

Técnico em Desenvolvimento de Sistemas

Definição de dados

Na computação, dados são toda a informação que pode ser traduzida eficientemente para a leitura, a transferência e o processamento.

O conceito de dados no contexto da computação tem suas raízes no trabalho de Claude Shannon, que foi um matemático norte-americano conhecido como o pai da teoria da informação. Ele introduziu conceitos digitais binários com base na aplicação da lógica booleana (ou booliana) de dois valores a circuitos eletrônicos, o que é hoje a base para toda a comunicação e computação desses dados.

Todos esses dados precisam ser armazenados de alguma maneira para que se possa acessá-los e alterá-los. Considerando que todo dado computacional é representado utilizando apenas o padrão de dois *bits* – 0 e 1 – e que esses *bits* representam toda e qualquer informação armazenada, os bancos de dados vieram para tornar o acesso a esses dados muito mais fácil, rápido e confiável.

Os *bits*, sim, com “b” minúsculo, têm sua própria unidade de medida. Observe:

Medidas comuns de armazenamento de dados	
Unidade	Valor
<i>bit</i>	1 <i>bit</i>
<i>byte</i>	8 <i>bits</i>
<i>kilobyte</i>	1024 <i>bytes</i>
<i>megabyte</i>	1024 <i>kilobytes</i>
<i>gigabyte</i>	1024 <i>megabytes</i>
<i>terabyte</i>	1024 <i>gigabytes</i>
<i>petabyte</i>	1024 <i>terabytes</i>
<i>exabyte</i>	1024 <i>petabytes</i>
<i>zettabyte</i>	1024 <i>exabytes</i>
<i>yottabyte</i>	1024 <i>zettabytes</i>
<i>brontobyte</i>	1024 <i>yottabytes</i>

Os dados armazenados com a utilização da SQL (*structured query language*), por sua vez, são os que ficam guardados em tabelas e podem ser de tipos variados, como números inteiros, cadeias de caracteres, números flutuantes (com casa decimal), datas, horas e minutos, valores booleanos (verdadeiro ou falso) e muitos outros tipos de valores, que você verá em seguida neste conteúdo.

Criação de banco de dados

Um banco de dados permite que sejam armazenados vários tipos de dados e seu custo de implementação e velocidade o tornam atualmente a maneira mais utilizada no planeta para se armazenar informação.

Um banco de dados nada mais é do que uma coleção de dados organizados de modo que seja possível efetuar buscas e transações fácil e rapidamente, mesmo quando se está trabalhando com grandes quantidades de dados. No contexto técnico, isso significa que a principal função do banco de dados é interagir com os sistemas a ele conectados, capturar, analisar, calcular e responder rápida e organizadamente.

Com um banco de dados, é possível também automatizar uma série de procedimentos utilizando a linguagem SQL. Por exemplo: ao invés de analisar a variação de preços em uma tabela de preços, você pode automatizar a criação de relatórios com estes valores e os avisos baseados neles.

No passado, apenas grandes empresas contavam com bancos de dados para gerir as suas informações, mas, atualmente, em qualquer computador simples, com *softwares* gratuitos, pode-se implementar um poderoso banco de dados pessoal ou que possa servir para armazenar os dados de qualquer programa de computador que se queira.

Para mais informações sobre a instalação e os primeiros passos com o MySQL Workbench, consulte o conhecimento **Banco de Dados**, desta unidade curricular.

Para criar uma nova base de dados, primeiramente abra o MySQL Workbench e conecte à instância local na tela inicial.

Ao conectar, nenhum banco de dados estará criado. Para criar um banco, utilize alguns recursos disponibilizados pelo próprio Workbench. Porém, pelo padrão da SQL, você pode utilizar o comando **CREATE DATABASE**, com a seguinte sintaxe:

```
CREATE DATABASE <nome do banco de dados>;
```

Neste conteúdo, você trabalhará com uma base de dados para uma farmácia. No Workbench, na aba **Query**, digite o comando de criação:

CREATE DATABASE farmacia;

Para executar esse comando, clique no ícone do raio logo acima do comando.

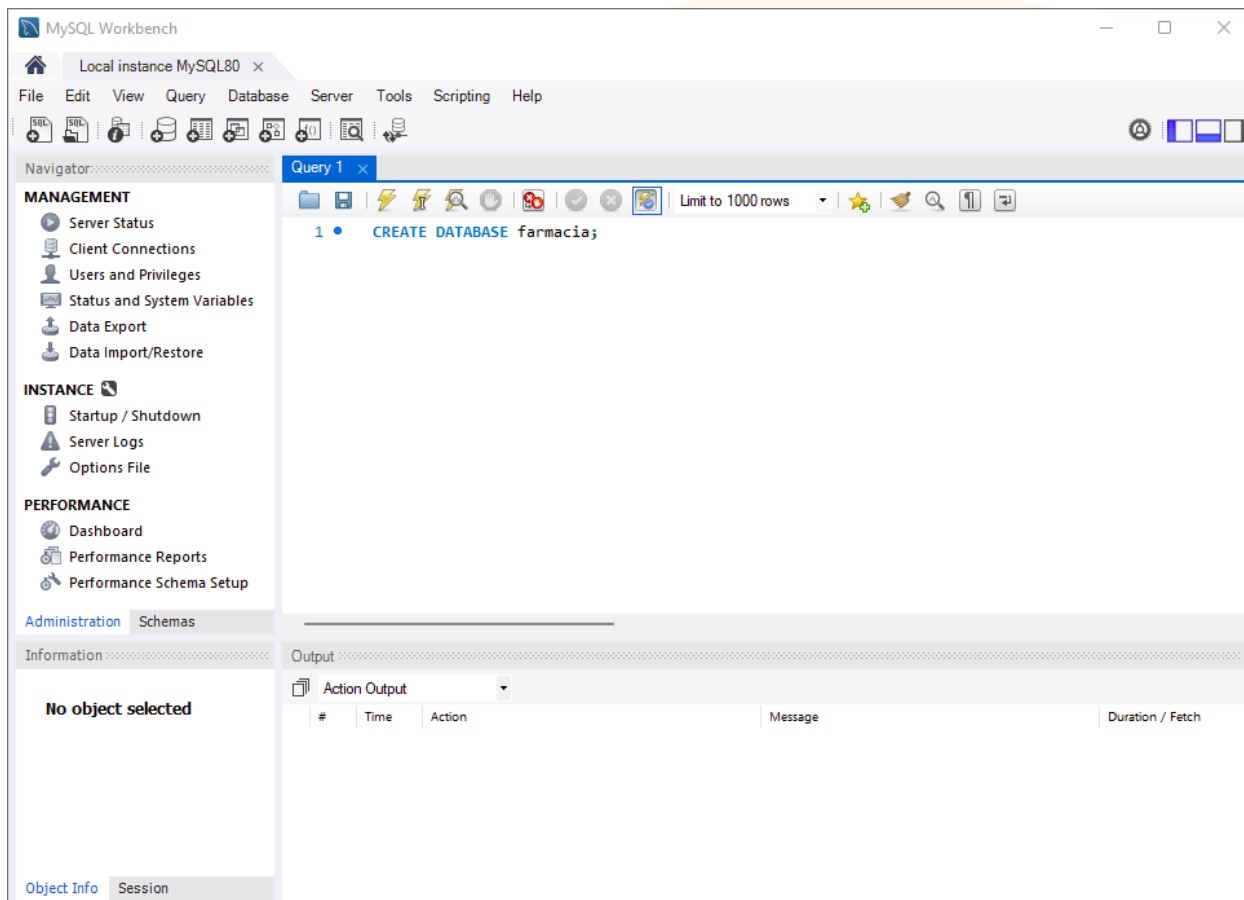
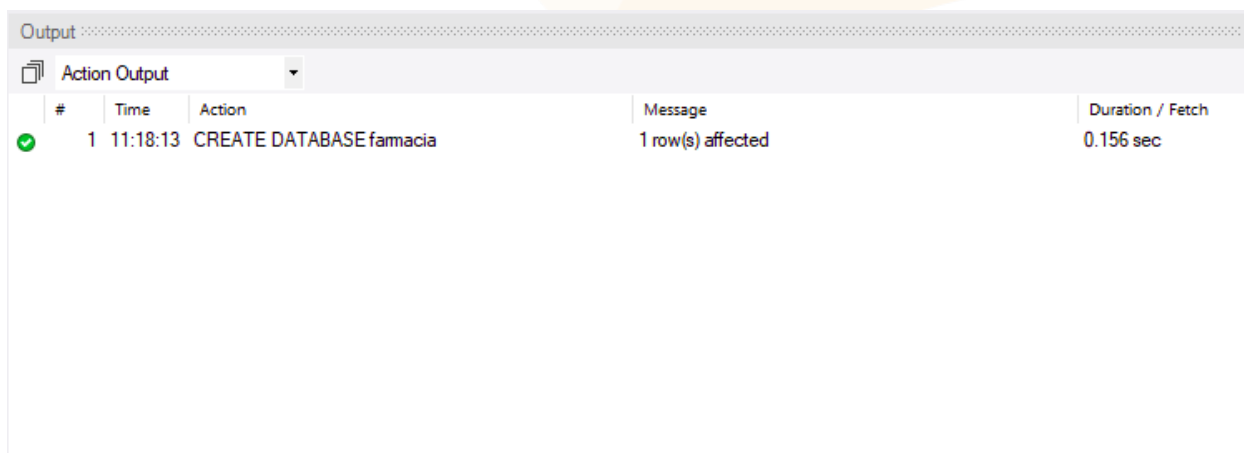


Figura 1 – Executando o comando para criar uma nova base de dados

A imagem mostra a tela principal que aparece após clicar para abrir o banco de dados da figura 2. No centro da imagem, na aba “Query 1”, está o comando “CREATE DATABASE farmacia”. Abaixo, consta a aba “Action Output”, que, até o momento, está em branco.

O ícone ⚡ executa as linhas selecionadas do *script* ou todos os comandos presentes na janela de **Query**, caso não haja seleção. No Windows, você pode utilizar o atalho **Ctrl + Shift + Enter**. O ícone 🛠, por sua vez, executa apenas a linha de *script* em que o cursor estiver no editor **Query**; no Windows, o atalho é **Ctrl + Enter**.

Após executar esse comando, você receberá na aba **Output**, na parte de baixo da janela, a saída dessa execução, em que mostrará a hora em que o comando foi executado, a ação que esse comando executou, quantas linhas foram afetadas (o que você aprenderá em breve) e quanto tempo o comando levou para ser executado.



The screenshot shows the 'Output' window in MySQL Workbench. The 'Action Output' tab is selected. It displays a table with the following data:

#	Time	Action	Message	Duration / Fetch
1	11:18:13	CREATE DATABASE farmacia	1 row(s) affected	0.156 sec

Figura 2 – Detalhe da aba **Output** na parte inferior da tela do Workbench após a execução do comando **CREATE DATABASE**

Com isso, você criou sua primeira base de dados, chamada de “farmácia”. Para listar as bases de dados que existem no servidor, você pode utilizar o comando:

```
SHOW DATABASES;
```

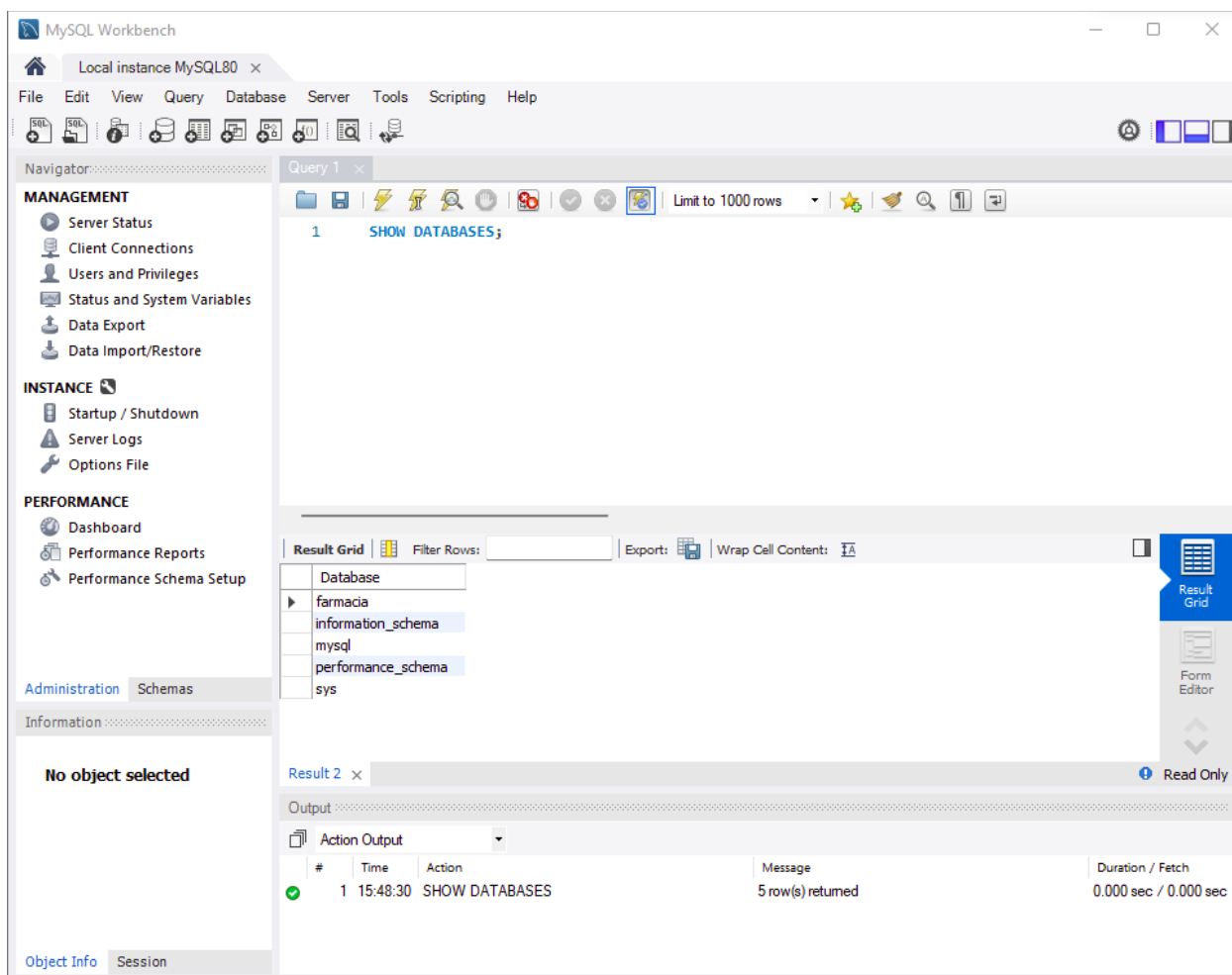



Figura 3 – Aba Result Grid, aberta depois de se executar o comando SHOW DATABASES

Quando você executar esse comando, um *grid* de resultado será mostrado com as informações da base de dados criada anteriormente, bem como a aba abaixo da execução do comando.

Visualmente, é possível acessar a aba **SCHEMAS** na seção **Navigador**, à esquerda. Ali estarão listados todos os bancos de dados presentes no servidor MySQL. O banco de dados **sys** já é criado automaticamente pelo sistema gerenciador de banco de dados (SGBD) e, em cada um deles, é possível expandir sua visão para listar as tabelas e outros recursos presentes. Pode ser necessário atualizar a visão pelo ícone  para mostrar o banco de dados recém-criado.

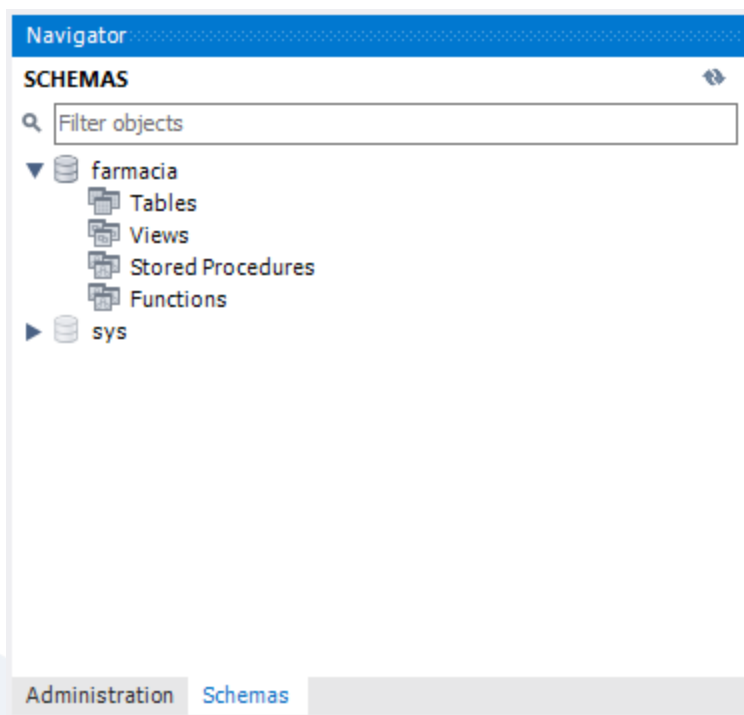


Figura 4 – Aba Schemas da seção Navigator, à esquerda na tela principal do Workbench

Como você pode ter percebido, o MySQL, assim como outros SGBDs, denomina banco de dados como *schemas*. *Schema* é a representação lógica que descreve a definição estrutural de um banco de dados. Na prática, são considerados sinônimos *schema* e *database*.

Para acionar o banco de dados e tornar possível fazer consultas sobre ele, aplique o comando **USE**, com a seguinte sintaxe.

```
USE <nome do banco de dados>;
```

Por último, você pode usar o comando **SHOW TABLES**; para listar as tabelas que estão na sua base de dados. Veja na figura a seguir os comandos **USE** para o banco de dados **farmacia** seguido do comando **SHOW TABLES** para mostrar todas as tabelas criadas no banco de dados selecionado.

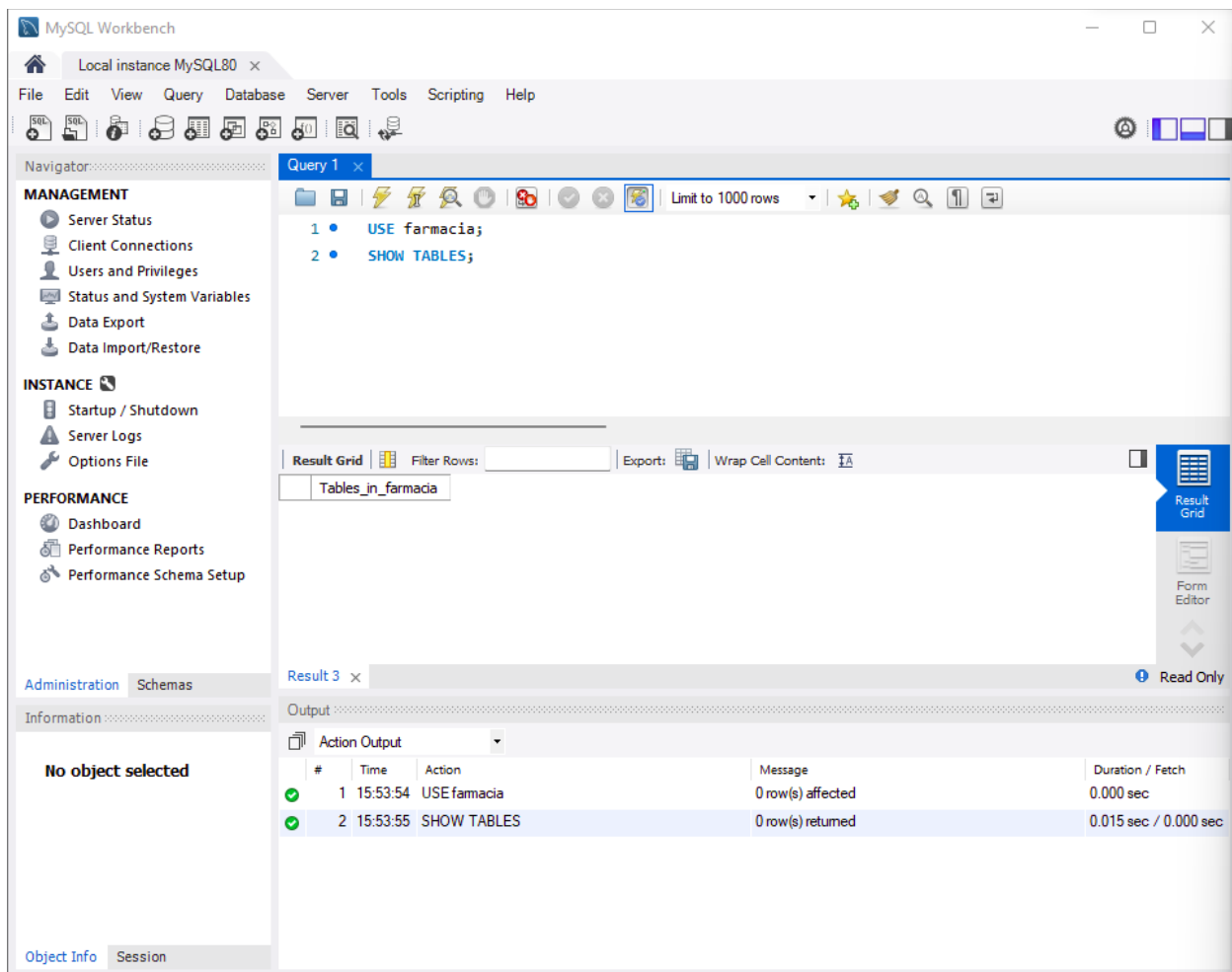


Figura 5 – Comandos **USE** e **SHOW TABLES**

Após a execução dos comandos, percebe-se que a base de dados ainda não tem nenhuma tabela. Como dito anteriormente, as tabelas também podem ser consultadas pela seção **Navigator**, na aba **Schemas**, expandindo o banco de dados selecionado.

Na aba **Schemas**, ao clicar duas vezes sobre um banco de dados, note que seu nome fica em negrito. Essa é uma operação correspondente ao comando **USE** aplicado anteriormente.

A seguir, você criará sua primeira tabela.

Criação de tabelas

Depois de criar o seu banco de dados, elabore então tabelas para armazenar os dados dos remédios do seu banco de dados **farmacia**.

As tabelas que serão criadas, são muito semelhantes a uma tabela no papel ou uma planilha eletrônica do Excel, formadas por linhas e colunas, sendo o número e a ordem das colunas fixos, cada coluna com o seu nome. Já o número de linhas é variável, pois é baseado em quanta informação “alimentará” essa tabela.

Cada coluna contém um tipo de dado e este tipo de dado restringe que valores possam ser colocados nela. Por exemplo, uma coluna que foi criada para armazenar valores numéricos não poderá conter cadeias de caracteres. Já uma coluna feita para armazenar cadeias de caracteres pode armazenar praticamente qualquer tipo de dado, no entanto, não poderá ser usada para realizar cálculos matemáticos com esses valores.

Para criar uma tabela de banco de dados, utilize o comando **CREATE TABLE**, que ainda permite definição de colunas e restrições (ou *constraints*), como chaves primárias e estrangeiras. A sintaxe geral de criação de tabelas é a seguinte:

```
CREATE TABLE <nome tabela> (  
<nome coluna 1> <tipo coluna 1> [restrições coluna 1],  
<nome coluna 2> <tipo coluna 2> [restrições coluna 2],  
...  
<nome coluna n> <tipo coluna n> [restrições coluna n],  
[definicoes de primary key]  
[definicoes de foreign key]  
);
```

Obs.: considere os itens entre < > como obrigatórios e os entre [] como opcionais.

Assim, na definição de uma tabela, é preciso, no mínimo, informar o nome dela e

suas colunas. Opcionalmente, é possível incluir restrições a cada coluna e definições de *constraints* de chaves (primárias ou estrangeiras) ou de índices.

Com a base de dados **Farmacia** criada, você poderá então criar uma nova tabela nessa base.

Como exemplo, defina a tabela **remedios**, que deve trazer como informações o nome do medicamento, a marca, o preço, a data de validade e uma informação indicando se ele é genérico ou não:

```
CREATE TABLE remedios (  
  nome VARCHAR(20),  
  marca VARCHAR(20),  
  preco DECIMAL(9,2),  
  generico CHAR(1),  
  validade DATE  
);
```

Antes de executar o *script*, pode ser necessário executar o comando **USE Farmacia**, para ativar o banco de dados ou ativar o banco por meio da aba **Schema**.

Antes de executar o *script*, pode ser necessário executar o comando **USE Farmacia**, para ativar o banco de dados ou ativar o banco por meio da aba **Schema**.

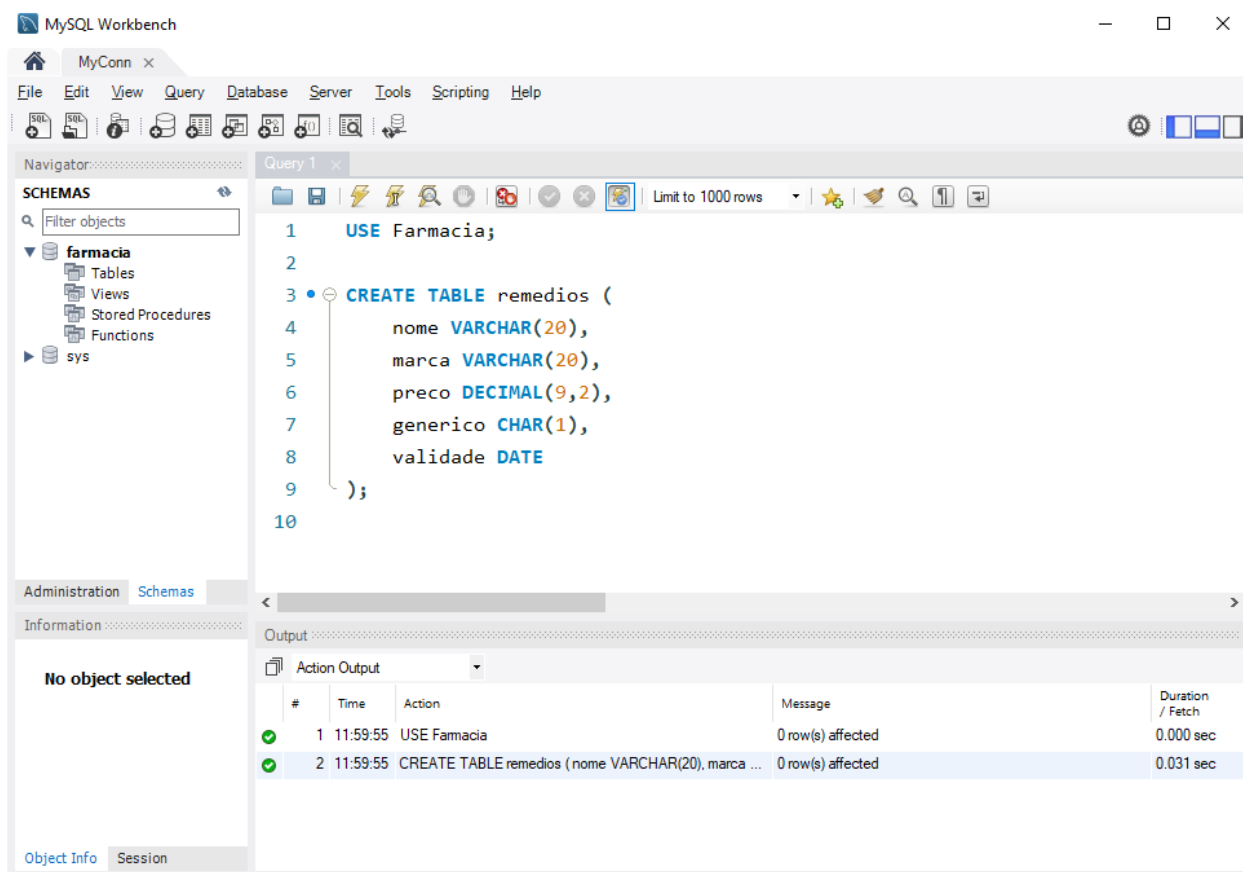


Figura 6 – Comando **CREATE TABLE** executado no Workbench

A imagem mostra a tela principal do MySQL Workbench com o comando, na aba “Query 1”:

```

CREATE TABLE remedios (
nome VARCHAR(20),
marca VARCHAR(20),
preco decimal(9,2),
generico CHAR(1),
validade DATE
);

```

Na aba “Action Output” está a confirmação de que o comando foi executado com sucesso.

Agora, quando você listar as tabelas da sua base de dados, terá listado o nome

da tabela “remédio” como resultado. Note que também é possível ver as tabelas criadas na aba lateral **Schema** após clicar no ícone **refresh**:

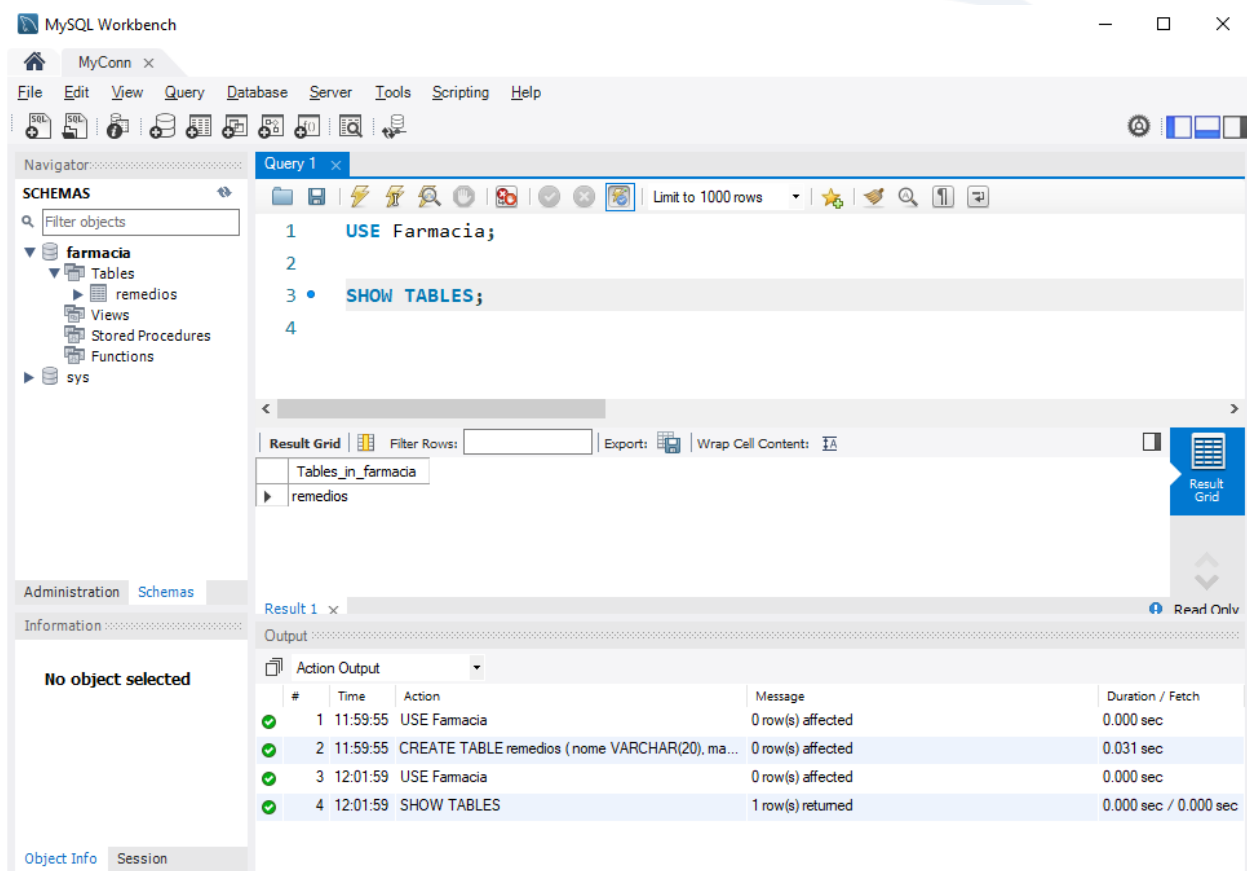


Figura 7 – Resultado do comando **SHOW TABLES**;

Com isso, você terá sua primeira tabela (remédios) em sua primeira base de dados (farmácia).

Colunas

Você, portanto, criará sua tabela com o comando:

```
CREATE TABLE remedios (
  nome VARCHAR(20),
  marca VARCHAR(20),
  preco DECIMAL(9,2),
```

```
generico CHAR(1),  
validade DATE  
);
```

Assim, você estará dizendo ao servidor para que crie uma tabela de nome **remedios** (**CREATE TABLE remedios**) e que, nessa tabela, haverá as seguintes colunas:



- nome (que deve armazenar até 20 caracteres)
- marca (que deve armazenar até 20 caracteres também)
- preco (que deve armazenar um número com casas decimais – até 9 dígitos e com duas casas depois da vírgula)
- generico (que deve armazenar um caractere – no caso, serão utilizadas as letras S e N)
- validade (que receberá uma data)

- ◆ nome (que deve armazenar até 20 caracteres)
- ◆ marca (que deve armazenar até 20 caracteres também)
- ◆ preco (que deve armazenar um número com casas decimais – até 9 dígitos e com duas casas depois da vírgula)
- ◆ generico (que deve armazenar um caractere – no caso, serão utilizadas as letras S e N)
- ◆ validade (que receberá uma data)

Dessa forma, está sendo definida cada uma das colunas para a tabela diretamente em sua criação. Cada coluna (ou atributo) corresponde a uma informação específica e de um tipo específico. Além disso, é possível incluir configurações ou restrições à coluna. A sintaxe geral é a seguinte:

<nome coluna> <tipo coluna> [restrições]

Entre as restrições, pode-se citar:

NULL ou NOT NULL

Define se uma coluna pode ou não armazenar valores nulos. Caso isso não seja informado, por padrão, considerar-se-á que são permitidos nulos.

Exemplo:

```
coluna1 VARCHAR(10) NOT NULL,  
coluna2 INT NULL,  
coluna3 DOUBLE -- equivalente a coluna 3 DOUBLE NULL
```

DEFAULT

Indica o valor padrão para essa coluna. Assim, caso, ao incluir um novo registro, não seja informado um valor para a coluna, o registro armazenará o valor padrão; caso contrário, usará o valor informado na inclusão do registro.

Exemplo:

```
coluna1 CHAR(1) NOT NULL DEFAULT('S')
```

AUTO_INCREMENT

Torna a coluna autoincremental (ou sequencial), ou seja, ao incluir um novo registro, a coluna usará o valor anterior guardado na coluna adicionado de um. É um recurso presente na maioria dos SGBDs, mas, para cada um, o comando é diferente (por exemplo: **IDENTITY** no MS SQL Server, **SEQUENCE** no Oracle, **SERIAL** no Postgre). No MySQL, a sintaxe é **AUTO_INCREMENT** e pode ser usada em apenas uma coluna, que precisa ser definida como chave.

Exemplo:

```
coluna1 INT NOT NULL AUTO_INCREMENT
```

PRIMARY KEY

Como você estudará ainda neste conteúdo, é possível definir na coluna se ela é uma chave primária. Geralmente as colunas *primary key* também são

autoincrementais.

Há ainda várias outras restrições mais complexas, mas que não são necessárias neste momento.

Observando a tabela **remedios**, você poderia incluir algumas restrições a ela. Suponha que todas as colunas, exceto marca, precisem ter valores informados. Além disso, considere que, por padrão, o remédio cadastrado não é um genérico. É possível propor então a seguinte cláusula de criação para a tabela:

```
CREATE TABLE remedios (  
  nome VARCHAR(20) NOT NULL,  
  marca VARCHAR(20),  
  preco DECIMAL(9,2) NOT NULL,  
  generico CHAR(1) NOT NULL DEFAULT('N'),  
  validade DATE NOT NULL  
);
```

Teste em sua máquina primeiro, excluindo a tabela com o comando **DROP TABLE**, cuja sintaxe é a seguinte:

```
DROP TABLE <nome da tabela>;
```

É necessário bastante atenção na execução desse comando, pois ele exclui definitivamente a estrutura de uma tabela e todos os dados contidos nela. Nesse caso, você pode excluir sem preocupações a tabela **remedios** usando o seguinte comando:

```
DROP TABLE remedios;
```

Depois disso, execute o comando **CREATE TABLE** atualizado e abordado há pouco.

Após a criação, será possível inserir alguns registros apenas para testar e se ter uma visualização mais concreta da tabela. Para isso, propõe-se o seguinte comando:

```
INSERT INTO remedios  
(nome,marca,preco,generico,  
validade) VALUES  
('Paracetamol','pfizer','5.50',  
'S',2022-10-10),  
('Dipirona','pfizer','8.50','N',  
'2022-05-08');
```

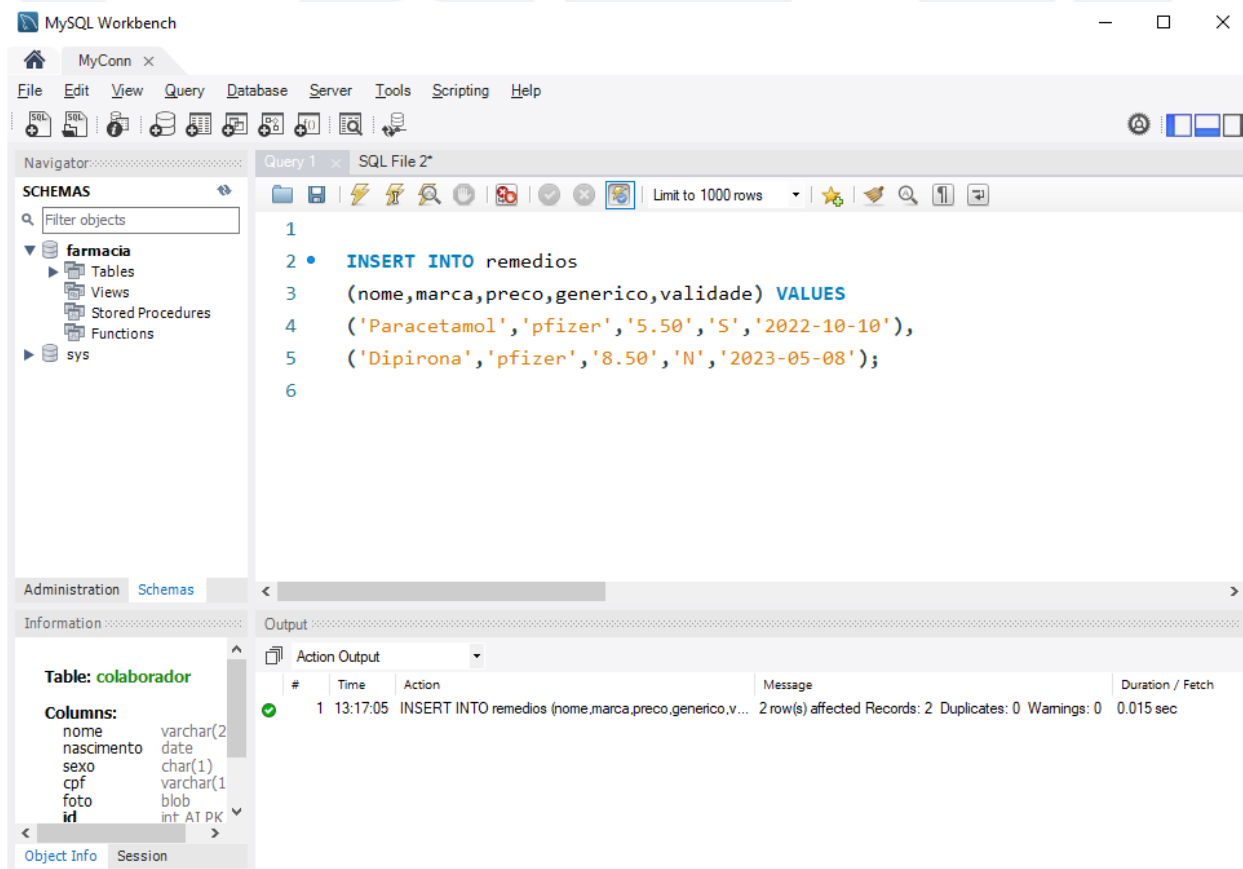


Figura 8 – Executando o comando INSERT para povoar a tabela e permitir testes

A imagem mostra a tela principal do MySQL Workbench com o comando: INSERT into remedios (nome,marca,preco,generico,validade) values ('Paracetamol','pfizer','5.50','S',2022-10-10), ('Dipirona','pfizer','8.50','N','2023-05-08'); na aba Query 1.

Na aba "Action Output", está a confirmação de que o comando foi executado com sucesso.

Você verá com mais detalhes os comandos de inclusão, exclusão e atualização de dados no conteúdo sobre manipulação de dados, desta unidade curricular.

Para visualizar os valores que você inseriu nas colunas da tabela, utilize o comando **SELECT**:

USE farmacia;

SELECT * FROM remedios;

A consulta retornará os dados cadastrados na tabela, mostrando-os na tela do Workbench abaixo da área de edição de código.

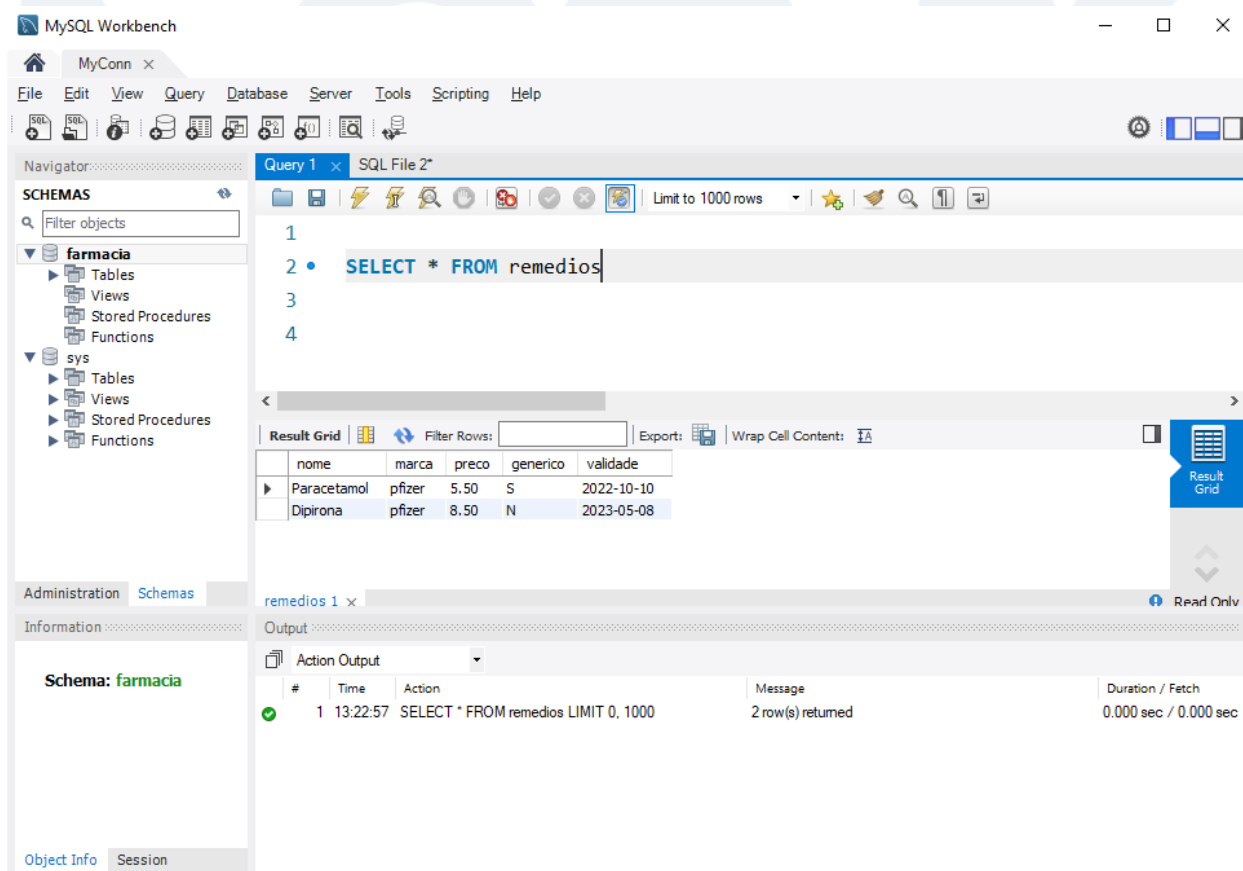


Figura 9 – Utilizando o comando SELECT para visualizar os dados incluídos na tabela remedios
Observe mais detalhadamente os comandos de pesquisa de dados no conteúdo **Consulta de dados**, desta unidade curricular.

Adicionando e removendo colunas

Uma vez construída a tabela, será possível incluir, alterar ou remover colunas. A sintaxe para inclusão de coluna é a seguinte:

```
ALTER TABLE <nome da tabela> ADD <nome da coluna nova> <tipo> [restrições];
```

Por exemplo, se fosse necessário incluir a coluna **validade** na tabela **remedios**, você poderia utilizar este comando:

```
ALTER TABLE remedios ADD validade DATE NOT NULL;
```

Por outro lado, caso você quisesse alterar o tipo, o nome ou a restrição de uma coluna, poderia utilizar a seguinte sintaxe no MySQL:

```
ALTER TABLE <nome da tabela> MODIFY COLUMN <nome coluna> <tipo> [restrições];
```

Caso preferisse que sua coluna **validade** tivesse tipo **DATETIME** ao invés de **DATE**, você poderia utilizar o seguinte comando:

```
ALTER TABLE remedios MODIFY COLUMN validade DATETIME NOT NULL;
```

Por fim, para excluir uma coluna, utilize esta sintaxe:

```
ALTER TABLE <nome da tabela> DROP COLUMN <nome coluna>;
```

Como exemplo, você pode excluir a coluna **validade** da seguinte maneira:

```
ALTER TABLE remedios DROP COLUMN validade;
```

Tipos de dados

Como visto anteriormente, ao definir uma coluna, é necessário estabelecer o tipo

de informação que ela deve carregar. A SQL estabelece alguns tipos próprios para números, texto, entre outros. Além disso, os SGBDs podem trazer variações dos tipos padrão ou tipos novos. Estude agora os tipos presentes no MySQL.

Tipos numéricos

Os tipos de dados numéricos que você utilizará em SQL consistem em números inteiros de dois, quatro e oito *bytes*, números flutuantes (com casa decimal) de quatro e oito *bytes*.

Os tipos de dados numéricos seguem esta tabela:

Nome	Tamanho	Faixa de Valores
TINYINT	1 <i>byte</i>	-128 a 127
<p>Número inteiro com a menor faixa de valores. Útil para poupar memória em colunas nas quais se sabe que não serão informados valores altos (por exemplo, a idade de uma pessoa).</p> <p>É possível usar o modificador UNSIGNED (sem sinal) para especificar que a coluna receberá apenas números positivos. Nesse caso, a faixa de valores se torna de 0 a 255</p> <p>Exemplo:</p> <p>coluna1 TINYINT--suporta até o número 127</p> <p>coluna2 TINYINT UNSIGNED --suporta até o número 255, mas apenas positivos</p>		
SMALLINT	2 <i>bytes</i>	-32768 a +32767
<p>Número inteiro com faixa pequena de valores. Com o modificador UNSIGNED, a faixa de valores fica de 0 a 65535.</p> <p>Exemplo:</p> <p>coluna1 SMALLINT --até 32767</p> <p>coluna2 SMALLINTUNSIGNED --até 65535, mas apenas positivos</p>		
INTEGER, INT	4 <i>bytes</i>	-2147483648 a +2147483647

Número inteiro. É o tipo mais usual, apesar de sua faixa de valores bastante ampla. A versão **UNSIGNED** amplia seu valor positivo máximo a 4294967295.

Exemplo:

coluna1 INT coluna2 INTEGER -- equivale à declaração anterior
coluna3 INT UNSIGNED --suporta apenas número positivos

BIGINT	8 bytes	-9223372036854775808 a 9223372036854775807
--------	---------	--

Número inteiro com faixa larga de valores. Utilizado em situações bastante específicas em que a coluna precisa armazenar números muito altos ou muito baixos. Deve ser usado com cuidado para não desperdiçar memória. Também conta com versão **UNSIGNED**, chegando a suportar o valor 18446744073709551615.

Exemplo:

coluna1 BIGINT

DECIMAL(m, n), NUMERIC(m, n)	variável	sem limite
---------------------------------	----------	------------

Número com ponto fixo e valor exato. Define a precisão (quantidade máxima de dígitos do valor armazenado) por m e a escala (quantidade exata de dígitos decimais) por n . Ou seja, armazenará números de até m dígitos no total, sendo desses, n dígitos decimais.

Exemplo:

coluna1 DECIMAL(5, 2)

A coluna acima armazenaria o número 389.75, por exemplo, normalmente. O número 250 seria armazenado como 250.00; o número 148.624 seria truncado para 148.62; já o número 15794 não seria armazenado, pois ultrapassa o limite de dígitos não decimais.

O valor máximo para m é 65 e para n é 30. Quando m não é informado, assume-se 10; já o valor padrão para n é 0.

FLOAT	4 bytes	-3.402823466E+38 a -1.175494351E-38
<p>Número de ponto flutuante e valor aproximado.</p> <p>Exemplo:</p> <p>coluna1 FLOAT</p>		
DOUBLE	8 bytes	-1.7976931348623157E+ 308 a -2.2250738585072014E- 308

Número de ponto flutuante e valor aproximado. Conta com dupla precisão, permitindo armazenamento de valores maiores.

Exemplo:

coluna1 DOUBLE

BIT(<i>n</i>)	até 8 <i>bytes</i>	
-----------------	--------------------	--

Representa valores binários. O parâmetro *n* representa a quantidade de *bits* que o valor armazenará, podendo ir de 1 até 64. Quando não informado, assume 1.

Exemplo:

coluna1 BIT

Pode ser usado especialmente para simular valores booleanos, utilizando menos espaço em memória do que os tipos inteiros. Para armazenar valores binários em colunas com esse tipo, usa-se a notação *b* (exemplo: b'1010')

Tipos lógicos

Apesar de ser possível utilizar BIT ou INT para representar valores booleanos (considerando 1 = verdadeiro e 0 = falso), a SQL e o MySQL contam com um tipo específico para tal: BOOLEAN ou BOOL que, na verdade, utilizam em seus bastidores

um tipo TINYINT de apenas um dígito. Observe o exemplo:

```
coluna1 BOOL
```

Pode-se preencher a coluna anterior com valores 0 e 1.

Tipos textuais

Valores textuais em SQL utilizam, por padrão, delimitação por aspas simples (') ao invés de aspas duplas, embora alguns SGBDs permitam o uso de aspas duplas. Assim, o valor '**algum texto**', por exemplo, seria válido para uma coluna textual em SQL.

Além disso, por padrão, a concatenação entre dois valores textuais se dá pelo símbolo ||; no MySQL, no entanto, por padrão, essa opção está desabilitada e utiliza-se a função **CONCAT()**. Por exemplo **CONCAT('informação ', 'textual!')** resulta na frase '**informação textual!**'.

Os tipos textuais em SQL e especificamente no MySQL são os seguintes:

- ◆ **CHAR(n)**: texto com tamanho fixo. O número de caracteres é definido pelo argumento *n* (se não informado, assume-se 1). Se os *n* caracteres não forem informados no valor, ele é completado com espaços em branco. Veja o exemplo:

```
coluna1 CHAR(1)
```

- ◆ **VARCHAR(n)**: texto com tamanho variável. A quantidade máxima (não fixa, como em *char*) de caracteres é especificada pelo argumento *n*, podendo chegar a 65535. É o tipo de texto mais usual. Confira o exemplo:

```
coluna1 VARCHAR(255)
```

- ◆ **TEXT**: texto de comprimento variável e (praticamente) ilimitado. Na realidade, uma coluna do tipo **TEXT** pode guardar até 65535 bytes de dados. Há variações desse tipo: **TINYTEXT** (255 caracteres), **MEDIUMTEXT** (cerca de 17 mil bytes) e **LONGTEXT** (mais de 4 bilhões de caracteres). Para textos curtos, prefira **VARCHAR**. Observe o exemplo:

```
coluna1 TEXT
```

Tipos para dados binários

Além dos tipos básicos, é possível guardar em tabelas de banco de dados relacionais dados binários – ou seja, informações expressas diretamente em bits –, ao invés de números ou letras. Isso é útil quando é necessário guardar imagens, vídeos ou outros arquivos de dados.

Os seguintes tipos são os mais usuais no MySQL:

- ◆ BLOB (binary large objects, ou objetos binários grandes): pode armazenar até 65535 bytes de dados
Veja o exemplo:

coluna1 BLOB

- ◆ MEDIUMBLOB: campo binário com capacidade de até 16777215 bytes de dados
- ◆ LONGBLOB: campo binário com capacidade de até 4294967295 bytes de dados

Campos BLOB não podem ser chaves primárias nem participar de group e order em cláusulas SELECT (veja mais sobre ORDER BY e GROUP BY nos conteúdos sobre consulta de dados e consulta com múltiplas tabelas, desta unidade). Esse tipo deve ser utilizado com cuidado, apenas quando necessário, pois pode fragmentar demais a base de dados e causar lentidão.

Tipos para data e hora

A SQL também tem tipos específicos para informações de data e hora. Por padrão, as datas são expressas no formato 'YYYY-MM-DD' (quatro dígitos de ano - dois dígitos de mês - dois dígitos de dia). Os tipos principais são os seguintes:

- ◆ **DATE**: expressa apenas uma data, sob formato 'YYYY-MM-DD' por padrão. Cada incremento representa um dia (ou seja, ao somar 1 em um campo deste tipo adiciona-se um dia à data armazenada. Pode armazenar datas de 1000-01-01 a 9999-12-31

Exemplo:

```
coluna1 DATE;
```

- ◆ **TIME** – Expressa apenas horas, minutos e segundos, sob formato padrão hh:mm:ss. O valor vai de '-838:59:59' a '838:59:59'.

Exemplo:

```
coluna1 TIME
```

- ◆ **DATETIME** – Expressa data e tempo, em formato padrão 'YYYY-MM-DD hh:mm:ss'. Os valores podem ir de '1000-01-01 00:00:00' a '9999-12-31 23:59:59'.

Exemplo:

```
coluna1 DATETIME
```

- ◆ **TIMESTAMP** – Marca de tempo, ou o número de segundos passados desde a data de 01/01/1970 (data Unix). O valor é armazenado no formato 'YYYY-MM-DD hh:mm:ss' e vai de '1970-01-01 00:00:01' a '2038-01-19 03:14:07'. A diferença com relação ao DATETIME é que o valor de TIMESTAMP é influenciado pelo time zone (ou o fuso horário) configurado no servidor.

Exemplo:

```
coluna1 TIMESTAMP
```

O comando **SET time_zone** pode modificar um valor armazenado. Por exemplo, considere que tenha sido configurado **SET time_zone='+0:00'** e incluído o valor '2008-01-01 00:00:01' na coluna do tipo **TIMESTAMP**. Em algum momento, poderá

haver a seguinte reconfiguração:

```
SET time_zone='+3:00'
```

Dessa forma, o valor da coluna passará a ser '2008-01-01 03:00:01'. Já os campos de tipo **DATETIME** não se alteram.

Chaves primárias e estrangeiras

O conceito de chave em um banco de dados consiste em ser um **valor único em uma linha que distingue esta de todas as outras linhas da tabela**. Esta seria, portanto, a chamada chave primária, que é usada para relacionar os dados de uma tabela com outra, sendo essa segunda chamada de chave estrangeira.

Na maioria dos casos, uma chave primária é muito importante para que se possa relacionar esta linha (que também pode ser chamada de tupla) com outras tabelas do banco de dados. Geralmente, ela vem na forma de um atributo, como, por exemplo, um código, que será utilizado em sua tabela a seguir.

Quando criar suas tabelas na base de dados, a chave primária (*primary key*) que será utilizada, no caso da sua lista de remédios, será uma coluna **Codigo**, para guardar o código do remédio.

Essa chave primária será um identificador único de cada remédio da sua tabela e é por meio desse valor que você conseguirá relacionar a tabela “remédios” com outras tabelas da sua base de dados.

Chaves primárias, como são utilizadas como um identificador único de uma tupla (linha), nunca poderão ser **NULL** – ou seja, sempre precisarão ter um valor.

Em SQL, **NULL** é uma representação de ausência de valor em um registro. É diferente de zero, que representa de fato um valor válido para coluna de tipo numérico.

Também é diferente de texto vazio, que igualmente constitui um valor válido para uma coluna de tipo textual. Operações matemáticas com **NULL** não geram valores válidos, mas geralmente os SGBDs tratam o resultado como **NULL**. O mesmo ocorre na concatenação entre um texto e **NULL**. As comparações de igualdade, maior e menor também não se aplicam a **NULL**. Você poderá obter mais detalhes sobre este assunto no conteúdo sobre consulta de dados, desta unidade curricular.

The logo for Senac, featuring the word "Senac" in a large, light blue, sans-serif font. Above the text is a stylized graphic element consisting of several overlapping, semi-transparent shapes in shades of blue and yellow, resembling a sunburst or a cluster of petals.

Definindo chave primária

A sintaxe de criação de uma tabela com uma chave primária pode se dar da seguinte maneira:

```
CREATE TABLE <nome tabela> (  
<nome coluna 1> <tipo coluna 1> NOT NULL [AUTO_INCREMENT],  
<nome coluna 2> <tipo coluna 2> [restrições]  
...  
PRIMARY KEY (nome coluna 1)  
);
```

Obs.: considere os itens entre < > como obrigatórios e os entre [] como opcionais.

Veja um exemplo de *script* para uma nova tabela chamada **Pessoa**, em que será utilizada uma coluna chamada **Id** como chave primária

```
CREATE TABLE Pessoa (  
Id int NOT NULL,  
UltimoNome varchar(20) NOT NULL,  
PrimeiroNome varchar(20),  
Idade int,  
PRIMARY KEY (ID)  
);
```

Apesar de chaves primárias poderem ser de qualquer tipo de dados, é comum que se usem tipos inteiros, como no exemplo anterior.

Caso a tabela já exista e ainda não tenha uma chave primária, utilize o comando **ALTER TABLE** para incluir este campo. No caso da tabela **Pessoa** mencionada, o

comando seria:

```
ALTER TABLE Pessoa ADD PRIMARY KEY (ID);
```

Voltando ao banco de dados **farmacia**, aplique uma coluna **código** chave primária para a tabela **remedios**. Note que, até agora, das colunas que constituem essa tabela, nenhuma dá a segurança de ser um valor único que identifique um registro de remédio. A coluna **nome**, apesar de constituir uma certa identidade a um remédio, poderá trazer valores repetidos – basta imaginar que um medicamento genérico de mesmo nome pode ser fabricado por várias marcas. Assim, podem ocorrer problemas na manipulação dessa tabela, caso não se tenha uma identificação confiável para cada registro incluído.

Exclua então a tabela atual **remedios** para criar agora a mesma tabela, porém com o campo **código**, que será um valor inteiro, autocriando-se e autoincrementando-se a cada nova linha adicionada à tabela. Após o comando **DROP TABLE** para exclusão, execute o seguinte código SQL.

```
CREATE TABLE remedios (  
codigo INT NOT NULL AUTO_INCREMENT,  
nome VARCHAR(20) NOT NULL,  
marca VARCHAR(20),  
preco DECIMAL(9,2) NOT NULL,  
generico CHAR(1) NOT NULL DEFAULT('N'),  
validade DATE NOT NULL,  
PRIMARY KEY (codigo)  
);
```

Note que, por definição, a coluna **código** receberá sempre um valor inteiro, que não pode ser nulo ou vazio, e que se autoincrementará a cada item novo na tabela (sendo o primeiro registro de valor 1).

O valor incluído como **AUTO_INCREMENT** existe, pois campos com essa característica receberão um número único e gerado automaticamente a cada vez que uma nova linha for criada. Na maioria das vezes, as chaves primárias recebem essa característica exatamente por se tratarem de campos que devem ser únicos para os usuários. É importante notar que **AUTO_INCREMENT** é aplicado apenas para valores inteiros.

Para testar, execute o seguinte comando de **INSERT**:

```
INSERT INTO remedios(codigo, nome, marca, preco, generico, validade)
VALUES (1, 'paracetamol', 'medvida', 8.50, 'S', '2023-01-01');
```

Em seguida, execute um **SELECT**:

```
SELECT * FROM remedios;
```

Observe que, no resultado, o registro com código 1 do remédio “paracetamol” é exibido.

Em seguida, inclua um novo remédio sem informar valor para a coluna **código**:

```
INSERT INTO remedios(nome, marca, preco, generico, validade)
VALUES ('dipirona', 'medvida', 4.50, 'S', '2024-01-01')
```

Logo após, utilize novamente o **SELECT**:

```
SELECT * FROM remedios;
```

Dessa vez, o resultado mostrará, além do registro 1, um novo registro com código 2 (aqui, o autoincremento exerceu seu propósito).

Código	Nome	Marca	Preço	Generico	Validade
1	paracetamol	medvida	8.50	S	2023-01-01
2	dipirona	medvida	4.50	S	2024-01-01

Tabela 1 – Resultado da consulta à tabela **remedios**

Semelhantemente ao caso da tabela **Pessoa**, seria possível aplicar o comando **AUTO_INCREMENT** à coluna **ID**.

```
CREATE TABLE Pessoa (  
ID int NOT NULL AUTO_INCREMENT,  
UltimoNome varchar(255) NOT NULL,  
PrimeiroNome varchar(255),  
Idade int,  
PRIMARY KEY (ID)  
);
```

É importante saber que a definição de chave primária pode vir diretamente na definição da coluna, como uma de suas restrições. Veja o exemplo no caso da tabela **Pessoa**, mostrada anteriormente:

```
CREATE TABLE Pessoa (  
ID int NOT NULL AUTO_INCREMENT PRIMARY KEY,  
UltimoNome varchar(255) NOT NULL,  
PrimeiroNome varchar(255),  
Idade int  
);
```

Em algumas situações, é necessário ou conveniente utilizar duas informações para identificar unicamente uma tupla em uma tabela. Para esses casos, é possível

definir uma **chave primária composta**, constituída de mais de uma coluna. Veja a sintaxe a seguir:

```
CREATE TABLE <nome tabela> (  
<nome coluna 1> <tipo coluna 1> NOT NULL,  
<nome coluna 2> <tipo coluna 2> NOT NULL,  
...  
PRIMARY KEY (nome coluna 1, nome coluna 2)  
);
```

Nessa sintaxe, mostra-se uma chave com duas colunas, mas podem ser incluídas mais de duas.

Como exemplo prático, imagine que haja a necessidade de identificar em um banco de dados uma pessoa a partir de seu RG. Sabe-se que o RG tem uma numeração própria em cada estado brasileiro. Assim, seria preciso identificar a pessoa a partir do RG e de seu estado de emissão. A tabela poderia ficar do seguinte modo:

```
CREATE TABLE PessoaRG(  
numero_rg INT NOT NULL,  
estado_emissao CHAR(2) NOT NULL,  
nome VARCHAR(255),  
data_nascimento DATE,  
PRIMARY KEY (numero_rg, estado_emissao)  
);
```

Para chaves primárias compostas, não se pode incluir **PRIMARY KEY** diretamente na definição de coluna.

Definição de chave estrangeira

Como você viu, a chave primária é muito importante para as bases de dados por serem este valor único em uma tabela para cada entrada de dados. As chaves estrangeiras, por sua vez, **são utilizadas para relacionar uma tabela com outra**.

Quando se modela um banco de dados e suas tabelas, é importante que sejam especificados quais campos serão utilizados como chaves primárias e quais serão utilizados como chave estrangeira, para que se saiba quais serão os relacionamentos e as estruturas intertabelas.

O fundamento-base de um banco de dados relacional é sua capacidade de criar relacionamento entre tabelas, para que o administrador desse banco possa facilmente relacionar dados entre uma tabela e outra. Esses relacionamentos são gerenciados por chaves estrangeiras, que são, essencialmente, campos em tabelas que referenciam chaves primárias de outras tabelas.

Não há chave estrangeira sem chave primária. Por exemplo, o campo **ID** da tabela **Pessoa** é uma chave primária; já em outras tabelas, referencia-se esse “id” como chave estrangeira, formando o *link* em comum entre essas tabelas.

A definição de uma chave estrangeira na criação de uma tabela obedece à seguinte sintaxe:

```
CREATE TABLE <nome tabela>(
<nome coluna 1> <tipo coluna> [restrições],
<nome coluna 2> <tipo coluna> [restrições],
...
<nome coluna n> <tipo coluna> [restrições]
[primary key],
[CONSTRAINT <nome para a chave>]
FOREIGN KEY <nome coluna m>
REFERENCES <nome outra tabela> (<chave da outra tabela>)
```

);

Veja no exemplo livre a seguir a definição de tabelas para “livros” e “autores”. Imagine que cada livro tenha um autor. Neste caso, a criação das tabelas **Autores** e **Livros** ficaria:

```
CREATE TABLE Autores (  
  IdAutor int NOT NULL AUTO_INCREMENT,  
  Nome varchar(255) NOT NULL,  
  PRIMARY KEY (IdAutor)  
);  
  
CREATE TABLE Livros (  
  IdLivro int NOT NULL AUTO_INCREMENT,  
  IdAutor int NOT NULL,  
  NomeLivro varchar(255) NOT NULL,  
  PRIMARY KEY (IdLivro),  
  FOREIGN KEY (IdAutor) REFERENCES Autores (IdAutor)  
);
```

Observe a maneira como esse relacionamento é especificado no modificador **FOREIGN KEY... REFERENCES**. A parte **FOREIGN KEY** especifica uma extremidade do relacionamento (o nome do campo na tabela atual), enquanto a parte **REFERENCES** especifica a outra extremidade do relacionamento (o nome do campo na tabela referenciada).

No exemplo anterior, por **IdAutor** na tabela **Livros** estar marcado como **NOT NULL**, será obrigatório sempre informar uma referência a algum autor cadastrado na tabela **Autores**. Caso seja informado para **Livros** um valor que não corresponda a uma chave presente na tabela **Autores**, a SQL emitirá um erro informando a quebra da integridade referencial.

Se a tabela fosse definida sem a chave estrangeira, mas fosse necessário defini-la posteriormente, seria preciso usar o seguinte comando:

```
ALTER TABLE Livros ADD FOREIGN KEY (IdAutor) REFERENCES Autores (IdAutor);
```

Outro detalhe é que a chave estrangeira poderia ser nomeada com a seguinte sintaxe:

```
CREATE TABLE Livros (  
  IdLivro int NOT NULL AUTO_INCREMENT,  
  IdAutor int NOT NULL,  
  NomeLivro varchar(255) NOT NULL,  
  PRIMARY KEY (IdLivro),  
  CONSTRAINT FK_Autores FOREIGN KEY (IdAutor) REFERENCES Autores (IdAutor)  
);
```

Também é possível definir chaves estrangeiras nomeadas usando **ALTER TABLE**:

```
ALTER TABLE Livros ADD CONSTRAINT FK_Autores FOREIGN KEY (IdAutor)  
REFERENCES Autores (IdAutor);
```

Nesta ligação entre as tabelas Livros e Autores, considera-se que o relacionamento é do tipo 1:N. Isso porque foi definido que cada livro tem um autor, mas cada autor pode ser referenciado por vários livros.

IdAutor (PRIMARY KEY)	AutorNome		IdLivro	NomeLivro	IdAutor (FOREIGN KEY)
1	J. R. R. Tolkien	1 N	100	O Senhor dos Anéis	1
2	Frank Herbert		101	O Hobbit	1
3	George Orwell		102	Duna	2
			103	1984	3
			104	A Revolução dos Bichos	3

Tabela 2 – Relacionamento 1:N entre **Autores** e **Livros**

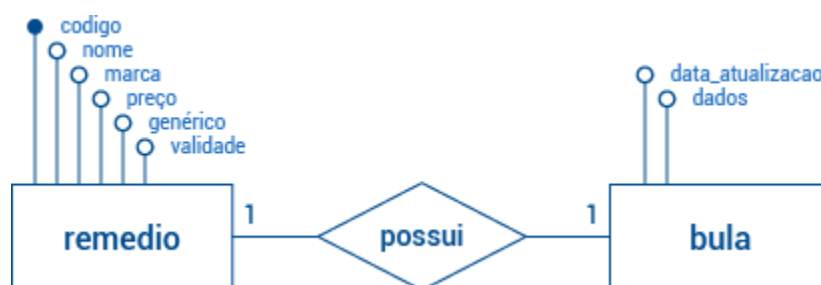
É possível dizer que os relacionamentos 1:N são os mais comuns em definição de tabelas por SQL. A partir deles, pode-se expandir para relacionamento N:N (utilizando uma tabela auxiliar) ou restringir para um relacionamento 1:1 (definindo a coluna de chave estrangeira como de valor único).

Voltando ao banco de dados de farmácias, serão agora criadas novas tabelas com relacionamentos variados.

Clique ou toque para visualizar o conteúdo.

Relacionamento 1:1

Imagine que seja necessário armazenar a bula digitalizada de um remédio. Cada remédio terá exatamente uma bula e cada bula corresponderá a exatamente um remédio. O relacionamento se torna o seguinte:

Figura 10 – Diagrama ER para o relacionamento 1:1 entre **remedio** e **bula**

No SQL, pode-se definir a tabela **bula** da seguinte maneira:

```
CREATE TABLE bula (  
id_remedio INT NOT NULL PRIMARY KEY,  
data_atualizacao DATETIME,  
dados BLOB NOT NULL,  
FOREIGN KEY (id_remedio) REFERENCES remedios(codigo)  
);
```

Neste caso, você está criando uma tabela cuja chave primária também é uma chave estrangeira, que referencia a tabela **remedios**. Você está definindo assim um relacionamento 1:1, uma vez que cada bula se referencia a exatamente 1 remédio (não pode haver duas bulas referenciando um mesmo remédio, já que a coluna de chave estrangeira é chave primária, que não pode ter valor repetido).

Relacionamento 1:N

Imagine, a seguir, que você expandirá um pouco mais seu banco, incluindo tabelas para clientes e vendas da farmácia. Cada cliente poderá fazer zero, uma ou mais compras e cada compra está associada a exatamente um cliente. A relação **1:N** proposta é a seguinte:



Figura 11 – Diagrama ER para o relacionamento 1:N entre **clientes** e **vendas**

A definição das tabelas pode ser feita do seguinte modo:


```
CREATE TABLE clientes (  
codigo INT NOT NULL AUTO_INCREMENT,  
nome VARCHAR(255) NOT NULL,  
cpf CHAR(11),  
PRIMARY KEY (codigo)  
);
```

Uma vez criada a tabela “clientes”, você pode incluir a tabela “vendas”, com referência a “clientes”:

```
CREATE TABLE vendas( codigo INT NOT NULL AUTO_INCREMENT, data  
DATETIME NOT NULL, total DECIMAL(8, 2), codigo_cliente INT NOT NULL, PRIMARY  
KEY (codigo), FOREIGN KEY (codigo_cliente) REFERENCES clientes(codigo) );
```

Note a necessidade de se definir uma coluna para guardar o código do cliente (**codigo_cliente**), que se tornará a referência da tabela **clientes** por meio da definição de chave estrangeira. Como cada venda pode guardar o código de apenas um cliente, mas esse código pode se repetir em várias vendas, estará assim sendo definida adequadamente a relação 1:N.

Relacionamento N:N

Por último, exercite um relacionamento N:N. Cada venda pode incluir um ou mais remédios. Cada remédio pode estar presente em uma ou mais vendas – note que aqui não se está falando do item unitário do remédio, mas sim de suas informações gerais. Imagine o relacionamento da seguinte maneira:

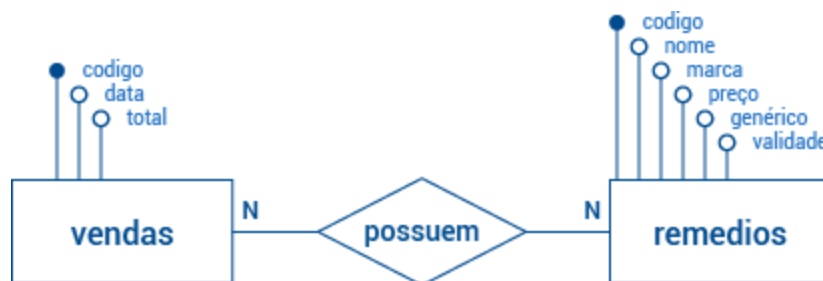


Figura 12 – Diagrama ER para o relacionamento N:N entre **vendas** e **remedios**.

Como definir esse tipo de relacionamento em tabelas usando SQL? Neste caso, não há uma maneira de fazer com que cada registro de venda referencie vários remédios e vice-versa – seria preciso uma espécie de coluna que contivesse vários valores, como um vetor ou uma lista, mas isso não é viável em SQL. Por isso, é necessária uma tabela auxiliar que realize essa ligação. Veja o *script* a seguir:

```
CREATE TABLE vendas_remedios (
codigo_venda INT NOT NULL,
codigo_remedio INT NOT NULL,
quantidade INT NOT NULL,
PRIMARY KEY (codigo_venda, codigo_remedio),
FOREIGN KEY (codigo_venda) REFERENCES vendas(codigo),
FOREIGN KEY (codigo_remedio) REFERENCES remedios(codigo)
);
```

A tabela **vendas_remedios** permite que cada venda se relacione com um ou mais registros de remédios e que cada remédio esteja ligado a uma ou mais vendas. Você está desse modo, portanto, definindo um relacionamento N:N. Considere ainda incluir uma informação importante: quantas unidades do remédio estão sendo vendidas na venda referenciada.

Observe que a chave primária aqui é composta, formada pelas colunas **codigo_venda** e **codigo_remedio**. Não se trata de uma obrigatoriedade, mas essa definição auxilia a determinar a restrição de que cada código de venda estará

associado a um código de remédio ao menos uma vez – ou seja, uma venda não repete um mesmo remédio.

Destruindo chaves primárias e estrangeiras

No MySQL, caso seja necessário destruir uma chave primária, você poderá usar o comando **ALTER TABLE** com a seguinte sintaxe:

```
ALTER TABLE <nome da tabela> DROP PRIMARY KEY;
```

Obviamente, pode-se redefinir a chave usando também **ALTER TABLE**.

Para destruição de uma chave estrangeira, é necessário conhecer o nome que essa chave recebeu. A sintaxe é a seguinte:

```
ALTER TABLE <nome da tabela> DROP FOREIGN KEY <nome da fk>;
```

Caso a chave estrangeira tenha sido definida com nome, a execução desse comando fica facilitada. Por exemplo, quando se define a tabela **Livros** com o *script* que consta a seguir, será possível conhecer o nome com o qual se pode remover a chave estrangeira.

```
CREATE TABLE Livros (  
  IdLivro int NOT NULL AUTO_INCREMENT,  
  IdAutor int NOT NULL,  
  NomeLivro varchar(255) NOT NULL,  
  PRIMARY KEY (IdLivro),  
  CONSTRAINT FK_Autores FOREIGN KEY (IdAutor) REFERENCES Autores (IdAutor)  
);
```

Neste caso, a destruição ficaria assim:

```
ALTER TABLE Livros DROP FOREIGN KEY FK_Autores;
```

Note, no entanto, que, nos outros exemplos, não foram atribuídos nomes às chaves estrangeiras, como na tabela **bula**, por exemplo:

```
CREATE TABLE bula(  
id_remedio INT NOT NULL PRIMARY KEY,  
data_atualizacao DATETIME,  
dados BLOB NOT NULL,  
FOREIGN KEY (id_remedio) REFERENCES remedios(codigo)  
);
```

Para se conseguir destruir a chave estrangeira, nesse caso, primeiro será preciso descobrir o nome do *constraint* atribuído automaticamente a ela. Pode-se obter isso por meio do seguinte comando:

```
SHOW CREATE TABLE bula;
```

Esse comando mostra como resultado o *script* SQL de criação da tabela informada (no caso, **bula**). O resultado do comando, portanto, será aproximadamente o seguinte:

```
CREATE TABLE `bula` (  
`id_remedio` int(11) NOT NULL,  
`data_atualizacao` datetime DEFAULT NULL,  
`dados` blob NOT NULL,  
PRIMARY KEY (`id_remedio`),  
CONSTRAINT `bula_ibfk_1` FOREIGN KEY (`id_remedio`) REFERENCES `remedios`  
(`codigo`)
```

```
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
```

Note que, internamente, o MySQL define um nome para as chaves estrangeiras – neste exemplo, a chave chama-se “bula_ibfk_1”. Assim, seria possível destruir a chave estrangeira com o comando:

```
ALTER TABLE bula DROP FOREIGN KEY bula_ibfk_1;
```

Que tal agora realizar alguns desafios?

Tente modelar um banco de dados simples para uma pequena loja com apenas três funcionários e um pequeno estoque. Os funcionários são:

- ◆ João da Silva (gerente)
- ◆ Marcos Palmeiras (caixa)
- ◆ Ana Maria Aparecida (vendedora)

No estoque, estão as roupas variadas e os materiais para artesanato.

Quais chaves estrangeiras e primárias você criaria e como faria isso? Como você controlaria este estoque? Você já é capaz de resolver esse desafio sem problemas!

Recobre o segundo desafio do material **Modelo de entidade e relacionamento**, que propôs a construção de um DER para uma festa junina. Crie, via SQL, as tabelas propostas pelo seu diagrama, incluindo suas chaves e seus relacionamentos.

Revisando e aplicando

No vídeo a seguir, você poderá expandir seu conhecimento e sua experiência

com novos exemplos de *scripts* SQL para criação de bancos de dados e tabelas. Acompanhe o caso de estudo passo a passo, pause ou retroceda sempre que necessário.

Encerramento

Com este conteúdo, você deve ser capaz de definir seu próprio banco de dados e suas próprias tabelas. Lembre-se que a definição das colunas vem de todo o estudo sobre as necessidades do cliente para o banco de dados. A partir dos requisitos e, preferencialmente, após o desenho de um diagrama ER, você poderá definir mais assertivamente as tabelas, as colunas e seus tipos, além de detectar com mais clareza o que identifica cada registro (definindo assim a chave primária) e como as tabelas se relacionam (estabelecendo dessa maneira as chaves estrangeiras).

O SQL torna-se, portanto, a ferramenta para concretizar esse estudo e criar as estruturas de dados que, de fato, receberão essas informações. Conhecer sua sintaxe é requisito básico para um bom profissional desenvolvedor de *software* ou de banco de dados.