrutor_id`, que também não aceita NULL. Se a regra 7 dissesse que uma turma pode ser aberta antes de ter instrutor, a mínima junto a **Instrutor** seria 0, e `instrutor_id` passaria a aceitar NULL: é a chave estrangeira sem valor que a integridade referencial admite nas ligações opcionais.

E se a mínima do lado n fosse 1? Suponha que a escola exigisse pelo menos uma turma para cada curso do catálogo. Essa regra não caberia em nenhuma coluna de **curso**, pelo mesmo motivo do telefone: a chave estrangeira garante que toda turma tem um curso, e não que todo curso tem uma turma.

No relacionamento N:N, nenhum dos dois lados resolve. Um aluno pode ter várias turmas, e uma turma pode ter vários alunos, então qualquer coluna, em qualquer das duas tabelas, teria de guardar vários valores. A saída é dar ao relacionamento uma tabela própria. Uma **tabela associativa** (*associative table*) é a tabela que representa um relacionamento N:N, com uma chave estrangeira para cada tabela relacionada. A **Matrícula** vira a tabela **matricula**, com a coluna `aluno_id`, que aponta para o aluno, e a coluna `turma_id`, que aponta para a turma. Cada matrícula é uma linha. A tabela dos alunos, **aluno**, sai da próxima seção; por enquanto, basta saber que o aluno de `id` 1 é a Ana.

Na [Figura 3.2](#) temos o N:N da **Matrícula** desdobrado em linhas. Cada linha da **matricula** liga um aluno a uma turma, e as duas setas mostram para onde apontam as duas chaves estrangeiras. A Ana, aluna de `id` 1, aparece destacada em duas linhas, uma para cada turma; a turma 11 também aparece em duas linhas, uma para a Ana e outra para a Carla. Nenhuma linha das três tabelas guarda mais de um valor numa coluna, e mesmo assim cada aluno pode ter várias turmas, e cada turma, vários alunos.

Os atributos do relacionamento vão para a tabela associativa. A data, o valor cobrado e a nota final viram as colunas `data_matricula`, `valor_cobrado` e `nota_final` da tabela **matricula**, que é o único lugar em que cabem: esses dados descrevem a ligação entre o aluno e a turma, e não algum dos dois isoladamente. A chave primária é a chave composta da [seção 3.2](#), o par `aluno_id` e `turma_id`, e é ela que cobra a regra 8: ninguém se matricula duas vezes na mesma turma.

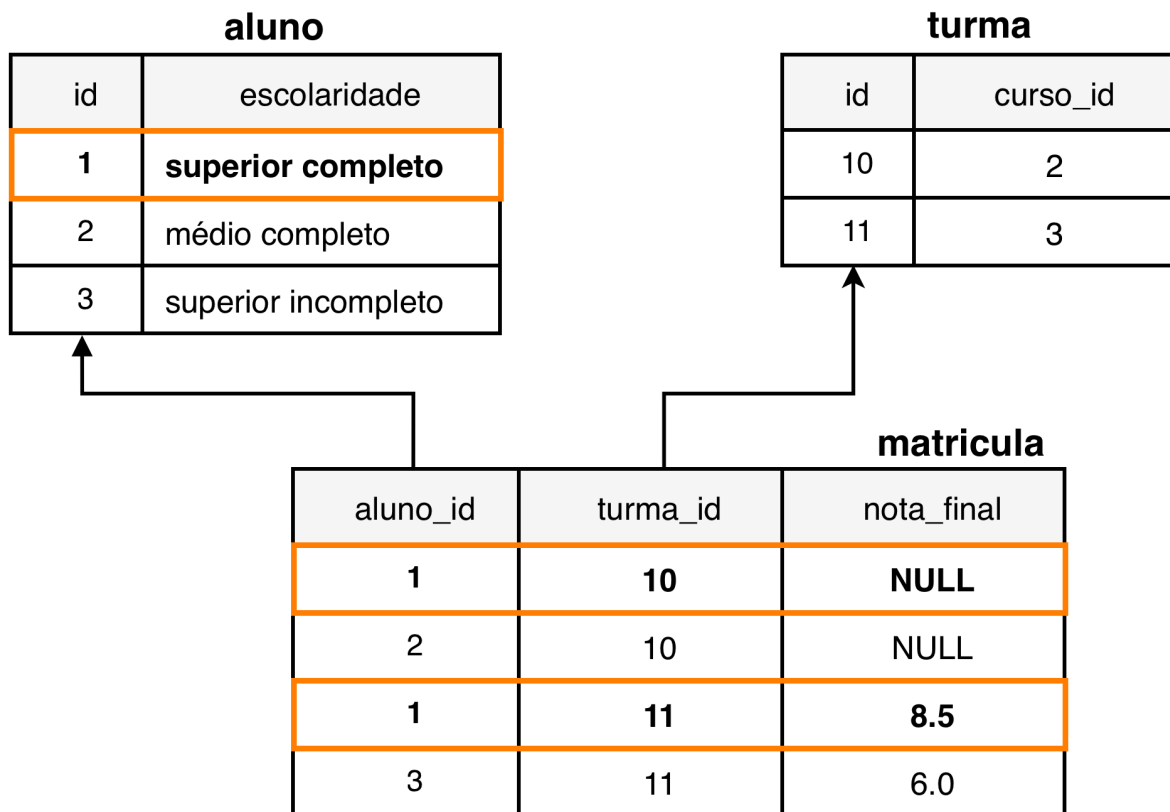


Figura 3.2: A tabela associativa matricula, entre as tabelas aluno e turma.

O **Pré-requisito** também é N:N e também vira uma tabela associativa, a `pre_requisito`, mas com uma particularidade: as duas chaves estrangeiras apontam para a mesma tabela, `curso`. Se as duas colunas se chamassem `curso_id`, não haveria como saber qual curso exige e qual é exigido. Por isso, a segunda coluna leva o nome do papel que o DER já registrava, e cada linha se lê como uma frase: o curso `curso_id` exige o curso `exigido_id`.

curso_id	exigido_id
2	1
3	1

A primeira linha diz que Python Essencial exige Lógica de Programação; a segunda, que Modelagem de Dados também exige. Lógica de Programação aparece duas vezes como curso exigido, porque, pela regra 2, um mesmo curso pode ser pré-requisito de vários. A chave primária é o par das duas colunas: o mesmo pré-requisito não é registrado duas vezes para o mesmo curso.

Falta o 1:1. Pela regra 9, cada matrícula gera no máximo um certificado, e todo certificado é de exatamente uma matrícula. A chave estrangeira poderia ficar em qualquer uma das duas tabelas, e é a cardinalidade mínima que decide qual é a melhor. Partindo de um certificado, a cardinalidade junto a **Matrícula** é (1,1): todo certificado tem uma matrícula. Partindo de uma matrícula, a cardinalidade junto a **Certificado** é (0,1), e muitas matrículas não têm certificado: na escola, a da Carla, com nota 6.0, e as duas da turma 10, ainda sem nota. Se a chave estrangeira ficasse na *matricula*, ela ficaria em NULL em todas essas linhas. Na tabela *certificado*, ela nunca fica sem valor, porque não existe certificado sem matrícula.

Como a matrícula tem chave composta, a chave estrangeira do certificado também é composta: as colunas *aluno_id* e *turma_id*, que juntas apontam para uma linha da *matricula*. Sem mais nada, porém, duas linhas de *certificado* poderiam apontar para a mesma matrícula, e o 1:1 viraria 1:N. Para impedir isso, o par *aluno_id* e *turma_id* é também uma chave alternativa da tabela *certificado* e, como toda chave alternativa, não pode se repetir.

Existe ainda uma terceira saída para o 1:1: juntar as duas tabelas numa só, com as colunas do certificado dentro da *matricula*. Ela evita uma tabela, mas tem o mesmo defeito da chave estrangeira na *matricula*: o código de verificação e a data de emissão ficariam em NULL em toda matrícula sem certificado. Se os dois lados fossem obrigatórios, toda matrícula teria um certificado, e a tabela única não teria colunas vazias; com a mínima 0 do lado do certificado, ela teria.

Resta a parte do DER que admite mais de uma solução: as especializações.

3.7 Do DER às tabelas: generalização e especialização

Pela regra 6, toda pessoa da escola é aluna, instrutora ou as duas coisas. No DER, essa regra virou a entidade **Pessoa**, com duas especializadas, **Aluno** e **Instrutor**. Nas tabelas, a especialização não tem uma conversão única, como o 1:N tem: há três alternativas, e escolher entre elas é uma das poucas decisões de verdade deste capítulo.

A primeira alternativa é uma tabela só para toda a hierarquia. A tabela *pessoa* ganharia também as colunas próprias do aluno e do instrutor, *escolaridade*, *data_admissao* e *valor_hora_aula*, e essas colunas ficariam em NULL nas linhas de quem não tem aquele papel. A segunda é uma tabela para cada entidade especializada, sem tabela para a entidade genérica: *aluno* e *instrutor*, cada uma com as próprias colunas e mais as colunas dos dados comuns, como o nome, o CPF e o e-mail. A terceira é uma tabela para a entidade genérica e uma para cada especializada: *pessoa* com os dados comuns, e *aluno* e *instrutor* só com os próprios dados, ligadas a *pessoa* por uma chave estrangeira.

Para escolher, faça três perguntas. Cada uma tem um motivo por trás.

- A especializada tem relacionamento próprio? O **Instrutor** tem a **Docência**. Na primeira alternativa, a da tabela única, a coluna *instrutor_id* da turma apontaria para *pessoa*, e o banco aceitaria como instrutor de uma turma qualquer pessoa, inclusive um aluno que nunca deu aula. Com uma tabela *instrutor*, a chave estrangeira aponta para ela, e só

quem está nessa tabela pode ser instrutor de uma turma. O mesmo vale para a **Matrícula**, que liga a turma a **Aluno**.

- A especialização é compartilhada? Na segunda alternativa, em que não existe a tabela **pessoa**, quem tivesse os dois papéis apareceria nas duas tabelas. Se a Marta se matriculasse um dia numa turma, o nome, o CPF e o e-mail dela passariam a estar em **instrutor** e em **aluno**, e a redundância do **capítulo 1** voltaria, com o risco de as duas cópias discordarem. Na especialização exclusiva, esse problema não existe, porque ninguém está nas duas especializadas.
- Quantos atributos próprios cada especializada tem? Na tabela única, cada atributo próprio vira uma coluna que fica em NULL nas linhas das outras especializadas. Com poucos atributos, isso custa pouco. Com muitos, a tabela se enche de colunas vazias, e fica difícil saber quais colunas valem para cada linha.

Para **Pessoa**, as duas primeiras perguntas decidem: o **Instrutor** tem relacionamento próprio, e a especialização é compartilhada. Fica a terceira alternativa, com as tabelas **pessoa**, **aluno** e **instrutor**. Em **aluno** e em **instrutor**, a coluna **id** é, ao mesmo tempo, a chave primária e uma chave estrangeira para **pessoa**: o **id** do aluno é o **id** da pessoa. A Ana, pessoa de **id** 1, está em **aluno** com o mesmo **id** 1; a Marta, pessoa de **id** 4, está em **instrutor** com o **id** 4. Assim, a tabela que aponta para um aluno, como a **matricula**, usa o mesmo número que identifica a pessoa.

id	nome	Em aluno?	Em instrutor?
1	Ana Lima	sim	não
2	Bruno Costa	sim	não
3	Carla Souza	sim	não
4	Marta Ribeiro	não	sim
5	Paulo Mendes	não	sim

Se a Marta se matricular numa turma, ela ganha uma linha em **aluno**, com o **id** 4, e nada do que já está em **pessoa** se repete.

As tabelas, porém, não cobram uma parte da regra 6. A especialização é total: toda pessoa é aluna ou instrutora. Nada nas três tabelas impede uma pessoa sem linha em **aluno** e sem linha em **instrutor**, pelo mesmo motivo do telefone: a chave estrangeira garante que todo aluno é uma pessoa, e não que toda pessoa tem um papel. A especialização total de **Pessoa** também vai para a lista.

Para **Turma**, as respostas são outras. Nem **Presencial** nem **On-line** tem relacionamento próprio; a especialização é exclusiva, pela regra 4; e cada especializada tem um único atributo, a sala ou o endereço da sala virtual. Nenhuma das perguntas pede tabelas separadas, e a tabela única custa apenas duas colunas que aceitam NULL, **sala** e **endereço_sala_virtual**. A turma 10, on-line, tem o endereço e NULL na sala; a turma 11, presencial, tem a sala e NULL no endereço.

id	sala	endereco_sala_virtual
10	NULL	https://aula.example.com/turma-10
11	Sala 2	NULL

Aqui, a tabela única tem ainda uma vantagem: a regra 4 inteira cabe nela. Dizer que toda turma é presencial ou on-line, nunca as duas, é o mesmo que dizer que, em cada linha, exatamente uma das duas colunas está preenchida. Essa condição envolve só colunas da própria linha, e o [capítulo 5](#) mostra como declará-la. Com três tabelas, garantir que cada turma esteja em exatamente uma das especializadas exigiria conferir linhas de tabelas diferentes ao mesmo tempo.

Repare na diferença em relação à [seção 2.9](#). Lá, vimos que a sala não é um atributo de **Turma** que fica em branco nas turmas on-line, e sim um atributo que só existe nas presenciais. Isso continua valendo no modelo conceitual. No modelo lógico, a tabela única representa esse “não existe” com um NULL que significa “não se aplica”, a segunda situação da [seção 3.4](#). O negócio não mudou; mudou a forma de representá-lo.

Agora temos todas as regras de conversão. Falta aplicá-las à escola inteira.

3.8 O estudo de caso em tabelas

Como fica o DER da escola inteiro, convertido em tabelas? Vamos aplicar as regras de conversão na mesma ordem em que a [seção 2.10](#) montou o diagrama, regra de negócio por regra de negócio.

- Pela regra 1, **Curso** vira a tabela **curso**, com a chave primária artificial **id** e o título como chave alternativa.
- Pela regra 2, o **Pré-requisito**, N:N, vira a tabela associativa **pre_requisito**, com as colunas **curso_id** e **exigido_id**, as duas apontando para **curso**.
- Pelas regras 3 e 4, **Turma** vira a tabela **turma**, com o código da escola na chave primária **id**, as datas e as vagas. A **Oferta**, 1:N, vira a chave estrangeira **curso_id**. As especializadas **Presencial** e **On-line** ficam na própria turma, nas colunas **sala** e **endereco_sala_virtual**.
- Pela regra 5, **Pessoa** vira a tabela **pessoa**, com a chave primária artificial **id** e o CPF como chave alternativa. O telefone, multivalorado, vira a tabela **telefone**.
- Pela regra 6, **Aluno** e **Instrutor** viram as tabelas **aluno** e **instrutor**, em que o **id** é o da pessoa.
- Pela regra 7, a **Docência**, 1:N, vira a chave estrangeira **instrutor_id** da tabela **turma**.
- Pela regra 8, a **Matrícula** vira a tabela associativa **matricula**, com a chave primária composta e os três atributos do relacionamento.
- Pela regra 9, **Certificado** vira a tabela **certificado**, com a chave primária artificial **id**, o código de verificação como chave alternativa e a chave estrangeira composta que aponta para a matrícula.

O resultado são nove tabelas, mostradas em três figuras. As figuras usam uma notação feita para mostrar a estrutura das tabelas sem os dados. Cada tabela é um retângulo com o nome no alto e uma linha para cada coluna. À esquerda da coluna, PK marca a chave primária, e FK, a chave estrangeira; à direita, NULL marca a coluna que aceita a falta de valor. Cada seta sai de uma chave estrangeira e chega à tabela referenciada, às vezes na linha da chave primária, às vezes na borda da tabela. Nos dois casos, ela aponta para a chave primária dessa tabela, a coluna marcada PK: na escola, toda chave estrangeira referencia uma chave primária, e nunca uma chave alternativa, como o *titulo* ou o *cpf*. Quando a chave estrangeira é composta, um colchete agrupa as colunas, e a seta é uma só. Uma tabela de contorno tracejado aparece completa em outra figura e, ali, mostra só as colunas de chave.

Na [Figura 3.3](#) temos o catálogo e as turmas. As duas setas que saem de *pre_requisito* chegam à mesma tabela, *curso*, e a turma aponta para o curso e para o instrutor. Na [Figura 3.4](#) temos as pessoas e os papéis delas: o telefone, o aluno e o instrutor apontam para a pessoa. Na [Figura 3.5](#) temos a matrícula, que aponta para o aluno e para a turma, e o certificado, que aponta para a matrícula pelo par de colunas agrupado pelo colchete. As três figuras se encontram nas tabelas *instrutor*, *aluno* e *turma*, que aparecem em mais de uma delas.

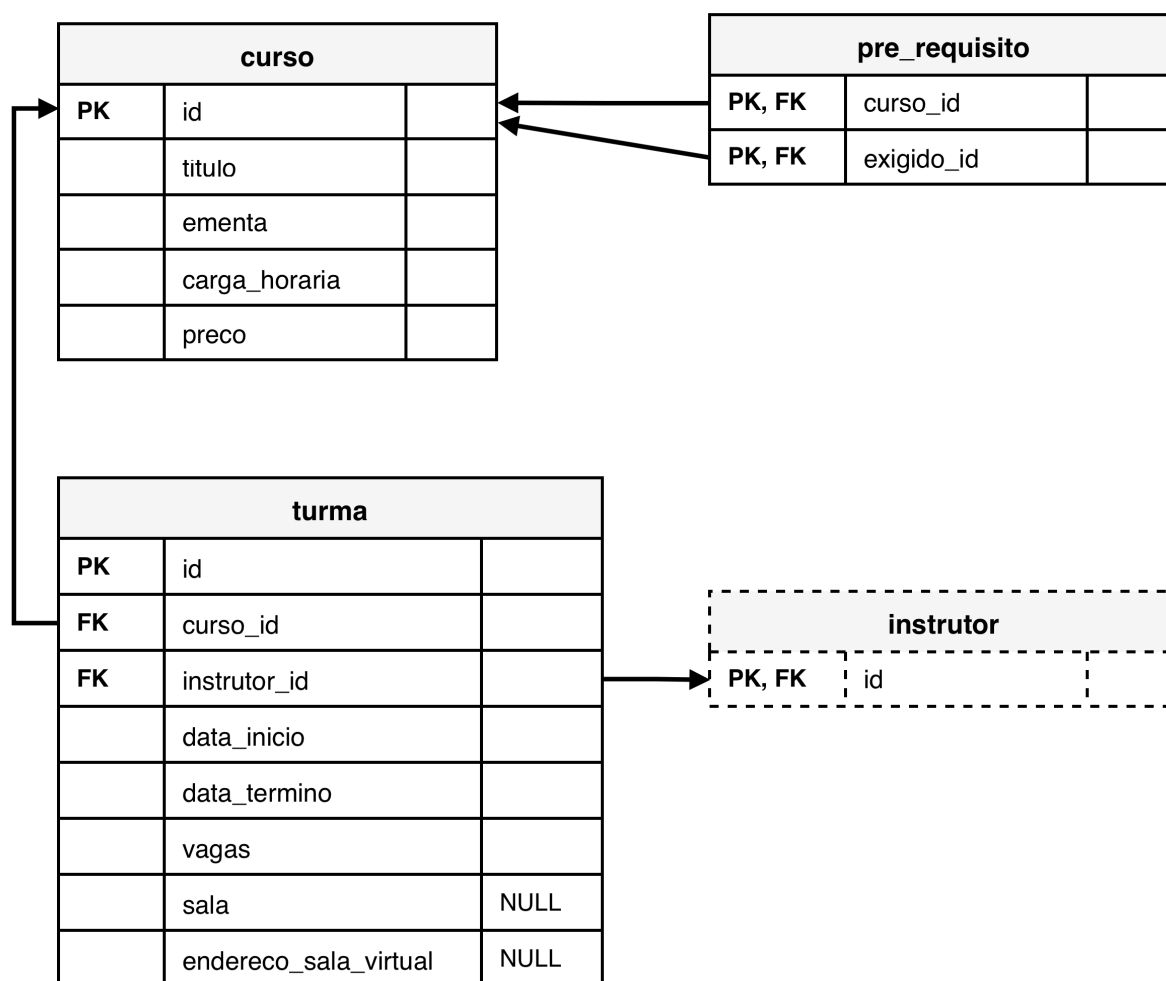


Figura 3.3: O esquema da escola, parte 1: cursos, pré-requisitos e turmas.

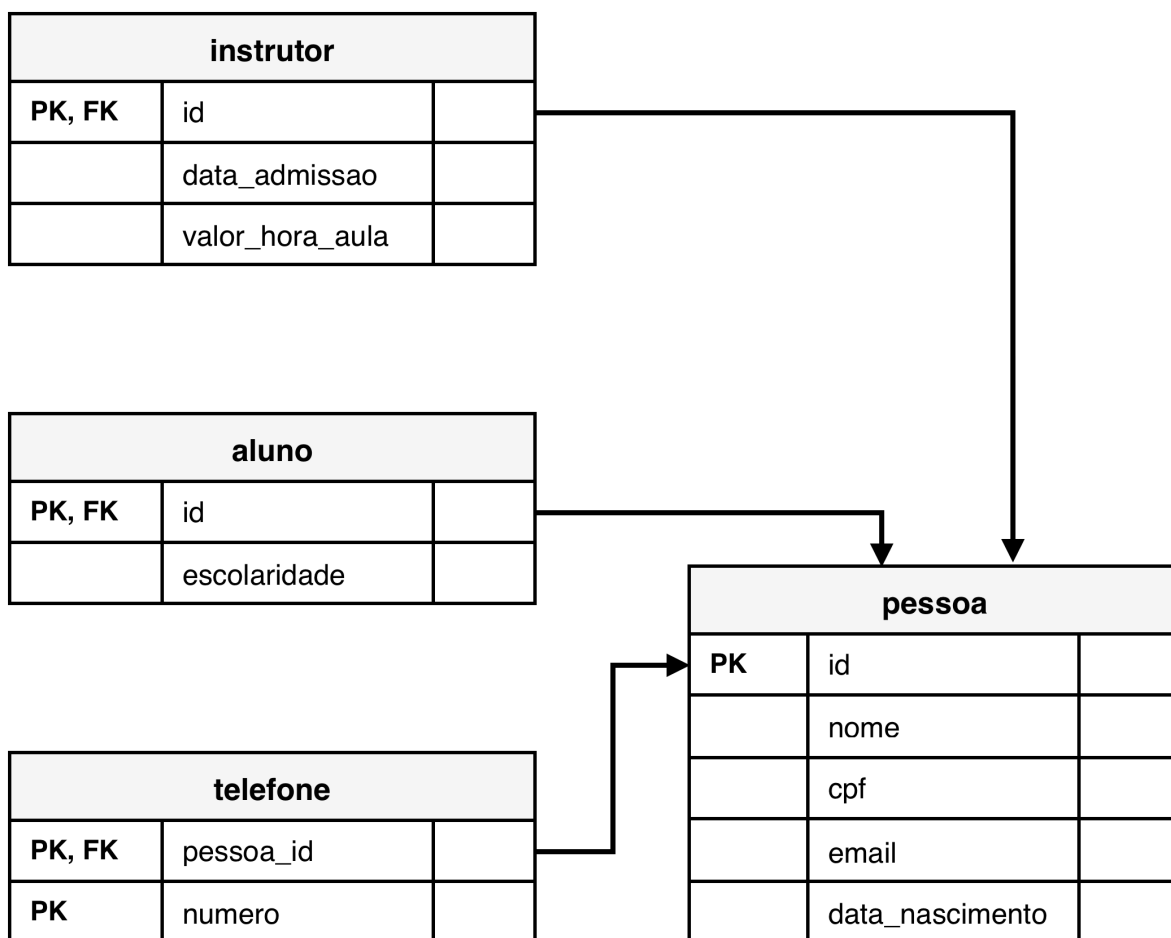


Figura 3.4: O esquema da escola, parte 2: pessoas, telefones, alunos e instrutores.

As figuras mostram as chaves primárias e as estrangeiras, mas não as alternativas. A tabela abaixo reúne todas as chaves candidatas, com as regras que as autorizam:

Tabela	Chave primária	Chave alternativa	Regras
curso	id	titulo	1
pre_requisito	curso_id, exigido_id	—	2
turma	id	—	3, 4, 7
pessoa	id	cpf	5
telefone	pessoa_id, numero	—	5
aluno	id	—	6
instrutor	id	—	6
matricula	aluno_id, turma_id	—	8
certificado	id	codigo_verificacao; o par aluno_id, turma_id	9

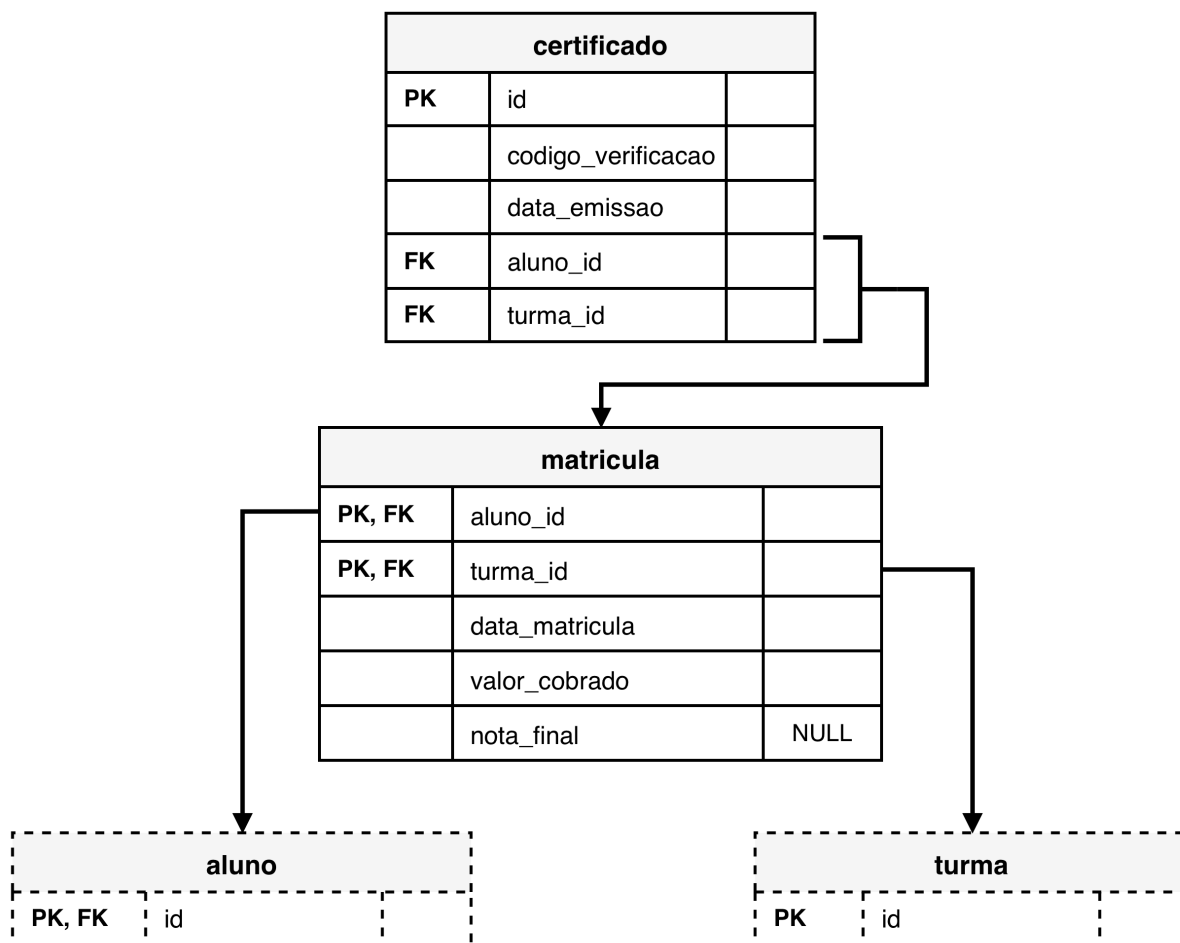


Figura 3.5: O esquema da escola, parte 3: matrículas e certificados.

Por fim, falta a lista das regras que as chaves e as decisões sobre o NULL não cobram. Ela reúne as regras que se perderam na conversão e as que o DER já não expressava, listadas na [seção 2.11](#):

- regra 4: em cada turma, exatamente uma das colunas `sala` e `endereco_sala_virtual` está preenchida;
- regra 5: toda pessoa tem pelo menos um telefone;
- regra 6: toda pessoa é aluna, instrutora ou as duas coisas;
- regra 8: a turma não passa do limite de vagas, a nota final vai de 0 a 10, e o valor cobrado é o preço do curso no dia da matrícula;
- regra 9: só a matrícula com nota final a partir de 7 tem certificado;
- regra 11: a transferência é só para outra turma do mesmo curso.

O banco consegue cobrar algumas dessas regras com outras restrições, que o [capítulo 5](#) apresenta; outras não cabem numa restrição simples. Enquanto isso, todas continuam escritas ao lado do modelo, como a [seção 2.11](#) recomendou.

As tabelas estão prontas. Mas como saber se elas estão bem desenhadas? O [capítulo 4](#) confere esse resultado por outro caminho, a normalização, e começa pela planilha em que a escola anotava tudo.

3.9 Resumo

Começamos pela correspondência entre os dois níveis: a entidade vira tabela, a ocorrência vira linha, o atributo vira coluna e o identificador vira chave. Vimos que, na teoria, a tabela se chama relação, termo que dá nome ao modelo relacional, e que as linhas de uma tabela não têm ordem nem se repetem.

Separamos os tipos de chave. A chave candidata é a coluna, ou o conjunto mínimo de colunas, cujo valor não se repete; das candidatas, uma é escolhida como chave primária, e as demais continuam valendo como chaves alternativas. A chave estrangeira aponta para linhas de outra tabela, ou da própria tabela. Comparamos a chave natural com a artificial e vimos que a escolha depende do motivo: a artificial protege contra valores do negócio que mudam, mas não dispensa a candidata natural, e a natural é a melhor escolha quando o dado existe justamente para identificar, como o código da turma.

Vimos o que o banco passa a cobrar quando as chaves são declaradas: a integridade de entidade, pela qual nenhuma parte da chave primária fica sem valor, e a integridade referencial, pela qual toda chave estrangeira aponta para uma linha que existe ou, numa ligação opcional, fica sem valor. Conhecemos o NULL, a marca da falta de valor, que serve tanto para o valor que ainda não existe quanto para o que não se aplica, e vimos que a cardinalidade mínima decide quais colunas o aceitam.

Com essas peças, convertemos o DER. O atributo multivalorado ganhou tabela própria. No 1:N, a chave estrangeira foi para o lado n; no N:N, o relacionamento virou uma tabela associativa; no 1:1, a chave estrangeira ficou do lado em que nunca fica sem valor. Para a especialização, escolhemos entre três alternativas com três perguntas e chegamos a respostas diferentes para **Pessoa** e para **Turma**. Por fim, montamos as nove tabelas da escola e listamos as regras que as chaves, sozinhas, não cobram.

No próximo capítulo, vamos conferir esse desenho com a normalização. Pegue um café: o ponto de partida é uma planilha bem bagunçada.

4 Normalização

“Na segunda e na terceira formas normais, um campo que não é chave precisa expressar um fato sobre a chave, a chave inteira e nada além da chave.”

William Kent (tradução livre)

No capítulo anterior, chegamos às nove tabelas da escola aplicando regras de conversão ao DER. Mas como saber se essas tabelas estão bem desenhadas? Até aqui, a única garantia era o cuidado com que montamos o DER e aplicamos as regras. Neste capítulo, vamos conferir o desenho por outro caminho, que não usa diagrama nenhum e olha só para as colunas de uma tabela e para o que o valor de uma coluna diz sobre o valor de outra.

Esse caminho serve também quando não há DER para conferir. Na vida real, é comum receber uma planilha ou um sistema que já está em uso, em vez de uma lista de regras de negócio, e precisar transformar o que já existe num banco bem desenhado. Era a situação da escola no capítulo 1, e é pela planilha da secretaria que vamos começar. No fim, vamos comparar o resultado com as tabelas do capítulo anterior.

O que será visto neste capítulo:

- [Redundância e anomalias](#)
- [Dependência funcional](#)
- [Primeira forma normal \(1FN\)](#)
- [Segunda forma normal \(2FN\)](#)
- [Terceira forma normal \(3FN\)](#)
- [Até onde normalizar](#)
- [Resumo](#)

4.1 Redundância e anomalias

No capítulo 1, vimos só um pedaço da planilha em que a secretaria anotava alunos, cursos e turmas. Aquele pedaço mostrava a linha da Ana em apenas três colunas: Nome, E-mail e Curso. A planilha inteira tem oito colunas e uma linha para cada matrícula. Essa planilha é o ponto de partida deste capítulo, porque os problemas que a normalização resolve estão todos nela.

Nome	CPF	E-mail	Telefones	Turma	Curso	Preço	Valor cobrado
Ana Lima	11111111111	ana.lima@example.com	10911111111, 10311111111	10	Python Essencial	720.00	720.00
Bruno Costa	22222222222	bruno.costa@example.com	10922222222	10	Python Essencial	720.00	690.00
Ana Lima	11111111111	ana.lima@example.com	10911111111, 10311111111	11	Modelagem de Dados	560.00	560.00
Carla Souza	33333333333	carla.souza@example.com	10933333333	11	Modelagem de Dados	560.00	560.00

Figura 4.1: A planilha da secretaria, com uma linha para cada matrícula. As duas linhas da Ana estão destacadas.

Na [Figura 4.1](#) temos as quatro matrículas da escola. Cada linha junta tudo o que a secretaria sabia sobre uma matrícula: quem é o aluno, como falar com ele, qual é a turma, de qual curso ela é, quanto o curso custa hoje e quanto o aluno pagou. A coluna Preço guarda o preço atual do curso, e a coluna Valor cobrado guarda o que o aluno pagou ao se matricular. O CPF e os telefones aparecem só com dígitos, como nas tabelas do capítulo 3.

Repare nas linhas destacadas. A Ana tem duas matrículas, e por isso o nome, o CPF, o e-mail e os dois telefones dela aparecem duas vezes. O mesmo acontece com os cursos: Python Essencial e o preço dele aparecem nas duas linhas da turma 10, e Modelagem de Dados e o preço dele, nas duas linhas da turma 11. É a redundância do capítulo 1, agora dentro de uma planilha só. E essa redundância cresce: cada matrícula nova repete mais uma vez os dados do aluno e os do curso.

A redundância, por si só, já dá trabalho: o mesmo dado precisa ser digitado várias vezes. O estrago maior vem de três problemas que ela provoca quando alguém altera a planilha, conhecidos como anomalias.

O primeiro é a **anomalia de atualização** (*update anomaly*): para alterar um único fato, é preciso alterar várias linhas, e quem esquece uma delas deixa a tabela discordando dela mesma. Se a Ana trocar de e-mail de novo, a secretaria precisa corrigir as duas linhas dela. Se corrigir só uma, a planilha passa a ter dois e-mails para a mesma pessoa, e a inconsistência do capítulo 1 volta, agora sem precisar de uma segunda planilha.

O segundo é a **anomalia de inserção** (*insertion anomaly*): um fato não pode ser registrado sem que outro, que ainda não existe, seja registrado junto. Lógica de Programação está no catálogo e custa 480.00, mas não tem turma nem matrícula. Onde a planilha guardaria esse preço? Cada linha é uma matrícula, e uma linha com o curso e o preço, mas sem aluno e sem turma, não é a matrícula de ninguém. O mesmo vale para o aluno: pela regra 8, a escola cadastra a pessoa como aluna antes de fechar a matrícula, e a planilha não tem lugar para um aluno que ainda não se matriculou.

O terceiro é a **anomalia de exclusão** (*deletion anomaly*): apagar um fato apaga junto outro que deveria continuar guardado. A Carla tem uma matrícula só. Se essa matrícula fosse cancelada e a secretaria apagassem a linha dela, o CPF, o e-mail e o telefone da Carla sumiriam junto, e a escola perderia o cadastro de uma aluna que poderia voltar no mês seguinte.

As três anomalias têm a mesma causa: a planilha guarda, numa tabela só, fatos sobre coisas diferentes. O nome e o e-mail dizem algo sobre o aluno; o curso diz algo sobre a turma; o preço, sobre o curso; e o valor cobrado, sobre a matrícula. Como cada linha é uma matrícula, os fatos sobre o aluno, a turma e o curso se repetem a cada matrícula e só existem enquanto houver uma matrícula para guardá-los.

Importante

Levar a planilha para um SGBD do jeito que ela está não resolve nenhuma dessas anomalias. Uma tabela com as mesmas oito colunas teria as mesmas repetições, e o banco aceitaria tanto a correção do e-mail da Ana numa linha só quanto a exclusão da única linha da Carla, porque as duas operações respeitam todas as regras declaradas. O SGBD cobra as regras que o desenho declara, mas não elimina a redundância sozinho: quem a elimina é o desenho das tabelas.

Para eliminar as anomalias, a planilha precisa ser dividida em tabelas menores, cada uma sobre um assunto só. A **normalização** (*normalization*) é o processo de dividir uma tabela em tabelas menores, guiado por critérios chamados formas normais, para que cada fato fique guardado num lugar só. Uma **forma normal** (*normal form*) é um critério que uma tabela atende ou não, e cada forma normal elimina um tipo de problema. Vamos ver as três primeiras, que são cumulativas: uma tabela só está na segunda forma normal se já estiver na primeira, e só está na terceira se já estiver na segunda. Codd apresentou a primeira forma normal em 1970, no artigo que propôs o modelo relacional, e definiu a segunda e a terceira num artigo de 1971.

Para dividir a planilha, porém, não basta olhar para ela e separar o que parece repetido. As formas normais se apoiam num conceito preciso, que diz o que o valor de uma coluna determina sobre o valor de outra: a dependência funcional.

4.2 Dependência funcional

Se você souber o CPF de uma linha da planilha, sabe também o nome do aluno dessa linha? Sabe: pela regra 5, cada CPF é de uma pessoa só, e cada pessoa tem um nome. E se você souber só o código da turma, sabe quanto o aluno pagou? Não sabe, e é essa diferença que as formas normais usam para decidir como dividir uma tabela.

Dizemos que uma coluna B depende funcionalmente de uma coluna A, ou que existe uma **dependência funcional** (*functional dependency*) de B em relação a A, quando cada valor de A corresponde a um único valor de B. Em outras palavras, duas linhas com o mesmo valor em A têm sempre o mesmo valor em B. A dependência se escreve $A \rightarrow B$ e se lê “A determina B”. A coluna à esquerda da seta é o **determinante** (*determinant*), e ele pode ser também um conjunto de colunas, escrito entre parênteses.

A planilha tem quatro dependências funcionais, e cada uma vem de uma regra de negócio:

Dependência	Regra	Por quê
CPF $\rightarrow$ Nome, E-mail	5	cada CPF é de uma pessoa, com um nome e um e-mail

Dependência	Regra	Por quê
Turma → Curso	3	cada turma é de um único curso
Curso → Preço	1	cada curso tem um preço atual
(CPF, Turma) → Valor cobrado	8	cada matrícula tem o seu valor cobrado

Na primeira linha, o CPF determina duas colunas, e as duas aparecem juntas à direita da seta. Na última, o determinante é o par CPF e Turma: nem o CPF sozinho nem a Turma sozinha determinam o valor cobrado, mas os dois juntos identificam uma matrícula, e a matrícula tem um valor cobrado só.

Repare que nenhuma dessas dependências foi tirada das quatro linhas da planilha. Uma dependência funcional precisa valer para todas as linhas que a tabela pode vir a ter, e não só para as que ela tem hoje, e quem diz o que a tabela pode vir a ter são as regras de negócio. Os dados de hoje até podem desmentir uma dependência. Veja as duas matrículas da turma 10:

Turma	Nome	Valor cobrado
10	Ana Lima	720.00
10	Bruno Costa	690.00

As duas linhas têm a mesma turma e valores cobrados diferentes, então a coluna Turma não determina a coluna Valor cobrado. Basta um par de linhas assim para derrubar a dependência. O motivo está na regra 8: o valor cobrado é o preço do curso no dia da matrícula, e o Bruno se matriculou antes de um reajuste de Python Essencial.

Mas os dados de hoje não conseguem provar uma dependência. Nas quatro linhas da planilha, cada curso aparece com uma única turma: Python Essencial só com a turma 10, e Modelagem de Dados só com a 11. Olhando apenas para os dados, Curso → Turma parece valer. Não vale. Pela regra 3, um curso pode ter várias turmas, e, no dia em que a escola abrir outra turma de Python Essencial, o mesmo curso vai aparecer com dois códigos de turma.

Atenção

Não deduza dependências funcionais dos dados de exemplo. Uma coincidência das linhas de hoje, como a de cada curso ter uma única turma, vira uma regra falsa, e a tabela acaba dividida no lugar errado: quem tomasse Curso → Turma como dependência guardaria a turma junto do curso, e a segunda turma do mesmo curso não teria onde ficar. As dependências saem das regras de negócio; os dados servem só para desmenti-las.

A dependência funcional também explica o que é uma chave. O par CPF e Turma não se repete entre as linhas, pela regra 8. Então cada valor do par corresponde a uma linha só e, portanto, a um único valor em cada uma das outras colunas: o par determina todas as colunas da tabela. É a chave candidata da [seção 3.2](#) vista por outro ângulo: um conjunto de colunas que determina todas as outras e do qual nenhuma coluna pode ser retirada sem que essa propriedade se perca.

A lista de dependências, porém, deixou de fora uma coluna da planilha: Telefones. O CPF não determina um telefone, porque a Ana tem dois. A célula dela guarda uma lista, e esse é o primeiro problema que a normalização resolve.

4.3 Primeira forma normal (1FN)

Olhe de novo para a célula de telefones da Ana na [Figura 4.1](#): ela guarda dois números. Para a secretaria, que só lia a planilha, esse arranjo funcionava. Para um banco, não: como encontrar quem tem o telefone 1031111111 sem desmontar o texto de cada célula? E como trocar só um dos dois números sem reescrever a célula inteira?

Um **valor atômico** (*atomic value*) é um valor que o negócio usa inteiro, sem precisar de partes dele. Uma tabela está na **primeira forma normal** (*first normal form*), ou **1FN**, quando cada coluna guarda um único valor atômico em cada linha e a tabela tem uma chave, de modo que nenhuma linha se repete. A célula de telefones da Ana falha na primeira condição, porque guarda dois valores.

Atômico não quer dizer impossível de dividir. O nome Ana Lima pode ser dividido em nome e sobrenome, e uma data, em dia, mês e ano. O que decide é o uso. A escola nunca precisa do sobrenome separado, e por isso o nome é atômico para ela; se precisasse listar os alunos em ordem de sobrenome, o nome viraria duas colunas. Já os dois telefones da Ana são valores que a escola usa separadamente, e a célula que junta os dois não é atômica.

Outro cuidado: atômico não quer dizer único. A 1FN não proíbe que o nome da Ana apareça em duas linhas. Cada uma dessas células guarda um valor só, e isso basta para a 1FN. A repetição entre linhas é assunto da 2FN e da 3FN.

A segunda condição pede uma chave. Qual seria a chave da planilha? O nome não serve, porque nenhuma regra impede duas pessoas com o mesmo nome. O CPF também não, porque a Ana tem duas linhas. O código da turma, sozinho, também não, porque a turma 10 tem duas linhas. Já o par CPF e Turma serve: pela regra 8, ninguém se matricula duas vezes na mesma turma. E cada parte do par é indispensável, como acabamos de ver.

Para resolver a célula de telefones, há três saídas. A primeira é repetir a linha para cada telefone. A matrícula da Ana na turma 10 viraria duas linhas, uma para cada número, e o telefone teria de entrar na chave, porque o par CPF e Turma passaria a se repetir. Os dados da Ana, que já apareciam duas vezes, passariam a aparecer quatro vezes. A segunda saída são colunas como Telefone 1 e Telefone 2, que a [seção 3.5](#) já descartou: a regra 5 não diz que o máximo é dois. A terceira, que é a escolhida, é tirar os telefones da planilha e guardá-los numa tabela própria, com uma linha para cada telefone e, ao lado de cada número, o CPF da pessoa a quem ele pertence. Esta é a tabela Telefone:

CPF	Número
1111111111	1091111111
1111111111	1031111111
2222222222	1092222222

CPF	Número
33333333333	10933333333

Por que só o CPF, e não o par CPF e Turma? Porque os telefones são da pessoa, e não da matrícula: os mesmos dois números da Ana apareciam nas duas linhas dela. Na tabela Telefone, cada número da Ana aparece uma vez só. A chave dessa tabela é o par CPF e Número: o CPF sozinho não basta, porque a Ana tem duas linhas, e o número sozinho também não, porque nenhuma regra diz que um número não se repete entre pessoas.

O resto da planilha continua igual, com sete colunas em vez de oito, e agora está na 1FN: cada célula guarda um valor só, e a chave é o par CPF e Turma.

Se a tabela Telefone parece familiar, não é coincidência. Na [seção 3.5](#), chegamos à tabela telefone pela regra de conversão do atributo multivalorado; aqui, chegamos à mesma tabela pela 1FN. A única diferença é o CPF no lugar de `pessoa_id`. A planilha não tem a chave artificial `id`, que a [seção 3.2](#) criou para a tabela `pessoa`, e a normalização trabalha só com as colunas que recebe. É a primeira vez que os dois caminhos se encontram, e não vai ser a última.

A 1FN, porém, não mexeu no nome e no e-mail da Ana, que continuam em duas linhas.

4.4 Segunda forma normal (2FN)

Depois da 1FN, cada célula da planilha guarda um valor só, e mesmo assim o nome e o e-mail da Ana continuam em duas linhas, assim como Python Essencial e o preço dele. Por que a repetição continua? A resposta está nas dependências entre as colunas da planilha.

Na [Figura 4.2](#) temos as sete colunas que ficaram na planilha depois da 1FN e uma seta para cada dependência da [seção 4.2](#). Cada seta sai de uma coluna só, menos a que chega ao Valor cobrado, que sai do colchete que agrupa o par CPF e Turma. Repare de onde sai cada seta. O Valor cobrado é determinado pela chave inteira. O Nome e o E-mail são determinados só pelo CPF, que é uma parte da chave. O Curso é determinado só pela Turma, a outra parte da chave, e o Preço também depende só da Turma, por meio do Curso.

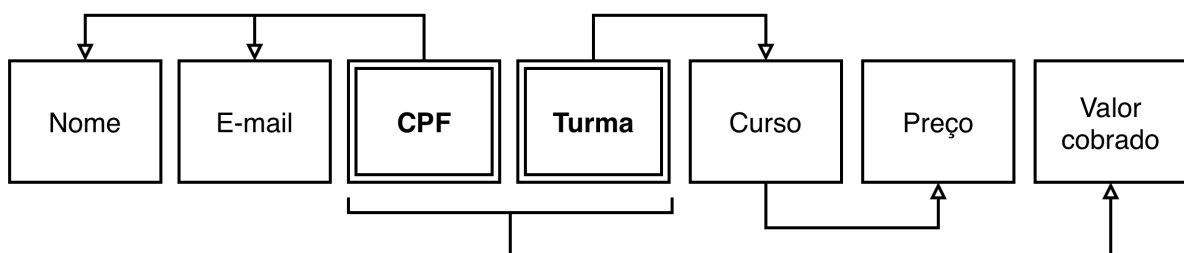


Figura 4.2: As dependências funcionais entre as colunas da planilha da secretaria, depois que os telefones saíram dela. As colunas CPF e Turma, juntas, formam a chave.

Há uma **dependência parcial** (*partial dependency*) quando uma coluna que não faz parte de nenhuma chave candidata depende só de uma parte de uma chave candidata composta. Uma tabela está na **segunda forma normal** (*second normal form*), ou **2FN**, quando está na 1FN e não tem dependência parcial. As duas definições seguem o artigo de Codd de 1971, e as duas falam de chave candidata, e não só de chave primária; o motivo aparece no fim desta seção.

A planilha tem duas dependências parciais: o Nome e o E-mail dependem só do CPF, e o Curso e o Preço dependem só da Turma. É por causa delas que os dados da Ana e os de Python Essencial se repetem. A redundância é o sintoma; o critério é a dependência parcial. O sintoma pode até faltar, como numa planilha em que cada aluno tivesse uma matrícula só, mas a dependência continuaria lá, e a repetição apareceria na segunda matrícula de alguém.

Daí sai uma consequência útil. Se todas as chaves candidatas de uma tabela têm uma coluna só, não existe parte de chave da qual uma coluna possa depender, e a tabela que está na 1FN já está na 2FN. A 2FN só dá trabalho quando há chave composta.

Para levar a planilha à 2FN, separamos cada dependência parcial numa tabela nova. Em toda dependência parcial, o determinante é só uma parte da chave da planilha. Esse determinante vai para a tabela nova, onde passa a ser a chave, e leva junto as colunas que dependem dele. Na planilha, ficam a chave inteira e as colunas que dependem dela inteira. O CPF, por exemplo, determina o Nome e o E-mail, e as três colunas formam a tabela Aluno, com o CPF como chave:

CPF	Nome	E-mail
11111111111	Ana Lima	ana.lima@example.com
22222222222	Bruno Costa	bruno.costa@example.com
33333333333	Carla Souza	carla.souza@example.com

A tabela Turma fica com a coluna Turma, que é a chave, e com o Curso e o Preço:

Turma	Curso	Preço
10	Python Essencial	720.00
11	Modelagem de Dados	560.00

E o que sobra da planilha é a tabela Matrícula, com o par CPF e Turma, que continua sendo a chave, e o Valor cobrado:

CPF	Turma	Valor cobrado
11111111111	10	720.00
22222222222	10	690.00
11111111111	11	560.00
33333333333	11	560.00

Confira o que aconteceu com as anomalias do aluno. A Ana aparece uma vez só na tabela Aluno, e, para trocar o e-mail dela, basta alterar uma linha. Se a matrícula da Carla fosse apagada, a linha dela na tabela Aluno continuaria lá. E um aluno que ainda não se matriculou ganha uma linha em Aluno sem precisar de linha em Matrícula, como a regra 8 pede.

Nada se perdeu na divisão. A tabela Matrícula guarda o CPF e a Turma de cada matrícula, e esses valores levam às linhas certas das tabelas Aluno e Turma. Juntando as três tabelas por essas colunas, é possível remontar a planilha da 1FN, linha por linha. Codd já mostrava isso em 1971: a tabela original pode ser recuperada a qualquer momento a partir das tabelas menores. Juntar tabelas numa consulta é assunto do [capítulo 6](#).

Atenção

Uma coluna de chave artificial não normaliza a tabela. Se a planilha ganhasse uma coluna Id, com um número diferente em cada linha, e só essa coluna fosse declarada como chave, a chave ficaria simples, e a tabela pareceria estar na 2FN. Mas o par CPF e Turma continuaria sem poder se repetir, pela regra 8, e por isso continuaria sendo uma chave candidata. O Nome continuaria dependendo só do CPF, e os dados da Ana continuariam em duas linhas. Antes de conferir uma forma normal, identifique todas as chaves candidatas que as regras de negócio criam, e não só a chave primária.

A 2FN resolveu o problema do aluno, mas não o do curso: Lógica de Programação continua sem lugar.

4.5 Terceira forma normal (3FN)

Lógica de Programação está no catálogo, custa 480.00 e ainda não tem turma. Depois da 2FN, onde a escola guarda esse preço? Na tabela Turma, não: cada linha dela é uma turma, e Lógica de Programação não tem nenhuma. A anomalia de inserção que a [seção 4.1](#) encontrou na planilha sobreviveu à 2FN, e a causa está na seta que a [Figura 4.2](#) desenha entre o Curso e o Preço.

Na tabela Turma, a chave é a coluna Turma, e o Preço depende dela, mas não diretamente: a turma determina o curso, e o curso determina o preço. Há uma **dependência transitiva** (*transitive dependency*) quando uma coluna que não faz parte de nenhuma chave candidata depende da chave por meio de outra coluna que também não faz parte de nenhuma chave candidata. Aqui, Turma → Curso → Preço. Uma tabela está na **terceira forma normal** (*third normal form*), ou **3FN**, quando está na 2FN e não tem dependência transitiva, ou seja, quando nenhuma coluna de fora das chaves candidatas depende de outra coluna de fora das chaves candidatas. A coluna Curso está fora das chaves candidatas da tabela Turma, porque, pela regra 3, o mesmo curso pode ter várias turmas.

Por causa da dependência transitiva, a tabela Turma ainda tem as três anomalias. A de inserção é a de Lógica de Programação. A de atualização aparece no próximo reajuste: o preço de Python Essencial está em todas as linhas das turmas desse curso, e o reajuste precisa alcançar todas elas. Hoje há uma turma por curso, mas a regra 3 permite várias, e a dependência vale para todas as linhas que a tabela pode vir a ter. A de exclusão aparece quando a única turma de um curso é excluída, o que a regra 10 permite enquanto a turma não tem matrículas: o preço do curso some junto com ela.

A solução segue o raciocínio da 2FN: a coluna Curso vira a chave de uma tabela própria, que fica também com a coluna Preço. A tabela Turma fica com a coluna Turma e com a coluna Curso, que diz de qual curso é cada turma:

Turma	Curso
10	Python Essencial
11	Modelagem de Dados

E a nova tabela Curso guarda o preço de cada curso, uma vez só:

Curso	Preço
Lógica de Programação	480.00
Python Essencial	720.00
Modelagem de Dados	560.00

Lógica de Programação finalmente tem onde ficar, mesmo sem turma. Um reajuste altera uma linha da tabela Curso, por mais turmas que o curso tenha. E excluir uma turma não apaga o preço de curso nenhum.

Agora que o preço mora na tabela Curso, olhe para a coluna Valor cobrado da tabela Matrícula. A Ana pagou 720.00 na turma 10, exatamente o preço de Python Essencial, e 560.00 na turma 11, exatamente o preço de Modelagem de Dados. Dá vontade de apagar a coluna e calcular o valor pelo curso da turma, não dá? O Bruno mostra por que não. Ele pagou 690.00, que era o preço de Python Essencial antes de um reajuste. Pela regra 8, o valor cobrado é o preço do curso no dia da matrícula, e não o preço atual. São dois fatos diferentes, que coincidem só enquanto o preço não muda. Se o valor cobrado fosse calculado a partir do preço atual, o reajuste reescreveria o que o Bruno pagou, e a escola deixaria de saber quanto cobrou dele. Por isso, a coluna fica, e ela não fere a 3FN: o Valor cobrado depende da chave inteira da tabela Matrícula, e de nenhuma outra coluna.

Com isso, chegamos ao fim da normalização: Aluno, Telefone, Curso, Turma e Matrícula, todas na 3FN. Falta comparar esse resultado com as tabelas da [seção 3.8](#):

Normalização	Capítulo 3	O que muda
Aluno	pessoa e aluno	chave artificial; especialização
Telefone	telefone	pessoa_id no lugar do CPF

Normalização	Capítulo 3	O que muda
Curso	curso	chave artificial; título como chave alternativa
Turma	turma	curso_id no lugar do título do curso
Matrícula	matricula	aluno_id no lugar do CPF

Cada tabela da normalização tem uma correspondente no capítulo 3, e as dependências são as mesmas. As diferenças têm três origens, e nenhuma delas é um erro de um dos dois caminhos.

A primeira são as chaves artificiais. A [seção 3.2](#) escolheu como chave primária uma coluna `id`, no lugar do CPF e do título, porque esses valores podem mudar. A normalização não toma essa decisão: ela trabalha com as chaves candidatas que existem, e o CPF e o título continuam sendo chaves candidatas nas duas versões.

A segunda é a especialização. A planilha só registra alunos, e por isso a normalização chegou a uma tabela Aluno. A divisão entre `pessoa`, `aluno` e `instrutor` vem da regra 6, que nenhuma coluna da planilha mostra.

A terceira é o que a planilha não tinha: os instrutores, os certificados, os pré-requisitos, as datas, as vagas e a nota. A normalização reorganiza as colunas que recebe e não cria as que faltam.

Uma armadilha aparece justamente nessa comparação. Na tabela `curso` do capítulo 3, a chave primária é o `id`, e o `preco` depende do `título`, que não é a chave primária. Isso fere a 3FN? Não, porque o `título` é uma chave candidata, a chave alternativa da tabela. A 3FN proíbe que uma coluna dependa de outra coluna de fora das chaves candidatas, e o `título` está entre elas. O mesmo vale para a tabela `pessoa`, em que o `nome` depende do `cpf`. Quem lê a definição como “nenhuma coluna pode depender de outra que não seja a chave primária” condena, sem motivo, toda tabela que tem uma chave artificial e uma chave alternativa. As três tabelas do capítulo 3 que ficaram fora da comparação, `pre_requisito`, `instrutor` e `certificado`, passam pela mesma conferência: nelas, toda coluna de fora das chaves candidatas depende só da chave inteira. As nove tabelas da escola estão na 3FN.

Os dois caminhos chegaram às mesmas dependências e às mesmas divisões, e isso dá confiança nos dois. A comparação mostra também até onde a normalização vai. A normalização confere as dependências entre as colunas que existem, mas não descobre o que ninguém registrou: ela confirma o modelo conceitual, e não o substitui.

4.6 Até onde normalizar

As nove tabelas da escola estão na 3FN. Dá para ir além da 3FN? E, no sentido contrário, faz sentido abrir mão de uma forma normal de propósito? As duas perguntas têm resposta, e as duas respostas dependem, mais uma vez, do que o negócio precisa.

A 3FN não é a última forma normal. Depois dela, a literatura define pelo menos mais três. A forma normal de Boyce-Codd é uma versão mais rigorosa da 3FN e exige que todo determinante seja uma chave candidata. A quarta e a quinta formas normais, propostas por Ronald Fagin em 1977 e em 1979, tratam de fatos multivalorados, como os telefones da Ana, quando uma mesma tabela junta mais de um fato desse tipo. Como avisei na Introdução, o livro para na 3FN: as tabelas da escola não têm as situações de que as formas seguintes tratam, e explicá-las exigiria uma teoria que não muda nenhuma decisão deste modelo.

A pergunta contrária pede mais cuidado. Toda divisão que a normalização fez tem um custo na hora de consultar. Na planilha, a secretaria via numa linha só o nome do aluno, o curso e o preço. Nas tabelas normalizadas, esses dados estão em tabelas diferentes, e a consulta que precisa de todos eles faz o SGBD encontrar, em cada tabela, as linhas que se correspondem e juntá-las. Kent já apontava esse custo em 1983: a normalização tende a penalizar a leitura, porque um dado que se obtinha de um registro só passa a ser buscado em vários.

A **desnormalização** (*denormalization*) é a decisão consciente de guardar um dado em mais de um lugar, abrindo mão de uma forma normal, para poupar esse trabalho numa consulta que se repete muito. Imagine que a escola exibisse no site a lista das turmas abertas, com o título do curso de cada uma, e que essa lista fosse consultada o tempo todo. Copiar o título do curso numa coluna da tabela `turma` deixaria a lista inteira numa tabela só, e o SGBD não precisaria consultar a tabela `curso` a cada acesso.

O custo é a volta das anomalias. O título passaria a estar em dois lugares, e, se a escola renomeasse Python Essencial, a alteração teria de chegar à tabela `curso` e a todas as turmas do curso; se uma delas ficasse para trás, o site mostraria o nome antigo. É a diferença entre as duas redundâncias do capítulo 1. A desnormalização só se justifica como redundância planejada, em que existe um mecanismo, definido junto com a cópia, que leva cada alteração a todas as cópias. A cópia que ninguém controla é justamente o problema que a normalização resolveu.

Por isso, a desnormalização tem condições. Ela parte de um modelo normalizado, porque só quem sabe de qual forma normal está abrindo mão sabe quais anomalias precisa controlar. Ela tem um motivo medido, uma consulta que de fato ficou lenta, e não uma que talvez fique; Kent registra que ninguém é obrigado a normalizar tudo quando os requisitos reais de desempenho entram na conta, e a palavra que importa aí é “reais”. E ela vem acompanhada do mecanismo que mantém as cópias iguais. Na escola, não há consulta lenta para justificar uma cópia: com três cursos e duas turmas, juntar as tabelas é um trabalho pequeno para o SGBD, e as tabelas ficam na 3FN. Outras formas de acelerar uma consulta sem mexer no modelo, como os índices, pertencem ao nível físico e ficam fora do livro, como a Introdução explicou.

Um último cuidado, com a palavra. Tabela que nunca foi normalizada não está desnormalizada. A planilha da secretaria não era desnormalizada; ela simplesmente nunca tinha sido normalizada. Desnormalizar é uma decisão tomada depois da normalização, com o custo conhecido, e não um nome elegante para o desenho que ninguém conferiu.

Com o desenho conferido, falta escrever as tabelas em SQL, e é isso que o [capítulo 5](#) faz.

4.7 Resumo

Começamos pela planilha da secretaria, a mesma de que o capítulo 1 mostrou um pedaço. Nela, vimos a redundância provocar três anomalias: a de atualização, em que alterar um fato exige alterar várias linhas; a de inserção, em que um fato não pode ser registrado sem outro que ainda não existe; e a de exclusão, em que apagar um fato apaga junto outro que deveria continuar guardado. Vimos também que levar a planilha para um SGBD do jeito que ela estava não resolveria nada, porque quem elimina a redundância é o desenho das tabelas.

Para dividir a planilha com critério, conhecemos a dependência funcional: a coluna B depende funcionalmente da coluna A quando cada valor de A corresponde a um único valor de B. Vimos que as dependências saem das regras de negócio, e não dos dados, que podem desmenti-las, mas não prová-las, e que a chave candidata é o conjunto mínimo de colunas que determina todas as outras.

Com as dependências em mãos, aplicamos três formas normais. Na 1FN, cada coluna passou a guardar um único valor atômico, e os telefones ganharam uma tabela própria. Na 2FN, eliminamos as dependências parciais, e os dados do aluno e os da turma saíram da matrícula. Na 3FN, eliminamos a dependência transitiva, e os dados do curso saíram da turma, o que deu um lugar a Lógica de Programação. No caminho, vimos que o valor cobrado não é cópia do preço, e sim outro fato, e que as definições falam de chaves candidatas, e não só da chave primária: uma coluna de chave artificial não normaliza tabela nenhuma.

Comparamos o resultado com as tabelas do capítulo 3 e vimos os dois caminhos chegarem às mesmas dependências e às mesmas divisões. As diferenças vieram das chaves artificiais, da especialização e do que a planilha não registrava. Por fim, vimos que existem formas normais além da 3FN, que o livro não desenvolve, e que desnormalizar é uma decisão legítima, com motivo medido e custo conhecido, que nada tem a ver com nunca ter normalizado.

No próximo capítulo, as tabelas saem do papel: vamos escrevê-las em SQL.

5 SQL: do modelo ao banco

“As restrições de integridade específicas de um banco de dados relacional precisam poder ser definidas na sublinguagem de dados relacional e guardadas no catálogo, e não nos programas de aplicação.”

Edgar F. Codd (tradução livre)

No capítulo anterior, conferimos o desenho das nove tabelas da escola com a normalização. Mas, até aqui, o modelo existe só no papel. As tabelas, as chaves e as regras de negócio estão em figuras e listas, e a única coisa que impede o cadastro de duas pessoas com o mesmo CPF é o cuidado de quem digita. Neste capítulo, vamos escrever o modelo em SQL, e é aqui que as regras de negócio passam a ser cobradas pelo banco.

É a ideia da epígrafe. Em 1985, Codd listou o que um SGBD precisava fazer para merecer o nome de relacional, e uma das exigências era que as regras ficassem guardadas no próprio banco, e não espalhadas pelos programas que gravam os dados. O [capítulo 1](#) mostrou por quê: a regra declarada uma vez no banco vale para todo programa, inclusive para o que ainda nem existe. Vamos começar pela linguagem e pelos tipos de dados, criar as tabelas, declarar as regras que elas cobram e, no fim, escrever a escola inteira em SQL. No caminho, vamos descobrir também quais regras o banco não consegue cobrar sozinho.

O que será visto neste capítulo:

- [O padrão SQL](#)
- [Um mapa da linguagem](#)
- [Tipos de dados](#)
- [Criando tabelas](#)
- [Restrições](#)
- [Ações referenciais](#)
- [Alterando e removendo estruturas](#)
- [O estudo de caso em SQL](#)
- [Resumo](#)

5.1 O padrão SQL

Se um dia a escola trocasse de SGBD, os comandos que vamos escrever neste capítulo continuariam funcionando? Em boa parte, sim, porque o SQL não pertence a nenhum fabricante: ele segue um padrão. Saber o que é esse padrão, e até onde ele vai, explica por que os comandos do livro são escritos do jeito que são.

O **padrão SQL** (*SQL standard*) é o conjunto oficial de regras da linguagem SQL. Ele é publicado com o número ISO/IEC 9075 pela ISO e pela IEC, que são organizações internacionais de padronização. A primeira versão foi aprovada em 1986 pelo ANSI, o órgão de padronização dos Estados Unidos, e adotada pela ISO em 1987. É por causa dessa origem que muita gente chama o padrão de “SQL ANSI”, nome que você vai encontrar em documentações e fóruns. O padrão continua sendo revisado, e a revisão vigente é a SQL:2023.

Por que seguir o padrão, se os exemplos usam o PostgreSQL como referência? Porque este é um livro de modelagem, e o modelo da escola não deveria depender de um SGBD. Um comando escrito na forma do padrão tem mais chance de funcionar do mesmo jeito em outro SGBD, e o que você aprende escrevendo dessa forma continua valendo fora do PostgreSQL. Mais chance, e não garantia: a própria documentação do PostgreSQL registra que nenhum SGBD declara seguir por completo nem mesmo o núcleo do padrão, a parte que todo SGBD deveria implementar. Cada SGBD implementa uma parte do padrão, acrescenta recursos próprios e, às vezes, faz a mesma coisa de outro jeito.

O PostgreSQL entra como referência de sintaxe. Todo comando do livro está escrito numa forma que o PostgreSQL aceita, e, quando existem duas formas, uma do padrão e outra própria do PostgreSQL, o texto usa a do padrão. Só que o PostgreSQL tem atalhos e tipos próprios que aparecem em quase todo material sobre ele, e não faria sentido fingir que eles não existem. Quando um deles aparecer, o livro avisa num bloco “Padrão SQL × PostgreSQL”, que mostra as duas formas e diz qual delas o livro usa.

Vale lembrar o combinado da Introdução: você não precisa executar os comandos deste capítulo nem os do próximo. Cada um mostra uma decisão de modelagem escrita na linguagem que o SGBD entende, e o que importa é entender a decisão. Antes do primeiro comando, porém, precisamos de um mapa dessa linguagem.

5.2 Um mapa da linguagem

Com quais comandos a escola cria as tabelas, cadastra a Ana e descobre quantos alunos há em cada turma? São comandos diferentes, cada um com um propósito, e o SQL tem muitos deles. Agrupá-los pelo que fazem ajuda a saber em que momento do trabalho cada um entra e em que parte do livro ele aparece.

Os comandos que criam, alteram e removem as estruturas do banco, como as tabelas, as colunas e as regras que as tabelas cobram, formam a **DDL**, linguagem de definição de dados (*Data Definition Language*): são o CREATE, o ALTER e o DROP. É com eles que o modelo lógico vira um banco, e por isso eles são o assunto deste capítulo. Os comandos que inserem, alteram e excluem linhas formam a **DML**, linguagem de manipulação de dados (*Data Manipulation Language*): são o INSERT, o UPDATE e o DELETE. A consulta, feita com o SELECT, lê as linhas sem alterar nenhuma. E os comandos de transação, como o START TRANSACTION, o COMMIT e o ROLLBACK, agrupam operações para que o SGBD execute todas ou nenhuma, como o [capítulo 1](#) explicou.

Grupo	Comandos	Para que serve	Onde
DDL	CREATE, ALTER, DROP	definir as estruturas	capítulo 5
DML	INSERT, UPDATE, DELETE	alterar as linhas	capítulo 6
consulta	SELECT	ler as linhas	capítulo 6
transação	START TRANSACTION, COMMIT, ROLLBACK	executar tudo ou nada	capítulo 6

Essa divisão é didática, e não uma regra da linguagem. As siglas variam conforme o autor: muitos materiais chamam a consulta de DQL (*Data Query Language*), enquanto a documentação do Oracle, por exemplo, põe o SELECT dentro da DML e trata a transação como um grupo à parte. A sigla importa pouco. O que importa é o que cada comando faz: o SELECT que lê as turmas e o UPDATE que altera uma turma são comandos de natureza diferente, qualquer que seja o nome do grupo de cada um.

Você vai encontrar ainda um quinto grupo, o de controle de acesso, às vezes chamado de DCL (*Data Control Language*), com o GRANT e o REVOKE, que dão e retiram as permissões de cada usuário sobre as tabelas. Ele fica fora do livro, como a Introdução avisou: decidir quem pode ver ou alterar cada tabela é administração do banco, e não modelagem.

Para quem modela, o grupo que interessa agora é o primeiro. É com ele que cada decisão dos capítulos anteriores vira uma instrução para o SGBD, a começar pela mais básica: que tipo de valor cada coluna guarda.

5.3 Tipos de dados

A carga horária de um curso é um número inteiro, o preço tem centavos, e a data de nascimento da Ana é uma data. Até aqui, o modelo da escola não dizia nada disso. Para criar uma tabela, porém, o SGBD precisa saber que tipo de valor cada coluna guarda, e essa escolha também é modelagem: ela decide quais valores a coluna aceita e o que se pode fazer com eles.

O **tipo de dado** (*data type*) de uma coluna é o conjunto de valores que ela aceita, junto com as operações que fazem sentido sobre esses valores. Se a coluna `carga_horaria` é de um tipo inteiro, o banco aceita 40 e recusa quarenta; e, como 40 é um número, dá para somar as cargas horárias dos cursos de um aluno. O tipo é a primeira regra que uma coluna cobra.

As regras da escola, porém, não dizem tudo o que é preciso saber para escolher o tipo. Nenhuma delas diz até quantos caracteres vai o título de um curso, nem com quantas casas decimais a nota é lançada, e o SQL obriga a decidir as duas coisas. Quem decide é o negócio, e não quem modela. Perguntamos à escola, e as respostas estão nos tipos desta seção. O livro usa só tipos do padrão SQL, e todos eles são aceitos pelo PostgreSQL. As tabelas da escola usam quatro desses tipos: `INTEGER`, `NUMERIC`, `VARCHAR` e `DATE`.

Para números inteiros, o tipo é `INTEGER`. A carga horária, o número de vagas e as chaves `id` são inteiros: a escola não oferece curso de 40,5 horas, e nenhuma turma tem meia vaga.

Para valores com casas decimais exatas, como o dinheiro, o tipo é `NUMERIC(p, s)`, em que `p` é o total de algarismos e `s` é a quantidade de algarismos depois da vírgula. O preço do curso é `NUMERIC(8, 2)`: oito algarismos, dois deles para os centavos, o que permite preços até 999999.99. Por que não um tipo de ponto flutuante, como o `REAL`, que também guarda números com casas decimais? Porque ele é aproximado. A documentação do PostgreSQL explica que alguns valores não podem ser convertidos exatamente para o formato interno desses tipos e são guardados como aproximações, e por isso recomenda o `NUMERIC` para valores monetários e para outras quantidades em que a exatidão é necessária. Em dinheiro, um centavo a mais ou a menos já é erro.

A nota final também é `NUMERIC`, com três algarismos, um depois da vírgula: `NUMERIC(3, 1)`. A escola lança as notas com uma casa decimal, como o 8.5 da Ana, e uma nota digitada com mais casas é arredondada para uma casa só. Repare que esse tipo aceita valores até 99.9, bem acima do 10 da regra 8. O tipo dá a forma do valor; o limite de 0 a 10 é outra regra, que a [seção 5.5](#) declara.

Para texto, o tipo é `VARCHAR(n)`, que aceita textos de até `n` caracteres e recusa os maiores. O título do curso é `VARCHAR(120)`, e a ementa, `VARCHAR(2000)`. Como escolher o `n`? O limite é uma regra: ele diz que um título com mais de 120 caracteres é um erro de digitação, e não um título. Mas um limite apertado demais recusa um dado legítimo. O wiki do PostgreSQL dá o exemplo do sobrenome: quem limita a coluna a 20 caracteres vai recusar, um dia, o cadastro de uma pessoa de verdade. Por isso, os limites da escola têm folga. O do e-mail, 254, é o tamanho do maior endereço que a RFC 5321, a norma que regula o envio de e-mails, admite.

Para datas, o tipo é `DATE`, que guarda dia, mês e ano: a data de nascimento, a de admissão do instrutor, as de início e de término da turma, a da matrícula e a da emissão do certificado. Guardar uma data como data, e não como texto, permite ao banco recusar um 31 de fevereiro e comparar uma data com outra, o que a [seção 5.5](#) usa para cobrar que a turma não termine antes de começar.

Falta cumprir a promessa do [capítulo 3](#): por que o CPF e o telefone ficam numa coluna de texto, só com os dígitos? O CPF tem 11 algarismos, e seria natural guardá-lo como número. Só que o CPF não é uma quantidade: ninguém soma CPFs nem calcula a média deles. Ele é um código formado por algarismos, e num código cada algarismo conta, inclusive os zeros à esquerda, que um tipo numérico descarta: guardado como número, o código 0123 vira 123. O mesmo vale para o telefone. Por isso, o CPF e o telefone são `VARCHAR(11)`. O padrão tem também o tipo `CHAR(n)`, de tamanho fixo, que parece feito para um código de 11 algarismos. Só que esse tipo completa com espaços o texto mais curto que o tamanho declarado, e o wiki do PostgreSQL recomenda não usá-lo.

E por que só os dígitos, sem os pontos e o hífen? Porque o mesmo CPF pode ser escrito de mais de um jeito: 111.111.111-11, 111111111-11 ou 11111111111. Para o banco, são três textos diferentes. Se cada atendente digitasse o CPF de um jeito, a proibição de dois cadastros com o mesmo CPF, declarada no banco, deixaria passar a Ana duas vezes, uma com pontos e outra sem. Com uma forma só, a dos dígitos, o mesmo CPF é sempre o mesmo texto, e a máscara, com os pontos e o hífen, fica para quando o número é exibido. A [seção 5.8](#) mostra a restrição que recusa o CPF digitado de outra forma.

Padrão SQL × PostgreSQL

Em quase todo material sobre PostgreSQL, você vai encontrar o tipo `TEXT`, que guarda textos de qualquer tamanho, sem limite declarado. Ele não é do padrão SQL, embora outros SGBDs tenham um tipo parecido. No padrão, o `VARCHAR` exige o tamanho, e os textos muito longos têm um tipo próprio, o `CHARACTER LARGE OBJECT`, que o PostgreSQL não implementa. O PostgreSQL também aceita o `VARCHAR` sem tamanho, que funciona como o `TEXT`. Este livro usa o `VARCHAR(n)` do padrão, com o limite de cada coluna escolhido com folga.

Com os tipos escolhidos, podemos criar a primeira tabela.

5.4 Criando tabelas

Como o catálogo de cursos da escola passa a existir dentro do banco? Com o `CREATE TABLE`, o comando que cria uma tabela. Ele recebe o nome da tabela e a lista das colunas, cada uma com o seu tipo. Vamos começar pela tabela `curso`, numa primeira versão:

```
CREATE TABLE curso (  
    id                INTEGER GENERATED ALWAYS AS IDENTITY,  
    titulo            VARCHAR(120),  
    ementa            VARCHAR(2000),  
    carga_horaria     INTEGER,  
    preco             NUMERIC(8,2)  
);
```

O comando diz ao SGBD: crie a tabela `curso`, com cinco colunas, nesta ordem. Cada linha entre os parênteses é uma coluna, com o nome à esquerda e o tipo à direita, e as colunas são separadas por vírgula. O ponto e vírgula encerra o comando.

Essa versão é válida, e o banco a aceitaria, mas ela quase não cobra regras da escola. Nada impede um curso sem título, dois cursos com o mesmo título ou uma carga horária negativa. As regras entram na próxima seção. Antes, vale entender três detalhes desse comando: por que os nomes estão em minúsculas e sem acento, por que as palavras-chave estão em maiúsculas e de onde vem o valor da coluna `id`.

Começemos pelos nomes, porque o [capítulo 3](#) prometeu explicar o formato deles: minúsculas, sem acento, com as palavras separadas por sublinhado, como em `carga_horaria`. O motivo está no que o SGBD faz com um nome escrito sem aspas. O padrão SQL manda converter esse nome para maiúsculas, e o PostgreSQL converte para minúsculas; a própria documentação do PostgreSQL registra que, nesse ponto, ele diverge do padrão. No PostgreSQL, então, `curso`, `Curso` e `CURSO` são o mesmo nome, porque os três viram `curso`. Para criar um nome que conserve as maiúsculas ou que tenha espaço, como `Carga Horaria`, é preciso escrevê-lo entre aspas. Com as aspas, a conversão não acontece: o nome fica exatamente como foi escrito, e todo comando que o usar vai precisar das aspas e da mesma grafia. Já um nome só com minúsculas, algarismos e sublinhado nunca precisa de aspas e é encontrado com qualquer mistura de maiúsculas e minúsculas que alguém digite. É a recomendação do wiki do PostgreSQL. E os acentos? O PostgreSQL aceita letras acentuadas num nome sem aspas, mas só garante a conversão para minúsculas das letras de A a Z. Nas letras acentuadas, a conversão depende da forma como o banco codifica os caracteres e pode não acontecer: num banco que não a faça, quem digitar `CARGA_HORÁRIA` não encontra a coluna `carga_horaria`, porque o `Á` continua maiúsculo. Num nome sem acento, a conversão vale para todas as letras. É por tudo isso que o modelo da escola chama de `carga_horaria` o que o DER chamava de carga horária.

As palavras-chave, como `CREATE TABLE` e `INTEGER`, estão em maiúsculas por convenção do livro, para separá-las à primeira vista dos nomes que nós escolhemos. O SGBD não faz diferença entre maiúsculas e minúsculas nas palavras-chave: `create table` funcionaria do mesmo jeito.

A primeira coluna, `id`, é a chave artificial que o [capítulo 3](#) escolheu para o curso, e quem gera o valor dela é o próprio banco. Quem pede isso ao banco é a cláusula `GENERATED ALWAYS AS IDENTITY`, que transforma `id` numa **coluna de identidade** (*identity column*): uma coluna cujo valor o SGBD gera sozinho, em sequência, a cada linha inserida. O primeiro curso cadastrado recebe o `id` 1, o segundo recebe o 2, e assim por diante. As tabelas `turma`, `aluno` e `instrutor` não usam esse recurso, porque nelas o valor do `id` não é gerado pelo banco. Na tabela `turma`, o `id` guarda o código que a escola atribui a cada turma. Nas tabelas `aluno` e `instrutor`, o `id` é o mesmo que a pessoa tem na tabela `pessoa`.

O ALWAYS diz que o valor vem sempre do banco: um comando que tente informar o id de um curso é recusado, a menos que peça, com uma cláusula própria, para substituir o valor gerado. A outra opção, BY DEFAULT, gera o valor só quando o comando não informa nenhum. O modelo da escola usa o ALWAYS porque o id do curso não significa nada para o negócio, e ninguém tem motivo para escolhê-lo à mão.

Um detalhe pega muita gente: a coluna de identidade gera valores em sequência, mas não impede que um valor se repita. A documentação do PostgreSQL lembra que a sequência pode ser reiniciada e que valores podem ser informados à mão. Quem impede a repetição é a chave primária, e é por ela que a próxima seção começa.

Padrão SQL × PostgreSQL

Em muito material sobre PostgreSQL, a coluna id aparece declarada como SERIAL, um atalho que o PostgreSQL oferecia antes de existirem as colunas de identidade e que não faz parte do padrão. O resultado parece o mesmo, um número gerado em sequência, mas a coluna de identidade é a forma do padrão, e o wiki do PostgreSQL recomenda usá-la nas aplicações novas. Este livro usa só GENERATED ALWAYS AS IDENTITY.

A tabela existe. Falta o principal: dizer ao banco quais regras ela cobra.

5.5 Restrições

A tabela curso da seção anterior aceita um curso sem título, dois cursos com o mesmo título e uma carga horária de -40. Nada disso combina com as regras da escola, e nada disso é recusado. Para o banco cobrar uma regra, ela precisa ser declarada na tabela.

Uma **restrição** (*constraint*) é uma condição declarada na tabela que toda linha precisa atender; o SGBD confere a condição em cada linha inserida ou alterada e recusa a linha que não a atende. É com as restrições que as decisões dos capítulos anteriores entram no banco. Veja a tabela curso completa:

```
CREATE TABLE curso (  
    id            INTEGER GENERATED ALWAYS AS IDENTITY,  
    titulo       VARCHAR(120) NOT NULL,  
    ementa       VARCHAR(2000) NOT NULL,  
    carga_horaria INTEGER NOT NULL,  
    preco        NUMERIC(8,2) NOT NULL,  
    CONSTRAINT curso_pkey PRIMARY KEY (id),  
    CONSTRAINT curso_titulo_key UNIQUE (titulo),  
    CONSTRAINT curso_carga_horaria_check  
        CHECK (carga_horaria > 0),  
    CONSTRAINT curso_preco_check CHECK (preco >= 0)  
);
```

O NOT NULL diz que a coluna não aceita NULL. Ele é a cardinalidade mínima da [seção 3.4](#) escrita em SQL: todo atributo com mínima 1 vira uma coluna NOT NULL. Pela regra 1, a escola guarda o título, a ementa, a carga horária e o preço de cada curso, e por isso as quatro colunas têm NOT NULL.

A restrição PRIMARY KEY declara a chave primária, e com ela o banco passa a cobrar a integridade de entidade da [seção 3.3](#): a coluna id não aceita valor repetido nem NULL. Por isso, as colunas da chave primária dispensam o NOT NULL, que a chave já inclui.

A restrição UNIQUE declara uma chave alternativa: a coluna não aceita valor repetido. A curso_titulo_key é a regra 1 escrita em SQL: não há dois cursos com o mesmo título. É ela que resolve o problema do aviso da [seção 3.2](#). Com uma chave artificial na chave primária, só o UNIQUE impede que Python Essencial seja cadastrado duas vezes, uma com o id 2 e outra com um id novo. Ao contrário da chave primária, o UNIQUE aceita linhas com NULL na coluna; quem impede o curso sem título é o NOT NULL da coluna titulo.

A restrição CHECK declara uma condição sobre os valores da linha. Na tabela curso, há duas: a carga horária é maior que zero, e o preço é maior que zero ou igual a zero. De onde vieram essas condições? Não das regras do [capítulo 2](#): a regra 1 diz que a escola guarda a carga horária e o preço, e não quais valores eles podem ter. Como nos tipos de dados, o SQL obriga a pensar nisso, e a resposta é da escola, e não de quem modela. Perguntamos, e a escola respondeu que a carga horária era sempre positiva e que o preço podia ser zero, porque a escola podia oferecer um curso gratuito. A diferença entre > 0 e ≥ 0 parece detalhe, mas é uma regra de negócio, e quem escolhesse sozinho entre os dois estaria inventando uma regra. As outras tabelas têm condições desse tipo, e todas vieram de respostas da escola: o término da turma não vem antes do início, toda turma tem pelo menos uma vaga, e nenhum curso é pré-requisito de si mesmo.

Repare na forma das restrições. Cada uma ganha um nome, depois da palavra CONSTRAINT, e fica no fim do comando, depois das colunas. O nome é necessário para alterar ou remover a restrição, como a [seção 5.7](#) vai mostrar, e, sem um nome declarado, cada SGBD inventa o seu. Os nomes seguem o formato dos que o próprio PostgreSQL gera: a tabela, a coluna e o tipo da restrição, com pkey para a chave primária, key para a chave alternativa, fkey para a chave estrangeira e check para a condição. O NOT NULL é a exceção e fica na própria coluna, porque o padrão SQL não o inclui entre as restrições que se declaram no fim da tabela.

A tabela turma traz as restrições que faltam: a chave estrangeira e uma condição que envolve mais de uma coluna.

```
CREATE TABLE turma (  
    id                INTEGER,  
    curso_id          INTEGER NOT NULL,  
    instrutor_id       INTEGER NOT NULL,  
    data_inicio        DATE NOT NULL,  
    data_termino       DATE NOT NULL,  
    vagas              INTEGER NOT NULL,  
    sala               VARCHAR(30),  
    endereco_sala_virtual VARCHAR(500),
```

```

CONSTRAINT turma_pkey PRIMARY KEY (id),
CONSTRAINT turma_curso_id_fkey
    FOREIGN KEY (curso_id) REFERENCES curso (id)
    ON DELETE RESTRICT,
CONSTRAINT turma_instrutor_id_fkey
    FOREIGN KEY (instrutor_id)
    REFERENCES instrutor (id),
CONSTRAINT turma_vagas_check CHECK (vagas > 0),
CONSTRAINT turma_periodo_check
    CHECK (data_termino >= data_inicio),
CONSTRAINT turma_modalidade_check CHECK (
    (sala IS NOT NULL
     AND endereco_sala_virtual IS NULL)
    OR (sala IS NULL
        AND endereco_sala_virtual IS NOT NULL)
)
);

```

A coluna `id` não tem `GENERATED`: o valor dela é o código que a escola dá à turma. A restrição `FOREIGN KEY` declara uma chave estrangeira e diz para onde ela aponta: `curso_id` referencia a coluna `id` da tabela `curso`, e `instrutor_id`, a coluna `id` da tabela `instrutor`. Com essas duas chaves estrangeiras, o banco cobra a integridade referencial, e o cadastro da turma 12 do curso 9, que a [Figura 3.1](#) mostrou, é recusado. A cardinalidade mínima continua decidindo o `NOT NULL`: pelas regras 3 e 7, toda turma tem um curso e um instrutor, e por isso as duas chaves estrangeiras não aceitam `NULL`. A linha `ON DELETE RESTRICT` é assunto da próxima seção.

Repare que a turma aponta para a tabela `instrutor`, que só aparece completa na [seção 5.8](#). Uma chave estrangeira só pode apontar para uma tabela que já existe, e por isso o banco só aceita este comando se a tabela `instrutor` já tiver sido criada. O mesmo vale para a `matricula`, mais adiante nesta seção, que aponta para as tabelas `aluno` e `turma`. A [seção 5.8](#) traz as nove tabelas na ordem em que podem ser criadas.

Agora, a promessa da [seção 3.7](#). Pela regra 4, toda turma é presencial ou on-line, nunca as duas, e na tabela única isso quer dizer que, em cada linha, exatamente uma das colunas `sala` e `endereco_sala_virtual` está preenchida. A `turma_modalidade_check` diz exatamente isso, com duas alternativas ligadas por `OR`: ou a sala está preenchida e o endereço não está, ou o endereço está preenchido e a sala não está. O `IS NOT NULL` pergunta se a coluna tem valor, e o `IS NULL`, se ela não tem. O banco recusa a turma com as duas colunas preenchidas e também a turma com as duas vazias.

O nome dessa restrição foge um pouco do formato. Quando a condição envolve mais de uma coluna, o PostgreSQL gera um nome sem coluna nenhuma, `turma_check`, e acrescenta um número a partir da segunda condição desse tipo na mesma tabela. Um nome assim não diz qual regra a restrição cobra. Por isso, quando a condição envolve várias colunas, o nome diz qual é a regra: `turma_modalidade_check` e `turma_periodo_check`.

Na tabela `matricula` aparecem os dois últimos casos: um CHECK sobre uma coluna que aceita NULL e um valor padrão.

```
CREATE TABLE matricula (  
    aluno_id          INTEGER,  
    turma_id          INTEGER,  
    data_matricula    DATE DEFAULT CURRENT_DATE NOT NULL,  
    valor_cobrado     NUMERIC(8,2) NOT NULL,  
    nota_final        NUMERIC(3,1),  
    CONSTRAINT matricula_pkey  
        PRIMARY KEY (aluno_id, turma_id),  
    CONSTRAINT matricula_aluno_id_fkey  
        FOREIGN KEY (aluno_id) REFERENCES aluno (id),  
    CONSTRAINT matricula_turma_id_fkey  
        FOREIGN KEY (turma_id) REFERENCES turma (id)  
        ON DELETE RESTRICT,  
    CONSTRAINT matricula_valor_cobrado_check  
        CHECK (valor_cobrado >= 0),  
    CONSTRAINT matricula_nota_final_check  
        CHECK (nota_final BETWEEN 0 AND 10)  
);
```

A chave primária é composta: as duas colunas vão juntas entre os parênteses, e é essa restrição que cobra a regra 8, pela qual ninguém se matricula duas vezes na mesma turma. A `matricula_nota_final_check` cobra a outra parte da regra 8: a nota vai de 0 a 10. Mas a nota da Ana na turma 10 ainda não foi lançada. Como a [seção 3.4](#) mostrou, a pergunta “essa nota está entre 0 e 10?” não tem resposta verdadeira nem falsa quando a nota é NULL. O CHECK recusa a linha só quando a condição é falsa, e por isso aceita a matrícula sem nota. É o que a regra 8 pede: a nota pode faltar, mas, quando existe, vai de 0 a 10.

Falta o DEFAULT, que não recusa nada e, por isso, não é uma restrição de verdade. Ele define o **valor padrão** (*default value*) da coluna, o valor que ela recebe quando o comando que insere a linha não informa nenhum. A escola registra cada matrícula no dia em que ela é fechada, e a coluna `data_matricula` tem o `CURRENT_DATE`, a data do dia, como valor padrão: a matrícula inserida sem data recebe a data de hoje.

O CHECK e o DEFAULT têm um limite em comum, e ele decide quais regras o banco consegue cobrar com eles: os dois só enxergam a linha que está sendo gravada. A documentação do PostgreSQL é explícita: a condição de um CHECK não pode conter consultas nem se referir a colunas de outras linhas, e o valor padrão também não pode conter consultas. Por isso, o valor cobrado, que pela regra 8 é o preço do curso no dia da matrícula, não pode ter esse preço como valor padrão: o preço está na tabela `curso`. Pelo mesmo motivo, a condição “a turma não passa do limite de vagas”, que depende de contar as linhas da `matricula`, não cabe num CHECK. A [seção 5.8](#) volta a essas regras.

5.6 Ações referenciais

O que acontece com os telefones da Ana se ela for excluída do cadastro? E com as turmas de Python Essencial, se alguém tentar excluir o curso? A [seção 3.3](#) deixou a resposta para este capítulo: depende da ação referencial declarada junto com a chave estrangeira. A regra 10 da escola pede respostas diferentes para casos diferentes, e é aqui que essas respostas viram SQL.

Uma **ação referencial** (*referential action*) é o que o SGBD faz com as linhas que apontam para uma linha quando ela é excluída ou quando o valor da chave dela é alterado. A ação é declarada na chave estrangeira, depois de `ON DELETE`, para a exclusão, ou de `ON UPDATE`, para a alteração.

Começemos pela primeira parte da regra 10: os telefones saem junto com a pessoa. A ação que faz isso é a `CASCADE`, e a tabela `telefone` fica assim:

```
CREATE TABLE telefone (  
    pessoa_id INTEGER,  
    numero    VARCHAR(11),  
    CONSTRAINT telefone_pkey  
        PRIMARY KEY (pessoa_id, numero),  
    CONSTRAINT telefone_pessoa_id_fkey  
        FOREIGN KEY (pessoa_id) REFERENCES pessoa (id)  
        ON DELETE CASCADE,  
    CONSTRAINT telefone_numero_check  
        CHECK (numero SIMILAR TO '[0-9]{10,11}'))  
);
```

Com a `ON DELETE CASCADE`, a exclusão de uma pessoa leva junto as linhas de `telefone` que apontam para ela. Se a Ana fosse excluída, as duas linhas com os telefones dela sairiam também, sem que ninguém precisasse excluí-las antes. A última restrição confere a forma do número. O `SIMILAR TO` verifica se o texto inteiro tem a forma descrita entre os apóstrofes: `[0-9]` quer dizer um algarismo qualquer, e `{10,11}`, que ele se repete de 10 a 11 vezes, o que corresponde ao DDD seguido de um número de 8 ou 9 algarismos. O número digitado com parênteses ou hífen não tem essa forma e é recusado.

Assim como a turma da [seção 5.5](#), a `telefone` aponta para uma tabela que só aparece completa na [seção 5.8](#): a `pessoa`. O banco só aceita este comando, e o da tabela `aluno`, mais adiante nesta seção, se a tabela `pessoa` já tiver sido criada.

A segunda parte da regra 10 diz o contrário: turma com matrículas não pode ser excluída, e curso com turmas também não. A ação que faz isso é a `RESTRICT`, com a qual o banco recusa a exclusão da linha enquanto houver alguma linha apontando para ela. Na [seção 5.5](#), a chave estrangeira `turma_curso_id_fkey` termina com `ON DELETE RESTRICT`, e a `matricula_turma_id_fkey` também. Com elas, o curso Python Essencial não pode ser excluído enquanto existir a turma 10, e a turma 10 não pode ser excluída enquanto tiver as matrículas da Ana e do Bruno.

E quando a chave estrangeira não declara ação nenhuma? Vale a ação padrão, a `NO ACTION`, que também recusa a exclusão. A diferença entre `NO ACTION` e `RESTRICT` só aparece quando a conferência da chave estrangeira é adiada para o fim de uma transação, recurso que o livro não usa; nos exemplos da escola, as duas recusam do mesmo jeito. Então por que escrever `RESTRICT`, se a ação padrão já recusaria? Porque, nessas duas chaves estrangeiras, a recusa é uma regra de negócio, e o `RESTRICT` escrito no comando deixa a regra 10 visível para quem lê o modelo. Nas chaves estrangeiras para as quais a escola não tem regra, como a que liga a matrícula ao aluno, fica a ação padrão: sem uma regra que mande excluir, recusar é a escolha que não perde dado.

A regra 10, porém, não diz o que acontece com os papéis quando a pessoa é excluída. Perguntamos à escola, e a resposta foi que a linha de `aluno` e a de `instrutor` saíam junto com a pessoa: o aluno não é outra coisa, e sim a própria pessoa no papel de aluno, e um papel sem a pessoa não faz sentido. A tabela `aluno` fica assim, e a `instrutor` segue o mesmo modelo:

```
CREATE TABLE aluno (
  id          INTEGER,
  escolaridade VARCHAR(30) NOT NULL,
  CONSTRAINT aluno_pkey PRIMARY KEY (id),
  CONSTRAINT aluno_id_fkey
    FOREIGN KEY (id) REFERENCES pessoa (id)
    ON DELETE CASCADE
);
```

As ações se encadeiam, e é na cadeia que se vê o que o modelo faz de verdade. Imagine que alguém tentasse excluir a Ana da tabela `pessoa`.

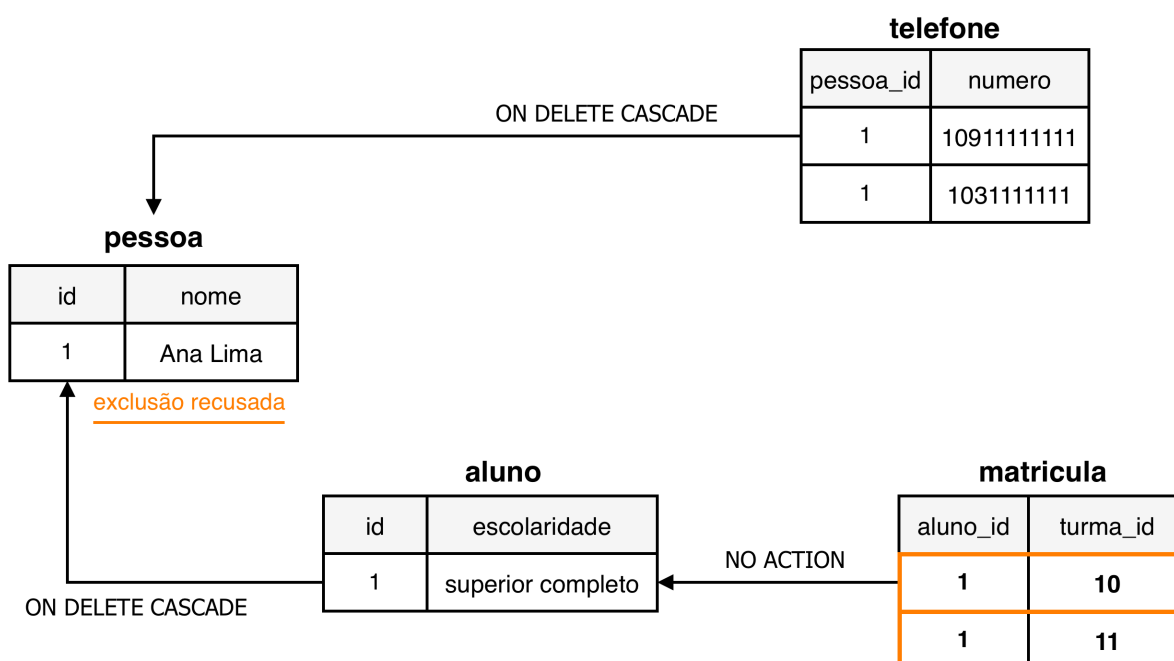


Figura 5.1: A exclusão da Ana alcançaria os telefones e a linha de aluno dela, mas as matrículas da Ana impedem a exclusão, e nenhuma linha é excluída.

Na [Figura 5.1](#) temos o caminho que essa exclusão percorreria. Os dois telefones da Ana sairiam por causa da `ON DELETE CASCADE` da tabela `telefone`, e a linha dela em `aluno` sairia pelo mesmo motivo. Só que essa linha de `aluno` é apontada pelas duas matrículas da Ana, e a chave estrangeira `matricula_aluno_id_fkey` não declara ação nenhuma: vale a ação padrão, que recusa. O banco recusa, então, o comando inteiro, e nada é excluído, nem os telefones. O SGBD executa um comando por inteiro ou não o executa, e por isso não deixa a exclusão pela metade, com a Ana sem telefones e ainda matriculada.

É o resultado que se espera quando a regra não manda excluir. As matrículas da Ana guardam a data, o valor cobrado e a nota de cada turma, e nenhuma regra autoriza que esses dados desapareçam junto com o cadastro dela. Já um aluno recém-cadastrado e ainda sem matrícula, situação que a regra 8 permite, seria excluído sem problema, com os telefones e a linha de `aluno` junto.

Até aqui, falamos da exclusão. A alteração da chave tem as mesmas ações, declaradas depois de `ON UPDATE`, mas na escola essa alteração praticamente não acontece, por causa de uma decisão da [seção 3.2](#). As chaves primárias de `curso`, `pessoa` e `certificado` são artificiais, e um valor sem significado não tem por que mudar. Se a chave primária da `pessoa` fosse o CPF, a correção de um CPF digitado errado mudaria a chave, e cada telefone, cada linha de `aluno` e cada matrícula daquela pessoa precisaria mudar junto; a `CASCADE`, declarada como `ON UPDATE CASCADE`, faria isso. Com a chave artificial, a correção do CPF altera uma coluna de uma linha, e nenhuma chave estrangeira percebe a mudança.

Há ainda duas ações que o modelo da escola não usa. A `SET NULL` deixa sem valor a chave estrangeira das linhas que apontavam para a linha excluída. Lembra-se da variação da [seção 3.6](#), em que a regra 7 permitiria uma turma sem instrutor? Nela, `instrutor_id` aceitaria `NULL`, e uma `ON DELETE SET NULL` deixaria sem instrutor as turmas do instrutor excluído, em vez de impedir a exclusão. Com a regra 7 como ela é, `instrutor_id` não aceita `NULL`, e essa ação esbarraria no `NOT NULL` da coluna. A `SET DEFAULT` funciona do mesmo jeito, mas põe na chave estrangeira o valor padrão da coluna, em vez do `NULL`.

Atenção

A `CASCADE` exclui linhas de outras tabelas sem aviso: quem exclui a pessoa não vê os telefones saírem. Declarada onde nenhuma regra de negócio manda, ela apaga dados que deveriam ficar, e a perda só aparece quando alguém procura por eles. Declare `CASCADE` só onde uma regra diz que as linhas dependentes saem junto; nos demais casos, deixe o banco recusar.

Com as ações, as tabelas estão completas. Mas o negócio muda, e o modelo muda junto.

5.7 Alterando e removendo estruturas

O que acontece com as tabelas da escola quando o negócio muda? Uma informação nova passa a ser guardada, uma regra deixa de valer, uma tabela perde a razão de existir. O modelo acompanha essas mudanças, e o banco também, com dois comandos: o `ALTER TABLE`, que altera uma tabela que já existe, e o `DROP TABLE`, que remove uma tabela. As mudanças desta seção são hipóteses, feitas só para mostrar os comandos; nas seções seguintes, o modelo da escola continua o mesmo.

Imagine que a escola, com o banco já em uso e a Marta e o Paulo cadastrados, passasse a publicar no site um minicurriculo de cada instrutor:

```
ALTER TABLE instrutor
  ADD COLUMN minicurriculo VARCHAR(1000);
```

O `ADD COLUMN` acrescenta a coluna no fim da tabela, e ela passa a existir também nas linhas que já estavam lá. Que valor a Marta e o Paulo teriam nessa coluna? Nenhum: as linhas que já existiam recebem `NULL`, ou o valor padrão, quando a coluna nova tem um `DEFAULT`. É por isso que a coluna nova aceita `NULL`. Se ela fosse declarada `NOT NULL` e sem valor padrão, o banco recusaria o comando, porque as linhas da Marta e do Paulo ficariam sem um valor obrigatório. Para tornar o minicurriculo obrigatório, a escola teria primeiro de escrever o de cada instrutor. Repare como o tempo encarece a decisão de tornar uma coluna obrigatória: uma coluna obrigatória criada junto com a tabela custa duas palavras no comando, e a mesma coluna acrescentada depois custa escrever o dado de cada linha que já existe.

Imagine agora que a ementa de alguns cursos passasse de 2.000 caracteres:

```
ALTER TABLE curso
  ALTER COLUMN ementa SET DATA TYPE VARCHAR(4000);
```

O `SET DATA TYPE` troca o tipo da coluna, e o banco converte os valores que já estão nela. Aumentar o limite de um texto é a troca mais tranquila, porque todo texto que cabia em 2.000 caracteres cabe em 4.000. A troca contrária, para um limite menor ou para outro tipo, pode esbarrar num valor que não cabe no tipo novo, e aí o banco recusa o comando.

As restrições também mudam. Imagine que a escola deixasse de oferecer cursos gratuitos. A restrição `curso_preco_check`, que aceita o preço zero, teria de dar lugar a uma que exige preço positivo:

```
ALTER TABLE curso
  DROP CONSTRAINT curso_preco_check RESTRICT;

ALTER TABLE curso
  ADD CONSTRAINT curso_preco_check CHECK (preco > 0);
```

O primeiro comando remove a restrição antiga, e é aqui que o nome escolhido na [seção 5.5](#) se paga: para remover uma restrição, é preciso dizer qual, e quem modelou sabe o nome de cada uma, porque foi quem o escolheu. Sem um nome declarado, seria preciso descobrir antes o nome que o SGBD inventou. O `RESTRICT` no fim do comando pede ao banco que recuse a remoção se alguma outra estrutura depender da restrição; nenhuma depende de uma restrição `CHECK`, e a remoção acontece. O segundo comando acrescenta a restrição nova, com o mesmo nome. Antes de aceitá-la, o banco confere as linhas que já estão na tabela: se algum curso tivesse preço zero, a restrição nova seria recusada, e a escola precisaria primeiro decidir o que fazer com esse curso.

Por fim, a tabela inteira. Imagine que a escola deixasse de emitir certificados:

```
DROP TABLE certificado RESTRICT;
```

O `DROP TABLE` remove a tabela e todas as linhas dela, sem pedir confirmação. O `RESTRICT`, aqui também, pede ao banco que recuse a remoção se alguma outra estrutura depender da tabela. Nenhuma tabela da escola aponta para `certificado`, e a remoção acontece. Com a tabela `curso`, a história é outra:

```
-- ERRO ESPERADO: turma e pre_requisito referenciam curso
DROP TABLE curso RESTRICT;
```

As chaves estrangeiras das tabelas `turma` e `pre_requisito` apontam para `curso`, e o banco recusa a remoção. Repare que a recusa não depende de haver linhas: mesmo sem nenhuma turma cadastrada, a chave estrangeira existe e depende da tabela `curso`. Para remover a tabela assim mesmo, seria preciso trocar o `RESTRICT` por `CASCADE`, e é aqui que mora a confusão mais comum. O `CASCADE` do `DROP TABLE` não remove as tabelas `turma` e `pre_requisito`, nem as linhas delas: ele remove a tabela `curso` e as restrições de chave estrangeira que apontavam para ela. As outras duas tabelas continuam lá, com as colunas `curso_id` e `exigido_id` guardando números que não apontam para nada, sem que o banco confira mais coisa alguma. É um `CASCADE` diferente do da [seção 5.6](#), que excluía linhas.

Padrão SQL × PostgreSQL

No PostgreSQL, o `RESTRICT` é o comportamento padrão do `DROP TABLE` e do `DROP CONSTRAINT`, e quase todo material escreve esses comandos sem ele, com o mesmo efeito. O padrão SQL exige que o comando diga se a remoção é `RESTRICT` ou `CASCADE`, e o livro o escreve sempre.

Com os comandos de definição apresentados, podemos escrever a escola inteira.

5.8 O estudo de caso em SQL

Como fica a escola inteira em SQL? São nove comandos `CREATE TABLE`, um para cada tabela das três figuras da [seção 3.8](#), e a ordem deles não é livre. Uma chave estrangeira só pode apontar para uma tabela que já existe, e por isso a tabela referenciada vem antes da que a referencia: curso vem antes de `pre_requisito` e de `turma`; pessoa, antes de `telefone`, `aluno` e `instrutor`; `instrutor`, antes de `turma`; e assim por diante. A ordem abaixo atende a todas essas dependências. Os comandos estão em três blocos: o catálogo de cursos, as pessoas e, por fim, as turmas, as matrículas e os certificados. O texto depois de cada bloco comenta só o que ainda não apareceu neste capítulo.

```
CREATE TABLE curso (
    id            INTEGER GENERATED ALWAYS AS IDENTITY,
    titulo        VARCHAR(120) NOT NULL,
    ementa        VARCHAR(2000) NOT NULL,
    carga_horaria INTEGER NOT NULL,
    preco        NUMERIC(8,2) NOT NULL,
    CONSTRAINT curso_pkey PRIMARY KEY (id),
    CONSTRAINT curso_titulo_key UNIQUE (titulo),
    CONSTRAINT curso_carga_horaria_check
        CHECK (carga_horaria > 0),
    CONSTRAINT curso_preco_check CHECK (preco >= 0)
);

CREATE TABLE pre_requisito (
    curso_id    INTEGER,
    exigido_id  INTEGER,
    CONSTRAINT pre_requisito_pkey
        PRIMARY KEY (curso_id, exigido_id),
    CONSTRAINT pre_requisito_curso_id_fkey
        FOREIGN KEY (curso_id) REFERENCES curso (id),
    CONSTRAINT pre_requisito_exigido_id_fkey
        FOREIGN KEY (exigido_id) REFERENCES curso (id),
    CONSTRAINT pre_requisito_distintos_check
        CHECK (curso_id <> exigido_id)
);
```

As duas chaves estrangeiras de `pre_requisito` apontam para a mesma tabela, `curso`, cada uma com o próprio nome. A `pre_requisito_distintos_check` impede que um curso seja registrado como pré-requisito de si mesmo: o `<>` quer dizer “diferente de”. Nenhuma regra do [capítulo 2](#) dizia isso, mas um curso que exigisse a si mesmo não poderia ser cursado por ninguém, e a escola confirmou a regra.

```
CREATE TABLE pessoa (
    id            INTEGER GENERATED ALWAYS AS IDENTITY,
    nome          VARCHAR(120) NOT NULL,
```

```

    cpf                VARCHAR(11) NOT NULL,
    email              VARCHAR(254) NOT NULL,
    data_nascimento    DATE NOT NULL,
    CONSTRAINT pessoa_pkey PRIMARY KEY (id),
    CONSTRAINT pessoa_cpf_key UNIQUE (cpf),
    CONSTRAINT pessoa_cpf_check
        CHECK (cpf SIMILAR TO '[0-9]{11}'))
);

CREATE TABLE telefone (
    pessoa_id INTEGER,
    numero    VARCHAR(11),
    CONSTRAINT telefone_pkey
        PRIMARY KEY (pessoa_id, numero),
    CONSTRAINT telefone_pessoa_id_fkey
        FOREIGN KEY (pessoa_id) REFERENCES pessoa (id)
        ON DELETE CASCADE,
    CONSTRAINT telefone_numero_check
        CHECK (numero SIMILAR TO '[0-9]{10,11}'))
);

CREATE TABLE aluno (
    id                INTEGER,
    escolaridade     VARCHAR(30) NOT NULL,
    CONSTRAINT aluno_pkey PRIMARY KEY (id),
    CONSTRAINT aluno_id_fkey
        FOREIGN KEY (id) REFERENCES pessoa (id)
        ON DELETE CASCADE
);

CREATE TABLE instrutor (
    id                INTEGER,
    data_admissao    DATE NOT NULL,
    valor_hora_aula   NUMERIC(6,2) NOT NULL,
    CONSTRAINT instrutor_pkey PRIMARY KEY (id),
    CONSTRAINT instrutor_id_fkey
        FOREIGN KEY (id) REFERENCES pessoa (id)
        ON DELETE CASCADE,
    CONSTRAINT instrutor_valor_hora_aula_check
        CHECK (valor_hora_aula > 0)
);

```

Na tabela `pessoa`, o CPF tem as duas restrições que a [seção 5.3](#) prometeu. A `pessoa_cpf_key` cobra a regra 5, pela qual não há duas pessoas com o mesmo CPF. A `pessoa_cpf_check` usa o `SIMILAR TO` que a [seção 5.6](#) apresentou na tabela `telefone`, agora com `{11}`: o CPF precisa ter exatamente 11 algarismos, e o CPF digitado com pontos ou hífen é recusado. A escolaridade do aluno é um texto livre, porque nenhuma regra lista os valores possíveis. Na tabela `instrutor`, a coluna `id` é, ao mesmo tempo, a chave primária e a chave estrangeira para `pessoa`, como na tabela `aluno`, e o valor da hora-aula precisa ser positivo.

```
CREATE TABLE turma (
    id                INTEGER,
    curso_id          INTEGER NOT NULL,
    instrutor_id      INTEGER NOT NULL,
    data_inicio       DATE NOT NULL,
    data_termino      DATE NOT NULL,
    vagas             INTEGER NOT NULL,
    sala              VARCHAR(30),
    endereco_sala_virtual VARCHAR(500),
    CONSTRAINT turma_pkey PRIMARY KEY (id),
    CONSTRAINT turma_curso_id_fkey
        FOREIGN KEY (curso_id) REFERENCES curso (id)
        ON DELETE RESTRICT,
    CONSTRAINT turma_instrutor_id_fkey
        FOREIGN KEY (instrutor_id)
        REFERENCES instrutor (id),
    CONSTRAINT turma_vagas_check CHECK (vagas > 0),
    CONSTRAINT turma_periodo_check
        CHECK (data_termino >= data_inicio),
    CONSTRAINT turma_modalidade_check CHECK (
        (sala IS NOT NULL
         AND endereco_sala_virtual IS NULL)
        OR (sala IS NULL
         AND endereco_sala_virtual IS NOT NULL)
    )
);

CREATE TABLE matricula (
    aluno_id          INTEGER,
    turma_id          INTEGER,
    data_matricula     DATE DEFAULT CURRENT_DATE NOT NULL,
    valor_cobrado      NUMERIC(8,2) NOT NULL,
    nota_final         NUMERIC(3,1),
    CONSTRAINT matricula_pkey
        PRIMARY KEY (aluno_id, turma_id),
    CONSTRAINT matricula_aluno_id_fkey
        FOREIGN KEY (aluno_id) REFERENCES aluno (id),
    CONSTRAINT matricula_turma_id_fkey
        FOREIGN KEY (turma_id) REFERENCES turma (id)
```

```

        ON DELETE RESTRICT,
    CONSTRAINT matricula_valor_cobrado_check
        CHECK (valor_cobrado >= 0),
    CONSTRAINT matricula_nota_final_check
        CHECK (nota_final BETWEEN 0 AND 10)
);

CREATE TABLE certificado (
    id                INTEGER GENERATED ALWAYS AS IDENTITY,
    codigo_verificacao VARCHAR(20) NOT NULL,
    data_emissao      DATE NOT NULL,
    aluno_id          INTEGER NOT NULL,
    turma_id          INTEGER NOT NULL,
    CONSTRAINT certificado_pkey PRIMARY KEY (id),
    CONSTRAINT certificado_codigo_verificacao_key
        UNIQUE (codigo_verificacao),
    CONSTRAINT certificado_aluno_id_turma_id_key
        UNIQUE (aluno_id, turma_id),
    CONSTRAINT certificado_aluno_id_turma_id_fkey
        FOREIGN KEY (aluno_id, turma_id)
        REFERENCES matricula (aluno_id, turma_id)
);

```

As tabelas turma e matricula são as mesmas da [seção 5.5](#). A tabela certificado traz a chave estrangeira composta da [seção 3.6](#): o par aluno_id e turma_id aponta, junto, para a chave primária da matricula, e as duas colunas aparecem juntas entre os parênteses dos dois lados. A certificado_aluno_id_turma_id_key declara o mesmo par como chave alternativa, e é ela que mantém o 1:1: sem essa restrição, duas linhas de certificado poderiam apontar para a mesma matrícula. O código de verificação é a outra chave alternativa, pela regra 9, e o tamanho de 20 caracteres deixa folga para a mudança de formato que a [seção 3.2](#) admitiu.

Com o modelo escrito, podemos cumprir a promessa da [seção 2.11](#) e ver, regra por regra, o que o banco passou a cobrar e o que ficou de fora:

Regra	Cobrada pelo banco	Não cobrada
1	curso_titulo_key; NOT NULL	—
2	chaves de pre_requisito	—
3	turma_curso_id_fkey; turma_pkey	—
4	turma_modalidade_check	—
5	pessoa_cpf_key; tabela telefone	pelo menos um telefone
6	aluno e instrutor apontam para pessoa	toda pessoa tem um papel
7	turma_instrutor_id_fkey; NOT NULL	—

Regra	Cobrada pelo banco	Não cobrada
8	matricula_pkey; matricula_nota_final_check	limite de vagas; valor cobrado igual ao preço do dia
9	chaves alternativas de certificado	nota mínima 7
10	ações referenciais	—
11	—	transferência para outra turma do mesmo curso

As regras que ficaram de fora têm um ponto em comum: para conferi-las, o banco precisaria olhar outras linhas além da que está sendo gravada. Para cobrar o limite de vagas, ele teria de contar as matrículas da turma; para cobrar a nota mínima, olhar a nota da matrícula ao gravar o certificado; para conferir o valor cobrado, consultar o preço do curso ao gravar a matrícula; para cobrar que toda pessoa tenha um telefone e um papel, procurar as linhas das tabelas `telefone`, `aluno` e `instructor` ao gravar a pessoa. E o `CHECK`, como vimos na [seção 5.5](#), só enxerga a linha que está sendo gravada.

O padrão SQL tem dois recursos para regras assim. O primeiro é a asserção, uma condição sobre o banco inteiro, declarada com o comando `CREATE ASSERTION`; o segundo é a consulta a outras tabelas dentro de um `CHECK`. O PostgreSQL não implementa nenhum dos dois, e a lista dos recursos do padrão que ele não implementa registra o segundo como omitido de propósito. No PostgreSQL, a saída é o **gatilho** (*trigger*), um procedimento que o SGBD executa automaticamente quando uma tabela é alterada e que pode conferir outras tabelas e recusar a alteração. Escrever um gatilho é programação, e não modelagem, e por isso o livro não vai além desta frase.

Isso não quer dizer que essas regras deixem de valer. Elas continuam escritas ao lado do modelo, como a [seção 2.11](#) recomendou, e quem grava os dados precisa cobrá-las, seja o programa da secretaria, seja um gatilho. A diferença é que, para essas regras, o banco não protege a escola sozinho, e quem modela precisa deixar isso registrado. A regra 11 é um caso à parte. A exigência do mesmo curso depende de outras linhas, como as demais regras da lista. Já a outra metade da regra diz que o aluno sai de uma turma e entra na outra, as duas coisas ou nenhuma, e é exatamente isso que uma transação garante, como o [capítulo 6](#) vai mostrar.

As nove tabelas existem, as restrições estão declaradas, e o banco está pronto para cobrar as regras. Só que ele ainda está vazio.

5.9 Resumo

Começamos pelo padrão SQL, o conjunto oficial de regras da linguagem, popularmente chamado de “SQL ANSI”. Vimos que seguir o padrão aumenta a chance de o modelo funcionar em outro SGBD, sem garanti-la, porque nenhum SGBD declara seguir por completo nem mesmo o núcleo do padrão, e que o PostgreSQL entra no livro como referência de sintaxe. Em seguida, agrupamos os comandos pelo que fazem: a DDL define as estruturas, a DML altera as linhas, a consulta lê as linhas, e os comandos de transação fazem o SGBD executar um grupo de operações por inteiro ou não executar nenhuma delas. Vimos também que essa divisão é didática.

Vimos que escolher o tipo de dado de uma coluna também é modelar, porque o tipo decide quais valores a coluna aceita. As tabelas da escola usam quatro tipos do padrão: `INTEGER`, `NUMERIC` para o dinheiro e a nota, que precisam de exatidão, `VARCHAR` com limites folgados e `DATE`. O CPF e o telefone ficaram em texto, só com os dígitos, porque são códigos, e não quantidades, e porque uma forma única de escrita é o que permite ao banco reconhecer um CPF repetido. Quando as regras não traziam o que era preciso saber para escolher o tipo, a resposta veio da escola, e não de quem modela.

Criamos a primeira tabela e vimos por que os nomes ficam em minúsculas, sem acento e sem aspas: sem aspas, o PostgreSQL converte o nome para minúsculas, e o padrão, para maiúsculas. Conhecemos a coluna de identidade, cujo valor o banco gera sozinho, e vimos que ela não impede a repetição. Depois, declaramos as restrições, cada uma ligada à regra que cobra: o `NOT NULL` escreve a cardinalidade mínima, a `PRIMARY KEY` e o `UNIQUE` declaram as chaves candidatas, a `FOREIGN KEY` cobra a integridade referencial, e o `CHECK` cobra condições sobre os valores da linha, como a regra 4, que coube inteira na tabela `turma`. O `DEFAULT` não recusa nada: ele define o valor que a coluna recebe quando o comando não informa nenhum.

Com as ações referenciais, a regra 10 virou SQL: os telefones e os papéis saem em cascata com a pessoa, e o curso com turmas e a turma com matrículas não podem ser excluídos. Vimos que as ações se encadeiam e que a recusa num ponto da cadeia recusa o comando inteiro. Vimos também que o modelo muda com o negócio, que o `ALTER TABLE` e o `DROP TABLE` acompanham essas mudanças, que o nome de cada restrição é o que permite removê-la e que o `CASCADE` do `DROP TABLE` remove as restrições que apontavam para a tabela removida, e não as outras tabelas nem as linhas delas.

Por fim, escrevemos as nove tabelas da escola, na ordem que as chaves estrangeiras impõem, e conferimos regra por regra o que o banco passou a cobrar. As regras que ficaram de fora dependem de outras linhas além da que está sendo gravada, e o `CHECK` não as alcança. Elas continuam valendo, escritas ao lado do modelo, e precisam ser cobradas por quem grava os dados.

No próximo capítulo, vamos pôr os dados da escola nessas tabelas e ver o modelo em uso.

6 SQL: o modelo em uso

“Todos os modelos estão errados, mas alguns são úteis.”

George E. P. Box (tradução livre)

No capítulo anterior, escrevemos as nove tabelas da escola em SQL e declaramos as regras que o banco cobra. Só que um banco vazio não responde nada. A escola quer saber quantos alunos há em cada turma, quais turmas ainda têm vaga, qual é a média das notas de cada curso e quem ainda está sem nota. Neste capítulo, vamos pôr os dados da escola nas tabelas e fazer essas perguntas ao banco. Se o modelo responder a todas, ele atende ao negócio; se alguma ficar sem resposta porque o dado de que ela precisa não está no banco, o problema está no modelo, e não na consulta.

A epígrafe é do estatístico George Box, que a escreveu em 1979 pensando em modelos estatísticos, mas ela vale para qualquer modelo, inclusive o de dados. Para Box, nenhum modelo simples representa com exatidão um sistema do mundo real. O modelo da escola também é uma simplificação: ele guarda o que as regras de negócio pediram e deixa de fora todo o resto. Por isso, a pergunta que importa não é se o modelo reproduz a escola por inteiro, e sim se ele é útil, e um modelo de dados é útil quando responde às perguntas do negócio. É isso que vamos conferir, com o mínimo de SQL necessário, porque este não é um curso de SQL. Como nos capítulos anteriores, você não precisa executar nada: o resultado de cada consulta aparece no texto, logo depois do comando.

O que será visto neste capítulo:

- [Inserindo linhas](#)
- [Consultando](#)
- [Juntando tabelas](#)
- [Resumindo dados](#)
- [Alterando e excluindo linhas](#)
- [Transações](#)
- [Dados derivados e visões](#)
- [Resumo](#)

6.1 Inserindo linhas

Como os três cursos da escola entram na tabela `curso`? Com o `INSERT`, o comando que insere linhas numa tabela. É o primeiro comando da DML que a [seção 5.2](#) apresentou, e é com ele que o banco deixa de estar vazio. Veja o primeiro curso:

```
INSERT INTO curso (titulo, ementa, carga_horaria, preco)
VALUES (
    'Lógica de Programação',
    'Algoritmos, variáveis, decisões e repetições.',
    40,
    480.00
);
```

O comando diz ao SGBD: insira na tabela `curso` uma linha com estes valores, nestas colunas. A lista entre os parênteses depois do nome da tabela diz quais colunas recebem valor, e os valores depois de `VALUES` vêm na mesma ordem: o primeiro vai para `titulo`, o segundo para `ementa`, e assim por diante. Com a lista escrita, o comando não depende da ordem em que as colunas foram criadas: ele continua certo mesmo que alguém acrescenta uma coluna à tabela, como a [seção 5.7](#) imaginou.

Repare que a coluna `id` não está na lista. Ela é a coluna de identidade da [seção 5.4](#), e quem dá o valor dela é o banco: este primeiro curso recebe o `id` 1. Como o valor vem da ordem de inserção, os cursos precisam entrar na ordem do estudo de caso para que Python Essencial fique com o `id` 2, e Modelagem de Dados, com o 3.

Cada tipo de valor tem a sua forma de escrita. O texto vai entre apóstrofes, como `'Lógica de Programação'`. O número vai sem apóstrofes, e as casas decimais vêm depois de um ponto, e não de uma vírgula, porque a vírgula separa um valor do outro. O PostgreSQL aceitaria a carga horária escrita como `'40'`, mas escrever um número entre apóstrofes confunde texto com número, e o livro não faz isso. A data, que aparece logo adiante, é escrita com a palavra `DATE` seguida da data entre apóstrofes, no formato ano, mês e dia: `DATE '2026-08-03'` é o dia 3 de agosto de 2026.

Os outros dois cursos entram do mesmo jeito:

```
INSERT INTO curso (titulo, ementa, carga_horaria, preco)
VALUES (
    'Python Essencial',
    'Python do zero: tipos, funções, coleções e arquivos.',
    60,
    720.00
);

INSERT INTO curso (titulo, ementa, carga_horaria, preco)
VALUES (
    'Modelagem de Dados',
```

```

    'Modelos conceitual e relacional, normalização e SQL.',
    40,
    560.00
);

```

Um mesmo INSERT também pode inserir várias linhas: basta escrever, depois de VALUES, uma lista de valores entre parênteses para cada linha, separando as listas por vírgula. É assim que entram os pré-requisitos, as pessoas e os telefones:

```

INSERT INTO pre_requisito (curso_id, exigido_id)
VALUES
    (2, 1),
    (3, 1);

INSERT INTO pessoa (nome, cpf, email, data_nascimento)
VALUES
    ('Ana Lima', '11111111111',
     'ana.lima@example.com', DATE '1998-04-12'),
    ('Bruno Costa', '22222222222',
     'bruno.costa@example.com', DATE '2001-09-30'),
    ('Carla Souza', '33333333333',
     'carla.souza@example.com', DATE '1995-01-22'),
    ('Marta Ribeiro', '44444444444',
     'marta.ribeiro@example.com', DATE '1988-06-15'),
    ('Paulo Mendes', '55555555555',
     'paulo.mendes@example.com', DATE '1983-11-03');

INSERT INTO telefone (pessoa_id, numero)
VALUES
    (1, '10911111111'),
    (1, '10311111111'),
    (2, '10922222222'),
    (3, '10933333333'),
    (4, '10944444444'),
    (5, '10955555555');

```

A ordem dos comandos é a mesma dos CREATE TABLE da [seção 5.8](#), e pelo mesmo motivo: uma chave estrangeira só aceita um valor que já existe na tabela referenciada. Os pré-requisitos só entram depois dos cursos, e os telefones só depois das pessoas. As pessoas receberam os valores de id de 1 a 5, na ordem em que aparecem no comando, e são esses valores que as tabelas aluno e instrutor repetem, porque o aluno e o instrutor são a própria pessoa:

```

INSERT INTO aluno (id, escolaridade)
VALUES
    (1, 'superior completo'),
    (2, 'médio completo'),

```

```

(3, 'superior incompleto');

INSERT INTO instrutor (id, data_admissao, valor_hora_aula)
VALUES
(4, DATE '2023-02-01', 90.00),
(5, DATE '2021-07-01', 110.00);

```

Faltam as turmas, as matrículas e o certificado:

```

INSERT INTO turma (id, curso_id, instrutor_id, data_inicio,
data_termino, vagas, sala, endereco_sala_virtual)
VALUES
(10, 2, 4, DATE '2026-08-03', DATE '2026-11-27', 30,
NULL, 'https://aula.example.com/turma-10'),
(11, 3, 5, DATE '2026-08-17', DATE '2026-09-25', 20,
'Sala 2', NULL);

INSERT INTO matricula (aluno_id, turma_id, data_matricula,
valor_cobrado, nota_final)
VALUES
(1, 10, DATE '2026-07-20', 720.00, NULL),
(2, 10, DATE '2026-07-06', 690.00, NULL),
(1, 11, DATE '2026-08-03', 560.00, 8.5),
(3, 11, DATE '2026-08-10', 560.00, 6.0);

INSERT INTO certificado (codigo_verificacao, data_emissao,
aluno_id, turma_id)
VALUES ('K7Q2-M9XA', DATE '2026-09-30', 1, 11);

```

No INSERT das turmas, o id é informado, porque é o código que a escola deu a cada turma. A palavra NULL registra que o dado está faltando: a turma 10 é on-line e não tem sala, e a turma 11 é presencial e não tem endereço de sala virtual. Nas matrículas, o NULL marca as notas que ainda não foram lançadas. O INSERT das matrículas informa a data, porque elas foram fechadas em julho e em agosto. No INSERT de uma matrícula fechada hoje, a coluna data_matricula poderia ficar fora da lista, e o valor padrão da [seção 5.5](#) poria nela a data do dia. O certificado recebe o id 1 do banco e aponta para a matrícula da Ana na turma 11 pelo par aluno_id e turma_id.

Com a carga completa, o banco guarda exatamente os dados do estudo de caso. Agora podemos cumprir a promessa do [capítulo 5](#) e ver o banco recusar dados que ferem as regras declaradas nele. Imagine que uma atendente, sem saber que a Ana já estava cadastrada, tentasse cadastrá-la de novo:

```

-- ERRO ESPERADO: o CPF 11111111111 já é da Ana
INSERT INTO pessoa (nome, cpf, email, data_nascimento)

```

```
VALUES ('Ana Lima', '1111111111',  
       'ana.lima@example.com', DATE '1998-04-12');
```

O banco recusaria o comando por causa da restrição `pessoa_cpf_key`, que cobra a regra 5: não há duas pessoas com o mesmo CPF. A coluna `id` não ajudaria a perceber a repetição, porque o banco daria a esse segundo cadastro um `id` novo; quem percebe a repetição é a chave alternativa, como a [seção 3.2](#) explicou. Agora, uma segunda matrícula da Ana na turma 10:

```
-- ERRO ESPERADO: a Ana já está matriculada na turma 10  
INSERT INTO matricula (aluno_id, turma_id, data_matricula,  
                      valor_cobrado)  
VALUES (1, 10, DATE '2026-07-21', 720.00);
```

A coluna `nota_final` ficou fora da lista, e por isso a linha receberia `NULL` nela. Mas a linha não entra: o par (1, 10) já existe, e a chave primária `matricula_pkey` cobra a regra 8, pela qual ninguém se matricula duas vezes na mesma turma. Os dois comandos seguintes tentam cadastrar uma turma 12:

```
-- ERRO ESPERADO: o curso 9 não existe  
INSERT INTO turma (id, curso_id, instrutor_id, data_inicio,  
                  data_termino, vagas, sala)  
VALUES (12, 9, 5, DATE '2026-09-14', DATE '2026-12-04', 20,  
       'Sala 1');  
  
-- ERRO ESPERADO: turma presencial e on-line ao mesmo tempo  
INSERT INTO turma (id, curso_id, instrutor_id, data_inicio,  
                  data_termino, vagas, sala, endereco_sala_virtual)  
VALUES (12, 2, 5, DATE '2026-09-14', DATE '2026-12-04', 20,  
       'Sala 1', 'https://aula.example.com/turma-12');
```

O primeiro é o cadastro que a [Figura 3.1](#) mostrou recusado: a restrição `turma_curso_id_fkey` não encontra o curso 9. O segundo aponta para um curso que existe, mas preenche a sala e o endereço da sala virtual, e a restrição `turma_modalidade_check` cobra a regra 4: toda turma é presencial ou on-line, nunca as duas.

Nos quatro casos, nada é gravado. E se um `INSERT` de várias linhas tiver uma linha só que fira uma restrição? O banco recusa o comando inteiro, e nenhuma das linhas entra, nem as que estavam certas. É a mesma propriedade que a [seção 5.6](#) mostrou na exclusão da Ana: o SGBD executa cada comando por inteiro ou não o executa.

Mas o banco só recusa o que fere uma regra declarada nele, e nem toda regra de negócio está declarada. A tabela da [seção 5.8](#) listou as regras que ficaram de fora, e uma delas é a nota mínima do certificado. Um certificado para a matrícula da Carla na turma 11, com nota 6.0, seria aceito sem reclamação, porque nenhuma restrição da tabela `certificado` olha a nota da matrícula. As restrições cobram as regras que foram declaradas, e só essas.

Com os dados no lugar, podemos começar a fazer perguntas.

6.2 Consultando

Quais cursos a escola oferece, em ordem alfabética, e quanto custa cada um? A resposta está na tabela `curso`, e quem a busca é o `SELECT`, o comando de consulta da [seção 5.2](#). Ele lê as linhas sem alterar nenhuma:

```
SELECT
    titulo,
    preco
FROM curso
ORDER BY titulo;
```

Resultado:

titulo	preco
Lógica de Programação	480.00
Modelagem de Dados	560.00
Python Essencial	720.00

O comando tem três partes, que se chamam cláusulas. O `SELECT` diz quais colunas o resultado mostra; o `FROM` diz de qual tabela elas vêm; e o `ORDER BY` diz em que ordem as linhas aparecem. O resultado de uma consulta também é uma tabela, com as colunas pedidas e uma linha para cada linha encontrada.

O `ORDER BY` cumpre uma promessa da [seção 3.1](#). A ordem das linhas de uma tabela não tem significado, e por isso o banco não promete ordem nenhuma quando a consulta não a pede: a documentação do PostgreSQL diz que, sem o `ORDER BY`, as linhas vêm na ordem que o SGBD achar mais rápida de produzir. Hoje, essa ordem pode até coincidir com a ordem de inserção, mas nada garante que continue assim. Se a ordem importa para quem lê o resultado, ela é pedida na consulta. Por isso, toda consulta deste capítulo que devolve mais de uma linha termina com um `ORDER BY`.

Quase sempre, a pergunta não é sobre todas as linhas, e sim sobre algumas. Quais matrículas foram feitas em agosto de 2026? Para escolher as linhas, a consulta ganha a cláusula `WHERE`, com uma condição:

```
SELECT
    aluno_id,
    turma_id,
    data_matricula
FROM matricula
WHERE data_matricula >= DATE '2026-08-01'
      AND data_matricula <= DATE '2026-08-31'
ORDER BY data_matricula;
```

Resultado:

aluno_id	turma_id	data_matricula
1	11	2026-08-03
3	11	2026-08-10

O banco confere a condição em cada linha da tabela e fica só com as linhas em que ela é verdadeira: as matrículas da Ana e da Carla na turma 11. As da turma 10 foram feitas em julho e ficam de fora. Das cláusulas, só o **WHERE** expressa uma condição; as outras dizem o que mostrar, de onde ler e em que ordem.

A condição usa os **operadores de comparação** (*comparison operators*), que comparam dois valores: = (igual), <> (diferente), < (menor), > (maior), <= (menor ou igual) e >= (maior ou igual). Você vai encontrar quem os chame de operadores “relacionais”, mas, em banco de dados, “relacional” já tem outro sentido, o do modelo relacional, e o livro fica com o nome que diz o que eles fazem. As comparações se combinam com três operadores lógicos: o **AND**, que exige as duas condições, como no exemplo; o **OR**, que se contenta com uma delas; e o **NOT**, que inverte uma condição.

Agora vem o efeito do **NULL** que a [seção 3.4](#) anunciou. Quais matrículas têm nota a partir de 7? E quais têm nota abaixo de 7? Parecem duas perguntas que, juntas, cobrem todas as matrículas. Vamos juntá-las com um **OR**:

```
SELECT
    aluno_id,
    turma_id,
    nota_final
FROM matricula
WHERE nota_final >= 7
    OR nota_final < 7
ORDER BY aluno_id;
```

Resultado:

aluno_id	turma_id	nota_final
1	11	8.5
3	11	6.0

Notou que faltam linhas? As matrículas da Ana e do Bruno na turma 10 não aparecem, embora a condição pareça cobrir qualquer nota. O motivo é que a comparação com **NULL** não dá verdadeiro nem falso. A nota da Ana na turma 10 ainda não existe, e a pergunta “essa nota é maior ou igual a 7?” não tem resposta: o resultado é desconhecido. O SQL trabalha com uma **lógica de três valores** (*three-valued logic*), em que uma condição pode ser verdadeira, falsa ou desconhecida, e o **WHERE** só fica com as linhas em que ela é verdadeira.

Os operadores lógicos também trabalham com os três valores. O OR dá verdadeiro se um dos lados for verdadeiro; sem nenhum lado verdadeiro, basta um lado desconhecido para o resultado ficar desconhecido, como nas matrículas da turma 10, em que os dois lados são desconhecidos. O AND dá falso se um dos lados for falso; sem nenhum lado falso, basta um lado desconhecido para o resultado ficar desconhecido. E o NOT de um valor desconhecido continua desconhecido. Por isso, nem NOT (nota_final >= 7) traz as matrículas sem nota: inverter uma resposta que não existe não produz uma resposta.

Para encontrar a falta de valor, o SQL tem um teste próprio, o IS NULL, que já apareceu na restrição turma_modalidade_check da [seção 5.5](#). Ele responde sempre com verdadeiro ou falso, nunca com desconhecido. Com esse teste, respondemos a uma das perguntas do negócio: quem ainda está sem nota?

```
SELECT
    aluno_id,
    turma_id
FROM matricula
WHERE nota_final IS NULL
ORDER BY aluno_id;
```

Resultado:

aluno_id	turma_id
1	10
2	10

São a Ana e o Bruno, na turma 10, que ainda está em andamento.

Atenção

Não escreva `nota_final = NULL` para procurar as notas que faltam. A comparação com NULL dá desconhecido em todas as linhas, inclusive nas que não têm nota, e a consulta devolve um resultado vazio, sem erro nenhum. Para perguntar se o valor falta, use `IS NULL`; para perguntar se ele existe, `IS NOT NULL`.

O resultado responde à pergunta, mas mostra números em vez de nomes. O nome da Ana está na tabela `pessoa`, e a consulta leu só a `matricula`. Para usar as duas tabelas na mesma consulta, precisamos juntá-las.

6.3 Juntando tabelas

Quem são, pelo nome, os alunos que ainda estão sem nota? A seção anterior encontrou as matrículas, mas a tabela `matricula` guarda só o `aluno_id`; o nome fica na tabela `pessoa`. A resposta precisa das duas tabelas, e é para isso que serve a **junção** (*join*): a operação que combina as linhas de duas tabelas, ligando cada linha de uma às linhas da outra que atendem a uma condição. Quase sempre, a condição é a igualdade entre uma chave estrangeira e a chave que ela referencia.

```
SELECT
    p.nome,
    m.turma_id
FROM matricula AS m
    JOIN pessoa AS p ON p.id = m.aluno_id
WHERE m.nota_final IS NULL
ORDER BY p.nome;
```

Resultado:

nome	turma_id
Ana Lima	10
Bruno Costa	10

O JOIN diz com qual tabela juntar, e o ON diz a condição. Para cada linha de `matricula`, o banco procura em `pessoa` a linha cujo `id` é igual ao `aluno_id` da matrícula e monta uma linha nova, com as colunas das duas. A matrícula da Ana na turma 10 tem o `aluno_id` 1 e encontra a linha da Ana; a do Bruno tem o 2 e encontra a linha do Bruno. Depois disso, o WHERE fica só com as matrículas sem nota, como na seção anterior.

A junção vai direto de `matricula` para `pessoa`, sem passar pela tabela `aluno`. Isso funciona por causa de uma decisão da [seção 3.7](#): o `id` do aluno é o mesmo `id` da pessoa, e por isso o `aluno_id` da matrícula serve para encontrar a linha da pessoa. A tabela `aluno` só entraria na consulta se a pergunta envolvesse a escolaridade.

Repare também nas letras depois de AS. São apelidos que a consulta dá às tabelas: `m` para `matricula` e `p` para `pessoa`. Numa junção, uma coluna com o mesmo nome pode existir nas duas tabelas, como a `id`, e por isso cada coluna é escrita com o nome da tabela na frente, separado por um ponto. O apelido encurta essa escrita: `p.nome` é a coluna `nome` da tabela `pessoa`.

Uma consulta pode juntar quantas tabelas a pergunta pedir, e aqui o [capítulo 4](#) deixou uma promessa. A normalização separou a planilha da secretaria em tabelas, e a [seção 4.4](#) garantiu que nada se perdia na divisão, porque juntar as tabelas pelas colunas que elas têm em comum remontava a planilha. Vamos remontá-la:

```

SELECT
    p.nome,
    m.turma_id,
    c.titulo,
    m.valor_cobrado
FROM matricula AS m
    JOIN pessoa AS p ON p.id = m.aluno_id
    JOIN turma AS t ON t.id = m.turma_id
    JOIN curso AS c ON c.id = t.curso_id
ORDER BY m.turma_id, p.nome;

```

Resultado:

nome	turma_id	titulo	valor_cobrado
Ana Lima	10	Python Essencial	720.00
Bruno Costa	10	Python Essencial	690.00
Ana Lima	11	Modelagem de Dados	560.00
Carla Souza	11	Modelagem de Dados	560.00

São as quatro linhas da planilha, na mesma ordem. A consulta mostra só quatro colunas para o resultado caber na página, mas o CPF, o e-mail e o preço do curso viriam do mesmo jeito, bastando acrescentá-los ao SELECT.

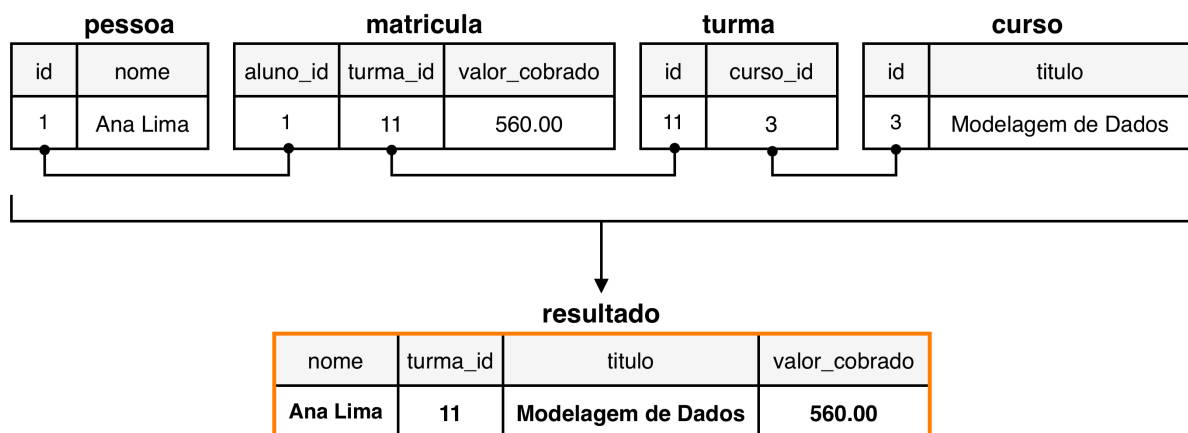


Figura 6.1: A matrícula da Ana na turma 11 encontra a linha da Ana em pessoa, a linha da turma 11 e, pela turma, a linha do curso de Modelagem de Dados; as quatro linhas formam uma linha só do resultado.

Na [Figura 6.1](#) temos o caminho de uma das linhas do resultado. A matrícula da Ana na turma 11 tem o `aluno_id` 1, que leva à linha dela em `pessoa`, e o `turma_id` 11, que leva à linha da turma. A turma, por sua vez, tem o `curso_id` 3, que leva ao curso de Modelagem de Dados. Cada ligação é uma chave estrangeira encontrando a chave que ela referencia, e as quatro linhas viram uma linha só do resultado. Repare que o título do curso não está guardado em lugar nenhum da matrícula: ele é buscado pela turma no momento da consulta. É por isso que a mudança do título de um curso, feita numa linha só da tabela `curso`, aparece em todas as consultas a partir de então.

Os telefones ficaram fora da consulta por um bom motivo. A Ana tem dois, e uma junção com a tabela `telefone` encontraria duas linhas para cada matrícula dela: o resultado repetiria as matrículas da Ana, uma vez com cada telefone. A planilha resolvia isso pondo os dois números numa célula só, justamente o que a [seção 4.3](#) proibiu. Se a secretaria quiser os telefones, a consulta mais clara é uma à parte, sobre `pessoa` e `telefone`.

A junção também responde a uma pergunta sobre o autorrelacionamento do pré-requisito: quais cursos cada curso exige, pelo título? A tabela `pre_requisito` tem duas chaves estrangeiras para a mesma tabela, `curso_id` e `exigido_id`, e a consulta precisa ler a tabela `curso` duas vezes, uma para cada papel:

```
SELECT
    c.titulo AS curso,
    e.titulo AS exige
FROM pre_requisito AS pr
    JOIN curso AS c ON c.id = pr.curso_id
    JOIN curso AS e ON e.id = pr.exigido_id
ORDER BY c.titulo;
```

Resultado:

curso	exige
Modelagem de Dados	Lógica de Programação
Python Essencial	Lógica de Programação

Aqui, o apelido deixa de ser um atalho e passa a ser necessário. A tabela `curso` aparece duas vezes na consulta, uma no papel de curso que exige e outra no papel de curso exigido, e só o apelido diz de qual das duas vem cada `titulo`: `c.titulo` é o título do curso que exige, e `e.titulo`, o do curso exigido. O `AS` também aparece depois das colunas, e ali ele dá o nome da coluna no resultado.

E as turmas de cada curso? Esta consulta parece simples, mas esconde uma armadilha:

```
SELECT
    c.titulo,
    t.id AS turma
FROM curso AS c
```

```
JOIN turma AS t ON t.curso_id = c.id
ORDER BY c.titulo;
```

Resultado:

titulo	turma
Modelagem de Dados	11
Python Essencial	10

Lógica de Programação sumiu. A junção monta linhas só com os pares que se correspondem, e esse curso não tem turma nenhuma, situação que a regra 3 permite. Se a pergunta era “quais cursos a escola tem e quais são as turmas de cada um”, a resposta está errada: um curso do catálogo ficou de fora. Para manter no resultado as linhas sem correspondente, existe a **junção externa** (*outer join*), a junção que mantém também as linhas de uma das tabelas que não encontram par na outra, preenchendo com NULL as colunas que vêm da outra tabela:

```
SELECT
    c.titulo,
    t.id AS turma
FROM curso AS c
    LEFT JOIN turma AS t ON t.curso_id = c.id
ORDER BY c.titulo;
```

Resultado:

titulo	turma
Lógica de Programação	NULL
Modelagem de Dados	11
Python Essencial	10

O LEFT diz que a tabela mantida é a que está à esquerda do JOIN, a `curso`. Lógica de Programação volta, com NULL no lugar da turma. É o curso que a planilha da secretaria não tinha onde guardar, a anomalia de inserção da [seção 4.1](#): nas tabelas normalizadas, ele existe na tabela `curso` mesmo sem turma, e a junção externa o mostra. Escolher entre a junção e a junção externa é responder a uma pergunta de negócio: as linhas sem correspondente fazem parte da resposta ou não?

Com as tabelas juntas, as perguntas que faltam pedem outra coisa: não as linhas, mas totais e médias tirados delas.

6.4 Resumindo dados

Quantos alunos há em cada turma? A resposta não é uma lista de matrículas, e sim um número para cada turma. Para calcular cada número, o banco precisa reunir as matrículas da mesma turma e contá-las. Quem faz a conta é uma **função de agregação** (*aggregate function*), uma função que recebe os valores de várias linhas e devolve um valor só; quem diz quais linhas formam cada grupo é a cláusula `GROUP BY`.

```
SELECT
    turma_id,
    COUNT(*) AS alunos
FROM matricula
GROUP BY turma_id
ORDER BY turma_id;
```

Resultado:

turma_id	alunos
10	2
11	2

Na [Figura 6.2](#) temos o que o `GROUP BY turma_id` faz com as quatro matrículas. O banco separa as linhas em grupos, um para cada valor de `turma_id`, e cada grupo vira uma linha só do resultado. O `COUNT(*)` conta as linhas de cada grupo: duas na turma 10, duas na turma 11. É a resposta à pergunta da escola: dois alunos em cada turma.

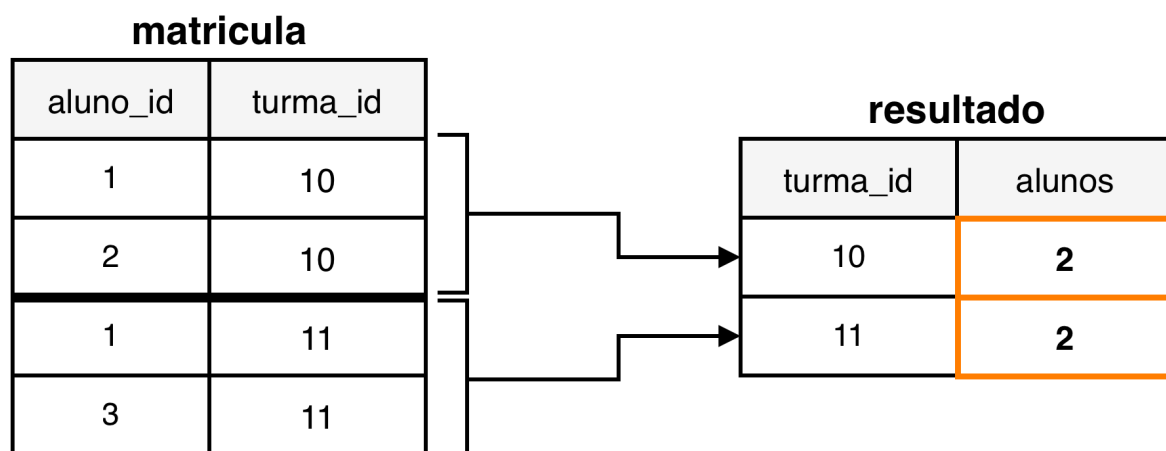


Figura 6.2: As quatro matrículas são separadas em dois grupos pela turma, e cada grupo vira uma linha do resultado, com a quantidade de matrículas do grupo.

Como cada grupo vira uma linha só, há uma regra para as colunas do SELECT: cada uma precisa estar no GROUP BY ou dentro de uma função de agregação. A `turma_id` pode aparecer, porque é igual em todas as linhas do grupo. A `aluno_id` não pode aparecer sozinha: o grupo da turma 10 tem os alunos 1 e 2, e a linha do resultado só tem lugar para um valor. Qual dos dois o banco mostraria? Como não há resposta, o banco recusa a consulta.

Outra pergunta do negócio: qual é a média das notas de cada curso? A nota está na `matricula`, e o curso de cada matrícula se descobre pela `turma`; a consulta junta as três tabelas, como na seção anterior, e agrupa pelo título do curso:

```
SELECT
  c.titulo,
  COUNT(*) AS matriculas,
  COUNT(m.nota_final) AS notas,
  CAST(AVG(m.nota_final) AS NUMERIC(4,2)) AS media
FROM matricula AS m
  JOIN turma AS t ON t.id = m.turma_id
  JOIN curso AS c ON c.id = t.curso_id
GROUP BY c.titulo
ORDER BY c.titulo;
```

Resultado:

titulo	matriculas	notas	media
Modelagem de Dados	2	2	7.25
Python Essencial	2	0	NULL

O AVG calcula a média. Em Modelagem de Dados, a média das notas da Ana e da Carla, 8.5 e 6.0, é 7.25. A média pode ter mais casas decimais que as notas, e o CAST, que converte um valor para outro tipo, a apresenta com duas casas.

Python Essencial é o caso interessante. As duas matrículas existem, e o COUNT(*) as conta, mas nenhuma tem nota. O COUNT(m.nota_final) conta só as linhas em que a nota tem valor, e dá zero. E o AVG ignora o NULL: ele calcula a média só das notas que existem, e, sem nenhuma nota, o resultado é NULL. É a resposta certa. A média das notas de Python Essencial ainda não existe, porque nenhuma nota foi lançada, e o banco diz exatamente isso. Lembra-se do aviso da [seção 3.4](#)? Se a escola guardasse 0 nas notas ainda não lançadas, esta consulta mostraria média zero para Python Essencial, e a média de uma turma com parte das notas lançadas cairia por causa das notas que ainda não existem.

Repare também no agrupamento pelo título. Ele funciona porque não há dois cursos com o mesmo título, pela regra 1; se houvesse, as matrículas dos dois cursos cairiam no mesmo grupo, e a média misturaria cursos diferentes. Quando cada grupo deve corresponder a uma coisa do negócio, como um curso, a coluna do agrupamento precisa identificar essa coisa, e quem garante isso é uma chave. Pelo mesmo motivo, uma consulta que agrupasse as matrículas pelo nome do aluno juntaria dois alunos homônimos num aluno só.

Com essas consultas, o modelo respondeu a três das quatro perguntas da escola. A que falta, sobre as vagas, exige um cuidado a mais e fica para a [seção 6.7](#). Antes, precisamos ver como as linhas mudam.

6.5 Alterando e excluindo linhas

O que acontece no banco quando a turma 10 terminar e a nota da Ana for lançada? A linha da matrícula dela já existe, com NULL na coluna `nota_final`, e não se trata de inserir uma linha nova: é preciso alterar a que existe. Quem faz isso é o UPDATE, e quem exclui linhas é o DELETE. Os exemplos desta seção são hipóteses, como as da [seção 5.7](#): nas seções seguintes, os dados continuam os da carga da [seção 6.1](#).

Imagine, então, que a turma 10 terminasse e que a Ana tirasse 9:

```
UPDATE matricula
SET nota_final = 9.0
WHERE aluno_id = 1
      AND turma_id = 10;
```

O UPDATE diz qual tabela muda; o SET, qual coluna recebe qual valor; e o WHERE, em quais linhas. A condição usa a chave primária da `matricula`, que é composta, e por isso as duas colunas aparecem: o `aluno_id` sozinho alcançaria as duas matrículas da Ana, e o `turma_id` sozinho, as matrículas da Ana e do Bruno na turma 10. Com as duas, a condição alcança exatamente uma linha.

É por isso que a alteração filtra pela chave. A chave é o que identifica uma linha, como a [seção 3.2](#) mostrou, e uma condição sobre ela alcança uma linha e só uma. Uma condição sobre o nome não tem essa garantia: se a escola tivesse duas alunas chamadas Ana Lima, um UPDATE com `WHERE nome = 'Ana Lima'` alteraria as duas. A chave primária serve, e as chaves alternativas também: o CPF identifica uma pessoa, e o título identifica um curso.

Outro caso. Imagine que a escola reajustasse o preço de Python Essencial para 760 reais:

```
UPDATE curso
SET preco = 760.00
WHERE id = 2;
```

Uma linha mudaria, e o preço novo valeria dali em diante para todas as consultas. E as matrículas da Ana e do Bruno em Python Essencial? Continuariam com 720.00 e 690.00, porque o valor cobrado não é uma cópia do preço, e sim o preço do dia da matrícula, como a [seção 4.5](#) mostrou. É o modelo funcionando: o reajuste altera o preço do catálogo sem reescrever o que cada aluno pagou.

Agora, uma exclusão. Imagine que a Ana cancelasse o telefone fixo:

```
DELETE FROM telefone
WHERE pessoa_id = 1
AND numero = '1031111111';
```

O DELETE diz de qual tabela excluir, e o WHERE, quais linhas. Aqui também a condição usa a chave primária, composta pela pessoa e pelo número. A Ana ficaria só com o celular, e a regra 5, que exige pelo menos um telefone, continuaria atendida. Se a Ana tivesse um telefone só e ele fosse excluído, o banco aceitaria a exclusão do mesmo jeito: essa parte da regra 5 está na lista do que o banco não cobra, na [seção 5.8](#).

As ações referenciais da [seção 5.6](#) valem para o DELETE, e é aqui que a recusa que aquela seção descreveu aparece num comando:

```
-- ERRO ESPERADO: a turma 10 é do curso Python Essencial
DELETE FROM curso
WHERE id = 2;
```

A chave estrangeira `turma_curso_id_fkey` foi declarada com `ON DELETE RESTRICT`, e a turma 10 aponta para o curso 2. O banco recusa a exclusão, como pede a regra 10: curso com turmas não pode ser excluído.

Atenção

O WHERE é o que limita o UPDATE e o DELETE, e sem ele os dois comandos alcançam todas as linhas da tabela. Um `UPDATE matricula SET nota_final = 9.0` sem WHERE daria nota 9 a todas as matrículas, inclusive às da turma 11, cujas notas já tinham sido lançadas; um `DELETE FROM telefone` sem WHERE apagaria os telefones de todas as pessoas. O banco não pede confirmação. Antes de alterar ou excluir, escreva a condição pela chave e confira, com um SELECT com o mesmo WHERE, quais linhas ela alcança.

Alterar e excluir linhas, um comando de cada vez, já está resolvido. Mas a regra 11 pede mais que isso: dois comandos que precisam acontecer juntos.

6.6 Transações

Como o banco transfere um aluno de turma? Pela regra 11, o aluno pode ser transferido para outra turma do mesmo curso: sai de uma e entra na outra, as duas coisas ou nenhuma. O [capítulo 1](#) apresentou a transação justamente com esse exemplo, e a [seção 5.8](#) deixou para este capítulo os comandos que a controlam. Para que haja uma transferência possível, precisamos de uma segunda turma de Python Essencial. Imagine que a escola abra a turma 12, presencial:

```
INSERT INTO turma (id, curso_id, instrutor_id, data_inicio,
                  data_termino, vagas, sala)
```

```
VALUES (12, 2, 5, DATE '2026-09-14', DATE '2026-12-04', 20,  
        'Sala 1');
```

É o mesmo código da turma recusada na [seção 6.1](#), mas a recusa não gravou nada, e o código continua livre. Esta turma aponta para Python Essencial, um curso que existe, e preenche só a sala, sem o endereço de sala virtual, que fica com NULL. Agora, imagine que o Bruno pedisse para sair da turma 10, on-line, e passar para a 12. A transferência tem dois movimentos: a matrícula na turma 10 sai, e uma matrícula na turma 12 entra. Perguntamos à escola o que a matrícula nova guardaria, e a resposta foi que ela conservaria a data e o valor cobrado da original, porque o Bruno já tinha pagado e o acordo era o mesmo; só a turma mudaria.

```
START TRANSACTION;  
  
DELETE FROM matricula  
WHERE aluno_id = 2  
      AND turma_id = 10;  
  
INSERT INTO matricula (aluno_id, turma_id, data_matricula,  
                       valor_cobrado)  
VALUES (2, 12, DATE '2026-07-06', 690.00);  
  
COMMIT;
```

O `START TRANSACTION` abre a transação, e os comandos seguintes passam a fazer parte dela. O `COMMIT` a encerra confirmando tudo o que foi feito: só nesse momento a transferência passa a valer. Se os dois comandos dessem certo, o Bruno terminaria matriculado na turma 12, com a data e o valor de antes, e sem matrícula na turma 10.

E se algo der errado no meio do caminho? Imagine que, no lugar da transferência anterior, alguém digitasse 21 em vez de 12 no `INSERT`:

```
-- ERRO ESPERADO: a turma 21 não existe  
START TRANSACTION;  
  
DELETE FROM matricula  
WHERE aluno_id = 2  
      AND turma_id = 10;  
  
INSERT INTO matricula (aluno_id, turma_id, data_matricula,  
                       valor_cobrado)  
VALUES (2, 21, DATE '2026-07-06', 690.00);  
  
ROLLBACK;
```

O DELETE daria certo, e a matrícula do Bruno na turma 10 sairia. O INSERT seria recusado, porque a chave estrangeira não encontraria a turma 21. Se cada comando valesse por conta própria, o Bruno ficaria sem turma nenhuma, exatamente o problema que o [capítulo 1](#) descreveu. Mas os dois comandos estariam na mesma transação, e o programa que pedisse a transferência, ao receber a recusa, encerraria a transação com o ROLLBACK, que desfaz tudo o que a transação tinha feito. A matrícula do Bruno na turma 10 voltaria a existir, como se nada tivesse acontecido.

E fora de uma transação aberta com o START TRANSACTION? No PostgreSQL, cada comando executado fora dela forma uma transação sozinho e é confirmado logo que termina. Foi o que aconteceu com todos os comandos deste capítulo até aqui. Por isso, um comando avulso já é executado por inteiro ou não é executado, como vimos na [seção 5.6](#) e na [seção 6.1](#). Neste modelo, aliás, a transferência caberia num comando só: um UPDATE que trocasse o turma_id da matrícula do Bruno de 10 para 12 faria os dois movimentos de uma vez, e também seria tudo ou nada. A transação explícita é o que estende essa garantia a um grupo de comandos, e é dela que se precisa sempre que uma operação do negócio exige mais de um comando, como a transferência escrita como saída e entrada.

Padrão SQL × PostgreSQL

Em muito material sobre PostgreSQL, a transação é aberta com BEGIN, que no PostgreSQL faz o mesmo que o START TRANSACTION. O START TRANSACTION é a forma do padrão SQL, e o livro usa só ela. Há também uma diferença de comportamento: no padrão, qualquer comando abre uma transação, que fica aberta até um COMMIT ou um ROLLBACK; o PostgreSQL confirma sozinho cada comando executado fora de uma transação aberta explicitamente, comportamento que a documentação chama de *autocommit*.

Uma transação tem quatro propriedades, resumidas na sigla **ACID** (*atomicity, consistency, isolation, durability*), que aparece num artigo de Theo Härder e Andreas Reuter de 1983. A transferência do Bruno mostra cada uma delas.

A **atomicidade** (*atomicity*) é a propriedade de a transação acontecer por inteiro ou não acontecer: ou o Bruno sai da turma 10 e entra na 12, ou nada muda. É o que o ROLLBACK garantiria no caso da turma 21.

A **consistência** (*consistency*) é a propriedade de a transação levar o banco de um estado que atende às regras para outro que também atende. O nome lembra a inconsistência do [capítulo 1](#), mas o sentido aqui é mais amplo: atender a todas as regras, e não só manter iguais as cópias de um dado. O banco confere as restrições declaradas, como a chave estrangeira que recusaria a turma 21. As regras que ficaram fora do banco continuam por conta de quem escreve a transação: a exigência de que a turma nova seja do mesmo curso, que a [seção 5.8](#) listou entre as não cobradas, depende do programa que pede a transferência.

O **isolamento** (*isolation*) é a propriedade de as alterações de uma transação ficarem invisíveis para as outras até ela terminar. Enquanto a transferência não chega ao COMMIT, ninguém mais vê o Bruno fora das duas turmas, e quem consulta as matrículas nesse intervalo continua vendo a matrícula dele na turma 10. O isolamento tem graus, que o padrão chama de níveis de isolamento, e escolher entre eles é um ajuste de concorrência que fica fora do livro.

A **durabilidade** (*durability*) é a propriedade de a transação confirmada não se perder. Depois que o COMMIT termina, a transferência sobrevive a uma queda de energia ou a uma falha do servidor: o SGBD grava as alterações num armazenamento permanente antes de avisar que a transação terminou.

Com as transações, a regra 11 sai do papel, com a ressalva de sempre: o banco garante o tudo ou nada, e a exigência do mesmo curso fica com quem grava os dados. Repare, ainda, que a transferência do Bruno liberaria uma vaga na turma 10 e ocuparia uma na 12. As vagas são justamente o assunto da pergunta da escola que ficou para depois na [seção 6.4](#), a última das quatro.

6.7 Dados derivados e visões

Quais turmas ainda têm vaga? É a última pergunta da escola, e a resposta não está guardada em coluna nenhuma. A tabela `turma` guarda o número de vagas, e a tabela `matricula`, as matrículas de cada turma; as vagas restantes saem de uma conta entre as duas. Um valor assim é um **dado derivado** (*derived data*), um dado que pode ser calculado a partir de outros dados que o banco já guarda.

A primeira ideia de muita gente é criar uma coluna `vagas_restantes` na tabela `turma` e atualizá-la a cada matrícula. Parece prático, mas essa coluna repetiria, em outra forma, uma informação que o banco já tem: as linhas da `matricula`. Toda matrícula, toda exclusão e toda transferência teria de atualizá-la também, e bastaria um programa deixar de atualizá-la uma vez para a coluna e as matrículas passarem a discordar. É a anomalia de atualização da [seção 4.1](#), agora entre duas tabelas. O dado derivado calculado no momento da consulta não tem esse problema: ele sai das matrículas que existem naquele instante e nunca fica desatualizado.

A conta em si é uma consulta com o que este capítulo já mostrou: as turmas, numa junção externa com as matrículas, agrupadas por turma. Como a escola vai fazer essa pergunta o tempo todo, vale guardar a consulta no banco, com um nome. Uma **visão** (*view*) é isso: uma consulta guardada no banco com um nome, que pode ser usada como se fosse uma tabela. Ela é criada com o `CREATE VIEW`:

```
CREATE VIEW turma_ocupacao AS
SELECT
    t.id,
    t.vagas,
    COUNT(m.aluno_id) AS matriculados,
    t.vagas - COUNT(m.aluno_id) AS vagas_restantes
FROM turma AS t
```

```
LEFT JOIN matricula AS m ON m.turma_id = t.id
GROUP BY t.id, t.vagas;
```

A junção é externa para que uma turma recém-aberta, ainda sem nenhuma matrícula, também apareça, com todas as vagas livres. E o COUNT conta a coluna `m.aluno_id`, e não as linhas, por causa dessa mesma turma. Na junção externa, a turma sem matrícula aparece numa linha só, com NULL nas colunas da matrícula; o COUNT(*) contaria essa linha e diria que a turma tem um aluno, enquanto o COUNT(`m.aluno_id`) ignora o NULL e diz zero, que é a resposta certa. A coluna `t.vagas` está no GROUP BY pela regra da [seção 6.4](#): ela aparece no SELECT fora de uma função de agregação.

Com a visão criada, a pergunta da escola vira uma consulta curta:

```
SELECT
    id,
    vagas_restantes
FROM turma_ocupacao
WHERE vagas_restantes > 0
ORDER BY id;
```

Resultado:

id	vagas_restantes
10	28
11	18

A turma 10 tem 30 vagas e duas matrículas; a 11, 20 vagas e duas matrículas. A visão não guarda essas vagas restantes. A documentação do PostgreSQL é explícita: a visão não guarda o resultado, e a consulta dela é executada toda vez que a visão é usada. Por isso, se o Bruno fosse transferido como na [seção 6.6](#), a próxima consulta à visão já mostraria a turma 10 com 29 vagas restantes, sem que ninguém atualizasse coisa alguma. Quem consulta a visão também não precisa saber como a conta é feita: a junção, o agrupamento e a subtração ficam escritos uma vez, no banco.

A idade das pessoas segue o mesmo raciocínio, e aqui fechamos uma conversa da [seção 2.4](#). A escola guarda a data de nascimento, e não a idade, porque a idade muda sozinha a cada aniversário. Quando alguém precisar da idade, ela é calculada na consulta, a partir da data de nascimento e da data do dia, e uma visão sobre a tabela `pessoa` poderia oferecer a idade já calculada, sempre certa.

Cuidado, porém, para não tratar como derivado o que não é. O valor cobrado da matrícula parece sair do preço do curso, mas a [seção 4.5](#) mostrou que ele é o preço do dia da matrícula, um fato próprio, que nenhuma conta sobre os dados de hoje reproduz. Ele continua guardado. A pergunta que separa um caso do outro é sempre a mesma: o valor pode ser recalculado, a qualquer momento, a partir do que o banco guarda? Se pode, ele é derivado e fica para a consulta; se não pode, ele é um fato próprio e precisa ser guardado.

E se a conta ficar pesada? Com milhares de turmas e matrículas, calcular as vagas a cada consulta tem um custo. Guardar o resultado numa coluna passa a ser uma opção, mas aí já é a desnormalização da [seção 4.6](#), com as mesmas condições: um motivo medido, e não presumido, e um mecanismo que mantenha a cópia igual às matrículas. Na escola, com duas turmas, a visão resolve.

6.8 Resumo

Começamos pondo os dados da escola no banco com o `INSERT`, sempre com a lista de colunas e sem o `id` das colunas de identidade, que o banco gera sozinho. A ordem dos comandos seguiu a das chaves estrangeiras, como a dos `CREATE TABLE`. Com a carga completa, vimos o banco recusar o CPF repetido, a matrícula repetida, a turma de um curso que não existe e a turma presencial e on-line ao mesmo tempo, e vimos também que ele aceitaria um certificado para uma matrícula com nota abaixo de 7, porque essa regra não está declarada.

Depois, consultamos. O `SELECT` escolhe as colunas, o `FROM` diz de onde elas vêm, o `WHERE` escolhe as linhas com uma condição, e o `ORDER BY` define a ordem, que sem ele não é garantida. Vimos que a comparação com `NULL` não dá verdadeiro nem falso, e sim desconhecido, e que por isso as matrículas sem nota escapam até de uma condição que parecia cobrir todas as notas. Para encontrar a falta de valor, o teste certo é o `IS NULL`.

A junção reuniu o que a normalização separou: a consulta sobre `matricula`, `pessoa`, `turma` e `curso` remontou a planilha da secretaria, linha por linha, sem guardar nada repetido. A junção externa manteve no resultado o curso sem turma, e vimos que escolher entre uma e outra é responder a uma pergunta de negócio. Com as funções de agregação e o `GROUP BY`, contamos os alunos de cada turma e calculamos a média das notas de cada curso; o `AVG` ignorou as notas que ainda não existem, e por isso a média de Python Essencial ficou sem valor, e não zero.

O `UPDATE` e o `DELETE` alteraram e excluíram linhas, sempre filtrando pela chave, porque só ela identifica uma linha, e vimos que, sem o `WHERE`, os dois alcançam a tabela inteira. Com o `START TRANSACTION`, o `COMMIT` e o `ROLLBACK`, a transferência da regra 11 passou a acontecer por inteiro ou não acontecer, e as propriedades ACID apareceram nela, uma a uma. Por fim, vimos que o dado derivado, como as vagas restantes, é calculado na consulta, e que a visão guarda essa consulta no banco, com um nome, sem guardar os dados.

O modelo respondeu às quatro perguntas da escola. Hora de um café: o caminho, da regra de negócio ao banco em uso, está completo.

Conclusão

Na Introdução, combinamos que, ao fim do livro, você saberia percorrer o caminho da regra de negócio ao SQL e justificar cada decisão tomada nesse caminho. Chegou a hora de conferir esse combinado e de apontar para onde ir depois daqui.

O que você sabe fazer agora

Você sabe começar pelo lugar certo: as regras de negócio. Diante de um problema como o da escola, você tira das regras as entidades, os atributos e os relacionamentos e desenha o DER. Sabe também que mais de um diagrama pode estar certo e que a escolha entre eles se justifica pelas regras. O telefone ficou como atributo multivalorado de **Pessoa** porque a escola guarda só o número de cada telefone; se cada telefone tivesse dados próprios, a entidade **Telefone** passaria a fazer mais sentido.

Depois, você transforma o diagrama em tabelas e escolhe a chave primária de cada uma. Sabe dizer por que o curso ganhou uma chave artificial, cujo valor o banco gera, e por que a turma ficou com o código que a própria escola atribui a ela. As duas chaves são estáveis, por motivos diferentes: o `id` do curso não significa nada para o negócio e por isso não tem motivo para mudar, e o código da turma foi criado pela escola justamente para identificar cada turma.

Você sabe conferir esse desenho pela normalização, partindo das dependências funcionais e chegando à 3FN. E sabe separar o que é repetição do que é fato próprio: o valor cobrado da matrícula parece uma cópia do preço do curso, mas é o preço do dia da matrícula, e por isso continua guardado.

Sabe também escrever o modelo em SQL, com os tipos, as chaves e as restrições que fazem o banco recusar dados inválidos. E sabe onde essa proteção termina: o banco só recusa o que fere uma regra declarada nele, e regras como a nota mínima do certificado ficam a cargo de quem grava os dados. Quem modela precisa saber quais regras estão nessa situação e deixá-las registradas.

Por fim, você sabe testar o modelo pelo uso. Com os dados do estudo de caso, o banco respondeu às quatro perguntas da escola: quantos alunos há em cada turma, quais turmas ainda têm vaga, qual é a média das notas de cada curso e quem ainda está sem nota. A resposta sobre as vagas veio de um dado derivado, calculado na consulta em vez de guardado numa coluna. É o teste que a epígrafe de George Box sugeriu no [capítulo 6](#): o modelo da escola não reproduz a escola inteira, mas é útil, porque responde ao que o negócio precisa saber.

Em todas essas decisões, a justificativa começou pelo mesmo lugar. Se alguém perguntar por que uma tabela foi desenhada daquele jeito, a sua resposta vai começar por uma regra de negócio. Esse foi o refrão do livro, do começo ao fim: a modelagem depende da regra de negócio.

Por onde continuar

O livro parou, de propósito, em vários pontos. Cada um deles é um caminho para quem quiser ir adiante, e todos partem de algo que você já viu aqui.

O primeiro é o modelo físico, que o [capítulo 1](#) apresentou e deixou de fora. É nele que entram os índices e o ajuste de desempenho: como o SGBD grava e encontra os dados quando as tabelas têm milhões de linhas, e não as poucas linhas da escola. As decisões desse nível dependem do SGBD escolhido, e por isso o estudo passa pela documentação desse SGBD.

O segundo é a normalização além da 3FN. A [seção 4.6](#) citou a forma normal de Boyce-Codd e a quarta e a quinta formas normais, sem desenvolvê-las. Elas tratam de situações que as tabelas da escola não têm, mas que aparecem em outros modelos.

O terceiro é a programação no banco. A [seção 5.8](#) mostrou que algumas regras não cabem numa restrição, como o limite de vagas e a nota mínima do certificado, e citou o gatilho como a saída do PostgreSQL para elas. Escrever gatilhos é programação, e não modelagem, mas é com eles que o banco passa a cobrar também essas regras.

O quarto é a concorrência. A [seção 6.6](#) apresentou o isolamento e deixou de fora os níveis de isolamento, que decidem o que uma transação enxerga das outras transações em andamento. Num sistema em que muitos usuários gravam dados ao mesmo tempo, a escolha do nível pesa sobre a correção dos dados e sobre o desempenho.

O quinto é a administração do banco: os usuários, as permissões e as cópias de segurança. A [seção 5.2](#) só situou os comandos GRANT e REVOKE, que dão e retiram as permissões. Administrar é o trabalho de quem mantém o banco funcionando, e não de quem o desenha, mas quem modela convive com esse trabalho.

O sexto é o próprio SQL. O [capítulo 6](#) usou o mínimo necessário para pôr o modelo em uso, e a linguagem vai muito além disso, com consultas dentro de outras consultas e com a cláusula HAVING, que filtra os grupos depois da agregação, entre outros recursos. Quanto mais SQL você souber, mais perguntas vai conseguir fazer ao modelo, e mais depressa vai perceber quando uma pergunta fica sem resposta porque o dado de que ela precisa não está no banco.

Para aprofundar, indico quatro fontes. *Projeto de Banco de Dados*, de Carlos Alberto Heuser, trata das duas etapas que este livro percorreu: a modelagem conceitual e o projeto lógico. *Sistemas de Banco de Dados*, de Ramez Elmasri e Shamkant B. Navathe, cobre o campo inteiro, inclusive assuntos que ficaram fora daqui, como os índices, o processamento de transações e o controle de concorrência. *Introdução a Sistemas de Bancos de Dados*, de C. J. Date, aprofunda a teoria do modelo relacional e da normalização. E a documentação do PostgreSQL, a referência de sintaxe de todo o livro, diz, na página de cada comando, o que ele faz e em que pontos difere do padrão SQL.

Obrigado por ter chegado até aqui. Se quiser acompanhar o que escrevo, você me encontra no meu site, <https://fabiojano.com>, no Instagram, [@fabiojaniolima](#), e no LinkedIn, [linkedin.com/in/fabiojano](https://www.linkedin.com/in/fabiojano). Para falar comigo, o e-mail é contato@fabiojano.com.

Glossário

- **ação referencial** (*referential action*): o que o SGBD faz com as linhas que apontam para uma linha quando ela é excluída ou quando o valor da chave dela é alterado. Ver [seção 5.6](#).
- **ACID** (*atomicity, consistency, isolation, durability*): sigla que reúne as quatro propriedades de uma transação: a atomicidade, a consistência, o isolamento e a durabilidade. Ver [seção 6.6](#).
- **anomalia de atualização** (*update anomaly*): problema em que alterar um único fato exige alterar várias linhas, e esquecer uma delas deixa a tabela discordando dela mesma. Ver [seção 4.1](#).
- **anomalia de exclusão** (*deletion anomaly*): problema em que apagar um fato apaga junto outro que deveria continuar guardado. Ver [seção 4.1](#).
- **anomalia de inserção** (*insertion anomaly*): problema em que um fato não pode ser registrado sem que outro, que ainda não existe, seja registrado junto. Ver [seção 4.1](#).
- **atomicidade** (*atomicity*): propriedade de a transação acontecer por inteiro ou não acontecer. Ver [seção 6.6](#).
- **atributo** (*attribute*): dado que se guarda sobre cada ocorrência de uma entidade. Ver [seção 2.4](#).
- **atributo de relacionamento** (*relationship attribute*): atributo que descreve a ligação entre ocorrências de entidades, e não alguma delas isoladamente. Ver [seção 2.8](#).
- **atributo multivalorado** (*multivalued attribute*): atributo que pode ter mais de um valor para a mesma ocorrência. Ver [seção 2.4](#).
- **atributo opcional**: atributo cuja cardinalidade mínima é zero, ou seja, que pode ficar sem valor. Ver [seção 2.4](#).
- **autorrelacionamento** (*recursive relationship*): relacionamento que associa ocorrências da mesma entidade. Ver [seção 2.6](#).
- **banco de dados** (*database*): coleção de dados relacionados entre si, organizada para atender a um propósito. Ver [seção 1.3](#).
- **cardinalidade** (*cardinality*): quantidade mínima e máxima de ocorrências de uma entidade que podem estar associadas, por meio de um relacionamento, a uma ocorrência da outra entidade. Ver [seção 2.7](#).
- **cardinalidade de atributo**: quantidade mínima e máxima de valores que um atributo pode ter para cada ocorrência. Ver [seção 2.4](#).
- **cardinalidade máxima** (*maximum cardinality*): parte da cardinalidade que diz se uma ocorrência pode se associar a no máximo uma ocorrência do outro lado (1) ou a várias (n). Ver [seção 2.7](#).
- **cardinalidade mínima** (*minimum cardinality*): parte da cardinalidade que diz se a associação é obrigatória (1) ou opcional (0). Ver [seção 2.7](#).

- **chave** (*key*): coluna, ou conjunto de colunas, que identifica cada linha da tabela ou aponta para uma linha de outra tabela. Ver [seção 1.4](#).
- **chave alternativa** (*alternate key*): chave candidata que não foi escolhida como chave primária e continua sem poder se repetir. Ver [seção 3.2](#).
- **chave artificial** (*surrogate key*): coluna criada só para identificar as linhas, com valores que não significam nada para o negócio. Ver [seção 3.2](#).
- **chave candidata** (*candidate key*): coluna, ou conjunto mínimo de colunas, cujo valor não se repete entre as linhas da tabela; mínimo quer dizer que nenhuma coluna pode ser retirada sem que o valor do que sobra passe a poder se repetir. Ver [seção 3.2](#).
- **chave composta** (*composite key*): chave formada por mais de uma coluna. Ver [seção 3.2](#).
- **chave estrangeira** (*foreign key*): coluna, ou conjunto de colunas, cujos valores apontam para linhas de outra tabela, ou da própria tabela, por meio da chave primária ou de outra chave candidata da tabela apontada. Ver [seção 3.2](#).
- **chave natural** (*natural key*): chave formada por dados que existem no negócio. Ver [seção 3.2](#).
- **chave primária** (*primary key*): chave candidata escolhida para identificar as linhas da tabela e servir de alvo às referências que vêm de outras tabelas. Ver [seção 3.2](#).
- **coluna** (*column*): parte da tabela que guarda o mesmo tipo de dado para todos os registros. Ver [seção 1.4](#).
- **coluna de identidade** (*identity column*): coluna cujo valor o SGBD gera sozinho, em sequência, a cada linha inserida. Ver [seção 5.4](#).
- **concorrência** (*concurrency*): acesso de várias pessoas ou programas aos mesmos dados ao mesmo tempo. Ver [seção 1.3](#).
- **consistência** (*consistency*): propriedade de a transação levar o banco de um estado que atende às regras para outro que também atende. Ver [seção 6.6](#).
- **dado**: sequência de símbolos que pode ser guardada e processada por uma máquina sem que a máquina precise entender o que os símbolos querem dizer. Ver [seção 1.1](#).
- **dado derivado** (*derived data*): dado que pode ser calculado a partir de outros dados que o banco já guarda. Ver [seção 6.7](#).
- **DDL** (*Data Definition Language*): linguagem de definição de dados, o grupo de comandos que criam, alteram e removem as estruturas do banco, como CREATE, ALTER e DROP. Ver [seção 5.2](#).
- **dependência funcional** (*functional dependency*): relação entre colunas em que cada valor de uma coluna, ou de um conjunto de colunas, corresponde a um único valor de outra coluna; escreve-se $A \rightarrow B$. Ver [seção 4.2](#).
- **dependência parcial** (*partial dependency*): dependência de uma coluna que não faz parte de nenhuma chave candidata em relação a só uma parte de uma chave candidata composta. Ver [seção 4.4](#).
- **dependência transitiva** (*transitive dependency*): dependência de uma coluna que não faz parte de nenhuma chave candidata em relação à chave, por meio de outra coluna que também não faz parte de nenhuma chave candidata. Ver [seção 4.5](#).
- **DER** (*Entity-Relationship Diagram*): Diagrama Entidade-Relacionamento, o desenho que representa um MER. Ver [seção 2.2](#).

- **desnormalização** (*denormalization*): decisão consciente de guardar um dado em mais de um lugar, abrindo mão de uma forma normal, para poupar trabalho numa consulta que se repete muito. Ver [seção 4.6](#).
- **determinante** (*determinant*): coluna, ou conjunto de colunas, que fica à esquerda da seta numa dependência funcional. Ver [seção 4.2](#).
- **DML** (*Data Manipulation Language*): linguagem de manipulação de dados, o grupo de comandos que inserem, alteram e excluem linhas, como INSERT, UPDATE e DELETE. Ver [seção 5.2](#).
- **durabilidade** (*durability*): propriedade de a transação confirmada não se perder. Ver [seção 6.6](#).
- **entidade** (*entity*): conjunto de coisas do negócio, concretas ou abstratas, sobre as quais se guardam dados. Ver [seção 2.3](#).
- **entidade associativa** (*associative entity*): relacionamento redefinido para ser tratado como entidade, de modo que possa participar de outros relacionamentos. Ver [seção 2.8](#).
- **especialização compartilhada** (*overlapping specialization*): especialização em que uma ocorrência da entidade genérica pode estar em mais de uma especializada ao mesmo tempo. Ver [seção 2.9](#).
- **especialização exclusiva** (*disjoint specialization*): especialização em que uma ocorrência da entidade genérica está em uma especializada só. Ver [seção 2.9](#).
- **especialização parcial** (*partial specialization*): especialização em que pode haver ocorrência da entidade genérica fora de todas as especializadas. Ver [seção 2.9](#).
- **especialização total** (*total specialization*): especialização em que toda ocorrência da entidade genérica está em pelo menos uma especializada. Ver [seção 2.9](#).
- **forma normal** (*normal form*): critério que uma tabela atende ou não; cada forma normal elimina um tipo de problema do desenho. Ver [seção 4.1](#).
- **função de agregação** (*aggregate function*): função que recebe os valores de várias linhas e devolve um valor só. Ver [seção 6.4](#).
- **gatilho** (*trigger*): procedimento que o SGBD executa automaticamente quando uma tabela é alterada e que pode conferir outras tabelas e recusar a alteração. Ver [seção 5.8](#).
- **generalização e especialização** (*generalization and specialization*): recurso do MER para representar uma entidade genérica, com o que todas as ocorrências têm em comum, e entidades especializadas, com o que só uma parte delas tem. Ver [seção 2.9](#).
- **herança** (*inheritance*): propriedade pela qual cada ocorrência de uma entidade especializada tem também os atributos e os relacionamentos da entidade genérica. Ver [seção 2.9](#).
- **identificador** (*identifier*): atributo, ou conjunto de atributos e relacionamentos, cujo valor distingue uma ocorrência da entidade de todas as outras. Ver [seção 2.5](#).
- **identificador composto** (*composite identifier*): identificador formado por mais de um atributo ou relacionamento. Ver [seção 2.5](#).
- **inconsistência**: situação em que cópias do mesmo dado guardam valores diferentes. Ver [seção 1.2](#).
- **informação**: o que um ou mais dados passam a significar para alguém que conhece o contexto deles. Ver [seção 1.1](#).

- **integridade** (*integrity*): garantia de que os dados obedecem às regras declaradas no banco. Ver [seção 1.3](#).
- **integridade de entidade** (*entity integrity*): regra pela qual nenhuma parte da chave primária pode ficar sem valor. Ver [seção 3.3](#).
- **integridade referencial** (*referential integrity*): regra pela qual toda chave estrangeira aponta para uma linha que existe na tabela referenciada ou, quando a ligação é opcional, fica sem valor. Ver [seção 3.3](#).
- **isolamento** (*isolation*): propriedade de as alterações de uma transação ficarem invisíveis para as outras até ela terminar. Ver [seção 6.6](#).
- **junção** (*join*): operação que combina as linhas de duas tabelas, ligando cada linha de uma às linhas da outra que atendem a uma condição. Ver [seção 6.3](#).
- **junção externa** (*outer join*): junção que mantém também as linhas de uma das tabelas que não encontram par na outra, preenchendo com NULL as colunas que vêm da outra tabela. Ver [seção 6.3](#).
- **linha** (*row*): parte da tabela que guarda um registro. Ver [seção 1.4](#).
- **lógica de três valores** (*three-valued logic*): lógica em que uma condição pode ser verdadeira, falsa ou desconhecida. Ver [seção 6.2](#).
- **MER** (*Entity-Relationship Model*): Modelo Entidade-Relacionamento, forma de descrever os dados de um negócio por meio de entidades, dos atributos delas e dos relacionamentos entre elas. Ver [seção 2.2](#).
- **modelo conceitual** (*conceptual model*): descrição dos dados do negócio sem depender de nenhum SGBD. Ver [seção 1.5](#).
- **modelo de dados** (*data model*): descrição dos dados que um banco guarda e de como eles se organizam. Ver [seção 1.5](#).
- **modelo físico** (*physical model*): descrição de como um SGBD específico grava e encontra os dados. Ver [seção 1.5](#).
- **modelo lógico** (*logical model*): descrição dos dados do jeito que o tipo de SGBD escolhido os representa; no modelo relacional, em tabelas, colunas e chaves. Ver [seção 1.5](#).
- **modelo relacional** (*relational model*): forma de representar dados, proposta por Codd em 1970, em que todos os dados ficam em tabelas. Ver [seção 1.4](#).
- **normalização** (*normalization*): processo de dividir uma tabela em tabelas menores, guiado pelas formas normais, para que cada fato fique guardado num lugar só. Ver [seção 4.1](#).
- **NoSQL**: etiqueta para os bancos que não seguem o modelo relacional. Ver [seção 1.4](#).
- **NULL**: marca que indica que uma coluna não tem valor numa linha; não é zero nem texto vazio. Ver [seção 3.4](#).
- **ocorrência** (*instance*): cada elemento do conjunto que forma uma entidade. Ver [seção 2.3](#).
- **operador de comparação** (*comparison operator*): operador que compara dois valores, como =, <>, <, >, <= e >=. Ver [seção 6.2](#).
- **padrão SQL** (*SQL standard*): conjunto oficial de regras da linguagem SQL, publicado pela ISO e pela IEC com o número ISO/IEC 9075 e popularmente chamado de “SQL ANSI”. Ver [seção 5.1](#).

- **papel** (*role*): função que a ocorrência de uma entidade cumpre num relacionamento. Ver [seção 2.6](#).
- **primeira forma normal (1FN)** (*first normal form*): forma normal em que cada coluna guarda um único valor atômico em cada linha e a tabela tem uma chave, de modo que nenhuma linha se repete. Ver [seção 4.3](#).
- **redundância**: o mesmo dado guardado em mais de um lugar. Ver [seção 1.2](#).
- **regra de negócio** (*business rule*): afirmação sobre o funcionamento da organização que diz quais dados ela guarda ou quais limites esses dados precisam respeitar. Ver [seção 2.1](#).
- **relação** (*relation*): nome que a teoria do modelo relacional dá à tabela, tomado da matemática por Codd. Ver [seção 3.1](#).
- **relacionamento** (*relationship*): associação entre ocorrências de entidades. Ver [seção 2.6](#).
- **relacionamento 1:1** (*one-to-one*): relacionamento com cardinalidade máxima 1 dos dois lados. Ver [seção 2.7](#).
- **relacionamento 1:N** (*one-to-many*): relacionamento com cardinalidade máxima 1 de um lado e n do outro. Ver [seção 2.7](#).
- **relacionamento N:N** (*many-to-many*): relacionamento com cardinalidade máxima n dos dois lados. Ver [seção 2.7](#).
- **restrição** (*constraint*): condição declarada na tabela que toda linha precisa atender; o SGBD recusa a linha que não a atende. Ver [seção 5.5](#).
- **segunda forma normal (2FN)** (*second normal form*): forma normal em que a tabela está na 1FN e não tem dependência parcial. Ver [seção 4.4](#).
- **SGBD** (*Database Management System*): sistema gerenciador de banco de dados, o software que cria, guarda e controla um banco de dados. Ver [seção 1.3](#).
- **SQL**: linguagem com que os programas fazem pedidos a um SGBD relacional. Ver [seção 1.3](#).
- **tabela** (*table*): no modelo relacional, estrutura que reúne os dados de um mesmo assunto, em linhas e colunas. Ver [seção 1.4](#).
- **tabela associativa** (*associative table*): tabela que representa um relacionamento N:N, com uma chave estrangeira para cada tabela relacionada. Ver [seção 3.6](#).
- **terceira forma normal (3FN)** (*third normal form*): forma normal em que a tabela está na 2FN e não tem dependência transitiva. Ver [seção 4.5](#).
- **tipo de dado** (*data type*): conjunto de valores que uma coluna aceita, junto com as operações que fazem sentido sobre esses valores. Ver [seção 5.3](#).
- **transação** (*transaction*): conjunto de operações que o SGBD executa por inteiro ou não executa. Ver [seção 1.3](#).
- **valor atômico** (*atomic value*): valor que o negócio usa inteiro, sem precisar de partes dele. Ver [seção 4.3](#).
- **valor padrão** (*default value*): valor que a coluna recebe quando o comando que insere a linha não informa nenhum. Ver [seção 5.5](#).
- **visão** (*view*): consulta guardada no banco com um nome, que pode ser usada como se fosse uma tabela. Ver [seção 6.7](#).