K19-k03 - SQL e Modelo Relacional

TREINAMENTOS

SQL e Modelo Relacional

Modelo Relacional e SQL

30 de julho de 2012

Sobre a K19

A K19 é uma empresa especializada na capacitação de desenvolvedores de software. Sua equipe é composta por profissionais formados em Ciência da Computação pela Universidade de São Paulo (USP) e que possuem vasta experiência em treinamento de profissionais para área de TI.

O principal objetivo da K19 é oferecer treinamentos de máxima qualidade e relacionados às principais tecnologias utilizadas pelas empresas. Através desses treinamentos, seus alunos se tornam capacitados para atuar no mercado de trabalho.

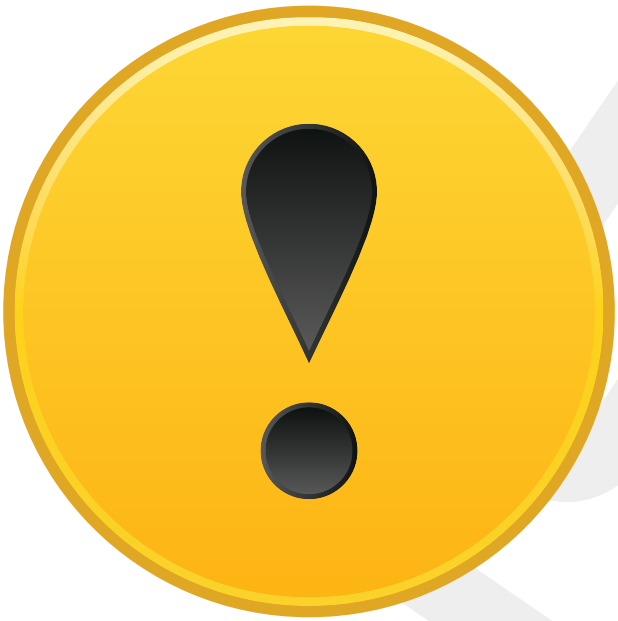
Visando a máxima qualidade, a K19 mantém as suas apostilas em constante renovação e melhoria, oferece instalações físicas apropriadas para o ensino e seus instrutores estão sempre atualizados didática e tecnicamente.

Seguro Treinamento

Na K19 o aluno faz o curso quantas vezes quiser!

Comprometida com o aprendizado e com a satisfação dos seus alunos, a K19 é a única que possui o Seguro Treinamento. Ao contratar um curso, o aluno poderá refazê-lo quantas vezes desejar mediante a disponibilidade de vagas e pagamento da franquia do Seguro Treinamento.

As vagas não preenchidas até um dia antes do início de uma turma da K19 serão destinadas ao alunos que desejam utilizar o Seguro Treinamento. O valor da franquia para utilizar o Seguro Treinamento é 10% do valor total do curso.



Termo de Uso

Termo de Uso

Todo o conteúdo desta apostila é propriedade da K19 Treinamentos. A apostila pode ser utilizada livremente para estudo pessoal . Além disso, este material didático pode ser utilizado como material de apoio em cursos de ensino superior desde que a instituição correspondente seja reconhecida pelo MEC (Ministério da Educação) e que a K19 seja citada explicitamente como proprietária do material.

É proibida qualquer utilização desse material que não se enquadre nas condições acima sem o prévio consentimento formal, por escrito, da K19 Treinamentos. O uso indevido está sujeito às medidas legais cabíveis.

Conheça os nossos cursos

K01- Lógica de Programação

K11 - Orientação a Objetos em Java

K12 - Desenvolvimento Web com JSF2 e JPA2

K21 - Persistência com JPA2 e Hibernate

K22 - Desenvolvimento Web Avançado com JFS2, EJB3.1 e CDI

K23 - Integração de Sistemas com Webservices, JMS e EJB

K31 - C# e Orientação a Objetos

K32 - Desenvolvimento Web com ASP.NET MVC

www.k19.com.br/cursos

CAPÍTULO 1

INTRODUÇÃO

SGBD

Qualquer empresa necessita armazenar os dados relacionados ao seu negócio. Por exemplo, uma livraria deve manter as informações dos livros que são comercializados por ela. Um banco precisa registrar os dados dos seus clientes. Uma escola deve guardar as informações dos seus alunos.

Hoje em dia, utilizar papel para registrar os dados de uma empresa não é uma boa alternativa. O espaço físico necessário gera custos altos para empresa. Em geral, a consulta das informações registradas é demorada. O risco de um acidente destruir os dados armazenados em papel é alto.

Em vários aspectos, utilizar computadores para o armazenamento de dados é uma abordagem melhor do que utilizar papel. Os dados podem ser armazenados, por exemplo, em arquivos de texto ou planilhas. Contudo, existem sistemas especializados na persistência de dados que oferecem recursos mais sofisticados e eficientes para esse tipo de objetivo. Esses sistemas são conhecidos como **Sistemas Gerenciadores de Bancos de Dados - SGBD**.

Os principais SGBDs adotados nas empresas utilizam o **Modelo Relacional** para definir a organização das informações armazenadas e a linguagem **SQL** para permitir a manipulação desses dados.

Eis uma lista dos SGBDs mais utilizados nas empresas:

- MySQL Server
- Oracle Database
- SQL Server
- PostgreSQL

MySQL Server

Neste treinamento, utilizaremos o MySQL Server, que é mantido pela Oracle e amplamente utilizado em aplicações comerciais. Para instalar o MySQL Server, você pode consultar o artigo disponível em nosso site: http://www.k19.com.br/artigos/instalando-mysql/

Bases de dados (*Databases*)

Um sistema gerenciador de banco de dados é capaz de gerenciar informações de diversos sistemas ao mesmo tempo. Por exemplo, as informações dos clientes de um banco, além dos produtos de uma loja virtual ou dos livros de uma livraria.

Suponha que os dados fossem mantidos sem nenhuma separação lógica. Implementar regras de segurança específicas seria extremamente complexo. Tais regras criam restrições quanto ao conteúdo que pode ser acessado por cada usuário. Por exemplo, determinado usuário poderia ter permissão de acesso aos dados dos clientes do banco, mas não às informações dos produtos da loja virtual, ou dos livros da livraria.

Para obter uma organização melhor, os dados são armazenados separadamente em um SGBD. Daí surge o conceito de **base de dados** (database). Uma base de dados é um agrupamento lógico das informações de um determinado domínio.

Criando uma base de dados no MySQL Server

Para criar uma base de dados no MySQL Server, podemos utilizar o comando **CREATE DATABASE**.

```
mysql> CREATE DATABASE livraria;
Query OK, 1 row affected (0.02 sec)
```

Terminal 1.1: Criando uma base de dados.

Podemos utilizar o comando **SHOW DATABASES** para listar as bases de dados existentes.

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| livraria           |
| mysql              |
| test               |
+--------------------+
4 rows in set (0.03 sec)
```

Terminal 1.2: Listando as bases de dados existentes.

Repare que, além da base de dados **livraria**, há outras três bases. Essas bases foram criadas automaticamente pelo próprio MySQL Server para teste ou para armazenar configurações.

Quando uma base de dados não é mais necessária, ela pode ser removida através do comando **DROP DATABASE**.

```
mysql> DROP DATABASE livraria;
Query OK, 0 rows affected (0.08 sec)
```

Terminal 1.3: Destruindo uma base de dados.

Tabelas

Um servidor de banco de dados é dividido em bases de dados com o intuito de separar as informações de domínios diferentes. Nessa mesma linha de raciocínio, podemos dividir os dados de uma base a fim de agrupá-los segundo as suas correlações. Essa separação é feita através de **tabelas**. Por exemplo, no sistema de um banco, é interessante separar o saldo e o limite de uma conta, do nome e CPF de um cliente. Então, poderíamos criar uma tabela para os dados relacionados às contas e outra para os dados relacionados aos clientes.

Cliente		
nome	idade	cpf
José	27	31875638735
Maria	32	30045667856

Conta		
numero	saldo	limite
1	1000	500
2	2000	700

Tabela 1.1: Tabelas para armazenar os dados relacionados aos clientes e às contas

Uma tabela é formada por **registros** (linhas) e os registros são formados por **campos** (colunas). Por exemplo, considere uma tabela para armazenar as informações dos clientes de um banco. Cada registro dessa tabela armazena em seus campos os dados de um determinado cliente.

Tabelas no MySQL Server

As tabelas no MySQL Server são criadas através do comando **CREATE TABLE**. Na criação de uma tabela, é necessário definir quais são os nomes e os tipos das colunas.

```
mysql> CREATE TABLE `livraria`.`Livro` (
    -> `titulo` VARCHAR(255),
    -> `preco` DOUBLE
    -> )
    -> ENGINE=MyISAM;
Query OK, 0 rows affected (0.14 sec)
```

Terminal 1.4: Criando uma tabela.

As tabelas de uma base de dados podem ser listadas através do comando **SHOW TABLES**. Antes de utilizar esse comando, devemos selecionar uma base de dados através do comando **USE**.

```
mysql> USE livraria;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SHOW TABLES;
+--------------------+
| Tables_in_livraria |
+--------------------+
| Livro              |
+--------------------+
1 row in set (0.00 sec)
```

Terminal 1.5: Listando as tabelas de uma base de dados.

Se uma tabela não for mais desejada, ela pode ser removida através do comando **DROP TABLE**.

```
mysql> DROP TABLE Livro;
Query OK, 0 rows affected (0.00 sec)
```

Terminal 1.6: Destruindo uma tabela.

Também podemos alterar a estrutura de uma tabela com o comando **ALTER TABLE**.

```
mysql> ALTER TABLE Livro RENAME livros;
Query OK, 0 rows affected (0.00 sec)
```

Terminal 1.7: Alterando o nome da tabela.

```
mysql> ALTER TABLE Livro ADD paginas INTEGER;
Query OK, 0 rows affected (0.00 sec)
```

Terminal 1.8: Adicionando uma coluna.

```
mysql> ALTER TABLE Livro DROP COLUMN paginas;
Query OK, 0 rows affected (0.00 sec)
```

Terminal 1.9: Removendo uma coluna.

CRUD

As operações básicas para manipular os dados persistidos são: inserir, ler, alterar e remover.

Essas operações são realizadas através de uma linguagem de consulta denominada **SQL** (*Structured Query Language*). Essa linguagem oferece quatro comandos básicos: **INSERT**, **SELECT**, **UPDATE** e **DELETE**. Esses comandos são utilizados para inserir, ler, alterar e remover registros, respectivamente.

```
mysql> INSERT INTO Livro (titulo, preco) VALUES ('Java', 98.75);
Query OK, 1 row affected (0.00 sec)
```

Terminal 1.10: Inserindo um registro.

```
mysql> SELECT * FROM Livro;
+--------+-------+
| titulo | preco |
+--------+-------+
| Java   | 98.75 |
+--------+-------+
1 row in set (0.00 sec)
```

Terminal 1.11: Selecionando registros.

```
mysql> UPDATE Livro SET preco = 115.9 WHERE titulo = 'Java';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Terminal 1.12: Alterando registros.

```
mysql> SELECT * FROM Livro;
+--------+-------+
| titulo | preco |
+--------+-------+
| Java   | 115.9 |
+--------+-------+
1 row in set (0.00 sec)
```

Terminal 1.13: Selecionando registros.

```
mysql> DELETE FROM Livro WHERE titulo = 'Java';
Query OK, 1 row affected (0.00 sec)
```

Terminal 1.14: Removendo registros.

```
mysql> SELECT * FROM Livro;
Empty set (0.00 sec)
```

Terminal 1.15: Selecionando registros.

Restrições

Podemos estabelecer algumas restrições sobre os valores armazenados nas tabelas para manter a consistência dos dados. Por exemplo, é possível obrigar que um determinado campo possua sempre um valor não nulo.

No MySQL Server, quando criamos uma tabela, podemos adicionar a restrição **NOT NULL** nas colunas que são obrigatórias.

```
mysql> CREATE TABLE `livraria`.`Livro` (
    -> `titulo` VARCHAR(255) NOT NULL,
    -> `preco` DOUBLE NOT NULL
    -> )
    -> ENGINE=MyISAM;
Query OK, 0 rows affected (0.14 sec)
```

Terminal 1.16: Aplicando o comando `NOT NULL` *nas colunas obrigatórias.*

Também podemos definir, no MySQL Server, que uma coluna não pode possuir valores repetidos através do comando **UNIQUE**.

```
mysql> CREATE TABLE `livraria`.`Livro` (
    -> `titulo` VARCHAR(255) NOT NULL UNIQUE,
    -> `preco` DOUBLE NOT NULL
    -> )
    -> ENGINE=MyISAM;
Query OK, 0 rows affected (0.14 sec)
```

Terminal 1.17: Aplicando o comando `UNIQUE` *na coluna* `titulo`.

Exercícios de Fixação

1. Abra um terminal, crie e acesse uma pasta com o seu nome.

```
cosen@k19:~$ mkdir rafael
cosen@k19:~$ cd rafael/
cosen@k19:~/rafael$
```

Terminal 1.18: Criando e acessando uma pasta com o seu nome.

2. Estando dentro da sua pasta, acesse o **MySQL Server** utilizando o usuário **root** e a senha **root**.

```
k19@k19-11:~/rafael$ mysql -u root -p
Enter password:
```

Terminal 1.19: Logando no MySQL Server.

3. Caso exista uma base de dados chamada **livraria**, remova-a. Utilize o comando **SHOW DATABASES** para listar as bases de dados existentes e o comando **DROP DATABASE** para remover a base livraria se ela existir.

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| livraria           |
```

```
| mysql              |
| test               |
+--------------------+
4 rows in set (0.00 sec)

mysql> DROP DATABASE livraria;
Query OK, 1 row affected (0.12 sec)
```

Terminal 1.20: Listando as bases de dados existentes e removendo a base `livraria`.

4 Crie uma nova base de dados chamada **livraria**. Utilize o comando **CREATE DATABASE**. Você vai utilizar esta base nos exercícios seguintes.

```
mysql> CREATE DATABASE livraria;
Query OK, 1 row affected (0.00 sec)
```

Terminal 1.21: Criando a base `livraria`.

5 Abra um editor de texto e digite o código abaixo para criar uma tabela com o nome **Editora**. Depois salve o arquivo com o nome `create-table-editora.sql` dentro da pasta com o seu nome.

```
1  USE livraria;
2  CREATE TABLE Editora (
3    id BIGINT NOT NULL AUTO_INCREMENT,
4    nome VARCHAR (255) NOT NULL,
5    email VARCHAR (255) NOT NULL,
6  )
7  ENGINE = InnoDB;
```

Código SQL 1.1: Criando a tabela `Editora`

6 Dentro do terminal, use o comando `source` para executar o arquivo que você acabou de criar.

```
mysql> source create-table-editora.sql
Database changed
Query OK, 0 rows affected (0.08 sec)
```

Terminal 1.22: Executando a tabela `Editora`.

7 Abra um novo editor de texto e digite o código abaixo para criar uma tabela com o nome **Livro**. Em seguida, salve o arquivo com o nome `create-table-livro.sql` dentro da pasta com o seu nome.

```
1  USE livraria;
2  CREATE TABLE Livro (
3    id BIGINT NOT NULL AUTO_INCREMENT,
4    titulo VARCHAR(255) NOT NULL,
5    preco DOUBLE NOT NULL,
6  )
7  ENGINE = InnoDB;
```

Código SQL 1.2: Criando a tabela Livro

8 Dentro do terminal, use o comando `source` para executar o código do arquivo `create-table-livro.sql`.

```
mysql> source create-table-livro.sql
Database changed
Query OK, 0 rows affected (0.08 sec)
```

Terminal 1.23: Executando a tabela `Livro`.

9 Abra um novo editor de texto e digite o código abaixo para adicionar alguns registros na tabela

Editora. Depois salve o arquivo com o nome `adicionando-registros-editora.sql` dentro da pasta com o seu nome.

```
1 INSERT INTO Editora (nome, email) VALUES ('Oreilly', 'oreilly@email.com');
2
3 INSERT INTO Editora (nome, email) VALUES ('Wrox', 'wrox@email.com');
4
5 INSERT INTO Editora (nome, email) VALUES ('Apress', 'apress@email.com');
```

Código SQL 1.3: Adicionando registros na tabela `Editora`

10 Dentro do terminal, execute o arquivo que você acabou de criar para adicionar alguns registro na tabela **Editora**.

```
mysql> source adicionando-registros-editora.sql
Query OK, 1 row affected (0.03 sec)

Query OK, 1 row affected (0.04 sec)

Query OK, 1 row affected (0.04 sec)
```

Terminal 1.24: Inserindo editoras.

11 Abra um novo editor de texto e digite o código abaixo para adicionar alguns registros na tabela **Livro**. Depois salve o arquivo com o nome `adicionando-registros-livro.sql` dentro da pasta com o seu nome.

```
1 INSERT INTO Livro (titulo, preco) VALUES ('Aprendendo C#', 89.90);
2
3 INSERT INTO Livro (titulo, preco) VALUES ('Introdução ao JSF 2', 122.90);
4
5 INSERT INTO Livro (titulo, preco) VALUES ('JSF 2 Avançado', 149.90);
```

Código SQL 1.4: Adicionando alguns registros na tabela `Livro`

12 Dentro do terminal, execute o arquivo que você acabou de criar para adicionar alguns registros na **Livro**.

```
mysql> source adicionando-registros-livro.sql
Query OK, 1 row affected (0.02 sec)

Query OK, 1 row affected (0.04 sec)

Query OK, 1 row affected (0.04 sec)
```

Terminal 1.25: Inserindo livros.

13 Consulte os registros da tabela Editora e da tabela Livro. Utilize o comando **SELECT**.

```
mysql> SELECT * FROM Editora;
+----+---------+-------------------+
| id | nome    | email             |
+----+---------+-------------------+
|  1 | Oreilly | oreilly@email.com |
|  2 | Wrox    | wrox@email.com    |
|  3 | Apress  | apress@email.com  |
+----+---------+-------------------+
3 rows in set (0.00 sec)
```

Terminal 1.26: Selecionando as editoras.

```
mysql> SELECT * FROM Livro;
+----+-----------------------+-------+
```

```
| id | titulo                | preco |+----+-----------------------+-------+|  1 | Aprendendo C#
|  89.9 ||  2 | Introdução ao JSF 2   | 122.9 ||  3 | JSF 2 Avançado        | 149.9 |+----+------+
3 rows in set (0.00 sec)
```

Terminal 1.27: Selecionando os livros.

14 Altere alguns dos registros da tabela Livro. Utilize o comando **UPDATE**.

```
mysql> UPDATE Livro SET preco=92.9 WHERE id=1;
Query OK, 1 row affected (0.07 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Terminal 1.28: Alterando livros.

15 Altere alguns dos registros da tabela Editora. Utilize o comando **UPDATE**.

```
mysql> UPDATE Editora SET nome='OReilly' WHERE id=1;
Query OK, 1 row affected (0.09 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Terminal 1.29: Alterando editoras.

16 Remova alguns registros da tabela Livro. Utilize o comando **DELETE**.

```
mysql> DELETE FROM Livro WHERE id=2;
Query OK, 1 row affected (0.07 sec)
```

Terminal 1.30: Removendo livros.

CAPÍTULO 2

CONSULTAS

SELECT

Através do comando `SELECT`, podemos recuperar as informações armazenadas em um banco de dados. Para utilizar o comando `SELECT`, devemos indicar as tabelas que serão consultadas e as colunas que queremos recuperar.

No exemplo abaixo, as colunas `nome` e `email` são recuperadas da tabela `Aluno`.

```
1 SELECT nome, email FROM Aluno;
```

Código SQL 2.1: Recuperando as colunas `nome` *e* `email` *da tabela* `Aluno`

Aluno	
nome	email
Rafael Cosentino	rafael.cosentino@k19.com.br
Jonas Hirata	jonas.hirata@k19.com.br

Tabela 2.1: Resultado da consulta: `SELECT nome, email FROM Aluno`

Quando todas as colunas devem ser recuperadas, é mais prático utilizar o caractere "*". Veja o exemplo abaixo:

```
1 SELECT * FROM Aluno;
```

Código SQL 2.2: Recuperando todas as colunas da tabela `Aluno`

Aluno			
nome	email	telefone	altura
Rafael Cosentino	cosentino@k19.com.br	11 23873791	1.87
Jonas Hirata	hirata@k19.com.br	11 23873791	1.76

Tabela 2.2: Resultado da consulta: `SELECT * FROM Aluno`

É possível recuperar colunas de várias tabelas. Nesse caso, os registros das tabelas consultadas são "cruzados". Veja o exemplo abaixo:

```
1 SELECT * FROM Aluno, Professor;
```

Código SQL 2.3: Recuperando todas as colunas das tabelas `Aluno` *e* `Professor`

Aluno x Professor					
nome	email	telefone	altura	nome	codigo
Rafael Cosentino	cosentino@k19.com.br	11 23873791	1.87	Marcelo Martins	1
Rafael Cosentino	cosentino@k19.com.br	11 23873791	1.87	Rafael Lobato	2
Jonas Hirata	hirata@k19.com.br	11 23873791	1.76	Marcelo Martins	1
Jonas Hirata	hirata@k19.com.br	11 23873791	1.76	Rafael Lobato	2

Tabela 2.3: Resultado da consulta: `SELECT * FROM Aluno, Professor`

Duas tabelas podem ter colunas com o mesmo nome. Nesse caso, para recuperá-las, devemos eliminar a ambiguidade utilizando os nomes das tabelas.

```
1 SELECT Aluno.nome, Professor.nome FROM Aluno, Professor;
```

Código SQL 2.4: Recuperando as colunas `nome` *das tabelas* `Aluno` *e* `Professor`

Aluno x Professor	
nome	nome
Rafael Cosentino	Marcelo Martins
Rafael Cosentino	Rafael Lobato
Jonas Hirata	Marcelo Martins
Jonas Hirata	Rafael Lobato

Tabela 2.4: Resultado da consulta: `SELECT Aluno.nome, Professor.nome FROM Aluno, Professor`

As tabelas e as colunas consultadas podem ser "apelidadas". Esse recurso pode facilitar a criação das consultas e a análise dos resultados. Para isso, devemos utilizar o comando `AS`.

```
1 SELECT a.nome AS aluno_nome, p.nome AS professor_nome FROM Aluno AS a, Professor AS p;
```

Código SQL 2.5: Utilizando "apelidos" nas tabelas e colunas

Aluno x Professor	
aluno_nome	professor_nome
Rafael Cosentino	Marcelo Martins
Rafael Cosentino	Rafael Lobato
Jonas Hirata	Marcelo Martins
Jonas Hirata	Rafael Lobato

Tabela 2.5: Resultado da consulta: `SELECT a.nome AS aluno_nome, p.nome AS professor_nome FROM Aluno AS a, Professor AS p`

WHERE

Os resultados de uma consulta podem ser filtrados através do comando `WHERE`. Veja o exemplo abaixo.

```
1 SELECT * FROM Aluno WHERE altura > 1.80;
```

Código SQL 2.6: Aplicando o comando `WHERE`

Aluno			
nome	email	telefone	altura
Rafael Cosentino	cosentino@k19.com.br	11 23873791	1.87

Tabela 2.6: Resultado da consulta: `SELECT * FROM Aluno WHERE altura > 1.80`

Eis uma lista de algumas funções e operadores de comparação do MySQL Server que podem ser utilizados com o comando `WHERE`:

- =

```
1 SELECT * FROM Aluno WHERE altura = 1.8;
```

- <> ou !=

```
1 SELECT * FROM Aluno WHERE altura <> 1.8;
2 SELECT * FROM Aluno WHERE altura != 1.8;
```

- <=

```
1 SELECT * FROM Aluno WHERE altura <= 1.8;
```

- <

```
1 SELECT * FROM Aluno WHERE altura < 1.8;
```

- >=

```
1 SELECT * FROM Aluno WHERE altura >= 1.8;
```

- >

```
1 SELECT * FROM Aluno WHERE altura > 1.8;
```

- IS

```
1 SELECT * FROM Aluno WHERE aprovado IS TRUE;
```

- IS NOT

```
1 SELECT * FROM Aluno WHERE aprovado IS NOT TRUE;
```

- IS NULL

```
1 SELECT * FROM Aluno WHERE nome IS NULL;
```

- IS NOT NULL

```
1 SELECT * FROM Aluno WHERE nome IS NOT NULL;
```

- BETWEEEN . . . AND . . .

```
1 SELECT * FROM Aluno WHERE altura BETWEEN 1.5 AND 1.8;
```

- NOT BETWEEEN . . . AND . . .

```
1 SELECT * FROM Aluno WHERE altura NOT BETWEEEN 1.5 AND 1.8;
```

- LIKE

```
1 SELECT * FROM Aluno WHERE nome LIKE 'Rafael%';
```

- NOT LIKE

```
1 SELECT * FROM Aluno WHERE nome NOT LIKE 'Rafael%';
```

- IN()

```
1 SELECT * FROM Aluno WHERE altura IN (1.5, 1.6, 1.7, 1.8);
```

- NOT IN()

```
1 SELECT * FROM Aluno WHERE altura NOT IN (1.5, 1.6, 1.7, 1.8);
```

Eis uma lista dos operadores lógicos do MySQL Server que também podem ser utilizados com o comando `WHERE`.

- NOT ou !

```
1 SELECT * FROM Aluno WHERE NOT altura = 1.80;
2 SELECT * FROM Aluno WHERE ! (altura = 1.80);
```

- AND ou &&

```
1 SELECT * FROM Aluno WHERE altura < 1.8 AND nome LIKE 'Rafael%';
2 SELECT * FROM Aluno WHERE altura < 1.8 && nome LIKE 'Rafael%';
```

- OR ou ||

```
1 SELECT * FROM Aluno WHERE altura < 1.8 OR nome LIKE 'Rafael%';
2 SELECT * FROM Aluno WHERE altura < 1.8 || nome LIKE 'Rafael%';
```

- XOR

```
1 SELECT * FROM Aluno WHERE altura < 1.8 XOR nome LIKE 'Rafael%';
```

Exercícios de Fixação

1 Crie uma tabela `Aluno` com as colunas `nome` (`VARCHAR(255)`), `email` (`VARCHAR(255)`) `telefone` (`VARCHAR(10)`), `altura` (`DECIMAL(3,2)`) e `aprovado` (`TINYINT(1)`).

```
1 CREATE TABLE Aluno(
```

```
  nome VARCHAR(255),
  email VARCHAR(255),
  telefone VARCHAR(255),
  altura DECIMAL(3,2),
  aprovado TINYINT(1)
)
ENGINE = InnoDB;
```

Código SQL 2.27: Resposta do exercício

2 Insira alguns registros na tabela evitando valores repetidos.

3 Utilizando a cláusula `WHERE` refaça os exemplos criando uma consulta para cada tipo de operador.

```
SELECT * FROM Aluno WHERE altura = 1.8;
SELECT * FROM Aluno WHERE altura != 1.8;
SELECT * FROM Aluno WHERE altura <= 1.8;
SELECT * FROM Aluno WHERE altura < 1.8;
SELECT * FROM Aluno WHERE altura >= 1.8;
SELECT * FROM Aluno WHERE altura > 1.8;
SELECT * FROM Aluno WHERE aprovado IS TRUE;
SELECT * FROM Aluno WHERE aprovado IS NOT TRUE;
SELECT * FROM Aluno WHERE nome IS NULL;
SELECT * FROM Aluno WHERE nome IS NOT NULL;
SELECT * FROM Aluno WHERE altura BETWEEN 1.5 AND 1.8;
SELECT * FROM Aluno WHERE altura NOT BETWEEEN 1.5 AND 1.8;
SELECT * FROM Aluno WHERE nome LIKE 'Rafael%';
SELECT * FROM Aluno WHERE nome NOT LIKE 'Rafael%';
SELECT * FROM Aluno WHERE altura IN (1.5, 1.6, 1.7, 1.8);
SELECT * FROM Aluno WHERE altura NOT IN (1.5, 1.6, 1.7, 1.8);
SELECT * FROM Aluno WHERE NOT altura = 1.80;
SELECT * FROM Aluno WHERE altura < 1.8 AND nome LIKE 'Rafael%';
SELECT * FROM Aluno WHERE altura < 1.8 OR nome LIKE 'Rafael%';
SELECT * FROM Aluno WHERE altura < 1.8 XOR nome LIKE 'Rafael%';
```

Código SQL 2.28: Resposta do exercício

Obs: nas consultas utilize valores que façam sentido de acordo com os valores que você inseriu na tabela.

ORDER BY

Os resultados de uma consulta podem ser ordenados através do comando `ORDER BY`. Para utilizar esse comando, é necessário indicar as colunas que serão utilizadas na ordenação dos registros. Veja o exemplo abaixo.

```
SELECT * FROM Aluno ORDER BY altura;
```

Código SQL 2.29: Ordenando os registros da tabela `Aluno` *pela coluna* `altura`

Aluno			
nome	email	telefone	altura
Jonas Hirata	hirata@k19.com.br	11 23873791	1.76
Rafael Cosentino	cosentino@k19.com.br	11 23873791	1.87

Tabela 2.7: Resultado da consulta: `SELECT * FROM Aluno ORDER BY altura`

No exemplo acima, dois registros podem possuir a mesma altura. É possível definir uma segunda coluna para "desempatar". Analogamente, podemos definir uma terceira coluna depois uma quarta e assim sucessivamente. Observe o código abaixo.

```
1 SELECT * FROM Aluno ORDER BY altura, nome;
```

Código SQL 2.30: Definindo uma sequência de colunas para realizar a ordenação dos registros

Para obter uma ordenação invertida, devemos aplicar o comando DESC. Esse comando é o contrário do comando ASC. Esses comandos são utilizados na consulta abaixo.

```
1 SELECT * FROM Aluno ORDER BY altura DESC, nome ASC;
```

Código SQL 2.31: Aplicando os comandos DESC *e* ASC

Essa última consulta, ordena os registros utilizando a coluna altura de forma decrescente. Caso ocorra um "empate" a coluna nome será utilizada de forma crescente para tentar "desempatar".

Exercícios de Fixação

4 Utilizando a tabela Aluno crie uma consulta que traga todos os alunos, sendo que os primeiro devem ser listados os alunos aprovados e em seguida os reprovados.

```
1 SELECT * FROM Aluno ORDER BY aprovado DESC;
```

Código SQL 2.32: Resposta do exercício

5 Utilizando a tabela Aluno crie uma consulta que traga todos os alunos aprovados ordenados pelo e-mail. Para desempate utilize as colunas altura (na orderm descrescente) e depois nome.

```
1 SELECT * FROM Aluno WHERE aprovado = 1 ORDER BY email, altura DESC, nome;
```

Código SQL 2.33: Resposta do exercício

Funções de Agrupamento

O resultado de uma consulta pode ser processado e algumas informações podem ser obtidas. Por exemplo, podemos obter o valor máximo ou mínimo de uma coluna numérica. É possível contabilizar a quantidade de registros obtidos através de uma consulta. Também podemos calcular a soma ou a média de uma coluna numérica entre outras informações.

Eis uma lista com as principais funções de agrupamento do MySQL Server e a sintaxe para aplicá-las:

- COUNT

```
1 SELECT COUNT(*) FROM Aluno;
```

- AVG

```
SELECT AVG(altura) FROM Aluno;
```

- SUM

```
SELECT SUM(altura) FROM Aluno;
```

- MAX

```
SELECT MAX(altura) FROM Aluno;
```

- MIN

```
SELECT MIN(altura) FROM Aluno;
```

- VARIANCE

```
SELECT VARIANCE(altura) FROM Aluno;
```

- STD ou STDDEV

```
SELECT STD(altura) FROM Aluno;
SELECT STDDEV(altura) FROM Aluno;
```

Exercícios de Fixação

6 Utilizando a tabela `Aluno` crie uma consulta calcule a média das alturas dos alunos reprovados.

```
SELECT AVG(altura) FROM Aluno WHERE aprovado = 0;
```

Código SQL 2.41: Resposta do exercício

7 Utilizando a tabela `Aluno` crie uma consulta calcule a variância das alturas dos alunos com mais de 1,7m.

```
SELECT VARIANCE(altura) FROM Aluno WHERE altura > 1.7;
```

Código SQL 2.42: Resposta do exercício

GROUP BY

Os registros obtidos através de uma consulta podem ser agrupados com o comando `GROUP BY` e uma função de agrupamento pode ser aplicada nos grupos obtidos.

Por exemplo, queremos saber quantos alunos foram aprovados e quantos foram reprovados. Para isso, é necessário agrupar os alunos aprovados e os reprovados e depois contabilizar a quantidade de registros de cada grupo. Veja a consulta abaixo.

```
SELECT aprovado, COUNT(*) FROM Aluno GROUP BY aprovado;
```

Podemos agrupar os registros utilizando várias colunas. Por exemplo, queremos saber quantos homens e quantas mulheres foram aprovados ou reprovados.

```
1 SELECT sexo, aprovado, COUNT(*) FROM Aluno GROUP BY sexo, aprovado;
```

Exercícios de Fixação

8 Utilizando a tabela `Aluno` crie uma consulta que calcule o número de alunos aprovados cujos nomes começam com a letra **A** ou terminam com a letra **A**, mas que não começam textbfe terminam com a letra **A**. Dê um apelido para a coluna com o número de alunos aprovados.

```
1 SELECT COUNT(*) AS total_aprovados
2 FROM Aluno WHERE aprovado = 1 AND nome LIKE 'A%' XOR '%A'
3 GROUP BY aprovado;
```

Código SQL 2.45: Resposta do exercício

DISTINCT

Resultados repeditos de uma consulta podem ser eliminados através do comando `DISTINCT`. Por exemplo, queremos obter uma lista das cidades onde os alunos nasceram.

```
1 SELECT DISTINCT(cidade) FROM Aluno;
```

LIMIT

A quantidade de resultados de uma consulta pode ser limitada através do comando `LIMIT`. Na consulta abaixo, os 10 primeiros registros da tabela `Aluno` são recuperados. Se a quantidade de registros nessa tabela for inferior a 10, todos os registros são recuperados.

```
1 SELECT * FROM Aluno LIMIT 10;
```

Também podemos descartar os primeiros registros do resultado de uma consulta. Para isso, basta passar dois parâmetros para o comando `LIMIT`.

```
1 SELECT * FROM Aluno LIMIT 5, 10;
```

No exemplo acima, os 5 primeiros registros da tabela `Aluno` são descartados. O resultado dessa consulta conterá no máximo 10 registros a partir do sexto.

CAPÍTULO 3

RELACIONAMENTOS

UNIQUE

Em alguns casos nossas tabelas precisam ter a garantia de que uma determinada informação seja única dentre os registros. Por exemplo, uma tabela `Cliente` poderíamos ter uma coluna `cpf` para representar o número do CPF de um determinado cliente. Nesse caso seria interessante garantir que não sejam inseridos na tabela dois clientes com o mesmo CPF ou que um cliente não seja inserido duas vezes.

Para evitar esse tipo de problema poderíamos realizar uma consulta na tabela `Cliente` antes de fazermos a inserção afim de verificarmos se já existe algum cliente cadastrado com o CPF que desejamos inserir. Essa abordagem não seria tão ruim se as operações realizadas em um banco de dados não ocorressem de forma concorrente.

Como esse cenário é muito comum, geralmente os SGBDs disponibilizam formas de garantirmos a unicidade de um registro. No caso do MySQL, podemos utilizar a restrição `UNIQUE`.

```
1  CREATE TABLE Cliente(
2    nome VARCHAR(255),
3    cpf VARCHAR(20) UNIQUE
4  )
5  ENGINE = InnoDB;
```

Código SQL 3.1: Utilizando a restrição `UNIQUE`

Também podemos apenas alterar uma coluna, caso a tabela já exista.

```
1  ALTER TABLE Cliente ADD UNIQUE (cpf);
```

Código SQL 3.2: Adicionando a restrição `UNIQUE` *em uma tabela existente*

Em uma tabela podemos ter quantas colunas com a restrição `UNIQUE` forem necessárias. Por exemplo, na tabela `Aluno` poderíamos ter a coluna `primeiro_nome` definida com a restrição `UNIQUE` e a coluna `sexo` sem a restrição. Ao tentarmos inserir um aluno do sexo masculino com o `primeiro_nome` Yuki poderemos ter um problema, pois em alguns países o nome Yuki pode ser usado tanto para homens quanto para mulheres. Nesse caso poderíamos definir a restrição `UNIQUE` em um índice composto pelas colunas `primeiro_nome` e `sexo`.

```
1  CREATE TABLE Aluno (
2    id INT NOT NULL,
3    primeiro_nome VARCHAR (255) NOT NULL,
4    sexo VARCHAR (255) NOT NULL,
5    UNIQUE INDEX(primeiro_nome, sexo)
6  )
7  ENGINE = InnoDB ;
```

Código SQL 3.3: Utilizando a restrição `UNIQUE` *em índices compostos*

Fica claro que no exemplo dado acima a nossa tabela `Aluno` permitiria, no máximo, a inserção de dois alunos com o primeiro nome Yuki: um do sexo masculino e outro do sexo feminino. Para resolver esse problema podemos adicionar outras colunas ao nosso índice composto que possui a restrição UNIQUE, por exemplo.

Exercícios de Fixação

1. Reproduza o exemplo anterior da tabela `Cliente` e tente inserir alguns registros com valores repetidos na coluna `cpf`.

```
1  CREATE TABLE Cliente(
2    nome VARCHAR(255),
3    cpf VARCHAR(20) UNIQUE
4  )
5  ENGINE = InnoDB;
```

Código SQL 3.4: Resposta do exercício

2. Reproduza o exemplo anterior da tabela `Aluno`. Não se esqueça do índice composto. Insira alguns registros para testar a restrição `UNIQUE` e observe os resultados.

```
1  CREATE TABLE Aluno (
2    id INT NOT NULL,
3    primeiro_nome VARCHAR (255) NOT NULL,
4    sexo VARCHAR (255) NOT NULL,
5    UNIQUE INDEX(primeiro_nome, sexo)
6  )
7  ENGINE = InnoDB;
```

Código SQL 3.5: Resposta do exercício

Exercícios Complementares

1. Crie ou atualize uma tabela `Livro` para que ela contenha, pelo menos, as colunas `titulo` e `autor`, ambas `VARCHAR(255)`.

2. Insira alguns valores na tabela `Livro`. Tente inserir valores com títulos e autores repetidos. Tente inserir valores com títulos e autores repetidos e observe os resultados.

3. Remova todos os registros da tabela `Livro` e faça uma alteração na mesma para que não seja permitida a inserção de valores repetidos nas colunas `titulo` e `autor`.

4. Tente inserir valores com títulos e autores repetidos e observe os resultados.

Chaves Primárias

Já vimos que em alguns momentos as tabelas necessitam a garantia da unicidade de um registro. Em alguns casos isso não basta, pois além da unicidade precisamos garantir que o valor de uma coluna não seja nulo e que tal valor seja suficiente para identificar um registro. Para situações como essa devemos utilizar a restrição `PRIMARY KEY` (chave primária).

Uma **chave primária** deve conter valores únicos, não nulos e uma tabela pode conter apenas uma coluna como chave primária.

É uma prática muito comum criarmos uma coluna com o nome `id` para armazenarmos um código de identificação do nosso registro dentro de uma tabela.

```
1  CREATE TABLE Cliente(
2    id INT NOT NULL,
3    cpf VARCHAR(20) UNIQUE,
4    nome VARCHAR(255),
5    PRIMARY KEY (id)
6  )
7  ENGINE = InnoDB;
```

Código SQL 3.8: Utilizando a restrição `PRIMARY ID`

Chaves Estrangeiras

Uma coluna com a restrição `FOREIGN KEY` (chave estrangeira) faz referência à uma chave primária definida em uma outra tabela. O uso das chaves estrangeiras nos traz alguns benefícios como previnir que uma operação realizada no banco de dados possa corromper a relação entre duas tabelas ou que dados inválidos sejam inseridos em uma coluna com a restrição `FOREIGN KEY`.

```
1  CREATE TABLE Conta(
2    id INT NOT NULL,
3    numero INT UNIQUE,
4    saldo DECIMAL(14,2),
5    limite DECIMAL(14,2),
6    PRIMARY KEY (id),
7    FOREIGN KEY (banco_id) REFERENCES Banco(id)
8  )
9  ENGINE = InnoDB;
```

Código SQL 3.9: Utilizando a restrição `FOREIGN ID`

Por enquanto a definição de chave estrangeira pode parecer um tanto vaga, porém tópicos a seguir seu funcionamento e utilidade poderão ser observadas mais claramente.

One to One

Suponha que nos foi dada a tarefa de modelar o banco de dados de uma rede social. Em algum momento iremos modelar a tabela de usuários e poderíamos chegar a algo semelhante a isso:

```
1  CREATE TABLE Usuario(
2    id INT NOT NULL,
3    nome VARCHAR(255),
4    nome_usuario VARCHAR(10),
5    senha VARCHAR(10),
6    email VARCHAR(100),
7    sexo TINYINT(1),
8    profissao VARCHAR(255),
9    onde_estudou VARCHAR(255),
10   hobbies VARCHAR(255),
11   gosto_musical VARCHAR(255),
12   PRIMARY KEY (id)
13 )
```

```
14 ENGINE = InnoDB;
```

Código SQL 3.10: Modelando a tabela `Usuario` *de uma rede social*

Não há nada de errado com a nossa tabela `Usuario`, entretanto podemos dividir a tabela em duas: uma apenas para as informações pertinentes à conta do usuário na rede social e outra para as suas informações pessoais (perfil).

```
1  CREATE TABLE Usuario(
2    id INT NOT NULL,
3    nome_usuario VARCHAR(10),
4    senha VARCHAR(10),
5    email VARCHAR(100),
6    PRIMARY KEY (id)
7  )
8  ENGINE = InnoDB;
9
10 CREATE TABLE Perfil(
11   id INT NOT NULL,
12   nome VARCHAR(255),
13   sexo TINYINT(1),
14   profissao VARCHAR(255),
15   onde_estudou VARCHAR(255),
16   hobbies VARCHAR(255),
17   gosto_musical VARCHAR(255),
18   PRIMARY KEY (id),
19   FOREIGN KEY (id) REFERENCES Usuario(id)
20 )
21 ENGINE = InnoDB;
```

Código SQL 3.11: Exemplo de relaciomento One to One

No exemplo acima acabamos de definir um relacionamento **One to One** (um para um), no qual 1 usuário está para 1 perfil assim como 1 perfil está para 1 usuário.

Repare no uso da chave estrangeira `id` na tabela `Perfil`. A coluna `id` da tabela `Perfil` faz referência à coluna `id` da tabela `Usuario` e, por ser uma chave estrangeira, o MySQL não permitirá que um valor inválido (`id` inexistente de usuário) seja atribuído à coluna `id` da tabela `Perfil`. Sem a restrição `FOREIGN KEY` poderíamos atribuir qualquer número inteiro.

Ainda com relação à chave estrangeira, se tentarmos remover do banco de dados um usuário que tenha uma entrada relacionada à ele na tabela `Perfil`, o MySQL nos informará que a operação não é permitida. Para que possamos remover o usuário devemos primeiro remover o registro relacionado da tabela `Perfil` e em seguida remover o registro do usuário.

Exercícios de Fixação

3 Reproduza o exemplo anterior das tabelas `Usuario` e `Perfil`. Insira alguns registros para testar as restrições e relacionamentos. Observe os resultados.

```
1  CREATE TABLE Usuario(
2    id INT NOT NULL,
3    nome_usuario VARCHAR(10),
4    senha VARCHAR(10),
5    email VARCHAR(100),
6    PRIMARY KEY (id)
7  )
8  ENGINE = InnoDB;
```

```
9
10 CREATE TABLE Perfil(
11   id INT NOT NULL,
12   nome VARCHAR(255),
13   sexo TINYINT(1),
14   profissao VARCHAR(255),
15   onde_estudou VARCHAR(255),
16   hobbies VARCHAR(255),
17   gosto_musical VARCHAR(255),
18   PRIMARY KEY (id),
19   FOREIGN KEY (id) REFERENCES Usuario(id)
20 )
21 ENGINE = InnoDB;
```

Código SQL 3.12: Resposta do exercício

Exercícios Complementares

5 Crie uma tabela `Livro` que contenha **apenas** as colunas `id`, `isbn` e `titulo`. Caso a tabela já exista, remova e crie ela novamente.

6 Crie também uma tabela `LivroDetalhe` que contenha as informações adicionais dos livros. Faça com que essa tabela contenha uma coluna que será a chave primária da tabela, assim como uma chave estrangeira para relacionarmos esta tabela com a tabela `Livro`.

7 Adicione alguns registros nas tabelas criadas para testar as restrições e o relacionamento one to one.

One to Many ou Many to One

Para ilustrar o relacionamento **One to Many** ou **Many to One**, vamos voltar ao exemplo da conta bancária utilizado anteriormente:

```
1  CREATE TABLE Conta(
2    id INT NOT NULL,
3    numero INT UNIQUE,
4    saldo DECIMAL(14,2),
5    limite DECIMAL(14,2),
6    banco_id INT,
7    PRIMARY KEY (id),
8    FOREIGN KEY (banco_id) REFERENCES Banco(id)
9  )
10 ENGINE = InnoDB;
```

Código SQL 3.15: Exemplo de relaciomento One to Many

No exemplo acima vimos apenas uma das pontas do relacionamento. Vamos ver como seria a outra ponta, ou seja, a tabela Banco:

```
1  CREATE TABLE Banco(
2    id INT NOT NULL,
3    nome VARCHAR(255),
4    endereco VARCHAR(255)
5    PRIMARY KEY (id)
6  )
7  ENGINE = InnoDB;
```

Código SQL 3.16: Exemplo de relaciomento One to Many - continuação

As tabelas `Banco` e `Conta` possuem um relacionamento One to Many, pois um banco pode **possuir diversas** (*many*) contas enquanto que uma conta **pertence a um único** (*one*) banco.

Exercícios de Fixação

4 Reproduza o exemplo anterior das tabelas `Conta` e `Banco`. Insira alguns registros para testar as restrições e relacionamentos. Observe os resultados.

```
1  CREATE TABLE Banco(
2    id INT NOT NULL,
3    nome VARCHAR(255),
4    endereco VARCHAR(255),
5    PRIMARY KEY (id)
6  )
7  ENGINE = InnoDB;
8
9  CREATE TABLE Conta(
10   id INT NOT NULL,
11   numero INT UNIQUE,
12   saldo DECIMAL(14,2),
13   limite DECIMAL(14,2),
14   banco_id INT,
15   PRIMARY KEY (id),
16   FOREIGN KEY (banco_id) REFERENCES Banco(id)
17 )
18 ENGINE = InnoDB;
```

Código SQL 3.17: Resposta do exercício

Exercícios Complementares

8 Crie ou altere duas tabelas: `Editora` e `Livro`. Crie as colunas e restrições necessárias para obtermos um relacionamento one to many entre elas. Dica: uma editora pode publicar diversos livros e um livro só pode pertencer à apenas uma editora.

9 Insira alguns valores para testar as restrições e relacionamentos. Observe os resultados.

Many to Many

Em um relacionamento **Many to Many** vários registros de uma tabela podem estar relacionados com vários registros de outra tabela. Para que isso seja possível é necessário que exista uma tabela intermediária que chamaremos de tabela de relacionamento. Ela recebe esse nome justamente por existir apenas para representar o relacionamento entre duas outras tabelas.

Para ilustrarmos o relacionamento **Many to Many** vamos pegar o exemplo do sistema de cadastro de alunos de uma escola. Nesse sistema um aluno pode se matricular em diversas turmas e uma turma pode ter matriculados diversos alunos.

```
CREATE TABLE Aluno(
  id INT NOT NULL,
  nome VARCHAR(255),
  email VARCHAR(255),
  data_nascimento DATETIME,
  PRIMARY KEY (id)
)
ENGINE = InnoDB;
```

Código SQL 3.19: Exemplo de relaciomento Many to Many - Tabela Aluno

```
CREATE TABLE Turma(
  id INT NOT NULL,
  inicio DATETIME,
  fim DATETIME,
  observacoes LONGTEXT,
  PRIMARY KEY (id)
)
ENGINE = InnoDB;
```

Código SQL 3.20: Exemplo de relaciomento Many to Many - Tabela Turma

Repare que tanto tabela `Aluno` quanto na tabela `Turma` não encontramos referências de uma para a outra. As o relacionamento será definido da tabela a seguir:

```
CREATE TABLE AlunoTurma(
  aluno_id INT NOT NULL,
  turma_id INT NOT NULL,
  PRIMARY KEY (aluno_id, turma_id),
  FOREIGN KEY (aluno_id) REFERENCES Aluno(id),
  FOREIGN KEY (turma_id) REFERENCES Turma(id)
)
ENGINE = InnoDB;
```

Código SQL 3.21: Exemplo de relaciomento Many to Many - Tabela AlunoTurma

Definindo as colunas `aluno_id` e `turma_id` como chave primária composta garantimos que cada registro será único e não nulo. Além disso, como ambas colunas também são chaves estrangeiras não será possível inserir um `id` inválido tanto na coluna `aluno_id` quanto na coluna `turma_id`.

Exercícios de Fixação

5 Reproduza o exemplo anterior das tabelas `Aluno`, `Turma` e `AlunoTurma`. Insira alguns registros para testar as restrições e relacionamentos. Observe os resultados.

```
CREATE TABLE Aluno(
  id INT NOT NULL,
  nome VARCHAR(255),
  email VARCHAR(255),
  data_nascimento DATETIME,
  PRIMARY KEY (id)
)
ENGINE = InnoDB;

CREATE TABLE Turma(
  id INT NOT NULL,
  inicio DATETIME,
  fim DATETIME,
  observacoes LONGTEXT,
```

```
15    PRIMARY KEY (id)
16  )
17  ENGINE = InnoDB;
18
19  CREATE TABLE AlunoTurma(
20    aluno_id INT NOT NULL,
21    turma_id INT NOT NULL,
22    PRIMARY KEY (aluno_id, turma_id),
23    FOREIGN KEY (aluno_id) REFERENCES Aluno(id),
24    FOREIGN KEY (turma_id) REFERENCES Turma(id)
25  )
26  ENGINE = InnoDB;
```

Código SQL 3.22: Resposta do exercício

Exercícios Complementares

10 Crie ou altere duas tabelas: `Autor` e `Livro`. Defina algumas colunas em ambas tabelas tendo em mente que um autor pode escrever diversos livros e um livro pode ser escrito por um ou mais autores.

11 Crie uma tabela de relacionamento entre as tabelas `Autor` e `Livro`. Não se esqueça das restrições.

12 Insira alguns valores para testar as restrições e relacionamentos. Observe os resultados.

SUBQUERIES, JOINS E UNIONS

CAPÍTULO 4

Conforme trabalhamos com um banco de dados é inevitável que em algum momento seja necessário gerar uma consulta que trabalhe com mais de uma tabela ao mesmo tempo. Neste capítulo iremos abordar as principais técnicas utilizadas em situações como esta e identificar em quais situações devemos aplicá-las.

Subqueries

Uma **subquery** é uma query como qualquer outra, porém ela é executada dentro de uma outra query de SELECT, INSERT, UPDATE ou DELETE. A função da subquery é produzir um resultado que será utilizado pela query que a contém. Alguns autores se referem à subquery como **query interna** e a query que a contém como **query externa**.

Subqueries podem ser utilizadas em qualquer parte de uma query onde uma expressão é aceita. Além disso, subqueries podem ocorrer em outras subqueries e assim por diante, ou seja, em uma query podemos encontrar vários níveis de subqueries.

Vamos supor que no sistema de cadastro de alunos de uma escola tenhamos a tabela `Nota` na qual ficam registradas as notas dos alunos em cada turma.

```
1  CREATE TABLE Nota(
2    id INT NOT NULL,
3    aluno_id INT,
4    turma_id INT,
5    nota DECIMAL(4,2),
6    PRIMARY KEY (id),
7    FOREIGN KEY (aluno_id) REFERENCES Aluno(id),
8    FOREIGN KEY (turma_id) REFERENCES Turma(id)
9  )
10 ENGINE = InnoDB;
```

Código SQL 4.1: Tabela Nota

Se quisermos saber quais foram os alunos que tiraram uma nota maior que a média das notas de cada turma, poderíamos realizar a seguinte consulta:

```
1  SELECT *
2  FROM Nota AS n1
3  WHERE n1.nota > (
4    SELECT AVG(n2.nota)
5    FROM Nota AS n2
6    WHERE n2.turma_id = n1.turma_id
7  );
```

Código SQL 4.2: Consultando notas acima da média da turma

No exemplo acima utilizamos uma subquery na cláusula `WHERE`. Repare que na subquery utiliza-

mos o valor `n1.turma` proveniente da query externa. Isso nos mostra que a subquery é dependente da query que a contém e, por isso, a chamamos de **subquery correlacionada**.

Uma subquery correlacionada, devido à sua dependência de um valor da query externa, pode custar muito processamento, pois cada registro encontrado pela query externa irá executar a subquery.

Quando uma subquery não necessita de nenhum valor da query externa nós as chamamos de **subquery independente**. Diferentemente de uma subquery correlacionada, a subquery independente pode ser executada apenas uma vez mesmo que a query externa retorne mais de um registro.

```
1  SELECT n1.*, (
2    SELECT MAX(n2.nota)
3    FROM Nota AS n2
4    WHERE n2.turma_id = 1
5  ) AS maior_nota
6  FROM Nota AS n1
7  WHERE n1.turma_id = 1;
```

Código SQL 4.3: Consultando notas de uma turma

No exemplo acima utilizamos uma subquery como um campo virtual da query externa para obtermos a maior nota de uma determinada turma. Como o valor `turma_id` não depende de um valor da query externa nossa subquery é independente.

Exercícios de Fixação

1. Reproduza os exemplos anteriores da tabela `Nota`. Observe os resultados.

```
1  SELECT *
2  FROM Nota AS n1
3  WHERE n1.nota > (
4    SELECT AVG(n2.nota)
5    FROM Nota AS n2
6    WHERE n2.turma_id = n1.turma_id
7  );
8
9  SELECT n1.*, (
10   SELECT MAX(n2.nota)
11   FROM Nota AS n2
12   WHERE n2.turma_id = 1
13 ) AS maior_nota
14 FROM Nota AS n1
15 WHERE n1.turma_id = 1;
```

Código SQL 4.4: Resposta do exercício

Exercícios Complementares

1. Crie ou altere a tabela `Livro`. A tabela deve conter uma coluna `preco` que irá armazenar o preço de cada livro. Crie uma consulta que devolva todas as colunas de todos os livros registrados. Além das colunas normais da tabela, crie uma coluna virtual que irá conter a média dos preços dos livros.

2. Ainda utilizando a tabela `Livro`, crie uma consulta que devolva todas as colunas de todos os

livros registrados cujos preços sejam **superiores** em relação aos livros mais baratos.

3 Na tabela `Livro` crie a coluna `autor_id` caso ela ainda não exista. Também crie ou altere a tabela `Autor` com, pelo menos, as colunas `id` e `nome`. Crie uma consulta que devolva todos os livros escritos por autores cujos nomes começam com a letra **A**.

Joins

Utilizamos joins do SQL para extrairmos as informações de uma ou mais tabelas em um único conjunto de resultados baseando-se nos relacionamentos entre as colunas das tabelas envolvidas.

Até agora criamos as nossas tabelas definindo uma coluna como chave primária em cada uma delas. Não fizemos isso à toa, pois agora veremos como utilizar esse relacionamento entre colunas de tabelas diferentes em uma única consulta.

Vamos voltar ao exemplo da rede social. Quando modelamos as tabelas separamos as informações do usuário em duas tabelas: `Usuario` com as informações pertinentes à conta do usuário na rede social e `Perfil` com as informações pessoais do mesmo.

```
1  CREATE TABLE Usuario(
2    id INT NOT NULL,
3    nome_usuario VARCHAR(10),
4    senha VARCHAR(10),
5    email VARCHAR(100),
6    PRIMARY KEY (id)
7  )
8  ENGINE = InnoDB;
9
10 CREATE TABLE Perfil(
11   id INT NOT NULL,
12   nome VARCHAR(255),
13   sexo TINYINT(1),
14   profissao VARCHAR(255),
15   onde_estudou VARCHAR(255),
16   hobbies VARCHAR(255),
17   gosto_musical VARCHAR(255),
18   PRIMARY KEY (id),
19   FOREIGN KEY (id) REFERENCES Usuario(id)
20 )
21 ENGINE = InnoDB;
```

Código SQL 4.8: Tabela `Usuario` *e* `Perfil`

Para trazer as informações das duas tabelas em um único conjunto de resultados utilizaremos a instrução `JOIN`.

```
1  SELECT *
2  FROM Usuario AS u
3  JOIN Perfil AS p
```

Código SQL 4.9: Cruzando os dados dos usuários e seus perfis

Repare que o resultado obtido não foi o desejado, pois para cada registro da tabela `Usuario` foi feita uma relação com todos os registros da tabela `Perfil`. Isso ocorreu porque não informamos qual a coluna queremos utilizar para definir o relacionamento entre as tabelas.

Para definirmos qual a coluna irá definir o relacionamento entre as tabelas devemos utilizar a

instrução JOIN juntamente com a instrução ON.

```
1  SELECT *
2  FROM Usuario AS u
3  JOIN Perfil AS p
4  ON u.id = p.id;
```

Código SQL 4.10: Consultando usuários e seus respectivos perfis

No exemplo acima utilizamos a instrução JOIN, porém o MySQL oferece outros tipos de joins. Abaixo segue uma lista com cada tipo:

- JOIN: retorna registros quando existe algum valor na coluna de relacionamento em pelo menos uma das tabelas.
- LEFT JOIN: retorna registros quando existe algum valor na coluna de relacionamento da tabela informada à esquerda na consulta.
- RIGHT JOIN: retorna registros quando existe algum valor na coluna de relacionamento da tabela informada à direita na consulta.

Exercícios de Fixação

2 Reproduza os exemplos anteriores sem o uso da instrução ON e depois com. Insira alguns registros nas tabelas e observe os resultados.

```
1  CREATE TABLE Usuario(
2    id INT NOT NULL,
3    nome_usuario VARCHAR(10),
4    senha VARCHAR(10),
5    email VARCHAR(100),
6    PRIMARY KEY (id)
7  )
8  ENGINE = InnoDB;
9
10 CREATE TABLE Perfil(
11   id INT NOT NULL,
12   nome VARCHAR(255),
13   sexo TINYINT(1),
14   profissao VARCHAR(255),
15   onde_estudou VARCHAR(255),
16   hobbies VARCHAR(255),
17   gosto_musical VARCHAR(255),
18   PRIMARY KEY (id),
19   FOREIGN KEY (id) REFERENCES Usuario(id)
20 )
21 ENGINE = InnoDB;
22
23 SELECT *
24 FROM Usuario AS u
25 JOIN Perfil AS p;
26
27 SELECT *
28 FROM Usuario AS u
29 JOIN Perfil AS p
30 ON u.id = p.id;
```

Código SQL 4.11: Resposta do exercício

Exercícios Complementares

4 Crie ou altere a tabela `Livro` e faça com que ela contenha a coluna `autor_id` caso esta ainda não exista. Também crie ou altere a tabela `Autor` com, pelo menos, as colunas `id` e `nome`. Crie uma consulta que devolva todos os livros escritos por autores cujos nomes começam com a letra **A**.

5 Crie uma consulta que gere como resultado uma lista com todos os autores que possuam livros publicados. Além disso, o resultado deve conter o número de livros que cada autor publicou.

6 Refaça o exercício anterior ordenando decrescentemente o resultado pelo nome do autor.

Unions

Nos exemplos anteriores conseguimos obter resultados de duas ou mais tabelas concatenando suas colunas ou criando campos virtuais.

Porém nem sempre é isso que desejamos. Às vezes queremos que duas ou mais tabelas sejam unidas aumentando o número de **registros**. Para atingirmos este objetivo devemos utilizar a instrução `UNION`.

Vamos supor que, em nossa rede social, armazenamos os usuários administrativos do site em uma tabela diferente dos usuários normais. A tabela poderia ser assim:

```
1  CREATE TABLE UsuarioAdministrativo(
2    id INT NOT NULL,
3    nome_usuario VARCHAR(10),
4    senha VARCHAR(10),
5    email VARCHAR(100),
6    grupo INT,
7    PRIMARY KEY (id)
8  )
9  ENGINE = InnoDB;
```

Código SQL 4.15: Tabela UsuarioAdministrativo

Para obter uma lista com o nome e e-mail de todos os usuários, inclusive os usuários administrativos utilizaremos a instrução `UNION`.

```
1  SELECT nome_usuario, email
2  FROM Usuario
3  UNION
4  SELECT nome_usuario, email
5  FROM UsuarioAdministrativo;
```

Código SQL 4.16: Obtendo a lista de todos os usuários da rede social

Repare que no primeiro e segundo `SELECT` escolhemos quais colunas queríamos no resultado. A instrução `UNION` nos obriga que cada `SELECT` retorne o mesmo número de colunas. Como a tabela `UsuarioAdministrador` possui uma coluna a mais, se tivéssemos utilizado o caractere *wildcard* `*` em ambas colunas, nossa consulta teria retornado um erro.

Por padrão a instrução `UNION` seleciona registros distintos. Portanto, caso um usuário administra-

tivo também seja um usuário normal da rede social, com nome e e-mail cadastrados com os mesmos valores nas duas tabelas, a consulta do exemplo acima nos teria retornado apenas um resultado para esse usuário.

Se quisermos que o resultado traga entradas duplicadas, devemos utilizar a instrução `UNION ALL`.

```
1  SELECT nome_usuario, email
2  FROM Usuario
3  UNION ALL
4  SELECT nome_usuario, email
5  FROM UsuarioAdministrativo;
```

Código SQL 4.17: Obtendo a lista de todos os usuários da rede social

Exercícios de Fixação

3 Reproduza os exemplos anteriores utilizando a instrução `UNION` e depois `UNION ALL`. Insira alguns registros nas tabelas e observe os resultados.

```
1  CREATE TABLE UsuarioAdministrativo(
2    id INT NOT NULL,
3    nome_usuario VARCHAR(10),
4    senha VARCHAR(10),
5    email VARCHAR(100),
6    grupo INT,
7    PRIMARY KEY (id)
8  )
9  ENGINE = InnoDB;
10
11 SELECT nome_usuario, email
12 FROM Usuario
13 UNION
14 SELECT nome_usuario, email
15 FROM UsuarioAdministrativo;
16
17 SELECT nome_usuario, email
18 FROM Usuario
19 UNION ALL
20 SELECT nome_usuario, email
21 FROM UsuarioAdministrativo;
```

Código SQL 4.18: Resposta do exercício

Exercícios Complementares

7 Utilizando as tabelas `Usuario` e `UsuarioAdministrativo` do exercício de fixação, crie uma consulta que gere uma lista com todos os usuarios (administrativos e normais). Além disso, quando um usuário não possuir um valor na coluna `nome_usuario`, imprima no seu lugar o e-mail deste usuário.

RESPOSTAS

APÊNDICE A

Resposta do Complementar 3.1

```
1  CREATE TABLE Livro (
2    titulo VARCHAR (255) NOT NULL,
3    autor VARCHAR (255) NOT NULL
4  )
5  ENGINE = InnoDB;
```

Código SQL 3.6: Resposta do exercício

Resposta do Complementar 3.3

```
1  ALTER TABLE Livro ADD UNIQUE INDEX(titulo, autor)
```

Código SQL 3.7: Resposta do exercício

Resposta do Complementar 3.5

```
1  CREATE TABLE Livro(
2    id INT NOT NULL,
3    isbn BIGINT,
4    titulo VARCHAR(255),
5    PRIMARY KEY (id)
6  )
7  ENGINE = InnoDB;
```

Código SQL 3.13: Resposta do exercício

Resposta do Complementar 3.6

```
1  CREATE TABLE LivroDetalhe(
2    id INT NOT NULL,
3    ano INT,
4    edicao INT,
5    preco DECIMAL(10,2),
6    PRIMARY KEY (id),
7    FOREIGN KEY (id) REFERENCES Livro(id)
8  )
9  ENGINE = InnoDB;
```

Código SQL 3.14: Resposta do exercício

Resposta do Complementar 3.8

```
1  CREATE TABLE Editora(
2    id INT NOT NULL,
3    nome VARCHAR(255),
4    PRIMARY KEY (id)
5  )
6  ENGINE = InnoDB;
7
8  CREATE TABLE Livro(
9    id INT NOT NULL,
10   titulo INT UNIQUE,
11   autor VARCHAR(255),
12   preco DECIMAL(14,2),
13   autor_id INT,
14   PRIMARY KEY (id),
15   FOREIGN KEY (autor_id) REFERENCES Autor(id)
16 )
17 ENGINE = InnoDB;
```

Código SQL 3.18: Resposta do exercício

Resposta do Complementar 3.10

```
1  CREATE TABLE Autor(
2    id INT NOT NULL,
3    nome VARCHAR(255),
4    PRIMARY KEY (id)
5  )
6  ENGINE = InnoDB;
7
8  CREATE TABLE Livro(
9    id INT NOT NULL,
10   titulo VARCHAR(255),
11   edicao INT,
12   preco DECIMAL(10,2),
13   isbn INT,
14   PRIMARY KEY (id)
15 )
16 ENGINE = InnoDB;
```

Código SQL 3.23: Resposta do exercício

Resposta do Complementar 3.11

```
1  CREATE TABLE AutorLivro(
2    autor_id INT NOT NULL,
3    livro_id INT NOT NULL,
4    PRIMARY KEY (autor_id, livro_id),
5    FOREIGN KEY (autor_id) REFERENCES Autor(id),
6    FOREIGN KEY (livro_id) REFERENCES Livro(id)
7  )
8  ENGINE = InnoDB;
```

Código SQL 3.24: Resposta do exercício

Resposta do Complementar 4.1

```sql
SELECT l1.*, (
  SELECT AVG(l2.preco)
  FROM Livro AS l2
) AS media_preco
FROM Livro AS l1;
```

Código SQL 4.5: Resposta do exercício

Resposta do Complementar 4.2

```sql
SELECT l1.*
FROM Livro AS l1
WHERE l1.preco > (
    SELECT MIN(l2.preco)
    FROM Livro AS l2
);
```

Código SQL 4.6: Resposta do exercício

Resposta do Complementar 4.3

```sql
SELECT *
FROM Livro
WHERE Livro.autor_id IN (
  SELECT id
  FROM Autor
  WHERE nome LIKE 'A%'
);
```

Código SQL 4.7: Resposta do exercício

Resposta do Complementar 4.4

```sql
SELECT Livro.*
FROM Livro
JOIN Autor
ON Livro.autor_id = Autor.id
WHERE Autor.nome LIKE 'A%';
```

Código SQL 4.12: Resposta do exercício

Resposta do Complementar 4.5

```sql
SELECT Autor.*, COUNT(Livro.id) AS total_livros
FROM Autor JOIN Livro
ON Livro.autor_id = Autor.id
GROUP BY Autor.id;
```

Código SQL 4.13: Resposta do exercício

Resposta do Complementar 4.6

```sql
SELECT * FROM (
  SELECT Autor.*, COUNT(Livro.id) AS total_livros
  FROM Autor JOIN Livro
  ON Livro.autor_id = Autor.id
  GROUP BY Autor.id
) AS A
ORDER BY A.nome DESC;
```

Código SQL 4.14: Resposta do exercício

Resposta do Complementar 4.7

```sql
SELECT nome_usuario, email
FROM Usuario
WHERE Usuario.nome_usuario IS NOT NULL
UNION ALL
SELECT nome_usuario, email
FROM UsuarioAdministrativo
WHERE UsuarioAdministrativo.nome_usuario IS NOT NULL
UNION ALL
SELECT nome_usuario, email
FROM Usuario
WHERE Usuario.nome_usuario IS NULL
UNION ALL
SELECT email, email
FROM UsuarioAdministrativo
WHERE UsuarioAdministrativo.nome_usuario IS NULL;
```

Código SQL 4.19: Resposta do exercício