



**alura** | [alura.com.br](https://alura.com.br)

# **SQL e Modelagem com banco de dados**

# Sumário

<b>1 Objetivos do curso</b>	<b>1</b>
1.1 O que é realmente importante?	1
1.2 Sobre os exercícios	1
1.3 Tirando dúvidas e indo além	2
<b>2 Meu problema</b>	<b>3</b>
2.1 Criando o nosso banco de dados	5
2.2 Começando um caderno novo: criando o banco	5
2.3 O padrão utilizado neste curso	6
2.4 A tabela de compras	6
2.5 Conferindo a existência de uma tabela	8
2.6 Inserindo registros no banco de dados	9
2.7 Seleção simples	10
2.8 A formatação de números decimais	10
2.9 A chave primária	11
2.10 Recriando a tabela do zero	13
2.11 Consultas com filtros	14
2.12 Modelando tabelas	19
2.13 Resumindo	21
2.14 Exercícios	21
<b>3 Atualizando e excluindo dados</b>	<b>23</b>
3.1 Utilizando o UPDATE	24
3.2 Atualizando várias colunas ao mesmo tempo	25
3.3 Utilizando uma coluna como referência para outra coluna	25
3.4 Utilizando o DELETE	26
3.5 Cuidados com o DELETE e UPDATE	27
3.6 Resumindo	27
<b>4 Alterando e restringindo o formato de nossas tabelas</b>	<b>29</b>

4.1 Restringindo os nulos	30
4.2 Adicionando Constraints	30
4.3 Valores Default	31
4.4 Evolução do banco	31
4.5 Resumindo	33
<b>5 Agrupando dados e fazendo consultas mais inteligentes</b>	<b>35</b>
5.1 Ordenando os resultados	37
5.2 Resumindo	41
<b>6 Juntando dados de várias tabelas</b>	<b>42</b>
6.1 Normalizando nosso modelo	47
6.2 One to Many/Many to One	50
6.3 FOREIGN KEY	50
6.4 Determinando valores fixos na tabela	58
6.5 Server SQL Modes	59
6.6 Resumindo	60
<b>7 Alunos sem matrícula e o Exists</b>	<b>62</b>
7.1 Subqueries	65
7.2 Resumindo	70
<b>8 Agrupando dados com GROUP BY</b>	<b>72</b>
8.1 Resumindo	79
<b>9 Filtrando agregações e o HAVING</b>	<b>81</b>
9.1 Condições com HAVING	83
9.2 Resumindo	86
<b>10 Múltiplos valores na condição e o IN</b>	<b>87</b>
10.1 Resumindo	93
<b>11 Sub-queries</b>	<b>94</b>
11.1 Resumindo	101
<b>12 Entendendo o LEFT JOIN</b>	<b>103</b>
12.1 RIGHT JOIN	109
12.2 JOIN ou SUBQUERY?	111
12.3 Resumindo	119
<b>13 Muitos alunos e o LIMIT</b>	<b>120</b>
13.1 Limitando e buscando a partir de uma quantidade específica	121
13.2 Resumindo	123



# OBJETIVOS DO CURSO

- que o aluno saia apto a utilizar qualquer sistema de banco de dados relacional (exemplos: MySQL, Oracle, PostgreSQL, SQL Server). Para isso usamos sempre que possível o padrão SQL que todos eles aceitam. Para acompanhar a apostila sugerimos o MySQL, para que passe pelos mesmos desafios e soluções que encontramos nesse curso. Após aprender a base a todos eles, estudar detalhes específicos de cada banco passa a ser bem mais simples.
- salientar que o uso de alguns conceitos, como as vantagens e desvantagens da modelagem, são entendidos por completo depois de um tempo de prática, além de mudarem com o tempo.
- mostrar que decorar comandos não é importante.

## 1.1 O QUE É REALMENTE IMPORTANTE?

Muitos livros, ao passar dos capítulos, mencionam todos os detalhes de uma ferramenta juntamente com seus princípios básicos. Isso acaba criando muita confusão, em especial porque o estudante não consegue distinguir exatamente o que é primordial aprender no início, daquilo que pode ser estudado mais adiante. Esse tipo de informação será adquirida com o tempo, e não é necessário no início: esse tipo de filtragem já fizemos e aplicamos aqui para você aprender a medida certa, e se aprofundar no momento que fizer sentido continuando seus estudos em outros cursos nossos no Alura.

Neste curso, separamos essas informações em quadros especiais, já que são informações extras. Ou então, apenas citamos num exercício e deixamos para o leitor procurar informações se for de seu interesse.

Por fim, falta mencionar algo sobre a prática, que deve ser tratada seriamente: todos os exercícios são muito importantes.

### O curso

Para aqueles que estão fazendo o curso online no [www.alura.com.br](http://www.alura.com.br), recomendamos estudarem em casa aquilo que foi visto durante a aula, tentando resolver os exercícios e desafios apresentados.

## 1.2 SOBRE OS EXERCÍCIOS

Os exercícios do curso variam de práticos até pesquisas na Internet, ou mesmo consultas sobre assuntos avançados em determinados tópicos para incitar a curiosidade do aprendiz na tecnologia.

## 1.3 TIRANDO DÚVIDAS E INDO ALÉM

Para tirar dúvidas dos exercícios, ou de desenvolvimento em geral, recomendamos o fórum do GUJ (<http://www.guj.com.br/>). O GUJ foi fundado por desenvolvedores do Alura e da Caelum, e hoje conta com mais de um milhão de mensagens.

Quando terminar essa aventura espero encontrá-lo no Alura, onde poderá continuar com diversos outros cursos na área de desenvolvimento e tecnologia.

<http://www.alura.com.br/>

Se o que você está buscando são livros de apoio, sugerimos conhecer a editora Casa do Código:

<http://www.casadocodigo.com.br>

## SOBRE O CURSO BANCO DE DADOS E MODELAGEM

Bem vindo ao curso de Banco de Dados e Modelagem do Alura. Nessa apostila você vai encontrar a transcrição do curso que pode ser encontrado em sua versão mais recente em [www.alura.com.br](http://www.alura.com.br).

Como desenvolvedor passei mais de 15 anos procurando exemplos reais de ensino para poder ajudar meus colegas desenvolvedores que estagiavam na minha equipe, e junto da equipe de instrutores do Alura, agrupei esses exemplos nesse curso.

Nosso foco aqui é mostrar o que é um banco de dados relacional, como pensar a modelagem de nossos dados e como criar, inserir, atualizar e pesquisar informações em uma base de dados. Tudo isso usando exemplos do mundo real, nada de abstrações genéricas que nos levam a questionar o uso real da tecnologia no dia a dia.

Como usamos o MySQL como software de banco de dados neste curso, você poderá executar todos os exemplos na sua máquina em casa mesmo.

# MEU PROBLEMA

Chegou o fim do mês e não sei direito onde meu dinheiro foi parar, no meu bolso não está. Nem na conta bancária. É um problema comum, e como podemos fazer para controlar os gastos?

A maneira mais simples de entender onde o dinheiro está indo é anotar todos os nossos gastos, sendo uma das mais tradicionais e antigas escrevê-los em um caderninho. Por exemplo dia 05/01/2016 gastei R\$ 20 em uma Lanchonete, escrevo isso bonitinho em uma linha.

R\$20 05/01/2016 Lanchonete

Tive um outro gasto de lanchonete no dia 06/01/2016 no valor de R\$ 15, vou e anoto novamente:

R\$20 05/01/2016 Lanchonete

R\$15 06/01/2016 Lanchonete

Por fim, minha esposa comprou um guarda-roupa pro quarto e ele ainda não chegou, então anotei:

R\$20 05/01/2016 Lanchonete

R\$15 06/01/2016 Lanchonete

R\$915,5 06/01/2016 Guarda-roupa (não recebi)

Repara que do jeito que anotei, as linhas acabam se assemelhando a uma tabelinha:

```
+-----+-----+-----+-----+
+ R$20   + 05/01/2016 + Lanchonete + recebida +
+ R$15   + 06/01/2016 + Lanchonete + recebida +
+ R$915,5 + 06/01/2016 + Guarda-roupa + não recebida +
+-----+-----+-----+-----+
```

Mas o que significa a primeira coluna dessa tabela mesmo? O valor gasto? Ok. E a segunda? É a data da compra? E a terceira, é o lugar que comprei ou são anotações relativas aos gastos? Repara que sem dar nome as colunas, minhas compras ficam confusas, a tabela fica estranha, e posso cada vez preencher com algo que *acho* que devo, ao invés de ter certeza do que cada campo significa. Como identificar cada um deles? Vamos dar um nome aos três campos que compõem a minha tabela: o valor, a data e as observações:

```
+-----+-----+-----+-----+
+ valor  + data    + observacoes + recebida +
+-----+-----+-----+-----+
+ R$20   + 05/01/2016 + Lanchonete + recebida +
+ R$15   + 06/01/2016 + Lanchonete + recebida +
+ R$915,5 + 06/01/2016 + Guarda-roupa + não recebida +
+-----+-----+-----+-----+
```

Posso também simplificar os dados da coluna *recebida* somente indicando se ela foi recebida (sim) ou não (não):

```
+-----+-----+-----+-----+
+ valor   + data       + observacoes + recebida   +
+-----+-----+-----+-----+
+ R$20    + 05/01/2016 + Lanchonete   + sim        +
+ R$15    + 06/01/2016 + Lanchonete   + sim        +
+ R$915,5 + 06/01/2016 + Guarda-roupa + não        +
+-----+-----+-----+-----+
```

Para montar essa tabela em um caderno eu tenho que ser capaz de traçar linhas bonitinhas e eu, pessoalmente, sou horrível para isso. No mundo tecnológico de hoje em dia, anotaríamos essa tabela em uma planilha eletrônica como o Excel:

valor	data	observacoes	recebida
20	30/05/2015	Lanchonete	sim
15	15/06/2015	Lanchonete	sim
1576,4	30/04/2015	Material de construcao	não
154	20/04/2015	Mercado	sim
20	20/05/2015	Mercado	sim
3500	21/05/2015	Televisao	não

Figura 2.1: Planilha exemplo

## UM BANCO PARA OS DADOS

Porém, além de guardar as informações, eu quero manter um histórico, tirar média semestral ou mensal, saber quanto que eu gastei com lanchonete ou mercado durante um período e realizar tarefas futuras como relatórios complexos. O Excel é uma ferramenta bem poderosa e flexível, porém, dependendo da quantidade de registros, a manipulação dos dados começa a ficar um pouco complicada em uma planilha eletrônica.

Para facilitar o nosso trabalho, existem softwares para armazenar, guardar os dados. Esses bancos de dados são sistemas que gerenciam desde a forma de armazenar a informação como também como consultá-la de maneira eficiente.

Os chamados Sistema de Gerenciamento de Banco de Dados (SGBD), nos permitem realizar todas essas tarefas sem nos preocuparmos com o caderninho ou a caneta. Claro que se vamos nos comunicar com alguém que vai gerenciar nossos dados precisamos falar uma língua comum, conversar em uma linguagem padrão que vários bancos utilizem.

Um padrão que surgiu para acessarmos e pesquisarmos dados armazenados e estruturados de uma forma específica é uma linguagem de consultas estruturadas, Structured Query Language, ou simplesmente SQL. Para conseguirmos utilizar essa linguagem, precisamos instalar um SGBD que será o nosso servidor de banco de dados, como por exemplo o MySQL, Oracle ou SQLServer.



Neste curso utilizaremos o MySQL, um SGBD gratuito que pode ser instalado seguindo as maneiras tradicionais de cada sistema operacional. Por exemplo, no Windows, o processo é realizado através do download de um arquivo .msi, enquanto no Linux (versões baseadas em Debian), pode ser feito através de um simples comando apt-get, e no MacOS pode ser instalado com um pacote .dmg. Baixe o seu instalador em <http://dev.mysql.com/downloads/mysql/>

## 2.1 CRIANDO O NOSSO BANCO DE DADOS

Durante todo o curso usaremos o terminal do MySQL. Existe também a interface gráfica do Workbench do MySQL, que não utilizaremos. Ao utilizarmos o terminal não nos preocupamos com o aprendizado de mais uma ferramenta que é opcional, podendo nos concentrar no SQL, que é o objetivo deste curso.

Abra o terminal do seu sistema operacional. No Windows, digite `cmd` no Executar. No Mac e Linux, abra o terminal. Nele, vamos entrar no MySQL usando o usuário `root`:

```
mysql -uroot -p
```

O `-u` indica o usuário `root`, e o `-p` é porque digitaremos a senha. Use a senha que definiu durante a instalação do MySQL, note que por padrão a senha pode ser em branco e, nesse caso, basta pressionar enter.

## 2.2 COMEÇANDO UM CADERNO NOVO: CRIANDO O BANCO

Agora que estamos conectados ao MySQL precisamos dizer a ele que queremos um caderno novo. Começaremos com um que gerenciará todos os dados relativos aos nossos gastos, permitindo que controlemos melhor nossos gastos, portanto quero um banco de dados chamado *ControleDeGastos*, pode parecer estranho, mas não existe um padrão para nomear um banco de dados, por isso utilizamos o padrão *CamelCase*. Para criar um banco de dados basta mandar o MySQL criar um banco:

```
mysql> CREATE DATABASE
```

Mas qual o nome mesmo? *ControleDeGastos*? Então:

```
mysql> CREATE DATABASE ControleDeGastos
```

Dou enter e o MySQL fica esperando mais informações:

```
mysql> CREATE DATABASE ControleDeGastos
->
```

Acontece que em geral precisamos notificar o MySQL que o comando que desejamos já foi digitado por completo. Para fazer isso usamos o caractere ponto e vírgula (;):

```
mysql> CREATE DATABASE ControleDeGastos
-> ;
Query OK, 1 row affected (0.01 sec)
```

Agora sim, ele percebeu que acabamos nosso comando (que começou na primeira linha) e indicou que o pedido (a query) foi executada com sucesso (OK), demorou 0.01 segundo e gerou 1 resultado (1 row affected).

Da mesma maneira que criamos esse banco, não é uma regra, mas é comum ter um banco de dados para cada projeto. Por exemplo:

```
mysql> CREATE DATABASE caelum
-> ;
Query OK, 1 row affected (0.01 sec)
```

```
mysql> CREATE DATABASE alura
-> ;
Query OK, 1 row affected (0.01 sec)
```

```
mysql> CREATE DATABASE casadocodigo
-> ;
Query OK, 1 row affected (0.01 sec)
```

Agora que tenho diversos bancos, como posso dizer ao MySQL que queremos usar aquele primeiro de todos? Quero dizer para ele, por favor, *use* o banco chamado *ControleDeGastos*. Usar em inglês, é *USE*, portanto:

```
mysql> USE ControleDeGastos;
Database changed
```

## 2.3 O PADRÃO UTILIZADO NESTE CURSO

Repare que as palavras que são de minha autoria, minha escolha, deixamos em minúsculo, enquanto as palavras que são específicas da linguagem SQL deixamos em maiúsculo. Esse é um padrão como existem diversos outros padrões. Adotamos este para que a qualquer momento que você veja um comando SQL neste curso, identifique o que é palavra importante (palavra chave) do SQL e o que é o nome de um banco de dados, uma tabela, uma coluna ou qualquer outro tipo de palavra que é uma escolha livre minha, como usuário do banco.

## 2.4 A TABELA DE COMPRAS

Agora que já tenho um caderno, já estou louco para escrever o meu primeiro gasto, que foi numa lanchonete. Por exemplo, peguei uma parte do meu caderno, de verdade, de 2016:

valor	data	observacoes	recebida
R\$20	05/01/2016	Lanchonete	sim
R\$15	06/01/2016	Lanchonete	sim
R\$915,5	06/01/2016	Guarda-roupa	não
...	...	...	...

Ok. Quero colocar a minha primeira anotação no banco. Mas se o banco é o caderno... como que o

banco sabe que existe uma tabela de valor, data e observações? Aliás, até mesmo quando comecei com o meu caderno, fui eu, Guilherme, que tive que desenhar a tabela com suas três coluninhas em cada uma das páginas que eu quis colocar seus dados.

Então vamos fazer a mesma coisa com o banco. Vamos dizer a ele que queremos ter uma tabela: descrevemos a estrutura dela através de um comando SQL. Nesse comando devemos dizer quais são os campos (valor, data e observações) que serão utilizados, para que ele separe o espaço específico para cada um deles toda vez que inserimos uma nova compra, toda vez que registramos um novo dado (um registro novo).

Para criar uma tabela, novamente falamos inglês, por favor senhor MySQL, crie uma tabela ( `CREATE TABLE` ) chamada *compras*:

```
mysql> CREATE TABLE compras;
ERROR 1113 (42000): A table must have at least 1 column
```

Como assim erro 1113? Logo após o código do erro, o banco nos informou que toda tabela deve conter pelo menos uma coluna. Verdade, não falei as colunas que queria criar. Vamos olhar novamente nossas informações:

valor	data	observacoes	recebida
R\$20	05/01/2016	Lanchonete	sim
R\$15	06/01/2016	Lanchonete	sim
R\$915,5	06/01/2016	Guarda-roupa	não
...	...	...	...

A estrutura da tabela é bem clara: são 4 campos distintos, valor que é um número com ponto decimal, data é uma data, observações que é um texto livre e recebida que é ou sim ou não.

Portanto vamos dizer ao banco que queremos esses 4 campos na nossa tabela, com uma única ressalva, que para evitar problemas de encoding usamos sempre nomes que usam caracteres simples, nada de acento para os nomes de nossas tabelas e colunas:

```
mysql> CREATE TABLE compras(
valor,
data,
observacoes,
recebida);
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '
data,
observacoes,
recebida)' at line 2
```

Ainda não foi dessa vez. Repara que o erro 1064 indica um erro de sintaxe no comando `CREATE TABLE`. Na verdade o banco de dados está interessado em saber qual o tipo de cada campo e justamente por isso devemos explicar campo a campo qual o tipo de cada coluna nossa. A coluna, campo, *valor* recebe um valor com possíveis casas decimais. Para isso podemos usar um tipo que representa números

decimais. No nosso caso suportando até 18 casas antes da vírgula e 2 depois dela:

```
valor DECIMAL(18,2),
```

Já o campo data é do tipo de data:

```
data DATE,
```

A coluna observações é um texto livre, que costuma ser chamado por um conjunto variável de caracteres, por isso *VARCHAR*. Assim como no caso do número decimal, devemos falar o tamanho máximo dessa coluna. No nosso caso serão até 255 caracteres:

```
observacoes VARCHAR(255),
```

Por fim chegamos a coluna *recebida*, que deve ser representada com valores do tipo *verdadeiro* ou *falso*, algo como *sim* ou *não*. No padrão de banco relacionais não existe um campo que aceite os valores *verdadeiro* e *falso*, um campo *booleano*. O que fazemos então é utilizar um campo numérico com os valores 0 para representar o negativo e 1 para o caso positivo. Uma coluna com um numerozinho, um *TINYINT*:

```
recebida TINYINT
```

Se vamos utilizar 0 e 1 para o *recebida*, nossos dados ficariam agora:

```
+-----+-----+-----+-----+
+ valor  + data    + observacoes + recebida +
+-----+-----+-----+-----+
+ R$20   + 05/01/2016 + Lanchonete  + 1        +
+ R$15   + 06/01/2016 + Lanchonete  + 1        +
+ R$915,5 + 06/01/2016 + Guarda-roupa + 0        +
+ ...   + ...      + ...         + ...      +
+-----+-----+-----+-----+
```

Agora que já sabemos como declarar cada campo, vamos a nova tentativa de criar nossa tabela:

```
mysql> CREATE TABLE compras(
valor DECIMAL(18,2),
data DATE,
observacoes VARCHAR(255),
recebida TINYINT);
Query OK, 0 rows affected (0.04 sec)
```

A tabela *compras* foi criada com sucesso.

## 2.5 CONFERINDO A EXISTÊNCIA DE UMA TABELA

Mas você confia em tudo que todo mundo fala? Vamos ver se realmente a tabela foi criada em nosso banco de dados. Se você possuísse um caderninho de tabelas, o que eu poderia fazer para pedir para me mostrar sua tabela de compras? Por favor, me descreva sua tabela de compras?

```
mysql> desc compras;
+-----+-----+-----+-----+-----+-----+
| Field          | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
```

```
| valor      | decimal(18,2) | YES | | NULL | |
| data       | date          | YES | | NULL | |
| observacoes | varchar(255)  | YES | | NULL | |
| recebida   | tinyint(4)    | YES | | NULL | |
+-----+-----+-----+-----+-----+
4 rows in set (0.01 sec)
```

E aí está nossa tabela de compras. Ela possui 4 colunas, que são chamados de *campos* (*fields*). Cada campo é de um tipo diferente (*decimal*, *date*, *varchar* e *tinyint*) e possui até mesmo informações extras que utilizaremos no decorrer do curso!

## 2.6 INSERINDO REGISTROS NO BANCO DE DADOS

Chegou a hora de usar nossa tabela: queremos inserir dados, então começamos pedindo para inserir algo em *compras*:

```
INSERT INTO compras
```

Não vamos cometer o mesmo erro da última vez. Se desejamos fazer algo com os 4 campos da tabela, temos que falar os valores que desejamos adicionar. Por exemplo, vamos pegar as informações da primeira linha da planilha:

valor	data	observacoes	recebida
20	30/05/2015	Lanchonete	sim

Figura 2.2: Planilha exemplo

Então nós temos uma compra com valor 20, observação *Lanchonete*, data 05/01/2016, e que foi recebida com sucesso (1):

```
mysql> INSERT INTO compras VALUES (20, 'Lanchonete', '05/01/2016', 1);
ERROR 1292 (22007): Incorrect date value: 'Lanchonete' for column 'data' at row 1
```

Parece que nosso sistema pirou. Ele achou que a *Lanchonete* era a data. *Lanchonete* era o campo *observacoes*. Mas... como ele poderia saber isso? Eu não mencionei para ele a ordem dos dados, então ele assumiu uma ordem determinada, uma ordem que eu não prestei atenção, mas foi fornecida quando executamos o `DESC compras` :

```
| valor      | decimal(18,2) | YES | | NULL | |
| data       | date          | YES | | NULL | |
| observacoes | varchar(255)  | YES | | NULL | |
| recebida   | tinyint(4)    | YES | | NULL | |
```

Tentamos novamente, agora na ordem correta dos campos:

```
mysql> INSERT INTO compras VALUES (20, '05/01/2016', 'Lanchonete', 1);
ERROR 1292 (22007): Incorrect date value: '05/01/2016' for column 'data' at row 1
```

Outro erro? Agora ele está dizendo que o valor para data está incorreto, porém, aqui no Brasil, usamos esse formato para datas... O que faremos agora? Que tal tentarmos utilizar o formato que já vem

por padrão no MySQL que é ano-mês-dia ?

Vamos tentar mais uma vez, porém, dessa vez, com a data '2016-01-05':

```
mysql> INSERT INTO compras VALUES (20, '2016-01-05', 'Lanchonete', 1);
Query OK, 1 row affected (0.01 sec)
```

Agora sim os dados foram inseridos com sucesso. Temos um novo registro em nossa tabela. Mas porque será que o MySQL adotou o formato ano-mês-dia ao invés de dia/mês/ano ? O formato ano-mês-dia é utilizado pelo padrão SQL, ou seja, por questões de padronização, ele é o formato mais adequado para que funcione para todos.

## 2.7 SELEÇÃO SIMPLES

Como fazemos para conferir se ele foi inserido? Pedimos para o sistema de banco de dados selecionar todos os campos (asterisco) da nossa tabela de compras. Queremos fazer uma consulta (uma *query*) de seleção (*SELECT*) na (*FROM*) tabela *compras*:

```
mysql> SELECT * FROM compras;
+-----+-----+-----+-----+
| valor | data      | observacoes | recebida |
+-----+-----+-----+-----+
| 20.00 | 2016-01-05 | Lanchonete  | 1        |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Perfeito! Registro inserido e selecionado com sucesso. Hora de revisar um errinho que deixamos para trás.

## 2.8 A FORMATAÇÃO DE NÚMEROS DECIMAIS

Desejamos inserir os dois próximos registros de nossa tabela:

```
+-----+-----+-----+-----+
+ valor  + data      + observacoes + recebida +
+-----+-----+-----+-----+
+ R$20   + 05/01/2016 + Lanchonete  + 1        +
+ R$15   + 06/01/2016 + Lanchonete  + 1        +
+ R$915,5 + 06/01/2016 + Guarda-roupa + 0        +
+ ...   + ...       + ...         + ...      +
+-----+-----+-----+-----+
```

Portanto o primeiro INSERT :

```
mysql> INSERT INTO compras VALUES (15, '2016-01-06', 'Lanchonete', 1);
Query OK, 1 row affected (0.01 sec)
```

E a segunda linha:

```
mysql> INSERT INTO compras VALUES (915,5, '2016-01-06', 'Guarda-roupa', 0);
ERROR 1136 (21S01): Column count doesn't match value count at row 1
```

O erro indica que o número de colunas não condiz com o número de colunas que temos na tabela. Vamos dar uma olhada dentro de *VALUES*?

```
(915,5, '2016-01-06', 'Guarda-roupa', 0)
```

A vírgula apareceu 4 vezes, portanto teríamos 5 campos diferentes! Repare que o valor 915,5 está formatado no estilo português do Brasil, diferente do estilo inglês americano, que usa ponto para definir o valor decimal, como em 915.5 . Alteramos e executamos:

```
mysql> INSERT INTO compras VALUES (915.5, '2016-01-06', 'Guarda-roupa', 0);
Query OK, 1 row affected (0.00 sec)
```

Podemos conferir os 3 registros que inserimos através do *SELECT* que fizemos antes:

```
mysql> SELECT * FROM compras;
+-----+-----+-----+-----+
| valor | data      | observacoes | recebida |
+-----+-----+-----+-----+
| 20.00 | 2016-01-05 | Lanchonete  | 1        |
| 15.00 | 2016-01-06 | Lanchonete  | 1        |
| 915.50 | 2016-01-06 | Guarda-roupa | 0        |
+-----+-----+-----+-----+
3 rows in set (0.01 sec)
```

Mas e se eu quisesse inserir os dados em uma ordem diferente? Por exemplo, primeiro informar a data , depois as observacoes , valor , e por fim se foi recebida ou não? Será que podemos fazer isso? Quando estamos fazendo um *INSERT* , podemos informar o que estamos querendo inserir por meio de parênteses:

```
mysql> INSERT INTO compras (data, observacoes, valor, recebida)
```

Note que agora estamos informando **explicitamente** a ordem dos valores que informaremos após a instrução *VALUES* :

```
mysql> INSERT INTO compras (data, observacoes, valor, recebida)
VALUES ('2016-01-10', 'Smartphone', 949.99, 0);
Query OK, 1 row affected (0,00 sec)
```

Se consultarmos a nossa tabela:

```
mysql> SELECT * FROM compras;
+-----+-----+-----+-----+
| valor | data      | observacoes | recebida |
+-----+-----+-----+-----+
| 20.00 | 2016-01-05 | Lanchonete  | 1        |
| 15.00 | 2016-01-06 | Lanchonete  | 1        |
| 915.50 | 2016-01-06 | Guarda-roupa | 0        |
| 949.99 | 2016-01-10 | Smartphone  | 0        |
+-----+-----+-----+-----+
4 rows in set (0,00 sec)
```

A nossa compra foi inserida corretamente, porém com uma ordem diferente da estrutura da tabela.

## 2.9 A CHAVE PRIMÁRIA

Agora se eu olhar essa minha tabela depois de 6 meses e encontrar essas linhas, como vou falar sobre uma das minhas compras? Tenho que falar "lembra da compra na lanchonete feita no dia 2016-01-05 no valor de 20.00 que já foi recebida?". Fica difícil referenciar cada linha da tabela por todos os valores dela. Seria muito mais fácil se eu pudesse falar: "sabe a compra 15?" ou "sabe a compra 37654?".

Fica difícil falar sobre um registro se eu não tenho nada identificando ele de maneira única. Por exemplo, como você sabe que eu sou o Guilherme, e não o Paulo? Pois meu CPF é um identificador único, (por exemplo) 222.222.222-22. Só eu tenho esse CPF e ele me identifica. O Paulo tem o dele, 333.333.333-33.

O mesmo vale para computadores de uma marca, por exemplo, o meu tem número de série 16X000015, e o computador que está ao lado do meu tem o número de série 16X000016. A única maneira de identificar unicamente uma coisa é tendo uma chave importantíssima, uma chave que é tão importante, que é primária aos brasileiros, aos computadores... e as minhas compras.

Mas como definir esses números únicos? Devo usar o ano de fabricação como no caso do computador que começa com 16 (ano 2016)? Ou sequencias com números diferentes como no caso do CPF?

Existem diversas maneiras de gerar esses números únicos, mas a mais simples de todas é: começa com 1. A primeira compra é a compra 1. A segunda compra é a 2. A terceira é a 3. Uma sequencia natural, que é incrementada automaticamente a cada nova compra, a cada novo registro.

É isso que queremos, um campo que seja nossa chave primária ( PRIMARY KEY ), que é um número inteiro ( INT ), e que seja automaticamente incrementado ( AUTO\_INCREMENT ). Só falta definir o nome dele, que como identifica nossa compra, usamos a abreviação `id`. Esse é um padrão *amplamente* utilizado. Portanto queremos um campo `id INT AUTO_INCREMENT PRIMARY KEY`.

Para fazer isso vamos alterar nossa tabela ( ALTER\_TABLE ) e adicionar uma coluna nova ( ADD\_COLUMN ):

```
mysql> ALTER TABLE compras ADD COLUMN id INT AUTO_INCREMENT PRIMARY KEY;
Query OK, 0 rows affected (0.07 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

E verificamos o resultado em nossa tabela:

```
mysql> SELECT * FROM compras;
+-----+-----+-----+-----+-----+
| valor | data      | observacoes | recebida | id |
+-----+-----+-----+-----+-----+
| 20.00 | 2016-01-05 | Lanchonete  | 1        | 1 |
| 15.00 | 2016-01-06 | Lanchonete  | 1        | 2 |
| 915.50 | 2016-01-06 | Guarda-roupa | 0        | 3 |
| 949.99 | 2016-01-10 | Smartphone  | 0        | 4 |
+-----+-----+-----+-----+-----+
4 rows in set (0,00 sec)
```

Repare que agora todas as compras podem ser identificadas unicamente!



## 2.10 RECRIANDO A TABELA DO ZERO

Claro que o padrão não é criar o `id` depois de um ano. Em geral criamos uma tabela já com o seu campo `id`. Vamos então adaptar nossa query para criar a tabela com todos os campos, inclusive o campo `id`:

```
CREATE TABLE compras(  
id INT AUTO_INCREMENT PRIMARY KEY,  
valor DECIMAL(18,2),  
data DATE,  
observacoes VARCHAR(255),  
recebida TINYINT);
```

```
ERROR 1050 (42S01): Table 'compras' already exists
```

Opa, como a tabela já existe não faz sentido recriá-la. Primeiro vamos jogar ela fora ( `DROP` ), inclusive com todos os dados que temos:

```
DROP TABLE compras;
```

Agora criamos a tabela com tudo o que conhecemos:

```
CREATE TABLE compras(  
id INT AUTO_INCREMENT PRIMARY KEY,  
valor DECIMAL(18,2),  
data DATE,  
observacoes VARCHAR(255),  
recebida TINYINT);
```

```
Query OK, 0 rows affected (0.04 sec)
```

Vamos verificar se ficou algum registro:

```
mysql> SELECT * FROM compras;  
Empty set (0,00 sec)
```

Ótimo! Conseguimos limpar a nossa tabela, agora podemos reenserrar nossos dados novamente:

```
mysql> INSERT INTO compras VALUES (20, '2016-01-05', 'Lanchonete', 1);  
ERROR 1136 (21S01): Column count doesn't match value count at row 1
```

Ops, agora a nossa tabela **não tem apenas 4 colunas** como antes, incluímos o `id` também, vejamos a descrição da tabela `compras`:

```
mysql> desc compras;  
+-----+-----+-----+-----+-----+-----+  
| Field      | Type          | Null | Key | Default | Extra          |  
+-----+-----+-----+-----+-----+-----+  
| id         | int(11)       | NO   | PRI | NULL    | auto_increment |  
| valor      | decimal(18,2) | YES  |     | NULL    |                |  
| data       | date          | YES  |     | NULL    |                |  
| observacoes | varchar(255)  | YES  |     | NULL    |                |  
| recebida   | tinyint(4)    | YES  |     | NULL    |                |  
+-----+-----+-----+-----+-----+-----+  
5 rows in set (0,00 sec)
```

Então precisaremos deixar **explícito** o que estamos inserindo:

```
mysql> INSERT INTO compras (valor, data, observacoes, recebida)
VALUES (20, '2016-01-05', 'Lanchonete', 1);
Query OK, 1 row affected (0,01 sec)
```

Agora sim a nossa compra foi cadastrada! Porém, tem algo um pouco estranho, e o `id` ? Será que foi inserido também? Vejamos:

```
mysql> SELECT * FROM compras;
+----+-----+-----+-----+-----+
| id | valor | data      | observacoes | recebida |
+----+-----+-----+-----+-----+
| 1  | 20.00 | 2016-01-05 | Lanchonete  | 1        |
+----+-----+-----+-----+-----+
1 row in set (0,00 sec)
```

Ele foi inserido automaticamente! Mas como isso aconteceu? Lembra que definimos o `id` como `AUTO_INCREMENT` ? Aquela propriedade que incrementa automaticamente a cada inserção. Então, é exatamente ela que incrementou o `id` pra nós! Agora basta apenas inserir as demais compras e verificar o resultado:

```
mysql> INSERT INTO compras (valor, data, observacoes, recebida)
VALUES (15, '2016-01-06', 'Lanchonete', 1);
Query OK, 1 row affected (0,00 sec)
```

```
mysql> INSERT INTO compras (valor, data, observacoes, recebida)
VALUES (915.50, '2016-01-06', 'Guarda-roupa', 0);
Query OK, 1 row affected (0,01 sec)
```

```
mysql> INSERT INTO compras (valor, data, observacoes, recebida)
VALUES (949.99, '2016-01-10', 'Smartphone', 0);
Query OK, 1 row affected (0,00 sec)
```

Vamos verificar como ficaram os registros das nossas compras:

```
mysql> SELECT * FROM compras;
+----+-----+-----+-----+-----+
| id | valor | data      | observacoes | recebida |
+----+-----+-----+-----+-----+
| 1  | 20.00 | 2016-01-05 | Lanchonete  | 1        |
| 2  | 15.00 | 2016-01-06 | Lanchonete  | 1        |
| 3  | 915.50 | 2016-01-06 | Guarda-roupa | 0        |
| 4  | 949.99 | 2016-01-10 | Smartphone  | 0        |
+----+-----+-----+-----+-----+
4 rows in set (0,00 sec)
```

Excelente! Agora o nosso sistema cadastra nossas compras da maneira esperada!

## 2.11 CONSULTAS COM FILTROS

Aprendemos a criar as nossas tabelas e inserir registros nelas, então vamos adicionar todas as nossas compras. Eu já tenho um arquivo com as minhas compras irei executá-lo, não se preocupe, nos exercícios forneceremos o link do arquivo. Saia do terminal com o comando `exit` :

```
mysql> exit
```

Execute o comando:

```
mysql -uroot -p ControleDeGastos < compras.sql
```

Agora todas as compras foram registradas no nosso banco de dados! Vamos consultar todas as compras novamente:

```
mysql> SELECT * FROM compras;
```

id	valor	data	observacoes	recebida
1	20.00	2016-01-05	Lanchonete	1
2	15.00	2016-01-06	Lanchonete	1
3	915.50	2016-01-06	Guarda-roupa	0
4	949.99	2016-01-10	Smartphone	0
5	200.00	2012-02-19	Material escolar	1
6	3500.00	2012-05-21	Televisao	0
7	1576.40	2012-04-30	Material de construcao	1
8	163.45	2012-12-15	Pizza pra familia	1
9	4780.00	2013-01-23	Sala de estar	1
10	392.15	2013-03-03	Quartos	1
11	1203.00	2013-03-18	Quartos	1
12	402.90	2013-03-21	Copa	1
13	54.98	2013-04-12	Lanchonete	0
14	12.34	2013-05-23	Lanchonete	0
15	78.65	2013-12-04	Lanchonete	0
16	12.39	2013-01-06	Sorvete no parque	0
17	98.12	2013-07-09	Hopi Hari	1
18	2498.00	2013-01-12	Compras de janeiro	1
19	3212.40	2013-11-13	Compras do mes	1
20	223.09	2013-12-17	Compras de natal	1
21	768.90	2013-01-16	Festa	1
22	827.50	2014-01-09	Festa	1
23	12.00	2014-02-19	Salgado no aeroporto	1
24	678.43	2014-05-21	Passagem pra Bahia	1
25	10937.12	2014-04-30	Carnaval em Cancun	1
26	1501.00	2014-06-22	Presente da sogra	0
27	1709.00	2014-08-25	Parcela da casa	0
28	567.09	2014-09-25	Parcela do carro	0
29	631.53	2014-10-12	IPTU	1
30	909.11	2014-02-11	IPVA	1
31	768.18	2014-04-10	Gasolina viagem Porto Alegre	1
32	434.00	2014-04-01	Rodeio interior de Sao Paulo	0
33	115.90	2014-06-12	Dia dos namorados	0
34	98.00	2014-10-12	Dia das crianças	0
35	253.70	2014-12-20	Natal - presentes	0
36	370.15	2014-12-25	Compras de natal	0
37	32.09	2015-07-02	Lanchonete	1
38	954.12	2015-11-03	Show da Ivete Sangalo	1
39	98.70	2015-02-07	Lanchonete	1
40	213.50	2015-09-25	Roupas	0
41	1245.20	2015-10-17	Roupas	0
42	23.78	2015-12-18	Lanchonete do Zé	1
43	576.12	2015-09-13	Sapatos	1
44	12.34	2015-07-19	Canetas	0
45	87.43	2015-05-10	Gravata	0
46	887.66	2015-02-02	Presente para o filhao	1

46 rows in set (0,00 sec)

A nossa consulta devolveu **todas** as nossas compras, mas eu quero saber a data de todas as compras

"baratas" ou no caso, com valor abaixo de 500. Em SQL podemos adicionar **filtros** com o argumento **WHERE** :

```
mysql> SELECT * FROM compras WHERE valor < 500;
```

id	valor	data	observacoes	recebida
1	20.00	2016-01-05	Lanchonete	1
2	15.00	2016-01-06	Lanchonete	1
5	200.00	2012-02-19	Material escolar	1
8	163.45	2012-12-15	Pizza pra familia	1
10	392.15	2013-03-03	Quartos	1
12	402.90	2013-03-21	Copa	1
13	54.98	2013-04-12	Lanchonete	0
14	12.34	2013-05-23	Lanchonete	0
15	78.65	2013-12-04	Lanchonete	0
16	12.39	2013-01-06	Sorvete no parque	0
17	98.12	2013-07-09	Hopi Hari	1
20	223.09	2013-12-17	Compras de natal	1
23	12.00	2014-02-19	Salgado no aeroporto	1
32	434.00	2014-04-01	Rodeio interior de Sao Paulo	0
33	115.90	2014-06-12	Dia dos namorados	0
34	98.00	2014-10-12	Dia das crianças	0
35	253.70	2014-12-20	Natal - presentes	0
36	370.15	2014-12-25	Compras de natal	0
37	32.09	2015-07-02	Lanchonete	1
39	98.70	2015-02-07	Lanchonete	1
40	213.50	2015-09-25	Roupas	0
42	23.78	2015-12-18	Lanchonete do Zé	1
44	12.34	2015-07-19	Canetas	0
45	87.43	2015-05-10	Gravata	0

24 rows in set (0,00 sec)

Verificamos as compras mais baratas, mas e para verificar as compras mais caras? Por exemplo, com valor acima de 1500? Usamos o **WHERE** novamente indicando que agora queremos valores acima de 1500:

```
mysql> SELECT * FROM compras WHERE valor > 1500;
```

id	valor	data	observacoes	recebida
6	3500.00	2012-05-21	Televisao	0
7	1576.40	2012-04-30	Material de construcao	1
9	4780.00	2013-01-23	Sala de estar	1
18	2498.00	2013-01-12	Compras de janeiro	1
19	3212.40	2013-11-13	Compras do mes	1
25	10937.12	2014-04-30	Carnaval em Cancun	1
26	1501.00	2014-06-22	Presente da sogra	0
27	1709.00	2014-08-25	Parcela da casa	0

8 rows in set (0,00 sec)

Sabemos todas as compras mais caras, porém eu só quero saber todas compras mais caras e que não foram entregues ao mesmo tempo. Para isso precisamos adicionar mais um filtro, ou seja, filtrar por compras acima de 1500 e também que não foram entregues:

```
mysql> SELECT * FROM compras WHERE valor > 1500 AND recebida = 0;
```

```
+-----+-----+-----+-----+-----+
```

```

| id | valor | data | observacoes | recebida |
+---+-----+-----+-----+-----+
| 6 | 3500.00 | 2012-05-21 | Televisao | 0 |
| 26 | 1501.00 | 2014-06-22 | Presente da sogra | 0 |
| 27 | 1709.00 | 2014-08-25 | Parcela da casa | 0 |
+---+-----+-----+-----+-----+
3 rows in set (0,00 sec)

```

Agora que podemos utilizar mais de um filtro vamos verificar todas as compras mais baratas (abaixo de 500) e mais caras (acima de 1500):

```

mysql> SELECT * FROM compras WHERE valor < 500 AND valor > 1500;
Empty set (0,00 sec)

```

Parece que não funcionou adicionar mais de um filtro para a mesma coluna... Vamos analisar um pouco a nossa *query*, será que realmente está fazendo sentido esse nosso filtro? Observe que estamos tentando pegar compras que tenham o valor abaixo de 500 **e ao mesmo tempo** tenha um valor acima de 1500. Se o valor for 300 é abaixo de 500, porém não é acima de 1500, se o valor for 1800 é acima de 1500, porém não é abaixo de 500, ou seja, **é impossível que esse filtro seja válido**. Podemos fazer um pequeno ajuste nesse filtro, podemos indicar que queremos valores que sejam menores que 500 **ou** maiores que 1500:

```

mysql> SELECT * FROM compras WHERE valor < 500 OR valor > 1500;
+---+-----+-----+-----+-----+
| id | valor | data | observacoes | recebida |
+---+-----+-----+-----+-----+
| 1 | 20.00 | 2016-01-05 | Lanchonete | 1 |
| 2 | 15.00 | 2016-01-06 | Lanchonete | 1 |
| 5 | 200.00 | 2012-02-19 | Material escolar | 1 |
| 6 | 3500.00 | 2012-05-21 | Televisao | 0 |
| 7 | 1576.40 | 2012-04-30 | Material de construcao | 1 |
| 8 | 163.45 | 2012-12-15 | Pizza pra familia | 1 |
| 9 | 4780.00 | 2013-01-23 | Sala de estar | 1 |
| 10 | 392.15 | 2013-03-03 | Quartos | 1 |
| 12 | 402.90 | 2013-03-21 | Copa | 1 |
| 13 | 54.98 | 2013-04-12 | Lanchonete | 0 |
| 14 | 12.34 | 2013-05-23 | Lanchonete | 0 |
| 15 | 78.65 | 2013-12-04 | Lanchonete | 0 |
| 16 | 12.39 | 2013-01-06 | Sorvete no parque | 0 |
| 17 | 98.12 | 2013-07-09 | Hopi Hari | 1 |
| 18 | 2498.00 | 2013-01-12 | Compras de janeiro | 1 |
| 19 | 3212.40 | 2013-11-13 | Compras do mes | 1 |
| 20 | 223.09 | 2013-12-17 | Compras de natal | 1 |
| 23 | 12.00 | 2014-02-19 | Salgado no aeroporto | 1 |
| 25 | 10937.12 | 2014-04-30 | Carnaval em Cancun | 1 |
| 26 | 1501.00 | 2014-06-22 | Presente da sogra | 0 |
| 27 | 1709.00 | 2014-08-25 | Parcela da casa | 0 |
| 32 | 434.00 | 2014-04-01 | Rodeio interior de Sao Paulo | 0 |
| 33 | 115.90 | 2014-06-12 | Dia dos namorados | 0 |
| 34 | 98.00 | 2014-10-12 | Dia das crianças | 0 |
| 35 | 253.70 | 2014-12-20 | Natal - presentes | 0 |
| 36 | 370.15 | 2014-12-25 | Compras de natal | 0 |
| 37 | 32.09 | 2015-07-02 | Lanchonete | 1 |
| 39 | 98.70 | 2015-02-07 | Lanchonete | 1 |
| 40 | 213.50 | 2015-09-25 | Roupas | 0 |
| 42 | 23.78 | 2015-12-18 | Lanchonete do Zé | 1 |
| 44 | 12.34 | 2015-07-19 | Canetas | 0 |
| 45 | 87.43 | 2015-05-10 | Gravata | 0 |

```

```
+-----+-----+-----+-----+-----+
32 rows in set (0,00 sec)
```

Perceba que agora temos todas as compras que possuem valores abaixo de 500 e acima de 1500. Suponhamos que fizemos uma compra com um valor específico, por exemplo 3500, e queremos saber apenas as compras que tiveram esse valor. O filtro que queremos utilizar é o **igual** ao valor desejado:

```
mysql> SELECT * FROM compras WHERE valor = 3500;
+-----+-----+-----+-----+-----+
| id | valor  | data      | observacoes | recebida |
+-----+-----+-----+-----+-----+
| 6  | 3500.00 | 2012-05-21 | Televisao   | 0        |
+-----+-----+-----+-----+-----+
1 row in set (0,00 sec)
```

Conseguimos realizar várias *queries* com muitos filtros, porém, eu quero saber dentre as compras realizadas, quais foram em **Lanchonete**. Então vamos verificar todas as observações que sejam iguais a Lanchonete:

```
mysql> SELECT * FROM compras WHERE observacoes = 'Lanchonete';
+-----+-----+-----+-----+-----+
| id | valor | data      | observacoes | recebida |
+-----+-----+-----+-----+-----+
| 1  | 20.00 | 2016-01-05 | Lanchonete  | 1        |
| 2  | 15.00 | 2016-01-06 | Lanchonete  | 1        |
| 13 | 54.98 | 2013-04-12 | Lanchonete  | 0        |
| 14 | 12.34 | 2013-05-23 | Lanchonete  | 0        |
| 15 | 78.65 | 2013-12-04 | Lanchonete  | 0        |
| 37 | 32.09 | 2015-07-02 | Lanchonete  | 1        |
| 39 | 98.70 | 2015-02-07 | Lanchonete  | 1        |
+-----+-----+-----+-----+-----+
7 rows in set (0,00 sec)
```

Agora preciso saber todas as minhas compras que foram **Parcelas**. Então novamente eu usarei o mesmo filtro, porém para Parcelas:

```
mysql> SELECT * FROM compras WHERE observacoes = 'Parcelas';
Empty set (0,00 sec)
```

Que estranho eu lembro de ter algum registro de parcela... Sim, existem registros de parcelas, porém não existe apenas uma observação "Parcela" e sim "Parcela do carro" ou "Parcela da casa", ou seja, precisamos filtrar as observações verificando se existe um **pedaço** do texto que queremos na coluna desejada. Para verificar um pedaço do texto utilizamos o argumento **LIKE** :

```
mysql> SELECT * FROM compras WHERE observacoes LIKE 'Parcela%';
+-----+-----+-----+-----+-----+
| id | valor  | data      | observacoes      | recebida |
+-----+-----+-----+-----+-----+
| 27 | 1709.00 | 2014-08-25 | Parcela da casa  | 0        |
| 28 | 567.09  | 2014-09-25 | Parcela do carro | 0        |
+-----+-----+-----+-----+-----+
2 rows in set (0,00 sec)
```

Perceba que utilizamos o "%". Quando adicionamos o "%" durante um filtro utilizando o **LIKE** significa que queremos todos os registros que iniciem com Parcela e que tenha qualquer tipo de

informação a direita, ou seja, se a observação for "Parcela da casa" ou "Parcela de qualquer coisa" ele retornará para nós, mas suponhamos que quiséssemos saber todas as compras com observações em que o "de" estivesse no meio do texto? Bastaria adicionar o "%" tanto no início quanto no final:

```
mysql> SELECT * FROM compras WHERE observacoes LIKE '%de%';
+-----+-----+-----+-----+-----+
| id | valor | data | observacoes | recebida |
+-----+-----+-----+-----+-----+
| 7 | 1576.40 | 2012-04-30 | Material de construcao | 1 |
| 9 | 4780.00 | 2013-01-23 | Sala de estar | 1 |
| 18 | 2498.00 | 2013-01-12 | Compras de janeiro | 1 |
| 20 | 223.09 | 2013-12-17 | Compras de natal | 1 |
| 32 | 434.00 | 2014-04-01 | Rodeio interior de Sao Paulo | 0 |
| 36 | 370.15 | 2014-12-25 | Compras de natal | 0 |
+-----+-----+-----+-----+-----+
6 rows in set (0,00 sec)
```

Veja que agora foram devolvidas todas as compras que possuem um "DE" na coluna observacoes independente de onde esteja.

## 2.12 MODELANDO TABELAS

Mas como foi que chegamos na tabela de compras mesmo? Começamos com uma idéia dos dados que gostaríamos de armazenar: nossas compras. Cada uma delas tem um valor, uma data, uma observação e se ela já foi recebida ou não. Seguindo essa idéia moldamos uma estrutura que compõe uma compra, o molde de uma compra:

```
Compra
==> tem um valor, com casas decimais
==> tem uma data
==> tem uma observação, texto
==> pode ter sido recebida ou não
```

Costumamos abreviar o modelo como:

```
Compra
- valor
- data
- observacoes
- recebida
```

E existem diversas maneiras de representar tal modelo através de diagramas. Nosso foco maior aqui será em pensarmos o modelo e como trabalharmos com o SQL, por isso não nos preocuparemos tanto com criar centenas de diagramas.

Após definirmos os campos importantes para nosso negócio, nossa empresa (o business), percebemos que era necessário identificar cada compra de maneira única, e para isso definimos uma chave primária, o id.

Tenho um novo cliente em minha empresa de desenvolvimento de software e este cliente é uma escola primária. Ela tem alunos bem novos e precisa manter um banco com as informações deles para

fazer diversas pesquisas.

Pensando sozinho, no meu trabalho, tento modelar um aluno:

```
Aluno
- id
- nome
- serie
- sala
- email
- telefone
- endereco
```

Pronto, apresento o modelo do meu banco para o cliente e ele reclama muito. A diretora comenta que quando tem que se comunicar a respeito do aluno não é o telefone ou email dele que ela precisa. Ela precisa conversar com os pais e por isso é necessário os campos de contato dos pais. Alteramos então nosso modelo:

```
Aluno
- id
- nome
- serie
- sala
- email
- telefone
- endereco
- nomeDoPai
- nomeDaMae
- telefoneDoPai
- telefoneDaMae
```

Um outro detalhe que sumiu é a questão da vacinação. A diretora gosta de conversar com os pais para indicar quando é época de vacinação. Ela sabe da importância do efeito de vacinação em grupo (todos devem ser vacinados, se um não é, existe chance de infectar muitos). Por isso ela sempre envia cartas (campo endereco) para os pais (campo nomeDoPai e nomeDaMae). Mas... como saber qual vacina ele precisa tomar? Depende da idade, sugerimos então armazenar a data de nascimento do aluno:

```
Aluno
- id
- nascimento
- nome
- serie
- sala
- email
- telefone
- endereco
- nomeDoPai
- nomeDaMae
- telefoneDoPai
- telefoneDaMae
```

Além disso, ela comenta que este ano o aluno está na terceira série, mas ano que vem estará na quarta série! Tanto a série quanto a sala são valores que mudam anualmente, e precisamos atualizá-los sempre, quase que todos os alunos tem sua série e sala atualizadas de uma vez só (algumas escolas permitem a troca de salas no meio do ano letivo). Para deixar isso claro, alteramos nossos campos,



refletindo que tanto a série quanto a sala são atuais:

```
Aluno
- id
- nascimento
- nome
- serieAtual
- salaAtual
- email
- telefone
- endereco
- nomeDoPai
- nomeDaMae
- telefoneDoPai
- telefoneDaMae
```

Esse nosso processo de criar, de moldar, de modelar uma tabela é o que chamamos de modelagem de dados. Na verdade estamos modelando a estrutura que será capaz de receber os dados. E não existem regras 100% fixas nesse processo, por isso mesmo é importantíssimo levantarmos as necessidades, os requisitos dos nossos clientes junto a eles. Conversando com eles, vendo o trabalho deles no dia a dia, entendemos o que eles precisam e como modelar essas informações no banco. E mesmo tendo modelado uma vez, pode ser que daqui um ano temos que evoluir nosso modelo.

## 2.13 RESUMINDO

Nós aprendemos a criar um banco de dados, criar tabelas como as planilhas do Excel e agora estamos aprendendo a manipular informações fazendo filtro com `WHERE` , `AND` , `OR` e `LIKE` . Vamos para os exercícios?

## 2.14 EXERCÍCIOS

1. Instale o servidor do MySQL em sua máquina. Qual o sistema operacional em que você fez sua instalação? Sentiu que algo podia ser simplificado no processo de instalação? Lembre-se que você pode realizar o download de ambos em <http://MySQL.com/downloads/MySQL>.

Você pode optar por baixar a versão mais atual.

1. Logue no MySQL, e comece criando o banco de dados:

```
mysql -uroot -p
CREATE DATABASE ControleDeGastos;
USE ControleDeGastos;
```

Agora vamos criar a tabela. Ela precisa ter os seguintes campos: `id` inteiro, `valor` número com vírgula, `data` , `observacoes` e um booleano para marcar se a compra foi `recebida` . A tabela deve-se chamar "compras".

1. [Clique aqui e faça o download](#) do arquivo .sql, e importe no MySQL:

```
mysql -u root -p ControleDeGastos < compras.sql
```

Em seguida, execute o `SELECT` para garantir que todas as informações foram adicionadas:

```
SELECT * FROM compras;
```

**DICA:** Salve o arquivo `compras.sql` em uma pasta de fácil acesso na linha de comando. Além disso, o arquivo deve estar no mesmo lugar onde você executará o comando.

1. Selecione `valor` e `observacoes` de todas as compras cuja data seja maior-ou-igual que 15/12/2012.
2. Qual o comando SQL para juntar duas condições diferentes? Por exemplo, `SELECT * FROM TABELA WHERE campo > 1000 campo < 5000`. Faça o teste e veja o resultado.
3. Vimos que todo texto é passado através de aspas simples ('). Posso passar aspas duplas (") no lugar?
4. Selecione todas as compras cuja data seja maior-ou-igual que 15/12/2012 e menor do que 15/12/2014.
5. Selecione todas as compras cujo valor esteja entre R\$15,00 e R\$35,00 e a observação comece com a palavra 'Lanchonete'.
6. Selecione todas as compras que já foram recebidas.
7. Selecione todas as compras que ainda não foram recebidas.
8. Vimos que para guardar o valor VERDADEIRO para a coluna `recebida`, devemos passar o valor 1. Para FALSO, devemos passar o valor 0. E quanto as palavras já conhecidas para verdadeiro e falso: `TRUE` e `FALSE`. Elas funcionam? Ou seja:

```
INSERT INTO compras (valor, data, observacoes, recebida) VALUES (100.0, '2015-09-08', 'COMIDA', TRUE) ;
```

Funciona? Faça o teste.

1. Selecione todas as compras com valor maior que 5.000,00 ou que já foram recebidas.
2. Selecione todas as compras que o valor esteja entre 1.000,00 e 3.000,00 ou seja maior que 5.000,00.

# ATUALIZANDO E EXCLUINDO DADOS

Cadastramos todas as nossas compras no banco de dados, porém, no meu rascunho onde eu coloco todas as minhas compras alguns valores não estão batendo. Vamos selecionar o valor e a observação de todas as compras com valores entre 1000 e 2000:

```
mysql> SELECT valor, observacoes FROM compras
WHERE valor >= 1000 AND valor <= 2000;
+-----+-----+
| valor | observacoes |
+-----+-----+
| 1576.40 | Material de construcao |
| 1203.00 | Quartos |
| 1501.00 | Presente da sogra |
| 1709.00 | Parcela da casa |
| 1245.20 | Roupas |
+-----+-----+
5 rows in set (0,00 sec)
```

Fizemos um filtro para um intervalo de valor, ou seja, **entre** 1000 e 2000. Em SQL, quando queremos filtrar um intervalo, podemos utilizar o operador `BETWEEN` :

```
mysql> SELECT valor, observacoes FROM compras
WHERE valor BETWEEN 1000 AND 2000;
+-----+-----+
| valor | observacoes |
+-----+-----+
| 1576.40 | Material de construcao |
| 1203.00 | Quartos |
| 1501.00 | Presente da sogra |
| 1709.00 | Parcela da casa |
| 1245.20 | Roupas |
+-----+-----+
5 rows in set (0,00 sec)
```

O resultado é o mesmo que a nossa *query* anterior, porém agora está mais clara e intuitiva.

No meu rascunho informa que a compra foi no ano de 2013, porém não foi feito esse filtro. Então vamos adicionar mais um filtro pegando o intervalo de 01/01/2013 e 31/12/2013:

```
mysql> SELECT valor, observacoes FROM compras
WHERE valor BETWEEN 1000 AND 2000
AND data BETWEEN '2013-01-01' AND '2013-12-31';
+-----+-----+
| valor | observacoes |
+-----+-----+
| 1203.00 | Quartos |
+-----+-----+
1 row in set (0,00 sec)
```

### 3.1 UTILIZANDO O UPDATE

Encontrei a compra que foi feita com o valor errado, no meu rascunho a compra de Quartos o valor é de 1500. Então agora precisamos **alterar** esse valor na nossa tabela. Para alterar/atualizar valores em SQL utilizamos a instrução `UPDATE` :

```
UPDATE compras
```

Agora precisamos adicionar o que queremos alterar por meio da instrução `SET` :

```
UPDATE compras SET valor = 1500;
```

Precisamos sempre **tomar cuidado** com a instrução `UPDATE` , pois o exemplo acima executa normalmente, porém o que ele vai atualizar? Não informamos em momento algum qual compra precisa ser atualizar, ou seja, ele iria **atualizar TODAS as compras** e não é isso que queremos. Antes de executar o `UPDATE` , precisamos verificar o `id` da compra que queremos alterar:

```
mysql> SELECT id, valor, observacoes FROM compras
WHERE valor BETWEEN 1000 AND 2000
AND data BETWEEN '2013-01-01' AND '2013-12-31';
+-----+-----+
| id | valor  | observacoes |
+-----+-----+
| 11 | 1203.00 | Quartos    |
+-----+-----+
1 row in set (0,00 sec)
```

Agora que sabemos qual é o `id` da compra, basta informá-lo por meio da instrução `WHERE` :

```
mysql> UPDATE compras SET valor = 1500 WHERE id = 11;
Query OK, 1 row affected (0,01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Vamos verificar se o valor foi atualizado:

```
mysql> SELECT valor, observacoes FROM compras
WHERE valor BETWEEN 1000 AND 2000
AND data BETWEEN '2013-01-01' AND '2013-12-31';
+-----+-----+
| valor  | observacoes |
+-----+-----+
| 1500.00 | Quartos    |
+-----+-----+
1 row in set (0,00 sec)
```

Analisando melhor essa compra eu sei que se refere aos móveis do quarto, mas uma observação como "Quartos" que dizer o que? Não é clara o suficiente, pode ser que daqui a 6 meses, 1 ano ou mais tempo eu nem saiba mais o que se refere essa observação e vou acabar pensando: "Será que eu comprei móveis ou comprei um quarto todo?". Um tanto confuso... Então vamos alterar também as observações e deixar algo mais claro, como por exemplo: "Reforma de quartos".

```
UPDATE compra SET observacoes = 'Reforma de quartos' WHERE id = 11;
```

Verificando a nova observação:

```
mysql> SELECT valor, observacoes FROM compras
-> WHERE valor BETWEEN 1000 AND 2000
-> AND data BETWEEN '2013-01-01' AND '2013-12-31';
+-----+-----+
| valor | observacoes |
+-----+-----+
| 1500.00 | Reforma de quartos |
+-----+-----+
1 row in set (0,00 sec)
```

## 3.2 ATUALIZANDO VÁRIAS COLUNAS AO MESMO TEMPO

Assim como usamos o `UPDATE` para atualizar uma única coluna, poderíamos atualizar dois ou mais valores, separando os mesmos com vírgula. Por exemplo, se desejasse atualizar o valor e as observações:

```
mysql> UPDATE compras SET valor = 1500, observacoes = 'Reforma de quartos cara'
-> WHERE id = 11;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

E o resultado:

```
mysql> SELECT valor, observacoes FROM compras WHERE id = 11;
+-----+-----+
| valor | observacoes |
+-----+-----+
| 1500.00 | Reforma de quartos cara |
+-----+-----+
1 row in set (0,00 sec)
```

Muito cuidado aqui! Diferentemente do `WHERE`, no caso do `SET` não utilize o operador `AND` para atribuir valores, como no exemplo a seguir, que **NÃO** é o que queremos:

```
errado: UPDATE compras SET valor = 1500 AND observacoes = 'Reforma de quartos novos' WHERE id = 8;
certo:  UPDATE compras SET valor = 1500, observacoes = 'Reforma de quartos novos' WHERE id = 8;
```

## 3.3 UTILIZANDO UMA COLUNA COMO REFERÊNCIA PARA OUTRA COLUNA

Esqueci de adicionar 10% de imposto que paguei na compra de id 11... portanto quero fazer algo como:

```
mysql> SELECT valor FROM compras WHERE id = 11;
+-----+
| valor |
+-----+
| 1500.00 |
+-----+
1 row in set (0,00 sec)
```

Descobri o valor atual, agora multiplico por 1.1 para aumentar 10% :  $1500 * 1.1$  são 1650 reais. Então atualizo agora manualmente:

```
UPDATE compras SET valor = 1650 AND observacoes = 'Reforma de quartos novos' WHERE id = 11;
```

Mas e se eu quisesse fazer o mesmo para diversas linhas?

```
mysql> SELECT valor FROM compras WHERE id > 11 AND id <= 14;
+-----+
| valor |
+-----+
| 402.90 |
| 54.98 |
| 12.34 |
+-----+
3 rows in set (0,00 sec)
```

UPDATE .... e agora???

E agora? Vou ter que calcular na mão cada um dos casos, com 10% a mais, e fazer uma linha de UPDATE para cada um? A solução é fazer um único UPDATE em que digo que quero alterar o valor para o valor dele mesmo, vezes os meus 1.1 que já queria antes:

```
UPDATE compras SET valor = valor * 1.1
WHERE id >= 11 AND id <= 14;
```

Nessa query fica claro que eu posso usar o valor de um campo (qualquer) para atualizar um campo da mesma linha. No nosso caso usamos o valor original para calcular o novo valor. Mas estou livre para usar outros campos da mesma linha, desde que faça sentido para o meu problema, claro. Por exemplo, se eu tenho uma tabela de produtos com os campos precoLiquidado eu posso atualizar o precoBruto quando o imposto mudar para 15%:

```
UPDATE produtos SET precoBruto = precoLiquidado * 1.15;
```

### 3.4 UTILIZANDO O DELETE

Observei o meu rascunho e percebi que essa compra eu não devia ter sido cadastrada na minha tabela de compras, ou seja, eu preciso excluir esse registro do meu banco de dados. Em SQL, quando queremos excluir algum registro, utilizamos a instrução DELETE :

```
DELETE FROM compras;
```

O DELETE tem o comportamento similar ao UPDATE , ou seja, **precisamos sempre tomar cuidado** quando queremos excluir algum registro da tabela. Da mesma forma que fizemos com o UPDATE , precisamos adicionar a instrução WHERE para informa o que queremos excluir, justamente para **evitamos a exclusão de todos os dados**:

```
mysql> DELETE FROM compras WHERE id = 11;
Query OK, 1 row affected (0,01 sec)
```

Se verificarmos novamente se existe o registro dessa compra:

```
mysql> SELECT valor, observacoes FROM compras
WHERE valor BETWEEN 1000 AND 2000
AND data BETWEEN '2013-01-01' AND '2013-12-31';
Empty set (0,00 sec)
```

A compra foi excluída conforme o esperado!

### 3.5 CUIDADOS COM O DELETE E UPDATE

Vimos como o `DELETE` e `UPDATE` podem ser **perigosos** em banco de dados, se não definirmos uma condição para cada uma dessas instruções podemos excluir/alterar **TODAS** as informações da nossa tabela. A boa prática para executar essas instruções é **sempre** escrever a instrução `WHERE` antes, ou seja, definir primeiro qual será a condição para executar essas instruções:

```
WHERE id = 11;
```

Então adicionamos o `DELETE` / `UPDATE` :

```
DELETE FROM compras WHERE id = 11;
```

```
UPDATE compras SET valor = 1500 WHERE id = 11;
```

Dessa forma garantimos que o nosso banco de dados não tenha risco.

### 3.6 RESUMINDO

Nesse capítulo aprendemos como fazer *queries* por meio de intervalos utilizando o `BETWEEN` e como alterar/excluir os dados da nossa tabela utilizando o `DELETE` e o `UPDATE` . Vamos para os exercícios?

## EXERCÍCIOS

1. Altere as compras, colocando a observação 'preparando o natal' para todas as que foram efetuadas no dia 20/12/2014.
2. Altere o VALOR das compras feitas antes de 01/06/2013. Some R\$10,00 ao valor atual.
3. Atualize todas as compras feitas entre 01/07/2013 e 01/07/2014 para que elas tenham a observação 'entregue antes de 2014' e a coluna recebida com o valor TRUE.
4. Em um comando `WHERE` é possível especificar um intervalo de valores. Para tanto, é preciso dizer qual o valor mínimo e o valor máximo que define o intervalo. Qual é o operador que é usado para isso?
5. Qual operador você usa para remover linhas de compras de sua tabela?
6. Exclua as compras realizadas entre 05 e 20 março de 2013.
7. Existe um operador lógico chamado `NOT` . Esse operador pode ser usado para negar qualquer condição. Por exemplo, para selecionar qualquer registro com data diferente de 03/11/2014, pode ser construído o seguinte `WHERE` :

```
WHERE NOT DATA = '2011-11-03'
```

Use o operador `NOT` e monte um `SELECT` que retorna todas as compras com valor diferente de R\$ 108,00.



# ALTERANDO E RESTRINGINDO O FORMATO DE NOSSAS TABELAS

Muitas vezes que inserimos dados em um banco de dados, precisamos saber quais são os tipos de dados de cada coluna da tabela que queremos popular. Vamos dar uma olhada novamente na estrutura da nossa tabela por meio da instrução `DESC` :

```
mysql> DESC compras;
```

```
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id         | int(11)       | NO   | PRI | NULL    | auto_increment |
| valor      | decimal(18,2) | YES  |     | NULL    |                |
| data       | date          | YES  |     | NULL    |                |
| observacoes | varchar(255)  | YES  |     | NULL    |                |
| recebida   | tinyint(4)    | YES  |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0,00 sec)
```

Como já vimos, o MySQL demonstra algumas características das colunas da nossa tabela. Vamos verificar a coluna `Null`, o que será que ela significa? Será que está dizendo que nossas colunas aceitam valores vazios? Então vamos tentar inserir uma compra com `observacoes` **vazia**, ou seja, **nula**, com o valor `null`:

```
INSERT INTO compras (valor, data, recebida, observacoes)
VALUES (150, '2016-01-04', 1, NULL);
Query OK, 1 row affected (0,01 sec)
```

Vamos verificar o resultado:

```
mysql> SELECT * FROM compras WHERE data = '2016-01-04';
+-----+-----+-----+-----+-----+
| id | valor | data      | observacoes | recebida |
+-----+-----+-----+-----+-----+
| 47 | 150.00 | 2016-01-04 | NULL        | 1        |
+-----+-----+-----+-----+-----+
1 row in set (0,00 sec)
```

O nosso banco de dados permitiu o registro de uma compra sem observação. Mas em qual momento informamos ao MySQL que queríamos valores nulos? Em nenhum momento! Porém, quando criamos uma tabela no MySQL e não informamos se queremos ou não valores nulos para uma determinada coluna, por padrão, o MySQL aceita os valores nulos.

Note que um texto vazio é diferente de não ter nada! É diferente de um texto vazio como `"`. Nulo é

nada, é a inexistência de um valor. Um texto sem caracteres é um texto com zero caracteres. Um valor nulo nem texto é, nem nada é. Nulo é o não ser. Uma questão filosófica milenar, discutida entre os melhores filósofos e programadores, além de ser fonte de muitos bugs. Cuidado.

## 4.1 RESTRINGINDO OS NULOS

Para resolver esse problema podemos criar restrições, *Constraints*, que tem a capacidade de determinar as regras que as colunas de nossas tabelas terão. Antes de configurar o *Constraints*, vamos verificar todos os registros que tiverem observações nulas e vamos apagá-los. Queremos selecionar todas as observações que são nulas, são nulas, SÃO NULAS, IS NULL :

```
mysql> SELECT * FROM compras WHERE observacoes IS NULL;
+-----+-----+-----+-----+
| id | valor | data      | observacoes | recebida |
+-----+-----+-----+-----+
| 47 | 150.00 | 2016-01-04 | NULL        | 1        |
+-----+-----+-----+-----+
1 row in set (0,00 sec)
```

Vamos excluir todas as compras que tenham as observações nulas:

```
DELETE FROM compras WHERE observacoes IS NULL;
Query OK, 1 row affected (0,01 sec)
```

## 4.2 ADICIONANDO CONSTRAINTS

Podemos definir *Constraints* no momento da criação da tabela, como no caso de definir que nosso valor e data não podem ser nulos ( NOT NULL ):

```
CREATE TABLE compras(
id INT AUTO_INCREMENT PRIMARY KEY,
valor DECIMAL(18,2) NOT NULL,
data DATE NOT NULL,
...
```

Porém, a tabela já existe! E não é uma boa prática excluir a tabela e cria-la novamente, pois podemos **perder registros!** Para fazer alterações na estrutura da tabela, podemos utilizar a instrução ALTER TABLE que vimos antes:

```
ALTER TABLE compras;
```

Precisamos especificar o que queremos fazer na tabela, nesse caso modificar uma coluna e adicionar uma *Constraints*:

```
mysql> ALTER TABLE compras MODIFY COLUMN observacoes VARCHAR(255) NOT NULL;
Query OK, 45 rows affected (0,04 sec)
Records: 45 Duplicates: 0 Warnings: 0
```

Observe que foram alteradas todas as 45 linhas existentes no nosso banco de dados. Se verificarmos a estrutura da nossa tabela novamente:

```
mysql> DESC compras;
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id         | int(11)       | NO   | PRI | NULL    | auto_increment |
| valor      | decimal(18,2) | YES  |     | NULL    |                |
| data       | date          | YES  |     | NULL    |                |
| observacoes | varchar(255)  | NO   |     | NULL    |                |
| recebida   | tinyint(4)    | YES  |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0,01 sec)
```

Na coluna *Null* e na linha das *observacoes* está informando que não é mais permitido valores nulos. Vamos testar:

```
mysql> INSERT INTO compras (valor, data, recebida, observacoes)
VALUES (150, '2016-01-04', 1, NULL);
ERROR 1048 (23000): Column 'observacoes' cannot be null
```

## 4.3 VALORES DEFAULT

Vamos supor que a maioria das compras que registramos não são entregues e que queremos que o próprio MySQL entenda que, quando eu não informar a coluna *recebida*, ela seja populada com o valor 0. No MySQL, além de *Constraints*, podemos adicionar valores padrões, no inglês *Default*, em uma coluna utilizando a instrução *DEFAULT* :

```
mysql> ALTER TABLE compras MODIFY COLUMN recebida tinyint(1) DEFAULT 0;
Query OK, 0 rows affected (0,00 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

Inserindo uma compra nova sem informa a coluna *recebida* :

```
INSERT INTO compras (valor, data, observacoes)
VALUES (150, '2016-01-05', 'Compra de teste');
Query OK, 1 row affected (0,01 sec)
```

Verificando o valor inserido na coluna *recebida*:

```
mysql> SELECT * FROM compras;
+-----+-----+-----+-----+-----+-----+
| id | valor  | data       | observacoes          | recebida |
+-----+-----+-----+-----+-----+-----+
| 1  | 20.00  | 2016-01-05 | Lanchonete           | 1        |
| 2  | 15.00  | 2016-01-06 | Lanchonete           | 1        |
| 3  | 915.50 | 2016-01-06 | Guarda-roupa         | 0        |
| ...| ...    | ...        | ...                   | ...      |
| 48 | 150.00 | 2016-01-05 | Compra de teste      | 0        |
+-----+-----+-----+-----+-----+-----+
46 rows in set (0,00 sec)
```

Se quisemos informar que o valor padrão deveria ser entregue, bastaria utilizar o mesmo comando, porém modificando o 0 para 1.

## 4.4 EVOLUÇÃO DO BANCO

Nossa escola deseja agora entrar em contato por email com todos os pais de alunos de um determinado bairro devido a maior taxa de ocorrência de dengue. Como fazer isso se o campo de endereço não obrigava o bairro? Podemos a partir de agora adicionar um novo campo:

```
Aluno
- id
- nascimento
- nome
- serieAtual
- salaAtual
- email
- telefone
- bairro
- endereco
- nomeDoPai
- nomeDaMae
- telefoneDoPai
- telefoneDaMae
```

Além disso, depois de um ano descobrimos um bug no sistema. Alguns alunos saíram do mesmo, como representar que não existe mais série atual e nem sala atual para esses alunos? Podemos escolher três abordagens:

1. marcar os campos serieAtual e salaAtual como NULL
2. marcar os campos como NULL e colocar um campo ativo como 0 ou 1
3. colocar um campo ativo como 0 ou 1

Repare que o primeiro e o último parecem ser simples de implementar, claramente mais simples que implementar os dois ao mesmo tempo (item 2). Mas cada um deles implica em um problema de modelagem diferente.

Se assumimos (1) e colocamos campos serieAtual e salaAtual como NULL, toda vez que queremos saber quem está inativo temos uma query bem estranha, que não parece fazer isso:

```
select * from Alunos where serieAtual is null and salaAtual is null
```

Sério mesmo? Isso significa que o aluno está inativo? Dois nulos? Além disso, se eu permito nulos, eu posso ter um caso muito estranho como o aluno a seguir:

```
Guilherme Silveira, serieAtual 8, salaAtual null
```

Como assim? É um aluno novo que alguém/o sistema esqueceu de colocar a sala? Ou é um aluno antigo que alguém/o sistema esqueceu de remover a série?

Por outro lado, na terceira abordagem, a do campo ativo, solto, gera um possível aluno como:

```
Guilherme Silveira, serieAtual 8, salaAtual B, ativo 0
```

Como assim, o Guilherme está inativo mas está na 8B ao mesmo tempo? Ou ele está na 8B ou não está, não tem as duas coisas ao mesmo tempo. Modelo estranho.

Por fim, a abordagem de modelagem 2, que utiliza o novo campo e o valor nulo, garante que as queries sejam simples:

```
select * from Alunos where ativo=false;
```

Mas ainda permite casos estranhos como:

```
Guilherme Silveira, serieAtual 8, salaAtual B, ativo 0  
Guilherme Silveira, serieAtual NULL, salaAtual B, ativo 0  
Guilherme Silveira, serieAtual NULL, salaAtual NULL, ativo 1
```

Poderíamos usar recursos bem mais avançados para tentar nos proteger de alguns desses casos, mas esses recursos são, em geral, específicos de um banco determinado. Por exemplo, funcionam no Oracle mas não no MySQL. Ou no MySQL mas não no PostgreSQL. Nesse curso focamos em padrões do SQL que conseguimos aplicar, em geral, para todos eles.

E agora? Três modelagens diferentes, todas resolvem o problema e todas geram outros problemas técnicos. Modelar bancos é tomar decisões que apresentam vantagens e desvantagens. Não há regra em qual das três soluções devemos escolher, escolha a que você julgar melhor e mais justa para sua empresa/sistema/ocasião. O mais importante é tentar prever os possíveis pontos de erro como os mencionados aqui. Conhecer as falhas em nosso desenho de modelagem (design) de antemão permite que não tenhamos uma surpresa desagradável quando descobrirmos elas de uma maneira infeliz no futuro.

## 4.5 RESUMINDO

Vimos que além de criar as tabelas em nosso banco de dados precisamos verificar se alguma coluna pode receber valores nulos ou não, podemos determinar essas regras por meio de *Constraints* no momento da criação da tabela ou utilizando a instrução `ALTER TABLE` se caso a tabela exista e queremos modificar sua estrutura. Também vimos que em alguns casos os valores de cada `INSERT` pode se repetir muito e por isso podemos definir valores padrões com a instrução `DEFAULT`.

## EXERCÍCIOS

1. Configure o valor padrão para a coluna `recebida`.
2. Configure a coluna `observacoes` para não aceitar valores nulos.
3. No nosso modelo atual, qual campo você deixaria com valores `DEFAULT` e quais não? Justifique sua decisão. Note que como estamos falando de modelagem, não existe uma regra absoluta, existe vantagens e desvantagens na decisão que tomar, tente citá-las.
4. `NOT NULL` e `DEFAULT` podem ser usados também no `CREATE TABLE`? Crie uma tabela nova e adicione *Constraints* e valores `DEFAULT`.

5. Reescreva o `CREATE TABLE` do começo do curso, marcando `observacoes` como nulo e valor padrão do `recebida` como 1.

# AGRUPANDO DADOS E FAZENDO CONSULTAS MAIS INTELIGENTES

Já adicionamos muitos dados em nosso banco de dados e seria mais interessante fazermos *queries* mais robustas, como por exemplo, saber o total que já gastei. O MySQL fornece a função `SUM()` que soma todos dos valores de uma coluna:

```
mysql> SELECT SUM(valor) FROM compras;
+-----+
| SUM(valor) |
+-----+
|    43967.91 |
+-----+
1 row in set (0,00 sec)
```

Vamos verificar o total de todas as compras recebidas:

```
mysql> SELECT SUM(valor) FROM compras WHERE recebida = 1;
+-----+
| SUM(valor) |
+-----+
|    31686.75 |
+-----+
1 row in set (0,00 sec)
```

Agora todas as compras que não foram recebidas:

```
mysql> SELECT SUM(valor) FROM compras WHERE recebida = 0;
+-----+
| SUM(valor) |
+-----+
|    12281.16 |
+-----+
1 row in set (0,00 sec)
```

Podemos também, contar quantas compras foram recebidas por meio da função `COUNT()` :

```
SELECT COUNT(*) FROM compras WHERE recebida = 1;
+-----+
| COUNT(*) |
+-----+
|         26 |
+-----+
```

Agora vamos fazer com que seja retornado a soma de todas as compras recebidas e não recebidas, porém retornaremos a coluna `recebida`, ou seja, em uma linha estará as compras recebidas e a sua soma e em outra as não recebidas e sua soma:

```
mysql> SELECT recebida, SUM(valor) FROM compras;
+-----+-----+
| recebida | SUM(valor) |
+-----+-----+
|         1 | 43967.91   |
+-----+-----+
1 row in set (0,00 sec)
```

Observe que o resultado não saiu conforme o esperado... Mas por que será que isso aconteceu? Quando utilizamos a função `SUM()` do MySQL ela soma todos os valores da coluna e retorna apenas uma única linha, pois é uma **função de agregação**! Para resolvermos esse problema, podemos utilizar a instrução `GROUP BY` que indica como a soma precisa ser agrupada, ou seja, some todas as compras recebidas e agrupe em uma linha, some todas as compras não recebidas e agrupe em outra linha:

```
mysql> SELECT recebida, SUM(valor) FROM compras GROUP BY recebida;
+-----+-----+
| recebida | SUM(valor) |
+-----+-----+
|         0 | 12281.16   |
|         1 | 31686.75   |
+-----+-----+
2 rows in set (0,00 sec)
```

O resultado foi conforme o esperado, porém note que o nome da coluna para a soma está um pouco estranho, pois estamos aplicando uma função ao invés de retornar uma coluna, seria melhor se o nome retornado fosse apenas "soma". Podemos nomear as colunas por meio da instrução `AS` :

```
mysql> SELECT recebida, SUM(valor) AS soma FROM compras
GROUP BY recebida;
+-----+-----+
| recebida | soma      |
+-----+-----+
|         0 | 12281.16  |
|         1 | 31686.75  |
+-----+-----+
2 rows in set (0,00 sec)
```

Também podemos aplicar filtros em *queries* que utilizam funções de agregação:

```
mysql> SELECT recebida, SUM(valor) AS soma FROM compras
WHERE valor < 1000
GROUP BY recebida;
+-----+-----+
| recebida | soma      |
+-----+-----+
|         0 | 4325.96   |
|         1 | 8682.83   |
+-----+-----+
2 rows in set (0,00 sec)
```

Suponhamos uma *query* mais robusta, onde podemos verificar em qual mês e ano a compra foi entregue ou não e o valor da soma. Podemos retornar a informação de ano utilizando a função `YEAR()` e a informação de mês utilizando a função `MONTH()` :

```
mysql> SELECT MONTH(data) as mes, YEAR(data) as ano, recebida,
SUM(valor) AS soma FROM compras
GROUP BY recebida;
```



```

+-----+-----+-----+-----+
| mes | ano | recebida | soma |
+-----+-----+-----+-----+
| 1 | 2016 | 0 | 12281.16 |
| 1 | 2016 | 1 | 31686.75 |
+-----+-----+-----+-----+
2 rows in set (0,00 sec)

```

Lembre-se que estamos lidando com uma função de agregação! Por isso precisamos informar todas as colunas que queremos **agrupar**, ou seja, a coluna de mês e de ano:

```

mysql> SELECT MONTH(data) as mes, YEAR(data) as ano, recebida,
SUM(valor) AS soma FROM compras
GROUP BY recebida, mes, ano;

```

```

+-----+-----+-----+-----+
| mes | ano | recebida | soma |
+-----+-----+-----+-----+
| 1 | 2013 | 0 | 12.39 |
| 1 | 2016 | 0 | 2015.49 |
| 4 | 2013 | 0 | 54.98 |
| 4 | 2014 | 0 | 434.00 |
| 5 | 2012 | 0 | 3500.00 |
| 5 | 2013 | 0 | 12.34 |
| 5 | 2015 | 0 | 87.43 |
| 6 | 2014 | 0 | 1616.90 |
| 7 | 2015 | 0 | 12.34 |
| 8 | 2014 | 0 | 1709.00 |
| 9 | 2014 | 0 | 567.09 |
| 9 | 2015 | 0 | 213.50 |
| 10 | 2014 | 0 | 98.00 |
| 10 | 2015 | 0 | 1245.20 |
| 12 | 2013 | 0 | 78.65 |
| 12 | 2014 | 0 | 623.85 |
| 1 | 2013 | 1 | 8046.90 |
| 1 | 2014 | 1 | 827.50 |
| 1 | 2016 | 1 | 35.00 |
| 2 | 2012 | 1 | 200.00 |
| 2 | 2014 | 1 | 921.11 |
| 2 | 2015 | 1 | 986.36 |
| 3 | 2013 | 1 | 795.05 |
| 4 | 2012 | 1 | 1576.40 |
| 4 | 2014 | 1 | 11705.30 |
| 5 | 2014 | 1 | 678.43 |
| 7 | 2013 | 1 | 98.12 |
| 7 | 2015 | 1 | 32.09 |
| 9 | 2015 | 1 | 576.12 |
| 10 | 2014 | 1 | 631.53 |
| 11 | 2013 | 1 | 3212.40 |
| 11 | 2015 | 1 | 954.12 |
| 12 | 2012 | 1 | 163.45 |
| 12 | 2013 | 1 | 223.09 |
| 12 | 2015 | 1 | 23.78 |
+-----+-----+-----+-----+
35 rows in set (0,00 sec)

```

## 5.1 ORDENANDO OS RESULTADOS

Conseguimos criar uma *query* bem robusta, mas perceba que as informações estão bem desordenadas, a primeira vista é difícil de ver a soma do mês de janeiro em todos os anos, como também

a soma de todos os meses em um único ano. Para podermos ordenar as informações das colunas, podemos utilizar a instrução `ORDER BY` passando as colunas que queremos que sejam ordenadas. Vamos ordenar por mês primeiro:

```
mysql> SELECT MONTH(data) as mes, YEAR(data) as ano, recebida,
SUM(valor) AS soma FROM compras
GROUP BY recebida, mes, ano
ORDER BY mes;
```

mes	ano	recebida	soma
1	2013	1	8046.90
1	2014	1	827.50
1	2016	1	35.00
1	2016	0	2015.49
1	2013	0	12.39
2	2014	1	921.11
2	2012	1	200.00
2	2015	1	986.36
3	2013	1	795.05
4	2014	0	434.00
4	2013	0	54.98
4	2014	1	11705.30
4	2012	1	1576.40
5	2013	0	12.34
5	2014	1	678.43
5	2015	0	87.43
5	2012	0	3500.00
6	2014	0	1616.90
7	2015	0	12.34
7	2015	1	32.09
7	2013	1	98.12
8	2014	0	1709.00
9	2014	0	567.09
9	2015	0	213.50
9	2015	1	576.12
10	2014	1	631.53
10	2015	0	1245.20
10	2014	0	98.00
11	2013	1	3212.40
11	2015	1	954.12
12	2012	1	163.45
12	2013	1	223.09
12	2015	1	23.78
12	2014	0	623.85
12	2013	0	78.65

35 rows in set (0,01 sec)

Já está melhor! Mas agora vamos ordenar por mês e por ano:

```
mysql> SELECT MONTH(data) as mes, YEAR(data) as ano, recebida,
SUM(valor) AS soma FROM compras
GROUP BY recebida, mes, ano
ORDER BY mes, ano;
```

mes	ano	recebida	soma
1	2013	1	8046.90
1	2013	0	12.39
1	2014	1	827.50

1	2016	1	35.00
1	2016	0	2015.49
2	2012	1	200.00
2	2014	1	921.11
2	2015	1	986.36
3	2013	1	795.05
4	2012	1	1576.40
4	2013	0	54.98
4	2014	0	434.00
4	2014	1	11705.30
5	2012	0	3500.00
5	2013	0	12.34
5	2014	1	678.43
5	2015	0	87.43
6	2014	0	1616.90
7	2013	1	98.12
7	2015	0	12.34
7	2015	1	32.09
8	2014	0	1709.00
9	2014	0	567.09
9	2015	0	213.50
9	2015	1	576.12
10	2014	1	631.53
10	2014	0	98.00
10	2015	0	1245.20
11	2013	1	3212.40
11	2015	1	954.12
12	2012	1	163.45
12	2013	1	223.09
12	2013	0	78.65
12	2014	0	623.85
12	2015	1	23.78

-----+-----+-----+-----+  
35 rows in set (0,00 sec)

Ficou mais fácil de visualizar, só que ainda não conseguimos verificar de uma forma clara a soma de cada mês durante um ano. Perceba que a instrução `ORDER BY` prioriza as colunas pela ordem em que são informadas, ou seja, quando fizemos:

```
ORDER BY mes, ano;
```

Informamos que queremos que dê prioridade à ordenação da coluna `mês` e as demais colunas sejam ordenadas de acordo com o `mês`! Isso significa que para ordenarmos o ano e depois o mês basta apenas colocar o ano no início do `ORDER BY` :

```
mysql> SELECT MONTH(data) as mes, YEAR(data) as ano, recebida,
SUM(valor) AS soma FROM compras
GROUP BY recebida, mes, ano
ORDER BY ano, mes;
```

mes	ano	recebida	soma
2	2012	1	200.00
4	2012	1	1576.40
5	2012	0	3500.00
12	2012	1	163.45
1	2013	1	8046.90
1	2013	0	12.39
3	2013	1	795.05
4	2013	0	54.98

5	2013	0	12.34
7	2013	1	98.12
11	2013	1	3212.40
12	2013	1	223.09
12	2013	0	78.65
1	2014	1	827.50
2	2014	1	921.11
4	2014	0	434.00
4	2014	1	11705.30
5	2014	1	678.43
6	2014	0	1616.90
8	2014	0	1709.00
9	2014	0	567.09
10	2014	1	631.53
10	2014	0	98.00
12	2014	0	623.85
2	2015	1	986.36
5	2015	0	87.43
7	2015	0	12.34
7	2015	1	32.09
9	2015	0	213.50
9	2015	1	576.12
10	2015	0	1245.20
11	2015	1	954.12
12	2015	1	23.78
1	2016	1	35.00
1	2016	0	2015.49

-----+-----+-----+-----+-----+  
35 rows in set (0,00 sec)

Agora conseguimos verificar de uma forma clara a soma dos meses em cada ano. Além da soma podemos também utilizar outras funções de agragação como por exemplo, a `AVG()` que retorna a média de uma coluna:

```
mysql> SELECT MONTH(data) as mes, YEAR(data) as ano, recebida,
AVG(valor) AS soma FROM compras
GROUP BY recebida, mes, ano
ORDER BY ano, mes;
```

mes	ano	recebida	soma
2	2012	1	200.000000
4	2012	1	1576.400000
5	2012	0	3500.000000
12	2012	1	163.450000
1	2013	0	12.390000
1	2013	1	2682.300000
3	2013	1	397.525000
4	2013	0	54.980000
5	2013	0	12.340000
7	2013	1	98.120000
11	2013	1	3212.400000
12	2013	1	223.090000
12	2013	0	78.650000
1	2014	1	827.500000
2	2014	1	460.555000
4	2014	0	434.000000
4	2014	1	5852.650000
5	2014	1	678.430000
6	2014	0	808.450000
8	2014	0	1709.000000

	9		2014		0		567.090000	
	10		2014		1		631.530000	
	10		2014		0		98.000000	
	12		2014		0		311.925000	
	2		2015		1		493.180000	
	5		2015		0		87.430000	
	7		2015		1		32.090000	
	7		2015		0		12.340000	
	9		2015		1		576.120000	
	9		2015		0		213.500000	
	10		2015		0		1245.200000	
	11		2015		1		954.120000	
	12		2015		1		23.780000	
	1		2016		1		17.500000	
	1		2016		0		671.830000	

+-----+-----+-----+-----+  
35 rows in set (0,00 sec)

## 5.2 RESUMINDO

Vimos como podemos fazer *queries* mais robustas e inteligentes utilizando funções de agregação, como por exemplo, a `SUM()` para somar e a `AVG()` para tirar a média. Vimos também que quando queremos retornar outras colunas ao utilizar funções de agregação, precisamos utilizar a instrução `GROUP BY` para determinar quais serão as colunas que queremos que seja feita o agrupamento e que nem sempre o resultado vem organizado e por isso, em determinados casos, precisamos utilizar a instrução `ORDER BY` para ordenar a nossa *query* por meio de uma coluna.

## EXERCÍCIOS

1. Calcule a média de todas as compras com datas inferiores a 12/05/2013.
2. Calcule a quantidade de compras com datas inferiores a 12/05/2013 e que já foram recebidas.
3. Calcule a soma de todas as compras, agrupadas se a compra recebida ou não.

# JUNTANDO DADOS DE VÁRIAS TABELAS

A nossa tabela de compras está bem populada, com muitas informações das compras realizadas, porém está faltando uma informação muito importante, que são os compradores. Por vezes foi eu quem comprou algo, mas também o meu irmão pode ter comprado ou até mesmo o meu primo... Como podemos fazer para identificar o comprador de uma compra? De acordo com o que vimos até agora podemos adicionar uma nova coluna chamada `comprador` com o tipo `varchar(200)` :

```
mysql> ALTER TABLE compras ADD COLUMN comprador VARCHAR(200);
Query OK, 0 rows affected (0,04 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

Vamos verificar como ficou a estrutura da nossa tabela:

```
mysql> DESC compras;
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id         | int(11)       | NO   | PRI | NULL    | auto_increment |
| valor      | decimal(18,2) | YES  |     | NULL    |                |
| data       | date          | YES  |     | NULL    |                |
| observacoes | varchar(255)  | NO   |     | NULL    |                |
| recebida   | tinyint(1)    | YES  |     | 0       |                |
| comprador  | varchar(200)  | YES  |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+
6 rows in set (0,00 sec)
```

Ótimo! Agora podemos adicionar os compradores de cada compra! Mas, antes de adicionarmos precisamos verificar novamente verificar as informações das compras e identificar quem foi que comprou:

```
mysql> SELECT id, valor, observacoes, data FROM compras;
+-----+-----+-----+-----+
| id | valor  | observacoes          | data      |
+-----+-----+-----+-----+
| 1  | 20.00  | Lanchonete           | 2016-01-05 |
| 2  | 15.00  | Lanchonete           | 2016-01-06 |
| 3  | 915.50 | Guarda-roupa         | 2016-01-06 |
| 4  | 949.99 | Smartphone           | 2016-01-10 |
| 5  | 200.00 | Material escolar     | 2012-02-19 |
| 6  | 3500.00 | Televisao            | 2012-05-21 |
| 7  | 1576.45 | Material de construcao | 2012-04-30 |
| 8  | 163.45 | Pizza pra familia    | 2012-12-15 |
| 9  | 4780.00 | Sala de estar        | 2013-01-23 |
| 10 | 392.15 | Quartos              | 2013-03-03 |
| 12 | 402.90 | Copa                 | 2013-03-21 |
| 13 | 54.98  | Lanchonete           | 2013-04-12 |
| 14 | 12.34  | Lanchonete           | 2013-05-23 |
```

15	78.65	Lanchonete	2013-12-04
16	12.39	Sorvete no parque	2013-01-06
17	98.12	Hopi Hari	2013-07-09
18	2498.00	Compras de janeiro	2013-01-12
19	3212.40	Compras do mes	2013-11-13
20	223.09	Compras de natal	2013-12-17
21	768.90	Festa	2013-01-16
22	827.50	Festa	2014-01-09
23	12.00	Salgado no aeroporto	2014-02-19
24	678.43	Passagem pra Bahia	2014-05-21
25	10937.12	Carnaval em Cancun	2014-04-30
26	1501.00	Presente da sogra	2014-06-22
27	1709.00	Parcela da casa	2014-08-25
28	567.09	Parcela do carro	2014-09-25
29	631.53	IPTU	2014-10-12
30	909.11	IPVA	2014-02-11
31	768.18	Gasolina viagem Porto Alegre	2014-04-10
32	434.00	Rodeio interior de Sao Paulo	2014-04-01
33	115.90	Dia dos namorados	2014-06-12
34	98.00	Dia das crianças	2014-10-12
35	253.70	Natal - presentes	2014-12-20
36	370.15	Compras de natal	2014-12-25
37	32.09	Lanchonete	2015-07-02
38	954.12	Show da Ivete Sangalo	2015-11-03
39	98.70	Lanchonete	2015-02-07
40	213.50	Roupas	2015-09-25
41	1245.20	Roupas	2015-10-17
42	23.78	Lanchonete do Zé	2015-12-18
43	576.12	Sapatos	2015-09-13
44	12.34	Canetas	2015-07-19
45	87.43	Gravata	2015-05-10
46	887.66	Presente para o filhao	2015-02-02
48	150.00	Compra de teste	2016-01-05

+-----+-----+-----+-----+-----+  
46 rows in set (0,00 sec)

Agora que já sei as informações das compras posso adicionar os seus compradores. Começaremos pelas 5 primeiras compras:

1	20.00	Lanchonete	2016-01-05
2	15.00	Lanchonete	2016-01-06
3	915.50	Guarda-roupa	2016-01-06
4	949.99	Smartphone	2016-01-10
5	200.00	Material escolar	2012-02-19

De acordo com as informações do meu rascunho, a primeira compra fui eu mesmo que comprei:

```
mysql> UPDATE compras SET comprador = 'Alex Felipe' WHERE id = 1;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

A segunda o meu primo e xará Alex Vieira que comprou:

```
mysql> UPDATE compras SET comprador = 'Alex Vieira' WHERE id = 2;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

O guarda-roupa foi o meu tio João da Silva que comprou:

```
mysql> UPDATE compras SET comprador = 'João da Silva' WHERE id = 3;
Query OK, 1 row affected (0,00 sec)
```

Rows matched: 1 Changed: 1 Warnings: 0

O smartphone fui eu:

```
mysql> UPDATE compras SET comprador = 'Alex Felipe' WHERE id = 4;
Query OK, 1 row affected (0,01 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

E o material escolar foi o meu tio João da Silva:

```
mysql> UPDATE compras SET comprador = 'João da Silva' WHERE id = 5;
Query OK, 1 row affected (0,01 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

Vejamos o resultado:

```
mysql> SELECT * FROM compras;
+----+-----+-----+-----+-----+-----+
| id | valor | data      | observacoes | recebida | comprador |
+----+-----+-----+-----+-----+-----+
| 1  | 20.00 | 2016-01-05 | Lanchonete   | 1        | Alex Felipe |
| 2  | 15.00 | 2016-01-06 | Lanchonete   | 1        | Alex Vieira |
| 3  | 915.50 | 2016-01-06 | Guarda-roupa | 0        | João da Silva |
| 4  | 949.99 | 2016-01-10 | Smartphone    | 0        | Alex Felipe |
| 5  | 200.00 | 2012-02-19 | Material escolar | 1        | João da Silva |
|... | ...   | ...       | ...          | ...      | ...        |
+----+-----+-----+-----+-----+-----+
46 rows in set (0,01 sec)
```

Pensando bem, não foi o meu tio João da Silva que comprou Material escolar e sim o meu tio João Vieira, eu devo ter anotado errado... Então vamos alterar:

```
mysql> UPDATE compras SET comprador = 'João vieira' WHERE comprador = 'João da Silva';
Query OK, 2 rows affected (0,01 sec)
Rows matched: 2 Changed: 2 Warnings: 0
```

Vejamos como ficou o resultado:

```
mysql> select * from compras limit 5;
+----+-----+-----+-----+-----+-----+
| id | valor | data      | observacoes | recebida | comprador |
+----+-----+-----+-----+-----+-----+
| 1  | 20.00 | 2016-01-05 | Lanchonete   | 1        | Alex Felipe |
| 2  | 15.00 | 2016-01-06 | Lanchonete   | 1        | Alex Vieira |
| 3  | 915.50 | 2016-01-06 | Guarda-roupa | 0        | João vieira |
| 4  | 949.99 | 2016-01-10 | Smartphone    | 0        | Alex Felipe |
| 5  | 200.00 | 2012-02-19 | Material escolar | 1        | João vieira |
+----+-----+-----+-----+-----+-----+
5 rows in set (0,00 sec)
```

Ops! Espera aí! O meu tio João da Silva sumiu!? E o meu tio João Vieira foi inserido com o sobrenome minúsculo... Vamos alterar novamente... Primeiro o nome do João Vieira:

```
mysql> UPDATE compras SET comprador = 'João Vieira' WHERE comprador = 'João vieira';
Query OK, 2 rows affected (0,01 sec)
Rows matched: 2 Changed: 2 Warnings: 0
```

Agora vamos reenserrar o meu tio João da Silva para a compra do guarda-roupa:



```
mysql> UPDATE compras SET comprador = 'João da Silva' WHERE id = 3;
Query OK, 1 row affected (0,01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Vamos verificar novamente a nossa tabela:

```
mysql> SELECT * FROM compras LIMIT 5;
+----+-----+-----+-----+-----+-----+
| id | valor | data      | observacoes | recebida | comprador |
+----+-----+-----+-----+-----+-----+
| 1  | 20.00 | 2016-01-05 | Lanchonete   | 1        | Alex Felipe |
| 2  | 15.00 | 2016-01-06 | Lanchonete   | 1        | Alex Vieira |
| 3  | 915.50 | 2016-01-06 | Guarda-roupa | 0        | João da Silva |
| 4  | 949.99 | 2016-01-10 | Smartphone   | 0        | Alex Felipe |
| 5  | 200.00 | 2012-02-19 | Material escolar | 1        | João Vieira |
|... | ...   | ...       | ...          | ...      | ...        |
+----+-----+-----+-----+-----+-----+
5 rows in set (0,00 sec)
```

Nossa, que trabalheira! Por causa de um errinho eu tive que **alterar tudo novamente!** Se eu não ficar atento, com certeza acontecerá uma caca gigante no meu banco. Além disso, eu preciso também adicionar o telefone do comprador para um dia, se for necessário, entrar em contato! Então, novamente, iremos adicionar um coluna nova com o nome `telefone` e tipo `VARCHAR(15)` :

```
mysql> ALTER TABLE compras ADD COLUMN telefone VARCHAR(30);
Query OK, 0 rows affected (0,06 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Vamos adicionar o telefone de cada um. Primeiro o meu telefone:

```
mysql> UPDATE compras SET telefone = '5571-2751' WHERE comprador = 'Alex Felipe';
Query OK, 2 rows affected (0,01 sec)
Rows matched: 2  Changed: 2  Warnings: 0
```

Agora o do meu xará, Alex Vieira:

```
mysql> UPDATE compras SET telefone = '5083-3884' WHERE comprador = 'Alex Vieira';
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Por fim, os telefones dos meus tios:

```
mysql> UPDATE compras SET telefone = '2220-4156' WHERE comprador = 'João da Silva';
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> UPDATE compras SET telefone = '2297-0033' WHERE comprador = 'João Vieira';
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Vejamos como ficou a nossa tabela:

```
SELECT * FROM compras;
+----+-----+-----+-----+-----+-----+-----+
| id | valor | data      | observacoes | recebida | comprador | telefone |
+----+-----+-----+-----+-----+-----+-----+
| 1  | 20.00 | 2016-01-05 | Lanchonete   | 1        | Alex Felipe | 5571-2751 |
| 2  | 15.00 | 2016-01-06 | Lanchonete   | 1        | Alex Vieira | 5083-3884 |
| 3  | 915.50 | 2016-01-06 | Guarda-roupa | 0        | João da Silva | 2220-4156 |
```

4	949.99	2016-01-10	Smartphone	0	Alex Felipe	5571-2751
5	200.00	2012-02-19	Material escolar	1	João Vieira	2297-0033
...	...	...	...	...	...	...

46 rows in set (0,00 sec)

Certo, agora nossos compradores possuem telefone também. Ainda faltam mais compras sem comprador, então continuaremos adicionando os compradores. A próxima compra foi o meu primo Alex Vieira que comprou. Vamos inseri-lo novamente:

```
mysql> UPDATE compras SET comprador = 'Alex Vieira' WHERE id = 6;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Consultando nossa tabela para verificar como está ficando:

```
mysql> SELECT * FROM compras;
```

id	valor	data	observacoes	recebida	comprador	telefone
1	20.00	2016-01-05	Lanchonete	1	Alex Felipe	5571-2751
2	15.00	2016-01-06	Lanchonete	1	Alex Vieira	5083-3884
3	915.50	2016-01-06	Guarda-roupa	0	João da Silva	2220-4156
4	949.99	2016-01-10	Smartphone	0	Alex Felipe	5571-2751
5	200.00	2012-02-19	Material escolar	1	João Vieira	2297-0033
6	3500.00	2012-05-21	Televisao	0	Alex Vieira	NULL
...	...	...	...	...	...	...

46 rows in set (0,00 sec)

Ah não! Esqueci de colocar o telefone. Deixa eu colocar:

```
mysql> UPDATE compras SET telefone = '5571-2751' WHERE id = 6;
Query OK, 1 row affected (0,00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Vamos ver se agora vai dar certo:

```
mysql> SELECT * FROM compras;
```

id	valor	data	observacoes	recebida	comprador	telefone
1	20.00	2016-01-05	Lanchonete	1	Alex Felipe	5571-2751
2	15.00	2016-01-06	Lanchonete	1	Alex Vieira	5083-3884
3	915.50	2016-01-06	Guarda-roupa	0	João da Silva	2220-4156
4	949.99	2016-01-10	Smartphone	0	Alex Felipe	5571-2751
5	200.00	2012-02-19	Material escolar	1	João Vieira	2297-0033
6	3500.00	2012-05-21	Televisao	0	Alex Vieira	5571-2751
...	...	...	...	...	...	...

46 rows in set (0,00 sec)

Poxa vida... Percebi que ao invés de colocar o telefone do meu primo acabei colocando o meu... Quanto sofrimento por causa de 2 colunas novas! Além de ter que me preocupar se os nomes dos compradores estão sendo exatamente iguais, agora eu preciso me preocupar se os telefones também estão corretos e mais, todas as vezes que eu preciso inserir um comprador eu preciso lembrar de inserir o seu telefone... E se agora eu precisasse adicionar o endereço, e-mail ou quaisquer informações do

comprador? Eu teria que **lembrar todas** essas informações **cada vez** que eu **inserir apenas uma compra!** Que horror!

Veja o quão problemático está sendo manter as informações de um comprador em uma tabela de compras. Além de trabalhosa, a inserção é bem problemática, pois há um grande risco de falhas no meu banco de dados como por exemplo: informações redundantes (que se repetem) ou então informações incoerentes (o mesmo comprador com alguma informação diferente). Com toda certeza não é uma boa solução para a nossa necessidade! Será que não existe uma forma diferente de resolver isso?

## 6.1 NORMALIZANDO NOSSO MODELO

Repara que temos dois elementos, duas entidades, em uma unica tabela: a compra e o comprador. Quando nos deparamos com esses tipos de problemas **criamos novas tabelas**. Então vamos criar uma tabela chamada compradores.

No nosso caso queremos que cada comprador seja representado pelo seu nome, endereço e telefone. Como já pensamos em modelagem antes, aqui fica o nosso modelo de tabela para representar os compradores:

```
mysql> CREATE TABLE compradores (
id INT PRIMARY KEY AUTO_INCREMENT,
nome VARCHAR(200),
endereco VARCHAR(200),
telefone VARCHAR(30)
);
Query OK, 0 rows affected (0,01 sec)
```

Vamos verificar a nossa tabela por meio do DESC :

```
mysql> DESC compradores;
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id         | int(11)       | NO   | PRI | NULL    | auto_increment |
| nome       | varchar(200)  | YES  |     | NULL    |                |
| endereco  | varchar(200)  | YES  |     | NULL    |                |
| telefone   | varchar(30)   | YES  |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0,00 sec)
```

Agora que criamos a nossa tabela, vamos inserir alguns compradores, no meu caso, as compras foram feitas apenas por mim e pelo meu tio João da Silva, então adicionaremos apenas 2 compradores:

```
mysql> INSERT INTO compradores (nome, endereco, telefone) VALUES
('Alex Felipe', 'Rua Vergueiro, 3185', '5571-2751');
Query OK, 1 row affected (0,01 sec)

mysql> INSERT INTO compradores (nome, endereco, telefone) VALUES
('João da Silva', 'Av. Paulista, 6544', '2220-4156');
Query OK, 1 row affected (0,01 sec)
```

Vamos verificar os nossos compradores:

```
mysql> SELECT * FROM compradores;
+-----+-----+-----+-----+
| id | nome          | endereco          | telefone |
+-----+-----+-----+-----+
| 1 | Alex Felipe   | Rua Vergueiro, 3185 | 5571-2751 |
| 2 | João da Silva | Av. Paulista, 6544  | 2220-4156 |
+-----+-----+-----+-----+
2 rows in set (0,00 sec)
```

Criamos agora duas tabelas diferentes na nossa base de dados, a tabela `compras` e a tabela `compradores`. Então não precisamos mais das inforções dos compradores na tabela `compras`, ou seja, vamos excluir as colunas `comprador` e `telefone`, mas como podemos excluir uma coluna? Precisamos alterar a estrutura da tabela, então começaremos pelo `ALTER TABLE`:

```
ALTER TABLE compras
```

Se para adicionar uma coluna utilizamos a instrução `ADD COLUMN`, logo, para excluir uma tabela, utilizaremos a instrução `DROP COLUMN`:

```
mysql> ALTER TABLE compras DROP COLUMN comprador;
Query OK, 0 rows affected (0,06 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

Vamos verificar a estrutura da nossa tabela:

```
mysql> DESC compras;
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id         | int(11)       | NO   | PRI | NULL    | auto_increment |
| valor      | decimal(18,2) | YES  |     | NULL    |                |
| data       | date          | YES  |     | NULL    |                |
| observacoes | varchar(255)  | NO   |     | NULL    |                |
| recebida   | tinyint(1)    | YES  |     | 0       |                |
| telefone   | varchar(30)   | YES  |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+
6 rows in set (0,00 sec)
```

A coluna `comprador` foi excluída com sucesso! Por fim, excluiríamos a coluna `telefone`:

```
mysql> ALTER TABLE compras DROP COLUMN telefone;
Query OK, 0 rows affected (0,03 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

Agora que excluímos todas as colunas relacionadas aos compradores da tabela `compras`, precisamos de alguma forma associar uma compra com um comprador. Então vamos criar uma nova coluna na tabela `compras` e adicionar o `id` do comprador nessa coluna:

```
mysql> ALTER TABLE compras ADD COLUMN id_compradores int;
Query OK, 0 rows affected (0,03 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

Por que o `id` dos compradores precisa ficar na tabela `compras` e não o `id` da tabela `compras` na tabela `compradores`? Precisamos pensar seguinte forma: uma compra só pode ser de 1 único comprador e o comprador pode ter 1 ou mais compras, como podemos garantir que **1 compra pertença**

**apenas a 1 comprador?** Fazendo com que a compra saiba quem é o seu **único comprador!** Ou seja, recebendo a chave do seu dono! Por isso adicionamos o `id` dos compradores na tabela de compras, **se fizéssemos o contrário**, ou seja, adicionássemos o `id` da compra na tabela compradores, iríamos permitir que **1 única compra** fosse realizada por 1 ou mais compradores, o que não faz sentido algum sabendo que as minhas compras só foram realizadas por mim apenas e as do meu tio por ele apenas.

Temos a nossa nova coluna para associar o comprador com a sua compra, então vamos fazer `UPDATE` na tabela compras inserindo o comprador de `id` 1 para a primeira metade das compras e inserindo a segunda metade para o comprador de `id` 2. Primeiro vamos verificar quantas compras nós temos usando a função `COUNT()` ;

```
mysql> SELECT count(*) FROM compras;
+-----+
| count(*) |
+-----+
|         46 |
+-----+
1 row in set (0,00 sec)
```

Vamos atualizar a primeira metade:

```
mysql> UPDATE compras SET id_compradores = 1 WHERE id < 22;
Query OK, 20 rows affected (0,01 sec)
Rows matched: 20  Changed: 20  Warnings: 0
```

Agora a segunda metade:

```
mysql> UPDATE compras SET id_compradores = 2 WHERE id > 21;
Query OK, 26 rows affected (0,01 sec)
Rows matched: 26  Changed: 26  Warnings: 0
```

Verificando as compras no nosso banco de dados:

```
mysql> SELECT * FROM compras;
```

id	valor	data	observacoes	recebida	id_compradores
1	20.00	2016-01-05	Lanchonete	1	1
2	15.00	2016-01-06	Lanchonete	1	1
3	915.50	2016-01-06	Guarda-roupa	0	1
4	949.99	2016-01-10	Smartphone	0	1
5	200.00	2012-02-19	Material escolar	1	1
6	3500.00	2012-05-21	Televisao	0	1
7	1576.40	2012-04-30	Material de construcao	1	1
8	163.45	2012-12-15	Pizza pra familia	1	1
9	4780.00	2013-01-23	Sala de estar	1	1
10	392.15	2013-03-03	Quartos	1	1
12	402.90	2013-03-21	Copa	1	1
13	54.98	2013-04-12	Lanchonete	0	1
14	12.34	2013-05-23	Lanchonete	0	1
15	78.65	2013-12-04	Lanchonete	0	1
16	12.39	2013-01-06	Sorvete no parque	0	1
17	98.12	2013-07-09	Hopi Hari	1	1
18	2498.00	2013-01-12	Compras de janeiro	1	1
19	3212.40	2013-11-13	Compras do mes	1	1
20	223.09	2013-12-17	Compras de natal	1	1
21	768.90	2013-01-16	Festa	1	1

22	827.50	2014-01-09	Festa	1	2
23	12.00	2014-02-19	Salgado no aeroporto	1	2
24	678.43	2014-05-21	Passagem pra Bahia	1	2
25	10937.12	2014-04-30	Carnaval em Cancun	1	2
26	1501.00	2014-06-22	Presente da sogra	0	2
27	1709.00	2014-08-25	Parcela da casa	0	2
28	567.09	2014-09-25	Parcela do carro	0	2
29	631.53	2014-10-12	IPTU	1	2
30	909.11	2014-02-11	IPVA	1	2
31	768.18	2014-04-10	Gasolina viagem Porto Alegre	1	2
32	434.00	2014-04-01	Rodeio interior de Sao Paulo	0	2
33	115.90	2014-06-12	Dia dos namorados	0	2
34	98.00	2014-10-12	Dia das crianças	0	2
35	253.70	2014-12-20	Natal - presentes	0	2
36	370.15	2014-12-25	Compras de natal	0	2
37	32.09	2015-07-02	Lanchonete	1	2
38	954.12	2015-11-03	Show da Ivete Sangalo	1	2
39	98.70	2015-02-07	Lanchonete	1	2
40	213.50	2015-09-25	Roupas	0	2
41	1245.20	2015-10-17	Roupas	0	2
42	23.78	2015-12-18	Lanchonete do Zé	1	2
43	576.12	2015-09-13	Sapatos	1	2
44	12.34	2015-07-19	Canetas	0	2
45	87.43	2015-05-10	Gravata	0	2
46	887.66	2015-02-02	Presente para o filho	1	2
48	150.00	2016-01-05	Compra de teste	0	2

46 rows in set (0,00 sec)

## 6.2 ONE TO MANY/MANY TO ONE

Repare que o que fizemos foi deixar claro que um comprador pode ter muitas compras (um para muitos), ou ainda que muitas compras podem vir do mesmo comprador (many to one), só depende do ponto de vista. Chamamos então de uma relação One to Many (ou Many to One).

## 6.3 FOREIGN KEY

Conseguimos associar uma compra com um comprador, então agora vamos buscar as informações das compras e seus respectivos compradores:

```
mysql> SELECT * FROM compras;
```

id	valor	data	observacoes	recebida	id_compradores
1	20.00	2016-01-05	Lanchonete	1	1
2	15.00	2016-01-06	Lanchonete	1	1
3	915.50	2016-01-06	Guarda-roupa	0	1
4	949.99	2016-01-10	Smartphone	0	1
5	200.00	2012-02-19	Material escolar	1	1
6	3500.00	2012-05-21	Televisao	0	1
7	1576.40	2012-04-30	Material de construcao	1	1
8	163.45	2012-12-15	Pizza pra familia	1	1
9	4780.00	2013-01-23	Sala de estar	1	1
10	392.15	2013-03-03	Quartos	1	1
12	402.90	2013-03-21	Copa	1	1
13	54.98	2013-04-12	Lanchonete	0	1
14	12.34	2013-05-23	Lanchonete	0	1

15	78.65	2013-12-04	Lanchonete	0	1
16	12.39	2013-01-06	Sorvete no parque	0	1
17	98.12	2013-07-09	Hopi Hari	1	1
18	2498.00	2013-01-12	Compras de janeiro	1	1
19	3212.40	2013-11-13	Compras do mes	1	1
20	223.09	2013-12-17	Compras de natal	1	1
21	768.90	2013-01-16	Festa	1	1
22	827.50	2014-01-09	Festa	1	2
23	12.00	2014-02-19	Salgado no aeroporto	1	2
24	678.43	2014-05-21	Passagem pra Bahia	1	2
25	10937.12	2014-04-30	Carnaval em Cancun	1	2
26	1501.00	2014-06-22	Presente da sogra	0	2
27	1709.00	2014-08-25	Parcela da casa	0	2
28	567.09	2014-09-25	Parcela do carro	0	2
29	631.53	2014-10-12	IPTU	1	2
30	909.11	2014-02-11	IPVA	1	2
31	768.18	2014-04-10	Gasolina viagem Porto Alegre	1	2
32	434.00	2014-04-01	Rodeio interior de Sao Paulo	0	2
33	115.90	2014-06-12	Dia dos namorados	0	2
34	98.00	2014-10-12	Dia das crianças	0	2
35	253.70	2014-12-20	Natal - presentes	0	2
36	370.15	2014-12-25	Compras de natal	0	2
37	32.09	2015-07-02	Lanchonete	1	2
38	954.12	2015-11-03	Show da Ivete Sangalo	1	2
39	98.70	2015-02-07	Lanchonete	1	2
40	213.50	2015-09-25	Roupas	0	2
41	1245.20	2015-10-17	Roupas	0	2
42	23.78	2015-12-18	Lanchonete do Zé	1	2
43	576.12	2015-09-13	Sapatos	1	2
44	12.34	2015-07-19	Canetas	0	2
45	87.43	2015-05-10	Gravata	0	2
46	887.66	2015-02-02	Presente para o filhao	1	2
48	150.00	2016-01-05	Compra de teste	0	2

46 rows in set (0,00 sec)

```
mysql> SELECT * FROM compradores;
```

id	nome	endereço	telefone
1	Alex Felipe	Rua Vergueiro, 3185	5571-2751
2	João da Silva	Av. Paulista, 6544	2220-4156

2 rows in set (0,00 sec)

Não... Não era isso que eu queria, eu quero que, em apenas uma *query*, nesse caso um `SELECT`, retorne tanto as informações da tabela `compras` e da tabela `compradores`. Será que podemos fazer isso? Sim, podemos! Lembra da instrução `FROM` que indica qual é a tabela que estamos buscando as informações? Além de buscar por uma única tabela, podemos indicar que queremos buscar por mais de 1 tabela separando as tabelas por `","`:

```
SELECT * FROM compras, compradores, tabela3, tabela4, ...;
```

Então vamos adicionar em um único `SELECT` as tabelas: `compras` e `compradores`:

```
mysql> SELECT * FROM compras, compradores;
```

id	valor	data	observacoes	recebida	id_compradores	id	nome

Dados Pessoais					Dados da Transação				
Nome	CPF	Data	Descrição	Valor	CPF	Data	Descrição	Valor	
Alex Felipe	123.456.789-01	2016-01-05	Lanchonete	20.00	987.654.321-09	2016-01-05	Lanchonete	20.00	
			Rua Vergueiro, 3185	5571-2751			Rua Vergueiro, 3185	5571-2751	
João da Silva	987.654.321-09	2016-01-05	Lanchonete	20.00	123.456.789-01	2016-01-05	Lanchonete	20.00	
			Av. Paulista, 6544	2220-4156			Av. Paulista, 6544	2220-4156	
Alex Felipe	123.456.789-01	2016-01-06	Lanchonete	15.00	987.654.321-09	2016-01-06	Lanchonete	15.00	
			Rua Vergueiro, 3185	5571-2751			Rua Vergueiro, 3185	5571-2751	
João da Silva	987.654.321-09	2016-01-06	Lanchonete	15.00	123.456.789-01	2016-01-06	Lanchonete	15.00	
			Av. Paulista, 6544	2220-4156			Av. Paulista, 6544	2220-4156	
Alex Felipe	123.456.789-01	2016-01-06	Guarda-roupa	915.50	987.654.321-09	2016-01-06	Guarda-roupa	915.50	
			Rua Vergueiro, 3185	5571-2751			Rua Vergueiro, 3185	5571-2751	
João da Silva	987.654.321-09	2016-01-06	Guarda-roupa	915.50	123.456.789-01	2016-01-06	Guarda-roupa	915.50	
			Av. Paulista, 6544	2220-4156			Av. Paulista, 6544	2220-4156	
Alex Felipe	123.456.789-01	2016-01-10	Smartphone	949.99	987.654.321-09	2016-01-10	Smartphone	949.99	
			Rua Vergueiro, 3185	5571-2751			Rua Vergueiro, 3185	5571-2751	
João da Silva	987.654.321-09	2016-01-10	Smartphone	949.99	123.456.789-01	2016-01-10	Smartphone	949.99	
			Av. Paulista, 6544	2220-4156			Av. Paulista, 6544	2220-4156	
Alex Felipe	123.456.789-01	2012-02-19	Material escolar	200.00	987.654.321-09	2012-02-19	Material escolar	200.00	
			Rua Vergueiro, 3185	5571-2751			Rua Vergueiro, 3185	5571-2751	
João da Silva	987.654.321-09	2012-02-19	Material escolar	200.00	123.456.789-01	2012-02-19	Material escolar	200.00	
			Av. Paulista, 6544	2220-4156			Av. Paulista, 6544	2220-4156	
Alex Felipe	123.456.789-01	2012-05-21	Televisao	3500.00	987.654.321-09	2012-05-21	Televisao	3500.00	
			Rua Vergueiro, 3185	5571-2751			Rua Vergueiro, 3185	5571-2751	
João da Silva	987.654.321-09	2012-05-21	Televisao	3500.00	123.456.789-01	2012-05-21	Televisao	3500.00	
			Av. Paulista, 6544	2220-4156			Av. Paulista, 6544	2220-4156	
Alex Felipe	123.456.789-01	2012-04-30	Material de construcao	1576.40	987.654.321-09	2012-04-30	Material de construcao	1576.40	
			Rua Vergueiro, 3185	5571-2751			Rua Vergueiro, 3185	5571-2751	
João da Silva	987.654.321-09	2012-04-30	Material de construcao	1576.40	123.456.789-01	2012-04-30	Material de construcao	1576.40	
			Av. Paulista, 6544	2220-4156			Av. Paulista, 6544	2220-4156	
Alex Felipe	123.456.789-01	2012-12-15	Pizza pra familia	163.45	987.654.321-09	2012-12-15	Pizza pra familia	163.45	
			Rua Vergueiro, 3185	5571-2751			Rua Vergueiro, 3185	5571-2751	
João da Silva	987.654.321-09	2012-12-15	Pizza pra familia	163.45	123.456.789-01	2012-12-15	Pizza pra familia	163.45	
			Av. Paulista, 6544	2220-4156			Av. Paulista, 6544	2220-4156	
Alex Felipe	123.456.789-01	2013-01-23	Sala de estar	4780.00	987.654.321-09	2013-01-23	Sala de estar	4780.00	
			Rua Vergueiro, 3185	5571-2751			Rua Vergueiro, 3185	5571-2751	
João da Silva	987.654.321-09	2013-01-23	Sala de estar	4780.00	123.456.789-01	2013-01-23	Sala de estar	4780.00	
			Av. Paulista, 6544	2220-4156			Av. Paulista, 6544	2220-4156	
Alex Felipe	123.456.789-01	2013-03-03	Quartos	392.15	987.654.321-09	2013-03-03	Quartos	392.15	
			Rua Vergueiro, 3185	5571-2751			Rua Vergueiro, 3185	5571-2751	
João da Silva	987.654.321-09	2013-03-03	Quartos	392.15	123.456.789-01	2013-03-03	Quartos	392.15	
			Av. Paulista, 6544	2220-4156			Av. Paulista, 6544	2220-4156	
Alex Felipe	123.456.789-01	2013-03-21	Copa	402.90	987.654.321-09	2013-03-21	Copa	402.90	
			Rua Vergueiro, 3185	5571-2751			Rua Vergueiro, 3185	5571-2751	
João da Silva	987.654.321-09	2013-03-21	Copa	402.90	123.456.789-01	2013-03-21	Copa	402.90	
			Av. Paulista, 6544	2220-4156			Av. Paulista, 6544	2220-4156	
Alex Felipe	123.456.789-01	2013-04-12	Lanchonete	54.98	987.654.321-09	2013-04-12	Lanchonete	54.98	
			Rua Vergueiro, 3185	5571-2751			Rua Vergueiro, 3185	5571-2751	
João da Silva	987.654.321-09	2013-04-12	Lanchonete	54.98	123.456.789-01	2013-04-12	Lanchonete	54.98	
			Av. Paulista, 6544	2220-4156			Av. Paulista, 6544	2220-4156	
Alex Felipe	123.456.789-01	2013-05-23	Lanchonete	12.34	987.654.321-09	2013-05-23	Lanchonete	12.34	
			Rua Vergueiro, 3185	5571-2751			Rua Vergueiro, 3185	5571-2751	
João da Silva	987.654.321-09	2013-05-23	Lanchonete	12.34	123.456.789-01	2013-05-23	Lanchonete	12.34	
			Av. Paulista, 6544	2220-4156			Av. Paulista, 6544	2220-4156	
Alex Felipe	123.456.789-01	2013-12-04	Lanchonete	78.65	987.654.321-09	2013-12-04	Lanchonete	78.65	
			Rua Vergueiro, 3185	5571-2751			Rua Vergueiro, 3185	5571-2751	
João da Silva	987.654.321-09	2013-12-04	Lanchonete	78.65	123.456.789-01	2013-12-04	Lanchonete	78.65	
			Av. Paulista, 6544	2220-4156			Av. Paulista, 6544	2220-4156	
Alex Felipe	123.456.789-01	2013-01-06	Sorvete no parque	12.39	987.654.321-09	2013-01-06	Sorvete no parque	12.39	
			Rua Vergueiro, 3185	5571-2751			Rua Vergueiro, 3185	5571-2751	
João da Silva	987.654.321-09	2013-01-06	Sorvete no parque	12.39	123.456.789-01	2013-01-06	Sorvete no parque	12.39	
			Av. Paulista, 6544	2220-4156			Av. Paulista, 6544	2220-4156	
Alex Felipe	123.456.789-01	2013-07-09	Hopi Hari	98.12	987.654.321-09	2013-07-09	Hopi Hari	98.12	
			Rua Vergueiro, 3185	5571-2751			Rua Vergueiro, 3185	5571-2751	





33	115.90	2014-06-12	Dia dos namorados		0	2	1	Alex F
elipe	Rua Vergueiro, 3185	5571-2751						
33	115.90	2014-06-12	Dia dos namorados		0	2	2	João d
a Silva	Av. Paulista, 6544	2220-4156						
34	98.00	2014-10-12	Dia das crianças		0	2	1	Alex F
elipe	Rua Vergueiro, 3185	5571-2751						
34	98.00	2014-10-12	Dia das crianças		0	2	2	João d
a Silva	Av. Paulista, 6544	2220-4156						
35	253.70	2014-12-20	Natal - presentes		0	2	1	Alex F
elipe	Rua Vergueiro, 3185	5571-2751						
35	253.70	2014-12-20	Natal - presentes		0	2	2	João d
a Silva	Av. Paulista, 6544	2220-4156						
36	370.15	2014-12-25	Compras de natal		0	2	1	Alex F
elipe	Rua Vergueiro, 3185	5571-2751						
36	370.15	2014-12-25	Compras de natal		0	2	2	João d
a Silva	Av. Paulista, 6544	2220-4156						
37	32.09	2015-07-02	Lanchonete		1	2	1	Alex F
elipe	Rua Vergueiro, 3185	5571-2751						
37	32.09	2015-07-02	Lanchonete		1	2	2	João d
a Silva	Av. Paulista, 6544	2220-4156						
38	954.12	2015-11-03	Show da Ivete Sangalo		1	2	1	Alex F
elipe	Rua Vergueiro, 3185	5571-2751						
38	954.12	2015-11-03	Show da Ivete Sangalo		1	2	2	João d
a Silva	Av. Paulista, 6544	2220-4156						
39	98.70	2015-02-07	Lanchonete		1	2	1	Alex F
elipe	Rua Vergueiro, 3185	5571-2751						
39	98.70	2015-02-07	Lanchonete		1	2	2	João d
a Silva	Av. Paulista, 6544	2220-4156						
40	213.50	2015-09-25	Roupas		0	2	1	Alex F
elipe	Rua Vergueiro, 3185	5571-2751						
40	213.50	2015-09-25	Roupas		0	2	2	João d
a Silva	Av. Paulista, 6544	2220-4156						
41	1245.20	2015-10-17	Roupas		0	2	1	Alex F
elipe	Rua Vergueiro, 3185	5571-2751						
41	1245.20	2015-10-17	Roupas		0	2	2	João d
a Silva	Av. Paulista, 6544	2220-4156						
42	23.78	2015-12-18	Lanchonete do Zé		1	2	1	Alex F
elipe	Rua Vergueiro, 3185	5571-2751						
42	23.78	2015-12-18	Lanchonete do Zé		1	2	2	João d
a Silva	Av. Paulista, 6544	2220-4156						
43	576.12	2015-09-13	Sapatos		1	2	1	Alex F
elipe	Rua Vergueiro, 3185	5571-2751						
43	576.12	2015-09-13	Sapatos		1	2	2	João d
a Silva	Av. Paulista, 6544	2220-4156						
44	12.34	2015-07-19	Canetas		0	2	1	Alex F
elipe	Rua Vergueiro, 3185	5571-2751						
44	12.34	2015-07-19	Canetas		0	2	2	João d
a Silva	Av. Paulista, 6544	2220-4156						
45	87.43	2015-05-10	Gravata		0	2	1	Alex F
elipe	Rua Vergueiro, 3185	5571-2751						
45	87.43	2015-05-10	Gravata		0	2	2	João d
a Silva	Av. Paulista, 6544	2220-4156						
46	887.66	2015-02-02	Presente para o filhao		1	2	1	Alex F
elipe	Rua Vergueiro, 3185	5571-2751						
46	887.66	2015-02-02	Presente para o filhao		1	2	2	João d
a Silva	Av. Paulista, 6544	2220-4156						
48	150.00	2016-01-05	Compra de teste		0	2	1	Alex F
elipe	Rua Vergueiro, 3185	5571-2751						
48	150.00	2016-01-05	Compra de teste		0	2	2	João d
a Silva	Av. Paulista, 6544	2220-4156						

+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

92 rows in set (0,00 sec)

Opa! Observe que essa *query* está um pouco estranha, pois ela devolveu 92 linhas, sendo que temos apenas 46 compras registradas! Duplicou os nossos registros... Esse problema aconteceu, pois o MySQL não sabe como associar a tabela compras e a tabela compradores. Precisamos informar ao MySQL por meio de qual coluna ele fará essa associação. Perceba que a coluna que contém a chave (*primary key*) do comprador é justamente a `id_compradores`, chamamos esse tipo de coluna de *FOREIGN KEY* ou chave estrangeira. Para juntarmos a chave estrangeira com a chave primária de uma tabela, utilizamos a instrução `JOIN`, informando a tabela e a chave que queremos associar:

```
mysql> SELECT * FROM compras JOIN compradores on compras.id_compradores = compradores.id;
```

id	valor	data	observacoes	recebida	id_compradores	id	nome
	endereço		telefone				
1	20.00	2016-01-05	Lanchonete	1	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
2	15.00	2016-01-06	Lanchonete	1	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
3	915.50	2016-01-06	Guarda-roupa	0	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
4	949.99	2016-01-10	Smartphone	0	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
5	200.00	2012-02-19	Material escolar	1	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
6	3500.00	2012-05-21	Televisao	0	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
7	1576.40	2012-04-30	Material de construcao	1	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
8	163.45	2012-12-15	Pizza pra familia	1	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
9	4780.00	2013-01-23	Sala de estar	1	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
10	392.15	2013-03-03	Quartos	1	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
12	402.90	2013-03-21	Copa	1	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
13	54.98	2013-04-12	Lanchonete	0	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
14	12.34	2013-05-23	Lanchonete	0	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
15	78.65	2013-12-04	Lanchonete	0	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
16	12.39	2013-01-06	Sorvete no parque	0	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
17	98.12	2013-07-09	Hopi Hari	1	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
18	2498.00	2013-01-12	Compras de janeiro	1	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
19	3212.40	2013-11-13	Compras do mes	1	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
20	223.09	2013-12-17	Compras de natal	1	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
21	768.90	2013-01-16	Festa	1	1	1	Alex F
elipe	Rua Vergueiro, 3185		5571-2751				
22	827.50	2014-01-09	Festa	1	2	2	João d
a Silva	Av. Paulista, 6544		2220-4156				
23	12.00	2014-02-19	Salgado no aeroporto	1	2	2	João d
a Silva	Av. Paulista, 6544		2220-4156				



nossa tabela compras. Observe esse INSERT :

```
mysql> INSERT INTO compras (valor, data, observacoes, id_compradores)
VALUES (1500, '2016-01-05', 'Playstation 4', 100);
Query OK, 1 row affected (0,00 sec)
```

Vamos verificar a nossa tabela compras:

```
mysql> SELECT * FROM compras WHERE id_compradores = 100;
+-----+-----+-----+-----+-----+-----+
| id | valor  | data      | observacoes | recebida | id_compradores |
+-----+-----+-----+-----+-----+-----+
| 49 | 1500.00 | 2016-01-05 | Playstation 4 | 0 | 100 |
+-----+-----+-----+-----+-----+-----+
1 row in set (0,00 sec)
```

Note que não existe o comprador com id 100, mas, mesmo assim, conseguimos adicioná-lo na nossa tabela de compras. Precisamos garantir a integridade dos nossos dados, informando ao MySQL que a coluna id\_compradores da tabela compras é uma *FOREIGN KEY* da tabela compradores, ou seja, só poderemos adicionar um id apenas se estiver registrado na tabela compradores. Antes de adicionarmos a *FOREIGN KEY*, precisamos excluir o registro com o id\_compradores = 100, pois não é possível adicionar uma *FOREIGN KEY* com dados que não exista na tabela que iremos referenciar.

```
mysql> DELETE FROM compras WHERE id_compradores = 100;
Query OK, 1 row affected (0,01 sec)
```

Quando adicionamos uma *FOREIGN KEY* em uma tabela, estamos adicionando uma Constraints, então precisaremos alterar a estrutura da tabela compras utilizando a instrução ALTER TABLE :

```
mysql> ALTER TABLE compras ADD CONSTRAINT fk_compradores FOREIGN KEY (id_compradores)
REFERENCES compradores (id);
Query OK, 46 rows affected (0,04 sec)
Records: 46 Duplicates: 0 Warnings: 0
```

Se tentarmos adicionar a compra anterior novamente:

```
mysql> INSERT INTO compras (valor, data, observacoes, id_compradores)
VALUES (1500, '2016-01-05', 'Playstation 4', 100);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails (`ControleDeGastos`.`compras`, CONSTRAINT `fk_compradores` FOREIGN KEY (`id_compradores`) REFERENCES `compradores` (`id`))
```

Agora o nosso banco de dados não permite a inserção de registros com compradores inexistentes! Se tentarmos adicionar essa mesma compra com um comprador existente:

```
INSERT INTO compras (valor, data, observacoes, id_compradores)
VALUES (1500, '2016-01-05', 'Playstation 4', 1);
Query OK, 1 row affected (0,00 sec)
```

se verificarmos essa compra:

```
mysql> SELECT * FROM compras WHERE observacoes = 'Playstation 4';
+-----+-----+-----+-----+-----+-----+
| id | valor  | data      | observacoes | recebida | id_compradores |
+-----+-----+-----+-----+-----+-----+
| 51 | 1500.00 | 2016-01-05 | Playstation 4 | 0 | 1 |
+-----+-----+-----+-----+-----+-----+
```

```
+-----+-----+-----+-----+-----+-----+
1 row in set (0,00 sec)
```

## 6.4 DETERMINANDO VALORES FIXOS NA TABELA

Conseguimos deixar a nossa base de dados bem robusta, restringindo as nossas colunas para não permitir valores nulos e não aceitar a inserção de compradores inexistentes, mas agora surgiu uma nova necessidade que precisa ser implementada, precisamos informar também qual é a forma de pagamento que foi realizada na compra, por exemplo, existem 2 formas de pagamento, boleto e crédito. Como poderíamos implementar essa nova informação na nossa tabela atualmente? Criamos uma nova coluna com o tipo `VARCHAR` e corremos o risco de inserir uma forma de pagamento inválida? Não parece uma boa solução... Que tal criarmos uma nova tabela chamada `id_forma_de_pagto` e fazermos uma *FOREIGN KEY*? Aparentemente é uma boa solução, porém iremos criar uma nova tabela para resolver um problema bem pequeno? Lembre-se que a cada *FOREIGN KEY* mais `JOIN` s as nossas *queries* terão... No MySQL, existe o tipo de dado `ENUM` que permite que informemos quais serão os dados que ele pode aceitar. Vamos adicionar o `ENUM` na nossa tabela de compras:

```
mysql> ALTER TABLE compras ADD COLUMN forma_pagto ENUM('BOLETO', 'CREDITO');
Query OK, 0 rows affected (0,04 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

Vamos tentar adicionar uma compra:

```
mysql> INSERT INTO compras (valor, data, observacoes, id_compradores, forma_pagto)
VALUES (400, '2016-01-06', 'SSD 128GB', 1, 'BOLETO');
Query OK, 1 row affected (0,00 sec)
```

Retornando essa nova compra:

```
mysql> SELECT * FROM compras WHERE observacoes = 'SSD 128GB';
+-----+-----+-----+-----+-----+-----+
| id | valor | data      | observacoes | recebida | id_compradores | forma_pagto |
+-----+-----+-----+-----+-----+-----+
| 52 | 400.00 | 2016-01-06 | SSD 128GB   | 0        | 1              | BOLETO      |
+-----+-----+-----+-----+-----+-----+
1 row in set (0,00 sec)
```

Mas e se tentarmos adicionar uma nova compra, porém com a forma de pagamento em dinheiro?

```
mysql> INSERT INTO compras (valor, data, observacoes, id_compradores, forma_pagto)
VALUES (80, '2016-01-07', 'Bola de futebol', 2, 'DINHEIRO');
Query OK, 1 row affected, 1 warning (0,00 sec)
```

Vamos verificar como ficou na nossa tabela de compras:

```
mysql> SELECT * FROM compras WHERE observacoes = 'Bola de futebol';
+-----+-----+-----+-----+-----+-----+
| id | valor | data      | observacoes | recebida | id_compradores | forma_pagto |
+-----+-----+-----+-----+-----+-----+
| 53 | 80.00 | 2016-01-07 | Bola de futebol | 0        | 2              |             |
+-----+-----+-----+-----+-----+-----+
1 row in set (0,00 sec)
```

## 6.5 SERVER SQL MODES

O MySQL impediu que fosse adicionado um valor diferente de "BOLETO" ou "CREDITO", mas o que realmente precisamos é que ele simplesmente não deixe adicionar uma compra que não tenha pelo menos uma dessas formas de pagamento. Além das configurações das tabelas, podemos também configurar o próprio servidor do MySQL. O servidor do MySQL opera em diferentes *SQL modes* e dentre esses modos, existe o *strict mode* que tem a finalidade de tratar valores inválidos que configuramos em nossas tabelas para instruções de `INSERT` e `UPDATE`, como por exemplo, o nosso `ENUM`. Para habilitar o *strict mode* precisamos alterar o *SQL mode* da **nossa sessão**. Nesse caso usaremos o modo "STRICT\_ALL\_TABLES":

```
mysql> SET SESSION sql_mode = 'STRICT_ALL_TABLES';
Query OK, 0 rows affected (0,00 sec)
```

Se quisermos verificar se o modo foi modificado podemos retornar esse valor por meio da instrução `SELECT` :

```
mysql> SELECT @@SESSION.sql_mode;
+-----+
| @@SESSION.sql_mode |
+-----+
| STRICT_ALL_TABLES  |
+-----+
1 row in set (0,00 sec)
```

Agora que configuramos o *SQL mode* do MySQL para impedir a inserção de valores inválidos, vamos apagar o último registro que foi inserido com valor inválido e tentar adicioná-lo novamente:

```
mysql> DELETE FROM compras WHERE observacoes = 'BOLA DE FUTEBOL';
Query OK, 1 row affected (0,00 sec)
```

Testando novamente a mesma inserção:

```
mysql> INSERT INTO compras (valor, data, observacoes, id_compradores, forma_pagto)
-> VALUES (80, '2016-01-07', 'BOLA DE FUTEBOL', 2, 'DINHEIRO');
ERROR 1265 (01000): Data truncated for column 'forma_pagto' at row 1
```

Perceba que agora o MySQL impediu que a linha fosse inserida, pois o valor não é válido para a coluna `forma_pagto`.

O *SQL mode* é uma configuração do servidor e nós alteramos apenas a *sessão que estávamos logado*, o que aconteceria se caso saímos da sessão que configuramos e entrássemos em uma nova? O MySQL adotaria o *SQL mode* padrão já configurado! Ou seja, teríamos que alterar novamente para o *strict mode*. Mas além da sessão, podemos fazer a configuração global do *SQL mode*.

```
mysql> SET GLOBAL sql_mode = 'STRICT_ALL_TABLES';
Query OK, 0 rows affected (0,00 sec)
```

Se verificarmos a configuração global para o *SQL mode*:

```
mysql> SELECT @@GLOBAL.sql_mode;
+-----+
```

```
| @@GLOBAL.sql_mode |
+-----+
| STRICT_ALL_TABLES |
+-----+
1 row in set (0,00 sec)
```

Agora, todas as vezes que entrarmos no MySQL, será adotado o *strict mode*.

O `ENUM` é uma boa solução quando queremos restringir valores específicos e já esperados em um coluna, porém não faz parte do padrão *ANSI* que é o padrão para a escrita de instruções `SQL`, ou seja, é um recurso exclusivo do MySQL e cada banco de dados possui a sua própria implementação para essa mesma funcionalidade.

## 6.6 RESUMINDO

Nesse capítulo aprendemos a fazer uma relação entre duas tabelas utilizando *FOREIGN KEYS*. Vimos também que para fazermos *queries* com duas tabelas diferentes utilizamos as chaves estrangeiras por meio da instrução `JOIN` que informa ao MySQL quais serão os critérios para associar as tabelas distintas. E sempre precisamos lembrar que, quando estamos lidando com *FOREIGN KEY*, precisamos criar uma Constraint para garantir que todas as chaves estrangeiras precisa existir na tabela que fazemos referência. Além disso, vimos também que quando queremos adicionar uma coluna nova para que aceite apenas determinados valores já esperado por nós, como é o caso da forma de pagamento, podemos utilizar o `ENUM` do MySQL como um solução, porém é importante lembrar que é uma solução que não faz parte do padrão *ANSI*, ou seja, cada banco de dados possui sua própria implementação. Vamos para os exercícios?

Vamos para os exercícios?

## EXERCÍCIOS

1. Crie a tabela `compradores` com `id`, `nome`, `endereco` e `telefone`.
2. Insira os compradores, Guilherme e João da Silva.
3. Adicione a coluna `id_compradores` na tabela `compras` e defina a chave estrangeira (*FOREIGN KEY*) referenciando o `id` da tabela `compradores`.
4. Atualize a tabela `compras` e insira o `id` dos compradores na coluna `id_compradores`.
5. Exiba o `NOME` do comprador e o `VALOR` de todas as compras feitas antes de 09/08/2014.
6. Exiba todas as compras do comprador que possui ID igual a 2.
7. Exiba todas as compras (mas sem os dados do comprador), cujo comprador tenha nome que começa com 'GUILHERME'.



8. Exiba o nome do comprador e a soma de todas as suas compras.
9. A tabela `compras` foi alterada para conter uma *FOREIGN KEY* referenciando a coluna `id` da tabela `compradores`. O objetivo é deixar claro para o banco de dados que `compras.id_compradores` está de alguma forma relacionado com a tabela `compradores` através da coluna `compradores.id`. Mesmo sem criar a *FOREIGN KEY* é possível relacionar tabelas através do comando `JOIN`.
10. Qual a vantagem em utilizar a *FOREIGN KEY* ?
11. Crie uma coluna chamada "forma\_pagto" do tipo `ENUM` e defina os valores: 'BOLETO' e 'CREDITO'.
12. Ative o *strict mode* na sessão que está utilizando para impossibilitar valores inválidos. Utilize o modo "STRICT\_ALL\_TABLES". E verifique se o *SQL mode* foi alterado fazendo um `SELECT` na sessão.
13. Tente inserir uma compra com forma de pagamento diferente de 'BOLETO' ou 'CREDITO', por exemplo, 'DINHEIRO' e verifique se o MySQL recusa a inserção.
14. Adicione as formas de pagamento para todas as compras por meio da instrução `UPDATE`.
15. Faça a configuração global do MySQL para que ele sempre entre no *strict mode*.

# ALUNOS SEM MATRÍCULA E O EXISTS

Para a segunda parte de nosso curso utilizaremos um conjunto de dados de um sistema de ensino online como exemplo. Não se preocupe, pois disponibilizaremos o arquivo para que você baixe e execute o script com todas as tabelas.

Continuaremos usando o terminal do MySQL durante o curso, porém, se você prefere uma outra interface, como por exemplo o *MySQL Workbench*, fique a vontade e usar o que for melhor para você.

Abra o terminal do MySQL com o comando `mysql -uroot -p` e crie a base de dados escola.

```
create database escola
```

```
Query OK, 1 row affected (0,01 sec)
```

Agora que criamos a nossa base de dados, podemos importar o arquivo `.sql` já existente. Saia do terminal e execute o arquivo na base de dados escola.

```
mysql -u root -p escola < escola.sql
```

Com o arquivo importado, podemos abrir novamente o MySQL, porém selecione a base de dados escola.

```
mysql -u root -p escola
```

Para verificar todas as tabelas da nossa base de dados podemos utilizar a instrução `SHOW TABLES` do MySQL:

```
SHOW TABLES;
```

```
+-----+
| Tables_in_escola |
+-----+
| aluno             |
| curso             |
| exercicio         |
| matricula         |
| nota              |
| resposta          |
| secao             |
+-----+
7 rows in set (0,00 sec)
```

Sabemos quais são as tabelas, porém precisamos saber mais sobre a estrutura dessas tabelas, então vamos utilizar a instrução `DESC`. Vamos verificar primeiro a tabela aluno:

```
DESC aluno;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
nome	varchar(255)	NO			
email	varchar(255)	NO			

Perceba que é uma tabela bem simples, onde serão armazenadas apenas as informações dos alunos. Vamos verificar a tabela curso:

```
DESC curso;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
nome	varchar(255)	NO			

Da mesma forma que a tabela aluno, a tabela curso armazenada apenas as informações dos cursos. Agora vamos verificar a tabela matricula:

```
DESC matricula;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
aluno_id	int(11)	NO		NULL	
curso_id	int(11)	NO		NULL	
data	datetime	NO		NULL	
tipo	varchar(20)	NO			

Conseguimos achar a primeira associação das tabelas, ou seja, a coluna `aluno_id` referencia a tabela `aluno` e a coluna `curso_id` referencia a tabela `curso`. Então vamos verificar quais são todos os cursos de um aluno. Para uma melhor compreensão de como é o resultado esperado dessa *query*, veja a planilha a seguir:

aluno	curso
Alex	SQL
Alex	Android
Guilherme	Java
Gabriel	C#
Marcos	HTML

Figura 7.1: Planilha exemplo

Como podemos ver, nós temos a lista de alunos e seus respectivos cursos. Então vamos começar a retornar todos os alunos que possuem uma matrícula, ou seja, vamos fazer um `JOIN` entre a tabela `aluno` e `matricula`:

```
SELECT a.nome FROM aluno a
JOIN matricula m ON m.aluno_id = a.id;
```

```
+-----+
| nome   |
+-----+
| João da Silva |
| Frederico José |
| Alberto Santos |
| Renata Alonso |
| Paulo José |
| Manoel Santos |
| Renata Ferreira |
| Paula Soares |
| Renata Alonso |
| Manoel Santos |
| João da Silva |
| Frederico José |
| Alberto Santos |
| Frederico José |
+-----+
```

Perceba que utilizamos `a.nome` , `m.aluno_id` e `a.id` , afinal, o que significa? Quando escrevemos `aluno a` , significa que estamos "apelidando" a tabela `aluno` , ou seja, todas as vezes que utilizarmos `a.alguma_coisa` , estaremos pegando alguma coluna da tabela `aluno` ! Em SQL, esses "apelidos" são conhecidos como *Alias*.

Os alunos foram retornados. Então agora vamos juntar a tabela `curso` e `aluno` com a tabela `matricula` ao mesmo tempo, porém, dessa vez vamos retornar o nome do aluno e o nome do curso:

```
SELECT a.nome, c.nome FROM aluno a
JOIN matricula m ON m.aluno_id = a.id
JOIN curso c ON m.curso_id = c.id;
```

```
+-----+-----+
| nome   | nome   |
+-----+-----+
| João da Silva | SQL e banco de dados |
| Frederico José | SQL e banco de dados |
| Alberto Santos | Scrum e métodos ágeis |
| Renata Alonso | C# e orientação a objetos |
| Paulo José | SQL e banco de dados |
| Manoel Santos | Scrum e métodos ágeis |
| Renata Ferreira | Desenvolvimento web com VRaptor |
| Paula Soares | Desenvolvimento mobile com Android |
| Renata Alonso | Desenvolvimento mobile com Android |
| Manoel Santos | SQL e banco de dados |
| João da Silva | C# e orientação a objetos |
| Frederico José | C# e orientação a objetos |
| Alberto Santos | C# e orientação a objetos |
| Frederico José | Desenvolvimento web com VRaptor |
+-----+-----+
```

Vamos verificar quantos alunos nós temos na nossa base de dados:

```
SELECT COUNT(*) FROM aluno;
```

```
+-----+
| COUNT(*) |
+-----+
```

```
+-----+
|      16 |
+-----+
```

## 7.1 SUBQUERIES

Observe que foram retornadas 14 linhas quando buscamos todos os alunos e seus cursos, ou seja, existem alunos que não tem matrícula! Como podemos verificar quais são os alunos que não estão matriculados? No MySQL, podemos utilizar a função `EXISTS()` para verificar se existe algum registro de acordo com uma determinada *query*:

```
SELECT a.nome FROM aluno a
WHERE EXISTS (SELECT m.id FROM matricula m WHERE m.aluno_id = a.id);
```

```
+-----+
| nome      |
+-----+
| João da Silva |
| Frederico José |
| Alberto Santos |
| Renata Alonso |
| Paulo José   |
| Manoel Santos |
| Renata Ferreira |
| Paula Soares |
+-----+
```

Repare que escrevemos uma *query* dentro de uma função, quando fazemos esse tipo de *query* chamamos de *subquery*. Mas o que aconteceu exatamente nessa *query*? Quando utilizamos o `EXISTS()` indicamos que queremos o retorno de todos os alunos nomes dos alunos ( `a.nome` ) que estão na tabela `aluno` , porém, queremos apenas se existir uma matrícula para esse aluno `EXISTS(SELECT m.id FROM matricula m WHERE m.aluno_id = a.id)` .

Perceba que novamente estamos retornando os alunos matriculados sendo que precisamos dos alunos que não estão matriculados. Nesse caso, podemos utilizar a instrução `NOT` para fazer a negação, ou seja, para retornar os alunos que **não** possuem matrícula:

```
SELECT a.nome FROM aluno a
WHERE NOT EXISTS (SELECT m.id FROM matricula m WHERE m.aluno_id = a.id);
```

```
+-----+
| nome      |
+-----+
| Paulo da Silva |
| Carlos Cunha  |
| Jose da Silva  |
| Danilo Cunha   |
| Zilmira José   |
| Cristaldo Santos |
| Osmir Ferreira |
| Claudio Soares |
+-----+
```

Conseguimos achar todos alunos que fazem parte do sistema e que não possuem uma matrícula, algo

que não é esperado... Provavelmente esses alunos haviam cancelado a matrícula mas ainda sim, existia o cadastro deles, sabendo dessas informações temos a capacidade de criar relatórios com informações relevantes para, por exemplo, o setor comercial dessa instituição entrar em contato com os alunos não matriculados e tentar efetuar uma venda.

A instituição subiu alguns exercícios e precisa saber quais desses exercícios não foram respondidos. Vamos novamente observar o resultado que se espera utilizando uma planilha como exemplo:

id	não respondidos
1	Exercícios 1
2	Exercícios 2
5	x

Figura 7.2: Planilha exemplo

Como havíamos visto anteriormente, existem as tabelas, `exercicio` e `resposta`, vamos uma olhada na tabela `exercicio`:

```
DESC exercicio;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
secao_id	int(11)	NO		NULL	
pergunta	varchar(255)	NO		NULL	
resposta_oficial	varchar(255)	NO		NULL	

Agora a tabela `respostas`:

```
DESC resposta;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
exercicio_id	int(11)	YES		NULL	
aluno_id	int(11)	YES		NULL	
resposta_dada	varchar(255)	YES		NULL	

Novamente encontramos uma outra associação, porém agora é entre `exercicio` e `resposta`. Então vamos pegar todos os exercícios que não foram respondidos utilizando novamente o `NOT EXISTS`:

```
SELECT * FROM exercicio e
WHERE NOT EXISTS (SELECT r.id FROM resposta r WHERE r.exercicio_id = e.id);
```

id	secao_id	pergunta	resposta_oficial
8	4	como funciona?	insert into (coluna1, coluna2) values (v1, v2)

```

| 9 |      5 | Como funciona a web?      | requisicao e resposta
|
| 10 |     5 | Que linguagens posso ajudar? | varias, java, php, c#, etc
|
| 11 |     6 | O que eh MVC?                | model view controller
|
| 12 |     6 | Frameworks que usam?         | vraptor, spring mvc, struts, etc
|
| 14 |     8 | O que é um interceptor?      | eh como se fosse um filtro que eh executado antes
|
| 15 |     8 | quando usar?                 | tratamento de excecoes, conexao com o banco de dados
|
+-----+-----+-----+-----+-----+-----+
-+

```

Se quisermos retornar da mesma forma que fizemos no exemplo da planilha, basta informar os campos desejados:

```

SELECT e.id, e.pergunta FROM exercicio e
WHERE NOT EXISTS (SELECT r.id FROM resposta r WHERE r.exercicio_id = e.id);

```

```

+-----+-----+-----+-----+-----+-----+
| id | pergunta                                     |
+-----+-----+-----+-----+-----+
| 8 | como funciona?                             |
| 9 | Como funciona a web?                         |
| 10 | Que linguagens posso ajudar?                 |
| 11 | O que eh MVC?                               |
| 12 | Frameworks que usam?                         |
| 14 | O que é um interceptor?                     |
| 15 | quando usar?                                |
+-----+-----+-----+-----+-----+

```

O pessoal do comercial da instituição, informou que existem alguns cursos que não tem nenhuma matrícula. Vamos verificar como é esperado desse resultado pela nossa planilha:

curso
C#
SQL
Java

Figura 7.3: Planilha exemplo

Da mesma forma que retornamos todos os exercícios que não tinha respostas, podemos retornar todos os cursos que não possuem matrícula:

```

SELECT c.nome FROM curso c
WHERE NOT EXISTS(SELECT m.id FROM matricula m WHERE m.curso_id = c.id);

```

```

+-----+-----+-----+-----+-----+-----+
| nome                                     |
+-----+-----+-----+-----+-----+
| Java e orientação a objetos             |
| Desenvolvimento mobile com iOS          |
| Ruby on Rails                           |
| PHP e MySQL                             |
+-----+-----+-----+-----+-----+

```

Veja que *queries* muito parecidas podem resolver problemas diferentes!

A instituição informou que tiveram vários exercícios que não foram respondidos pelos alunos nos cursos que foram realizados recentemente. Vamos verificar quem foram esses alunos, para verificarmos o motivo de não ter respondido, se foi um problema no sistema ou na base de dados... Novamente vamos verificar o que se espera desse resultado numa planilha:

alunos	curso
Paul	C#
Pedro	Java
João	C#

Figura 7.4: Planilha exemplo

Vamos tentar fazer essa *query*. Começaremos retornando o aluno juntando a tabela `aluno` com a tabela `matricula`.

```
SELECT a.nome FROM aluno a
JOIN matricula m ON m.aluno_id = a.id
```

Agora vamos juntar também a tabela `curso` e retornar os cursos também:

```
SELECT a.nome, c.nome FROM aluno a
JOIN matricula m ON m.aluno_id = a.id
JOIN curso c ON m.curso_id = c.id
```

Vamos testar e verificar como está a nossa *query* atualmente:

```
+-----+-----+
| nome          | nome          |
+-----+-----+
| João da Silva | SQL e banco de dados |
| Frederico José | SQL e banco de dados |
| Alberto Santos | Scrum e métodos ágeis |
| Renata Alonso  | C# e orientação a objetos |
| Paulo José    | SQL e banco de dados |
| Manoel Santos  | Scrum e métodos ágeis |
| Renata Ferreira | Desenvolvimento web com VRaptor |
| Paula Soares   | Desenvolvimento mobile com Android |
| Renata Alonso  | Desenvolvimento mobile com Android |
| Manoel Santos  | SQL e banco de dados |
| João da Silva  | C# e orientação a objetos |
| Frederico José | C# e orientação a objetos |
| Alberto Santos | C# e orientação a objetos |
| Frederico José | Desenvolvimento web com VRaptor |
+-----+-----+
```

Aparentemente está tudo certo, porém ainda precisamos informar que queremos apenas os alunos que não responderam os exercícios desses de algum desses cursos. Então adicionaremos agora o `NOT EXISTS()` :

```
SELECT a.nome, c.nome FROM aluno a
JOIN matricula m ON m.aluno_id = a.id
JOIN curso c ON m.curso_id = c.id
```



```
WHERE NOT EXISTS(SELECT r.aluno_id
FROM resposta r WHERE r.aluno_id = a.id);
```

nome	nome
Paulo José	SQL e banco de dados
Manoel Santos	Scrum e métodos ágeis
Renata Ferreira	Desenvolvimento web com VRaptor
Paula Soares	Desenvolvimento mobile com Android
Manoel Santos	SQL e banco de dados

Há uma regra no sistema em que não pode permitir que alunos que não estejam matriculados respondam os exercícios, ou seja, não pode existir uma resposta na tabela `resposta` com um `id` de um aluno (`aluno_id`) que não esteja matriculado. Vamos primeiro verificar todos os alunos **matriculados** que responderam os exercícios:

```
SELECT r.id, a.nome FROM aluno a
JOIN resposta r ON r.aluno_id = a.id
WHERE EXISTS(SELECT m.aluno_id FROM matricula m
WHERE m.aluno_id = a.id);
```

id	nome
1	João da Silva
2	João da Silva
3	João da Silva
4	João da Silva
5	João da Silva
6	João da Silva
7	João da Silva
8	Frederico José
9	Frederico José
10	Frederico José
11	Frederico José
12	Alberto Santos
13	Alberto Santos
14	Alberto Santos
15	Alberto Santos
16	Alberto Santos
17	Alberto Santos
18	Alberto Santos
19	Alberto Santos
20	Alberto Santos
21	Renata Alonso
22	Renata Alonso
23	Renata Alonso
24	Renata Alonso
25	Renata Alonso
26	Renata Alonso
27	Renata Alonso

Observe que repetiu alguns alunos, pois um aluno respondeu mais de uma questão. Vamos verificar agora os que responderam, porém **não estão matriculados**:

```
SELECT r.id, a.nome FROM aluno a
JOIN resposta r ON r.aluno_id = a.id
```

```
WHERE NOT EXISTS(SELECT m.aluno_id FROM matricula m
WHERE m.aluno_id = a.id);
```

Empty set (0,00 sec)

O resultado saiu conforme o esperado, isso significa, que não existem respostas de alunos que não possuem matrícula!

## 7.2 RESUMINDO

Nesse capítulo aprendemos a criar *queries* capazes de retornar valores caso exista, ou não, uma associação entre duas tabelas, como por exemplo, se um aluno está matriculado ou se um aluno respondeu algum exercício por meio da função `EXISTS()`. Aprendemos também o que são *subqueries* como no exemplo em que passamos uma *query* como parâmetro para a função `EXISTS()` resolver diversos problemas, além disso, analisamos que várias *queries* "similadres" tem a capacidade de resolver problemas distintos utilizando o `EXISTS()`. Vamos para os exercícios?

## EXERCÍCIOS

1. Baixe o schema do banco de dados [aqui](#)

Crie o banco de dados `escola` :

```
create database escola
```

Importe-o no seu Mysql com o seguinte comando, direto no terminal:

```
mysql -u root -p escola < escola.sql
```

Faça um `SELECT` qualquer para garantir que os dados estão lá.

DICA: Salve o arquivo `escola.sql` em um lugar de fácil acesso pelo terminal. Você deve rodar o comando para importar o schema no mesmo lugar onde estar o arquivo `escola.sql`. Por exemplo, salve o arquivo SQL na pasta `\aula-mysql`. Depois abra um terminal e entre nessa pasta:

```
cd \aula-mysql
mysql -u root -p escola < escola.sql
```

No Windows você pode usar o comando `dir` é para listar os arquivos da pasta atual na linha de comando. Ou seja, ao executar `dir` na pasta `aula-mysql` você deve encontrar o arquivo `escola.sql`.

1. Busque todos os alunos que não tenham nenhuma matrícula nos cursos.
2. Busque todos os alunos que não tiveram nenhuma matrícula nos últimos 45 dias, usando a instrução `EXISTS`.

3. É possível fazer a mesma consulta sem usar EXISTS ? Quais são?

# AGRUPANDO DADOS COM GROUP BY

A instrução solicitou a média de todos os cursos para fazer uma comparação de notas para verificar se todos os cursos possuem a mesma média, quais cursos tem menores notas e quais possuem as maiores notas. Vamos verificar a estrutura de algumas tabelas da nossa base de dados, começaremos pela tabela `curso` :

```
DESC curso;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
nome	varchar(255)	NO			

Agora vamos verificar a tabela `secao` :

```
DESC secao;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
curso_id	int(11)	NO		NULL	
titulo	varchar(255)	NO			
explicacao	varchar(255)	NO		NULL	
numero	int(11)	NO		NULL	

Já podemos perceber que existe uma relação entre `curso` de `secao` . Vamos também dar uma olhada na tabela `exercicio` :

```
DESC exercicio;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
secao_id	int(11)	NO		NULL	
pergunta	varchar(255)	NO		NULL	
resposta_oficial	varchar(255)	NO		NULL	

Observe que na tabela `exercicio` temos uma associação com a tabela `secao` . Agora vamos verificar a tabela `resposta` :

```
DESC resposta
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
exercicio_id	int(11)	YES		NULL	
aluno_id	int(11)	YES		NULL	
resposta_dada	varchar(255)	YES		NULL	

Podemos verificar que a tabela `resposta` está associada com a tabela `exercício`. Por fim, vamos verificar a tabela `nota`:

```
DESC nota;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
resposta_id	int(11)	YES		NULL	
nota	decimal(18,2)	YES		NULL	

Note que também a tabela `nota` possui uma associação, nesse caso com a tabela `resposta`.

Como vimos, existem muitas tabelas que podemos selecionar em nossa *query*, então vamos montar a nossa *query* por partes. Começaremos pela tabela `nota`:

```
SELECT n.nota FROM nota n;
```

nota
8.00
0.00
7.00
6.00
9.00
10.00
4.00
4.00
7.00
8.00
6.00
7.00
4.00
9.00
3.00
5.00
5.00
5.00
6.00
8.00
8.00
9.00
10.00
2.00
0.00
1.00
4.00

+-----+

Conseguimos pegar todas as notas, agora precisamos resolver a nossa primeira associação, nesse caso, juntar a tabela resposta com a tabela nota :

```
SELECT n.nota FROM nota n
JOIN resposta r ON n.resposta_id = r.id;
```

```
+-----+
| nota |
+-----+
| 8.00 |
| 0.00 |
| 7.00 |
| 6.00 |
| 9.00 |
| 10.00 |
| 4.00 |
| 4.00 |
| 7.00 |
| 8.00 |
| 6.00 |
| 7.00 |
| 4.00 |
| 9.00 |
| 3.00 |
| 5.00 |
| 5.00 |
| 5.00 |
| 6.00 |
| 8.00 |
| 8.00 |
| 9.00 |
| 10.00 |
| 2.00 |
| 0.00 |
| 1.00 |
| 4.00 |
+-----+
```

Devolvemos todas as notas associadas com a tabela resposta . Agora vamos para o próximo JOIN entre a tabela resposta e a tabela exercicio :

```
SELECT n.nota FROM nota n
JOIN resposta r ON n.resposta_id = r.id
JOIN exercicio e ON r.exercicio_id = e.id;
```

```
+-----+
| nota |
+-----+
| 8.00 |
| 0.00 |
| 7.00 |
| 6.00 |
| 9.00 |
| 10.00 |
| 4.00 |
| 4.00 |
| 7.00 |
| 8.00 |
| 6.00 |
+-----+
```

7.00
4.00
9.00
3.00
5.00
5.00
5.00
6.00
8.00
8.00
9.00
10.00
2.00
0.00
1.00
4.00

+-----+

Agora faremos a associação entre a tabela `exercicio` e a tabela `secao` :

```
SELECT n.nota FROM nota n
JOIN resposta r ON n.resposta_id = r.id
JOIN exercicio e ON r.exercicio_id = e.id
JOIN secao s ON e.secao_id = s.id;
```

nota
8.00
0.00
7.00
6.00
9.00
10.00
4.00
4.00
7.00
8.00
6.00
7.00
4.00
9.00
3.00
5.00
5.00
5.00
6.00
8.00
8.00
9.00
10.00
2.00
0.00
1.00
4.00

+-----+

Por fim, faremos a última associação entre a tabela `secao` e a tabela `curso` .

```
SELECT n.nota FROM nota n
JOIN resposta r ON n.resposta_id = r.id
JOIN exercicio e ON r.exercicio_id = e.id
```

```
JOIN secao s ON e.secao_id = s.id
JOIN curso c ON s.curso_id = c.id;
```

```
+-----+
| nota |
+-----+
| 8.00 |
| 0.00 |
| 7.00 |
| 6.00 |
| 9.00 |
| 10.00 |
| 4.00 |
| 4.00 |
| 7.00 |
| 8.00 |
| 6.00 |
| 7.00 |
| 4.00 |
| 9.00 |
| 3.00 |
| 5.00 |
| 5.00 |
| 5.00 |
| 6.00 |
| 8.00 |
| 8.00 |
| 9.00 |
| 10.00 |
| 2.00 |
| 0.00 |
| 1.00 |
| 4.00 |
+-----+
```

Fizemos todas associações que precisávamos, porém repare que ainda está retornando a nota de todos os alunos por curso uma a uma, porém nós precisamos da média por curso! No MySQL, podemos utilizar a função `AVG()` para tirar a média:

```
SELECT AVG(n.nota) FROM nota n
JOIN resposta r ON n.resposta_id = r.id
JOIN exercicio e ON r.exercicio_id = e.id
JOIN secao s ON e.secao_id = s.id
JOIN curso c ON s.curso_id = c.id;
```

```
+-----+
| AVG(n.nota) |
+-----+
| 5.740741 |
+-----+
```

Observe que foi retornado apenas um valor, será que essa média é igual para todos os cursos? Vamos tentar retornar os cursos e verificarmos:

```
SELECT c.nome, AVG(n.nota) FROM nota n
JOIN resposta r ON n.resposta_id = r.id
JOIN exercicio e ON r.exercicio_id = e.id
JOIN secao s ON e.secao_id = s.id
JOIN curso c ON s.curso_id = c.id;
```



nome	AVG(n.nota)
SQL e banco de dados	5.740741

Apenas 1 curso? Não era esse o resultado que esperávamos! Quando utilizamos a função `AVG()` ela calcula todos os valores existentes da *query* e retorna a média, porém em apenas uma linha! Para que a função `AVG()` calcule a média de cada curso, precisamos informar que queremos **agrupar** a média para uma determinada coluna, nesse caso, a coluna `c.nome`, ou seja, para cada curso diferente queremos que calcule a média. Para agruparmos uma coluna utilizamos a instrução `GROUP BY`, informando a coluna que queremos agrupar:

```
SELECT c.nome, AVG(n.nota) FROM nota n
JOIN resposta r ON n.resposta_id = r.id
JOIN exercicio e ON r.exercicio_id = e.id
JOIN secao s ON e.secao_id = s.id
JOIN curso c ON s.curso_id = c.id
GROUP BY c.nome;
```

nome	AVG(n.nota)
C# e orientação a objetos	4.857143
Desenvolvimento web com VRaptor	8.000000
Scrum e métodos ágeis	5.777778
SQL e banco de dados	6.100000

O pessoal do comercial da instituição informou que alguns alunos estão reclamando pela quantidade de exercícios nos cursos. Então vamos verificar quantos exercícios existem para cada curso. Primeiro vamos verificar quantos exercícios existem no banco usando a função `COUNT()`:

```
SELECT COUNT(*) FROM exercicio;
```

COUNT(*)
31

Retornamos a quantidade de todos os exercícios, porém nós precisamos saber o total de exercícios para cada curso, ou seja, precisamos juntar a tabela `curso`. Porém, para juntar a tabela `curso`, teremos que juntar a tabela `secao`:

```
SELECT COUNT(*) FROM exercicio e
JOIN secao s ON e.secao_id = s.id
```

Agora podemos juntar a tabela `curso` e retornar o nome do curso também:

```
SELECT c.nome, COUNT(*) FROM exercicio e
JOIN secao s ON e.secao_id = s.id
JOIN curso c ON s.curso_id = c.id;
```

nome	COUNT(*)
------	----------

```
+-----+-----+
| SQL e banco de dados |      31 |
+-----+-----+
```

Perceba que o resultado foi similar ao que aconteceu quando tentamos tirar a média sem agrupar! Então precisamos também informar que queremos agrupar a contagem pelo nome do curso. Então vamos adicionar o `GROUP BY` :

```
SELECT c.nome, COUNT(*) FROM exercicio e
JOIN secao s ON e.secao_id = s.id
JOIN curso c ON s.curso_id = c.id
GROUP BY c.nome;
```

```
+-----+-----+
| nome                                | COUNT(*) |
+-----+-----+
| C# e orientação a objetos          |         7 |
| Desenvolvimento web com VRaptor    |         7 |
| Scrum e métodos ágeis              |         9 |
| SQL e banco de dados                |         8 |
+-----+-----+
```

Note que o nome da coluna que conta todos os exercícios está um pouco estranha, vamos adicionar um alias para melhorar o resultado:

```
SELECT c.nome, COUNT(*) AS contagem FROM exercicio e
JOIN secao s ON e.secao_id = s.id
JOIN curso c ON s.curso_id = c.id
GROUP BY c.nome;
```

```
+-----+-----+
| nome                                | contagem |
+-----+-----+
| C# e orientação a objetos          |         7 |
| Desenvolvimento web com VRaptor    |         7 |
| Scrum e métodos ágeis              |         9 |
| SQL e banco de dados                |         8 |
+-----+-----+
```

Agora o relatório faz muito mais sentido.

Todo final de semestre nós precisamos enviar um relatório para o MEC informando quantos alunos estão matriculados em cada curso da instituição. Faremos novamente a nossa *query* por partes, vamos retornar primeiro todos os cursos:

```
SELECT c.nome FROM curso c
```

Vamos juntar a tabela `matricula` :

```
SELECT c.nome FROM curso c
JOIN matricula m ON m.curso_id = c.id
```

Agora vamos juntar os alunos também:

```
SELECT c.nome FROM curso c
JOIN matricula m ON m.curso_id = c.id
JOIN aluno a ON m.aluno_id = a.id;
```

Precisamos agora contar a quantidade de alunos:

```
SELECT c.nome, COUNT(a.id) AS quantidade FROM curso c
JOIN matricula m ON m.curso_id = c.id
JOIN aluno a ON m.aluno_id = a.id;
```

nome	quantidade
SQL e banco de dados	14

Lembre-se que precisamos agrupar a contagem pelo nome do curso:

```
SELECT c.nome, COUNT(a.id) AS quantidade FROM curso c
JOIN matricula m ON m.curso_id = c.id
JOIN aluno a ON m.aluno_id = a.id
GROUP BY c.nome;
```

nome	quantidade
C# e orientação a objetos	4
Desenvolvimento mobile com Android	2
Desenvolvimento web com VRaptor	2
Scrum e métodos ágeis	2
SQL e banco de dados	4

Agora conseguimos realizar o nosso relatório conforme o esperado.

## 8.1 RESUMINDO

Vimos nesse capítulo como podemos gerar relatórios utilizando funções como `AVG()` e `COUNT()`. Vimos também que, se precisamos retornar as colunas para verificar qual é o valor de cada linha, como por exemplo, a média de cada curso, precisamos agrupar essas colunas por meio do `GROUP BY`. Vamos para os exercícios?

## EXERCÍCIOS

1. Exiba a média das notas por curso.
2. Devolva o curso e as médias de notas, levando em conta somente alunos que tenham "Silva" ou "Santos" no sobrenome.
3. Conte a quantidade de respostas por exercício. Exiba a pergunta e o número de respostas.
4. Você pode ordenar pelo `COUNT` também. Basta colocar `ORDER BY COUNT(coluna)`.

Pegue a resposta do exercício anterior, e ordene por número de respostas, em ordem decrescente.

1. Podemos agrupar por mais de um campo de uma só vez. Por exemplo, se quisermos a média de

notas por aluno por curso, podemos fazer GROUP BY aluno.id, curso.id.

# FILTRANDO AGREGAÇÕES E O HAVING

Todo o fim de semestre, a instituição de ensino precisa montar os boletins dos alunos. Então vamos montar a *query* que retornará todas as informações para montar o boletim. Começaremos retornando todas as notas dos alunos:

```
SELECT n.nota FROM nota n
```

Agora vamos associar a com as respostas com as notas:

```
SELECT n.nota FROM nota n
JOIN resposta r ON r.id = n.resposta_id
```

Associaremos agora com os exercícios com as respostas:

```
SELECT n.nota FROM nota n
JOIN resposta r ON r.id = n.resposta_id
JOIN exercicio e ON e.id = r.exercicio_id
```

Agora associaremos a seção com os exercícios:

```
SELECT n.nota FROM nota n
JOIN resposta r ON r.id = n.resposta_id
JOIN exercicio e ON e.id = r.exercicio_id
JOIN secao s ON s.id = e.secao_id
```

Agora o curso com a seção:

```
SELECT n.nota FROM nota n
JOIN resposta r ON r.id = n.resposta_id
JOIN exercicio e ON e.id = r.exercicio_id
JOIN secao s ON s.id = e.secao_id
JOIN curso c ON c.id = s.curso_id
```

Por fim, a resposta com o aluno:

```
SELECT n.nota FROM nota n
JOIN resposta r ON r.id = n.resposta_id
JOIN exercicio e ON e.id = r.exercicio_id
JOIN secao s ON s.id = e.secao_id
JOIN curso c ON c.id = s.curso_id
JOIN aluno a ON a.id = r.aluno_id;
```

Verificando o resultado:

```
+-----+
| nota  |
+-----+
| 8.00  |
```

```

| 0.00 |
| 7.00 |
| 6.00 |
| 9.00 |
| 10.00 |
| 4.00 |
| 4.00 |
| 7.00 |
| 8.00 |
| 6.00 |
| 7.00 |
| 4.00 |
| 9.00 |
| 3.00 |
| 5.00 |
| 5.00 |
| 5.00 |
| 6.00 |
| 8.00 |
| 8.00 |
| 9.00 |
| 10.00 |
| 2.00 |
| 0.00 |
| 1.00 |
| 4.00 |
+-----+

```

Observe que estamos fazendo *queries* grandes, aconselhamos que, no momento que precisar fazer uma *query* complexa, faça *queries* menores e testes seus resultados, pois se existir alguma instrução errada, é mais fácil de identificar o problema.

Agora que associamos todas as nossas tabelas necessárias, vamos tirar a média com a função de agregação `AVG()` que é capaz de tirar médias, conforme visto durante o curso:

```

SELECT AVG(n.nota) FROM nota n
JOIN resposta r ON r.id = n.resposta_id
JOIN exercicio e ON e.id = r.exercicio_id
JOIN secao s ON s.id = e.secao_id
JOIN curso c ON c.id = s.curso_id
JOIN aluno a ON a.id = r.aluno_id;

```

```

+-----+
| AVG(n.nota) |
+-----+
| 5.740741 |
+-----+

```

Retornou a média, porém não queremos apenas a média! Precisamos também dos alunos e dos cursos. Então vamos adicioná-los:

```

SELECT a.nome, c.nome, AVG(n.nota) FROM nota n
JOIN resposta r ON r.id = n.resposta_id
JOIN exercicio e ON e.id = r.exercicio_id
JOIN secao s ON s.id = e.secao_id
JOIN curso c ON c.id = s.curso_id
JOIN aluno a ON a.id = r.aluno_id;

```

```

+-----+-----+-----+

```

nome	nome	AVG(n.nota)
João da Silva	SQL e banco de dados	5.740741

Lembre-se que estamos lidando com uma função de agregação, ou seja, se não informarmos a forma que ela precisa **agrupar** as colunas, ela retornará apenas uma linha! Porém, precisamos sempre pensar em qual tipo de agrupamento é necessário, nesse caso queremos que mostre a média de cada aluno, então agruparemos pelos alunos, porém também que queremos que a cada curso que o aluno fez mostre a sua :

```
SELECT a.nome, c.nome, AVG(n.nota) FROM nota n
JOIN resposta r ON r.id = n.resposta_id
JOIN exercicio e ON e.id = r.exercicio_id
JOIN secao s ON s.id = e.secao_id
JOIN curso c ON c.id = s.curso_id
JOIN aluno a ON a.id = r.aluno_id
GROUP BY a.nome, c.nome;
```

nome	nome	AVG(n.nota)
Alberto Santos	Scrum e métodos ágeis	5.777778
Frederico José	Desenvolvimento web com VRaptor	8.000000
Frederico José	SQL e banco de dados	5.666667
João da Silva	SQL e banco de dados	6.285714
Renata Alonso	C# e orientação a objetos	4.857143

## 9.1 CONDIÇÕES COM HAVING

Retornamos todas as médias dos alunos, porém a instituição precisa de um relatório separado para todos os alunos que reprovaram, ou seja, que tiraram nota baixa, nesse caso médias menores que 5. De acordo com o que vimos até agora bastaria adicionarmos um `WHERE` :

```
SELECT a.nome, c.nome, AVG(n.nota) FROM nota n
JOIN resposta r ON r.id = n.resposta_id
JOIN exercicio e ON e.id = r.exercicio_id
JOIN secao s ON s.id = e.secao_id
JOIN curso c ON c.id = s.curso_id
JOIN aluno a ON a.id = r.aluno_id
WHERE AVG(n.nota) < 5
GROUP BY a.nome, c.nome;
```

ERROR 1111 (HY000): Invalid use of group function

Nesse caso, estamos tentando adicionar condições para uma função de agregação, porém, quando queremos **adicionar condições para funções de agregação** precisamos utilizar o `HAVING` ao invés de `WHERE` :

```
SELECT a.nome, c.nome, AVG(n.nota) FROM nota n
JOIN resposta r ON r.id = n.resposta_id
JOIN exercicio e ON e.id = r.exercicio_id
JOIN secao s ON s.id = e.secao_id
JOIN curso c ON c.id = s.curso_id
```

```
JOIN aluno a ON a.id = r.aluno_id
HAVING AVG(n.nota) < 5
GROUP BY a.nome, c.nome;
```

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'GROUP BY a.nome, c.nome' at line 8

Além de utilizarmos o `HAVING` existe um pequeno detalhe, precisamos **sempre** agrupar antes as colunas pelo `GROUP BY` para depois utilizarmos o `HAVING` :

```
SELECT a.nome, c.nome, AVG(n.nota) FROM nota n
JOIN resposta r ON r.id = n.resposta_id
JOIN exercicio e ON e.id = r.exercicio_id
JOIN secao s ON s.id = e.secao_id
JOIN curso c ON c.id = s.curso_id
JOIN aluno a ON a.id = r.aluno_id
GROUP BY a.nome, c.nome
HAVING AVG(n.nota) < 5;
```

nome	nome	AVG(n.nota)
Renata Alonso	C# e orientação a objetos	4.857143

Agora conseguimos retornar o aluno que teve a média abaixo de 5. E se quiséssemos pegar todos os alunos que aprovaram? É simples, bastaria alterar o sinal para `>=` :

```
SELECT a.nome, c.nome, AVG(n.nota) FROM nota n
JOIN resposta r ON r.id = n.resposta_id
JOIN exercicio e ON e.id = r.exercicio_id
JOIN secao s ON s.id = e.secao_id
JOIN curso c ON c.id = s.curso_id
JOIN aluno a ON a.id = r.aluno_id
GROUP BY a.nome, c.nome
HAVING AVG(n.nota) >= 5;
```

nome	nome	AVG(n.nota)
Alberto Santos	Scrum e métodos ágeis	5.777778
Frederico José	Desenvolvimento web com VRaptor	8.000000
Frederico José	SQL e banco de dados	5.666667
João da Silva	SQL e banco de dados	6.285714

A instituição enviou mais uma solicitação de um relatório informando quais cursos tem poucos alunos para tomar uma decisão se vai manter os cursos ou se irá cancelá-los. Então vamos novamente fazer a nossa *query* por passos, primeiro vamos começar selecionando os cursos:

```
SELECT c.nome FROM curso c
```

Agora vamos juntar o curso com a matrícula e a matrícula com o aluno e verificar o resultado:

```
SELECT c.nome FROM curso c
JOIN matricula m ON m.curso_id = c.id
JOIN aluno a ON m.aluno_id = a.id;
```

```
+-----+
```



nome
SQL e banco de dados
SQL e banco de dados
Scrum e métodos ágeis
C# e orientação a objetos
SQL e banco de dados
Scrum e métodos ágeis
Desenvolvimento web com VRaptor
Desenvolvimento mobile com Android
Desenvolvimento mobile com Android
SQL e banco de dados
C# e orientação a objetos
C# e orientação a objetos
C# e orientação a objetos
Desenvolvimento web com VRaptor

Nossa *query* está funcionando, então vamos contar a quantidade de alunos com a função `COUNT()` :

```
SELECT c.nome, COUNT(a.id) FROM curso c
JOIN matricula m ON m.curso_id = c.id
JOIN aluno a ON m.aluno_id = a.id;
```

Há um detalhe nessa *query*, pois queremos contar todos os alunos de cada curso, ou seja, precisamos agrupar os cursos!

```
SELECT c.nome, COUNT(a.id) FROM curso c
JOIN matricula m ON m.curso_id = c.id
JOIN aluno a ON m.aluno_id = a.id
GROUP BY c.nome;
```

nome	COUNT(a.id)
C# e orientação a objetos	4
Desenvolvimento mobile com Android	2
Desenvolvimento web com VRaptor	2
Scrum e métodos ágeis	2
SQL e banco de dados	4

A *query* funcionou, porém precisamos saber apenas os cursos que tem poucos alunos, nesse caso, cursos que tenham menos de 10 alunos:

```
SELECT c.nome, COUNT(a.id) FROM curso c
JOIN matricula m ON m.curso_id = c.id
JOIN aluno a ON m.aluno_id = a.id
GROUP BY c.nome
HAVING COUNT(a.id) < 10;
```

nome	COUNT(a.id)
C# e orientação a objetos	4
Desenvolvimento mobile com Android	2
Desenvolvimento web com VRaptor	2
Scrum e métodos ágeis	2
SQL e banco de dados	4

Agora podemos enviar o relatório para a instituição.

## 9.2 RESUMINDO

Sabemos que para adicionarmos filtros apenas para colunas utilizamos a instrução `WHERE` e indicamos todas as peculiaridades necessárias, porém quando precisamos adicionar filtros para funções de agregação, como por exemplo o `AVG()`, precisamos utilizar a instrução `HAVING`. Além disso, é sempre bom lembrar que, quando estamos desenvolvendo *queries* grandes, é recomendado que faça passa-a-passa *queries* menores, ou seja, resolva os menores problemas juntando cada tabela por vez e teste para verificar se está funcionando, pois isso ajuda a verificar aonde está o problema da *query*. Vamos para os exercícios?

## EXERCÍCIOS

1. Qual é a principal diferença entre as instruções `having` e `where` do `sql`?
2. Devolva todos os alunos, cursos e a média de suas notas. Lembre-se de agrupar por aluno e por curso. Filtre também pela nota: só mostre alunos com nota média menor do que 5.
3. Exiba todos os cursos e a sua quantidade de matrículas. Mas, exiba somente cursos que tenham mais de 1 matrícula.
4. Exiba o nome do curso e a quantidade de seções que existe nele. Mostre só cursos com mais de 3 seções.

# MÚLTIPLOS VALORES NA CONDIÇÃO E O IN

O setor de financeiro dessa instituição, solicitou um relatório informando todas as formas de pagamento cadastradas no banco de dados para verificar se está de acordo com o que eles trabalham. Na base de dados, se verificarmos a tabela `matricula` :

```
DESC matricula;
```

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
aluno_id	int(11)	NO		NULL	
curso_id	int(11)	NO		NULL	
data	datetime	NO		NULL	
tipo	varchar(20)	NO			

Observe que existe a coluna `tipo` que representa qual é a forma de pagamento. E precisamos pegar um relatório da seguinte maneira:

Forma de pagamento
Forma 1
Forma 2
Forma 3

Figura 10.1: Planilha exemplo

Vamos selecionar apenas a coluna `tipo` da tabela `matricula` :

```
SELECT m.tipo FROM matricula m;
```

tipo
PAGA_PF
PAGA_PJ
PAGA_PF
PAGA_CHEQUE
PAGA_BOLETO
PAGA_PJ
PAGA_PF
PAGA_PJ
PAGA_PJ
PAGA_CHEQUE
PAGA_BOLETO
PAGA_PJ

```
| PAGA_PF      |
| PAGA_PJ      |
+-----+
```

Veja que foram retornados tipos de pagamento iguais, porém precisamos enviar um relatório apenas com os tipos de pagamento **distintos**. Para retornarmos os valores distintos de uma coluna podemos utilizar a instrução `DISTINCT` :

```
SELECT DISTINCT m.tipo FROM matricula m;
```

```
+-----+
| tipo    |
+-----+
| PAGA_PF |
| PAGA_PJ |
| PAGA_CHEQUE |
| PAGA_BOLETO |
+-----+
```

Conseguimos retornar o relatório das formas de pagamento, porém o setor financeiro ainda precisa saber de mais informações. Agora foi solicitado que enviásse um relatório com os cursos e a quantidade de alunos que possuem o tipo de pagamento PJ. Vamos verificar o exemplo em uma planilha:

curso	matriculas pj
C#	5
Java	2

Figura 10.2: Planilha exemplo

Sabemos que o relatório é sobre a quantidade de matrículas que foram pagas como PJ, precisamos contar, ou seja, usaremos a função `COUNT()` . Vamos começar a contar a quantidade de matrículas:

```
SELECT COUNT(m.id) FROM matricula m;
```

```
+-----+
| COUNT(m.id) |
+-----+
|          14 |
+-----+
```

Agora vamos juntar com a tabela `curso` e exibir o nome do curso:

```
SELECT c.nome, COUNT(m.id) FROM matricula m
JOIN curso c ON m.curso_id = c.id;
```

```
+-----+-----+
| nome                | COUNT(m.id) |
+-----+-----+
| SQL e banco de dados |          14 |
+-----+-----+
```

Observe que foi retornado apenas uma linha! Isso significa que a função `COUNT()` também é uma função de agregação, ou seja, se queremos adicionar mais colunas na nossa *query*, precisamos agrupá-las. Então vamos agrupar o nome do curso:

```
SELECT c.nome, COUNT(m.id) FROM matricula m
JOIN curso c ON m.curso_id = c.id
GROUP BY c.nome;
```

nome	COUNT(m.id)
C# e orientação a objetos	4
Desenvolvimento mobile com Android	2
Desenvolvimento web com VRaptor	2
Scrum e métodos ágeis	2
SQL e banco de dados	4

Conseguimos retornar todas os cursos e a quantidade de matrículas, porém precisamos filtrar por tipo de pagamento PJ. Então vamos adicionar um `WHERE` :

```
SELECT c.nome, COUNT(m.id) FROM matricula m
JOIN curso c ON m.curso_id = c.id
WHERE m.tipo = 'PAGA_PJ'
GROUP BY c.nome;
```

nome	COUNT(m.id)
C# e orientação a objetos	1
Desenvolvimento mobile com Android	2
Desenvolvimento web com VRaptor	1
Scrum e métodos ágeis	1
SQL e banco de dados	1

## FILTROS UTILIZANDO O IN

O setor financeiro da instituição precisa de mais detalhes sobre os tipos de pagamento de cada curso, eles precisam de um relatório similar ao que fizemos, porém para todos que sejam pagamento PJ e PF. Para diferenciar o tipo de pagamento, precisaremos adicionar a coluna do `m.tipo` :

```
SELECT c.nome, COUNT(m.id), m.tipo
FROM matricula m
JOIN curso c ON m.curso_id = c.id
WHERE m.tipo = 'PAGA_PJ'
OR m.tipo = 'PAGA_PF'
GROUP BY c.nome, m.tipo;
```

nome	COUNT(m.id)	tipo
C# e orientação a objetos	1	PAGA_PF
C# e orientação a objetos	1	PAGA_PJ
Desenvolvimento mobile com Android	2	PAGA_PJ
Desenvolvimento web com VRaptor	1	PAGA_PF
Desenvolvimento web com VRaptor	1	PAGA_PJ
Scrum e métodos ágeis	1	PAGA_PF
Scrum e métodos ágeis	1	PAGA_PJ
SQL e banco de dados	1	PAGA_PF
SQL e banco de dados	1	PAGA_PJ

Suponhamos que agora precisamos retornar também os que foram pagos em boleto ou cheque. O que poderíamos adicionar na *query*? Mais `OR` s?

```
SELECT c.nome, COUNT(m.id), m.tipo
FROM matricula m
JOIN curso c ON m.curso_id = c.id
WHERE m.tipo = 'PAGA_PJ'
OR m.tipo = 'PAGA_PF'
OR m.tipo = 'PAGA_BOLETO'
OR m.tipo = 'PAGA_CHEQUE'
OR m.tipo = '...'
OR m.tipo = '...'
GROUP BY c.nome, m.tipo;
```

Resolveria, mas perceba que a nossa *query* a cada novo tipo de pagamento a nossa *query* tende a crescer, dificultando a leitura... Em SQL, existe a instrução `IN` que permite especificarmos mais de um valor que precisamos filtrar ao mesmo tempo para uma determinada coluna:

```
SELECT c.nome, COUNT(m.id), m.tipo
FROM matricula m
JOIN curso c ON m.curso_id = c.id
WHERE m.tipo IN ('PAGA_PJ', 'PAGA_PF', 'PAGA_CHEQUE', 'PAGA_BOLETO')
GROUP BY c.nome, m.tipo;
```

nome	COUNT(m.id)	tipo
C# e orientação a objetos	1	PAGA_BOLETO
C# e orientação a objetos	1	PAGA_CHEQUE
C# e orientação a objetos	1	PAGA_PF
C# e orientação a objetos	1	PAGA_PJ
Desenvolvimento mobile com Android	2	PAGA_PJ
Desenvolvimento web com VRaptor	1	PAGA_PF
Desenvolvimento web com VRaptor	1	PAGA_PJ
Scrum e métodos ágeis	1	PAGA_PF
Scrum e métodos ágeis	1	PAGA_PJ
SQL e banco de dados	1	PAGA_BOLETO
SQL e banco de dados	1	PAGA_CHEQUE
SQL e banco de dados	1	PAGA_PF
SQL e banco de dados	1	PAGA_PJ

Se um novo tipo de pagamento for adicionado, basta adicionarmos dentro do `IN` e a nossa *query* funcionará corretamente.

A instituição nomeou 3 alunos como os mais destacados nos últimos cursos realizamos e gostaria de saber quais foram todos os cursos que eles fizeram. Os 3 alunos que se destacaram foram: João da Silva, Alberto Santos e a Renata Alonso. Vamos verificar quais são os `id` s desses alunos:

```
SELECT * FROM aluno;
```

id	nome	email
1	João da Silva	joao@dasilva.com
2	Frederico José	fred@jose.com
3	Alberto Santos	alberto@santos.com
4	Renata Alonso	renata@alonso.com

5	Paulo da Silva	paulo@dasilva.com
6	Carlos Cunha	carlos@cunha.com
7	Paulo José	paulo@jose.com
8	Manoel Santos	manoel@santos.com
9	Renata Ferreira	renata@ferreira.com
10	Paula Soares	paula@soares.com
11	Jose da Silva	jose@dasilva.com
12	Danilo Cunha	danilo@cunha.com
13	Zilmira José	Zilmira@jose.com
14	Cristaldo Santos	cristaldo@santos.com
15	Osmir Ferreira	osmir@ferreira.com
16	Claudio Soares	claudio@soares.com

O aluno João da Silva é 1, Alberto Santos 3 e Renata Alonso 4. Agora que sabemos os `id`s podemos verificar os seus cursos. Então vamos começar a nossa *query* retornando todos os cursos:

```
SELECT c.nome FROM curso c;
```

Agora vamos juntar o curso com a matrícula:

```
SELECT c.nome FROM curso c
JOIN matricula m ON m.curso_id = c.id;
```

Por fim, vamos juntar a matricula com o aluno e retornar o nome do aluno:

```
SELECT a.nome, c.nome FROM curso c
JOIN matricula m ON m.curso_id = c.id
JOIN aluno a ON m.aluno_id = a.id;
```

Fizemos todas as junções, agora só precisamos do filtro. Precisamos retornar os cursos dos **3 alunos ao mesmo tempo**, podemos utilizar a instrução `IN` :

```
SELECT a.nome, c.nome FROM curso c
JOIN matricula m ON m.curso_id = c.id
JOIN aluno a ON m.aluno_id = a.id
WHERE a.id IN (1,3,4);
```

nome	nome
João da Silva	SQL e banco de dados
Alberto Santos	Scrum e métodos ágeis
Renata Alonso	C# e orientação a objetos
Renata Alonso	Desenvolvimento mobile com Android
João da Silva	C# e orientação a objetos
Alberto Santos	C# e orientação a objetos

Retornamos todos os cursos dos 3 alunos, porém ainda tá um pouco desorganizado, então vamos ordenar pelo nome dos alunos utilizando o `ORDER BY` :

```
SELECT a.nome, c.nome FROM curso c
JOIN matricula m ON m.curso_id = c.id
JOIN aluno a ON m.aluno_id = a.id
WHERE a.id IN (1,3,4)
ORDER BY a.nome;
```

nome	nome
Alberto Santos	Scrum e métodos ágeis
Alberto Santos	C# e orientação a objetos
João da Silva	SQL e banco de dados
João da Silva	C# e orientação a objetos
Renata Alonso	C# e orientação a objetos
Renata Alonso	Desenvolvimento mobile com Android

Na instituição, serão lançados alguns cursos novos de *.NET* e o pessoal do comercial precisa divulgar esses cursos para os ex-alunos, porém apenas para os ex-alunos que já fizeram os cursos de C# e de SQL. Inicialmente vamos verificar os `id`s desses cursos:

```
SELECT * FROM curso;
```

id	nome
1	SQL e banco de dados
2	Desenvolvimento web com VRaptor
3	Scrum e métodos ágeis
4	C# e orientação a objetos
5	Java e orientação a objetos
6	Desenvolvimento mobile com iOS
7	Desenvolvimento mobile com Android
8	Ruby on Rails
9	PHP e MySQL

Curso de SQL é 1 e o curso de C# é 4. Construindo a nossa *query*, começaremos retornando o aluno:

```
SELECT a.nome FROM aluno a;
```

Então juntamos com a matrícula e o curso e vamos retornar quais foram os cursos realizados:

```
SELECT a.nome, c.nome FROM aluno a
JOIN matricula m ON m.aluno_id = a.id
JOIN curso c ON m.curso_id = c.id;
```

Agora utilizaremos o filtro para retornar tanto o curso de SQL(1), quanto o curso de C#(4):

```
SELECT a.nome, c.nome FROM aluno a
JOIN matricula m ON m.aluno_id = a.id
JOIN curso c ON m.curso_id = c.id
WHERE c.id IN (1, 4);
```

nome	nome
João da Silva	SQL e banco de dados
Frederico José	SQL e banco de dados
Renata Alonso	C# e orientação a objetos
Paulo José	SQL e banco de dados
Manoel Santos	SQL e banco de dados
João da Silva	C# e orientação a objetos
Frederico José	C# e orientação a objetos
Alberto Santos	C# e orientação a objetos



Novamente o resultado está desordenado, vamos ordenar pelo nome do aluno:

```
SELECT a.nome, c.nome FROM aluno a
JOIN matricula m ON m.aluno_id = a.id
JOIN curso c ON m.curso_id = c.id
WHERE c.id IN (1, 4)
ORDER BY a.nome;
```

nome	nome
Alberto Santos	C# e orientação a objetos
Frederico José	SQL e banco de dados
Frederico José	C# e orientação a objetos
João da Silva	SQL e banco de dados
João da Silva	C# e orientação a objetos
Manoel Santos	SQL e banco de dados
Paulo José	SQL e banco de dados
Renata Alonso	C# e orientação a objetos

Agora sabemos que apenas os alunos Frederico José e João da Silva, são os ex-alunos aptos para realizar os novos cursos de *.NET*.

## 10.1 RESUMINDO

Nesse capítulo vimos que quando precisamos saber todos os valores de uma determinada coluna podemos utilizar a instrução `DISTINCT` para retornar todos os valores **distintos**, ou seja, sem nenhuma repetição. Vimos também que quando precisamos realizar vários filtros para uma mesma coluna, podemos utilizar a instrução `IN` passando por parâmetro todos os valores que esperamos que seja retornado, ao invés de ficar preenchendo a nossa *query* com vários `OR` s. Vamos para os exercícios?

## EXERCÍCIOS

1. Exiba todos os tipos de matrícula que existem na tabela. Use `DISTINCT` para que não haja repetição.
2. Exiba todos os cursos e a sua quantidade de matrículas. Mas filtre por matrículas dos tipos PF ou PJ.
3. Traga todas as perguntas e a quantidade de respostas de cada uma. Mas dessa vez, somente dos cursos com ID 1 e 3.

# SUB-QUERIES

A instituição precisa de um relatório mais robusto, com as seguintes informações: Precisa do nome do aluno e curso, a média do aluno em relação ao curso e a diferença entre a média do aluno e a média geral do curso. Demonstrando em uma planilha, o resultado que se espera é o seguinte:

aluno	curso	média do aluno	diferença
Alex	Java	6	-1
Guilherme	SQL	6	0
Renata	C#	7	1

Figura 11.1: Planilha exemplo

Observe que o aluno Alex fez o curso de Java tirou 6 de média e a diferença entre a média dele e a média geral para o curso de Java foi -1, isso significa que a média geral do curso de Java é 7, ou seja,  $6 - 7$ . Vamos analisar o aluno Guilherme, veja que ele tirou 6 de média e a diferença foi 0, pois a média geral do curso de SQL é 6, ou seja,  $6 - 6$ . Por fim, a aluna Renata tirou média 7 no curso de C#, porém a diferença foi 1, isso significa que a média geral é 6, ou seja,  $7 - 6$ .

Como você montaria essa *query*? Aparentemente é um pouco complexa... Então vamos começar por partes da mesma forma que fizemos anteriormente. Começaremos pela tabela `nota` :

```
SELECT n.nota FROM nota n;
```

Agora vamos juntar as tabelas `resposta` e `exercicio` e vamos verificar o resultado:

```
SELECT n.nota FROM nota n
JOIN resposta r ON n.resposta_id = r.id
JOIN exercicio e ON r.exercicio_id = e.id;
```

```
+-----+
| nota |
+-----+
| 8.00 |
| 0.00 |
| 7.00 |
| 6.00 |
| 9.00 |
| 10.00 |
| 4.00 |
| 4.00 |
| 7.00 |
| 8.00 |
| 6.00 |
| 7.00 |
```

	4.00	
	9.00	
	3.00	
	5.00	
	5.00	
	5.00	
	6.00	
	8.00	
	8.00	
	9.00	
	10.00	
	2.00	
	0.00	
	1.00	
	4.00	
+	-----	+

A nossa *query* está funcionando. Vamos adicionar as tabelas de `secao` e `curso`, porém, dessa vez vamos adicionar o nome do curso:

```
SELECT c.nome, n.nota FROM nota n
JOIN resposta r ON n.resposta_id = r.id
JOIN exercicio e ON r.exercicio_id = e.id
JOIN secao s ON e.secao_id = s.id
JOIN curso c ON s.curso_id = c.id;
```

nome	nota
SQL e banco de dados	8.00
SQL e banco de dados	0.00
SQL e banco de dados	7.00
SQL e banco de dados	6.00
SQL e banco de dados	9.00
SQL e banco de dados	10.00
SQL e banco de dados	4.00
SQL e banco de dados	4.00
SQL e banco de dados	7.00
Desenvolvimento web com VRaptor	8.00
SQL e banco de dados	6.00
Scrum e métodos ágeis	7.00
Scrum e métodos ágeis	4.00
Scrum e métodos ágeis	9.00
Scrum e métodos ágeis	3.00
Scrum e métodos ágeis	5.00
Scrum e métodos ágeis	5.00
Scrum e métodos ágeis	5.00
Scrum e métodos ágeis	6.00
Scrum e métodos ágeis	8.00
C# e orientação a objetos	8.00
C# e orientação a objetos	9.00
C# e orientação a objetos	10.00
C# e orientação a objetos	2.00
C# e orientação a objetos	0.00
C# e orientação a objetos	1.00
C# e orientação a objetos	4.00

Por fim, juntaremos a tabela `aluno` com a tabela `resposta` e retornaremos o nome do aluno:

```
SELECT a.nome, c.nome, n.nota FROM nota n
```

```

JOIN resposta r ON n.resposta_id = r.id
JOIN exercicio e ON r.exercicio_id = e.id
JOIN secao s ON e.secao_id = s.id
JOIN curso c ON s.curso_id = c.id
JOIN aluno a ON r.aluno_id = a.id;

```

nome	nome	nota
João da Silva	SQL e banco de dados	8.00
João da Silva	SQL e banco de dados	0.00
João da Silva	SQL e banco de dados	7.00
João da Silva	SQL e banco de dados	6.00
João da Silva	SQL e banco de dados	9.00
João da Silva	SQL e banco de dados	10.00
João da Silva	SQL e banco de dados	4.00
Frederico José	SQL e banco de dados	4.00
Frederico José	SQL e banco de dados	7.00
Frederico José	Desenvolvimento web com VRaptor	8.00
Frederico José	SQL e banco de dados	6.00
Alberto Santos	Scrum e métodos ágeis	7.00
Alberto Santos	Scrum e métodos ágeis	4.00
Alberto Santos	Scrum e métodos ágeis	9.00
Alberto Santos	Scrum e métodos ágeis	3.00
Alberto Santos	Scrum e métodos ágeis	5.00
Alberto Santos	Scrum e métodos ágeis	5.00
Alberto Santos	Scrum e métodos ágeis	5.00
Alberto Santos	Scrum e métodos ágeis	6.00
Alberto Santos	Scrum e métodos ágeis	8.00
Renata Alonso	C# e orientação a objetos	8.00
Renata Alonso	C# e orientação a objetos	9.00
Renata Alonso	C# e orientação a objetos	10.00
Renata Alonso	C# e orientação a objetos	2.00
Renata Alonso	C# e orientação a objetos	0.00
Renata Alonso	C# e orientação a objetos	1.00
Renata Alonso	C# e orientação a objetos	4.00

Conseguimos retornar todas as notas do aluno e os cursos, porém nós precisamos das médias e não de todas as notas. Então vamos utilizar a função `AVG()` para retornar a média do aluno. Lembre-se que a função `AVG()` é uma função de agregação, ou seja, precisamos agrupar o aluno e o curso também:

```

SELECT a.nome, c.nome, AVG(n.nota) FROM nota n
JOIN resposta r ON n.resposta_id = r.id
JOIN exercicio e ON r.exercicio_id = e.id
JOIN secao s ON e.secao_id = s.id
JOIN curso c ON s.curso_id = c.id
JOIN aluno a ON r.aluno_id = a.id
GROUP BY a.nome, c.nome;

```

nome	nome	AVG(n.nota)
Alberto Santos	Scrum e métodos ágeis	5.777778
Frederico José	Desenvolvimento web com VRaptor	8.000000
Frederico José	SQL e banco de dados	5.666667
João da Silva	SQL e banco de dados	6.285714
Renata Alonso	C# e orientação a objetos	4.857143

Agora nós temos a média do aluno e seu respectivo curso, mas ainda falta a coluna da diferença que

calcula a diferença entre a média do aluno em um determinado curso e subtrai pela média geral. Porém ainda não temos a média geral, então como podemos pegar a média geral? Vamos verificar a tabela nota :

```
SELECT * FROM nota;
```

id	resposta_id	nota
1	1	8.00
2	2	0.00
3	3	7.00
4	4	6.00
5	5	9.00
6	6	10.00
7	7	4.00
8	8	4.00
9	9	7.00
10	10	8.00
11	11	6.00
12	12	7.00
13	13	4.00
14	14	9.00
15	15	3.00
16	16	5.00
17	17	5.00
18	18	5.00
19	19	6.00
20	20	8.00
21	21	8.00
22	22	9.00
23	23	10.00
24	24	2.00
25	25	0.00
26	26	1.00
27	27	4.00

Perceba que temos todas as notas, teoricamente as notas de todos os cursos, ou seja, para pegarmos a média geral usaremos o `AVG()` :

```
SELECT AVG(n.nota) FROM nota n;
```

AVG(n.nota)
5.740741

Conseguimos a média geral, agora vamos adicionar a coluna diferença. Antes de começar a fazer a coluna diferença vamos nomear a coluna de média do aluno para melhorar a visualização:

```
SELECT a.nome, c.nome, AVG(n.nota) as media_aluno FROM nota
...
```

A coluna diferença precisa da informação da `media_aluno` - media geral, porém, nós não temos nenhuma coluna para a média geral, e o resultado que precisamos está em uma query diferente... Como podemos resolver isso? Adicionando essa outra *query* dentro da *query* principal, ou seja, fazer uma

*subquery*:

```
SELECT a.nome, c.nome, AVG(n.nota) as media_aluno,
AVG(n.nota) - (SELECT AVG(n.nota) FROM nota n) as diferenca
FROM nota n
JOIN resposta r ON n.resposta_id = r.id
JOIN exercicio e ON r.exercicio_id = e.id
JOIN secao s ON e.secao_id = s.id
JOIN curso c ON s.curso_id = c.id
JOIN aluno a ON r.aluno_id = a.id
GROUP BY a.nome, c.nome;
```

nome	nome	media_aluno	diferenca
Alberto Santos	Scrum e métodos ágeis	5.777778	0.037037
Frederico José	Desenvolvimento web com VRaptor	8.000000	2.259259
Frederico José	SQL e banco de dados	5.666667	-0.074074
João da Silva	SQL e banco de dados	6.285714	0.544974
Renata Alonso	C# e orientação a objetos	4.857143	-0.883598

Observe que agora retornamos a diferença, mas será que essas informações batem? Que tal retornamos a média geral também?

```
SELECT a.nome, c.nome, AVG(n.nota) as media_aluno,
(SELECT AVG(n.nota) FROM nota n) as media_geral,
AVG(n.nota) - (SELECT AVG(n.nota) FROM nota n) as diferenca
FROM nota n
JOIN resposta r ON n.resposta_id = r.id
JOIN exercicio e ON r.exercicio_id = e.id
JOIN secao s ON e.secao_id = s.id
JOIN curso c ON s.curso_id = c.id
JOIN aluno a ON r.aluno_id = a.id
GROUP BY a.nome, c.nome;
```

nome	nome	media_aluno	media_geral	diferenca
Alberto Santos	Scrum e métodos ágeis	5.777778	5.740741	0.037037
Frederico José	Desenvolvimento web com VRaptor	8.000000	5.740741	2.259259
Frederico José	SQL e banco de dados	5.666667	5.740741	-0.074074
João da Silva	SQL e banco de dados	6.285714	5.740741	0.544974
Renata Alonso	C# e orientação a objetos	4.857143	5.740741	-0.883598

Conseguimos exibir o relatório como esperado, porém existe um pequeno detalhe. Note que o resultado da *subquery* (SELECT AVG(n.nota) FROM nota n) foi de apenas uma linha e é justamente por esse motivo que conseguimos efetuar operações aritméticas como, nesse caso, a subtração. Se o resultado fosse **mais de uma linha**, não seria possível realizar operações.

A instituição precisa de um relatório do aproveitamento dos alunos nos cursos, ou seja, precisamos saber se eles estão respondendo todos os exercícios, então iremos buscar o número de respostas que cada respondeu aluno individualmente. Vamos verificar o que é esperado do resultado em uma planilha:

aluno	resposta
Gabriel	10
Pedro	1
João	0

Figura 11.2: Planilha exemplo

Então primeiro começaremos retornando os alunos:

```
SELECT a.nome FROM aluno a;
```

Agora precisamos da quantidade de todas as respostas, então usaremos o `COUNT()` :

```
SELECT COUNT(r.id) FROM resposta r;
```

```
+-----+
| COUNT(r.id) |
+-----+
|          27 |
+-----+
```

Sabemos a *query* que conta as respostas e sabemos a *query* que retornam os alunos, então vamos adicionar a *query* que conta as respostas dentro da que retorna os alunos, ou seja, vamos fazer novamente uma *subquery*!

```
SELECT a.nome, (SELECT COUNT(r.id) FROM resposta r) AS quantidade_respostas FROM aluno a;
```

```
+-----+-----+
| nome          | quantidade_respostas |
+-----+-----+
| João da Silva |          27 |
| Frederico José |          27 |
| Alberto Santos |          27 |
| Renata Alonso  |          27 |
| Paulo da Silva |          27 |
| Carlos Cunha  |          27 |
| Paulo José     |          27 |
| Manoel Santos  |          27 |
| Renata Ferreira |          27 |
| Paula Soares   |          27 |
| Jose da Silva  |          27 |
| Danilo Cunha   |          27 |
| Zilmira José   |          27 |
| Cristaldo Santos |          27 |
| Osmir Ferreira |          27 |
| Claudio Soares |          27 |
+-----+-----+
```

Observe que os resultados da quantidade de respostas foram iguais para todos os alunos, pois não adicionamos nenhum filtro na *subquery*. Para resolver o problema, basta adicionar um `WHERE` indicando o que precisa ser filtrado, nesse caso, o `id` dos alunos retornados na *query* principal:

```
SELECT a.nome, (SELECT COUNT(r.id) FROM resposta r WHERE r.aluno_id = a.id) AS quantidade_respostas FROM aluno a;
```

```
+-----+-----+
```

nome	quantidade_respostas
João da Silva	7
Frederico José	4
Alberto Santos	9
Renata Alonso	7
Paulo da Silva	0
Carlos Cunha	0
Paulo José	0
Manoel Santos	0
Renata Ferreira	0
Paula Soares	0
Jose da Silva	0
Danilo Cunha	0
Zilmira José	0
Cristaldo Santos	0
Osmir Ferreira	0
Claudio Soares	0

A instituição precisa de uma relatório muito parecido com a *query* que acabamos de fazer, ela precisa saber quantas matrículas um aluno tem, ou seja, ao invés de resposta, informaremos as matrículas. Então vamos apenas substituir as informações das respostas pelas informações da matrícula:

```
SELECT a.nome, (SELECT COUNT(m.id) FROM matricula m WHERE m.aluno_id = a.id) AS quantidade_matricula
FROM aluno a;
```

nome	quantidade_matricula
João da Silva	2
Frederico José	3
Alberto Santos	2
Renata Alonso	2
Paulo da Silva	0
Carlos Cunha	0
Paulo José	1
Manoel Santos	2
Renata Ferreira	1
Paula Soares	1
Jose da Silva	0
Danilo Cunha	0
Zilmira José	0
Cristaldo Santos	0
Osmir Ferreira	0
Claudio Soares	0

Conseguimos pegar a quantidade de resposta e matricula de um determinado aluno, porém fizemos isso separadamente, porém agora precisamos juntar essas informações para montar em um único relatório que mostre, o nome do aluno, a quantidade de respostas e a quantidade de matrículas. Então vamos partir do princípio, ou seja, fazer a *query* que retorna todos os alunos:

```
SELECT a.nome FROM aluno a;
```

Agora vamos pegar a quantidade de respostas:

```
SELECT COUNT(r.id) FROM resposta r;
```



E então vamos pegar a quantidade de matrículas:

```
SELECT COUNT(m.id) FROM matricula m;
```

Temos todos os `SELECT` s que resolvem um determinado problema, ou seja, agora precisamos juntar todos eles para resolver a nova necessidade. Então vamos adicionar as duas *queries* que contam as matrículas e as respostas dentro da *query* principal, ou seja, a que retorna os alunos:

```
SELECT a.nome,  
(SELECT COUNT(m.id) FROM matricula m WHERE m.aluno_id = a.id) AS quantidade_matricula,  
(SELECT COUNT(r.id) FROM resposta r WHERE r.aluno_id = a.id) AS quantidade_respostas  
FROM aluno a;
```

nome	quantidade_matricula	quantidade_respostas
João da Silva	2	7
Frederico José	3	4
Alberto Santos	2	9
Renata Alonso	2	7
Paulo da Silva	0	0
Carlos Cunha	0	0
Paulo José	1	0
Manoel Santos	2	0
Renata Ferreira	1	0
Paula Soares	1	0
Jose da Silva	0	0
Danilo Cunha	0	0
Zilmira José	0	0
Cristaldo Santos	0	0
Osmir Ferreira	0	0
Claudio Soares	0	0

## 11.1 RESUMINDO

Vimos que nesse capítulo aprendemos a utilizar *subqueries* para resolver diversos problemas, como por exemplo contar a quantidade de matrículas ou de respostas de um aluno. Também vimos que podemos aplicar operações aritméticas utilizando *subqueries*, porém é sempre importante lembrar que só podemos realizar esse tipo de operações **desde que** a *subquery* retorne uma **única linha**. Então vamos para os exercícios?

## EXERCÍCIOS

1. Exiba a média das notas por aluno, além de uma coluna com a diferença entre a média do aluno e a média geral. Use sub-queries para isso.
2. Qual é o problema de se usar sub-queries?
3. Exiba a quantidade de matrículas por curso. Além disso, exiba a divisão entre matrículas naquele curso e matrículas totais.



# ENTENDENDO O LEFT JOIN

Os instrutores da instituição pediram um relatório com os alunos que são mais participativos na sala de aula, ou seja, queremos retornar os alunos que responderam mais exercícios. Consequentemente encontraremos também os alunos que não estão participando muito, então já aproveitamos e conversamos com eles para entender o que está acontecendo. Então começaremos retornando o aluno:

```
SELECT a.nome FROM aluno a;
```

Agora vamos contar a quantidade de respostas por meio da função `COUNT()` e agrupando pelo nome do aluno:

```
SELECT a.nome, COUNT(r.id) AS respostas
FROM aluno a
JOIN resposta r ON r.aluno_id = a.id
GROUP BY a.nome;
```

nome	respostas
Alberto Santos	9
Frederico José	4
João da Silva	7
Renata Alonso	7

Mas onde estão todos os meus alunos? Fugiram? Aparentemente essa *query* não está trazendo exatamente o que a gente esperava... Vamos contar a quantidade de alunos existentes:

```
SELECT COUNT(a.id) FROM aluno a;
```

COUNT(a.id)
16

Observe que existe 16 alunos no banco de dados, porém só foram retornados 4 alunos e suas respostas. Provavelmente não está sendo retornando os alunos que não possuem respostas! Vamos verificar o que está acontecendo exatamente. Vamos pegar um aluno que não foi retornado, como o de id 5. Quantas respostas ele tem?

```
SELECT r.id FROM resposta r WHERE r.aluno_id = 5;
```

id
----

```
|          0 |
+-----+
```

Tudo bem, ele não respondeu, mas como ele foi desaparecer daquela query nossa? Vamos pegar outro aluno que desapareceu, o de id 6:

```
SELECT r.id FROM resposta r WHERE r.aluno_id = 6;
```

```
+-----+
| id      |
+-----+
|          0 |
+-----+
```

Opa, parece que encontramos um padrão.

```
SELECT r.id FROM resposta r WHERE r.aluno_id = 7;
```

```
+-----+
| id      |
+-----+
|          0 |
+-----+
```

Sim, encontramos um padrão. Será que é verdadeiro essa teoria que está surgindo na minha cabeça? Alunos sem resposta desapareceram? Vamos procurar todos os alunos que não possuem nenhuma resposta. Isto é selecionar os alunos que não existe, resposta deste aluno:

```
SELECT a.nome FROM aluno a WHERE NOT EXISTS (SELECT r.id FROM resposta r WHERE r.aluno_id = a.id);
```

```
+-----+
| nome      |
+-----+
| Paulo da Silva |
| Carlos Cunha  |
| Paulo José    |
| Manoel Santos |
| Renata Ferreira |
| Paula Soares  |
| Jose da Silva  |
| Danilo Cunha   |
| Zilmira José   |
| Cristaldo Santos |
| Osmir Ferreira |
| Claudio Soares |
+-----+
```

Se verificarmos os nomes, realmente, todos os alunos que não tem respostas não estão sendo retornados naquela primeira query. Porém nós queremos também que retorne os alunos sem respostas... Vamos tentar de uma outra maneira, vamos retornar o nome do aluno e a resposta que ele respondeu:

```
SELECT a.nome, r.resposta_dada FROM aluno a
JOIN resposta r ON r.aluno_id = a.id;
```

```
+-----+-----+
-+
| nome      | resposta_dada |
|           |               |
```

```

+-----+-----+
-+
| João da Silva | uma selecao
|
| João da Silva | ixi, nao sei
|
| João da Silva | alterar dados
|
| João da Silva | esquecer o where e alterar tudo
|
| João da Silva | apagar coisas
|
| João da Silva | tb nao pode esquecer o where
|
| João da Silva | inserir dados
|
| Frederico José | buscar dados
|
| Frederico José | select campos from tabela
|
| Frederico José | alterar coisas
|
| Frederico José | ixi, nao sei
|
| Alberto Santos | tempo pra fazer algo
|
| Alberto Santos | 1 a 4 semanas
|
| Alberto Santos | melhoria do processo
|
| Alberto Santos | todo dia
|
| Alberto Santos | reuniao de status
|
| Alberto Santos | todo dia
|
| Alberto Santos | o quadro branco
|
| Alberto Santos | um metodo agil
|
| Alberto Santos | tem varios outros
|
| Renata Alonso | eh a internet
|
| Renata Alonso | browser faz requisicao, servidor manda resposta
|
| Renata Alonso | eh o servidor que lida com http
|
| Renata Alonso | nao sei
|
| Renata Alonso | banco de dados!
|
| Renata Alonso | eh colocar a app na internet
|
| Renata Alonso | depende da tecnologia, mas geralmente eh levar pra um servidor que ta na internet
+-----+-----+
-+

```

Agora vamos adicionar a coluna `id` da tabela `aluno` e `aluno_id` da tabela `resposta` :

```
SELECT a.id, a.nome, r.aluno_id, r.resposta_dada FROM aluno a
```

JOIN resposta r ON r.aluno\_id = a.id;

id	nome	aluno_id	resposta_dada
1	João da Silva	1	uma selecao
1	João da Silva	1	ixi, nao sei
1	João da Silva	1	alterar dados
1	João da Silva	1	eskecer o where e alterar tudo
1	João da Silva	1	apagar coisas
1	João da Silva	1	tb nao pode esquecer o where
1	João da Silva	1	inserir dados
2	Frederico José	2	buscar dados
2	Frederico José	2	select campos from tabela
2	Frederico José	2	alterar coisas
2	Frederico José	2	ixi, nao sei
3	Alberto Santos	3	tempo pra fazer algo
3	Alberto Santos	3	1 a 4 semanas
3	Alberto Santos	3	melhoria do processo
3	Alberto Santos	3	todo dia
3	Alberto Santos	3	reuniao de status
3	Alberto Santos	3	todo dia
3	Alberto Santos	3	o quadro branco
3	Alberto Santos	3	um metodo agil
3	Alberto Santos	3	tem varios outros
4	Renata Alonso	4	eh a internet
4	Renata Alonso	4	browser faz requisicao, servidor manda resposta
4	Renata Alonso	4	eh o servidor que lida com http
4	Renata Alonso	4	nao sei
4	Renata Alonso	4	banco de dados!
4	Renata Alonso	4	eh colocar a app na internet
4	Renata Alonso	4	depende da tecnologia, mas geralmente eh levar pra um servidor qu e ta na internet

-----+

O que o SQL faz também é que todos os alunos cujo o `id` não esteja na coluna `aluno_id` não serão retornados! Isto é, ele só trará para nós alguém que o `JOIN` tenha valor igual nas duas tabelas. Se só está presente em uma das tabelas, ele ignora.

```
SELECT a.id, a.nome, r.aluno_id, r.resposta_dada FROM aluno a
LEFT JOIN resposta r ON r.aluno_id = a.id;
```

3	Alberto Santos		3	reuniao de status
3	Alberto Santos		3	todo dia
3	Alberto Santos		3	o quadro branco
3	Alberto Santos		3	um metodo agil
3	Alberto Santos		3	tem varios outros
4	Renata Alonso		4	eh a internet
4	Renata Alonso		4	browser faz requisicao, servidor manda resposta
4	Renata Alonso		4	eh o servidor que lida com http
4	Renata Alonso		4	nao sei
4	Renata Alonso		4	banco de dados!
4	Renata Alonso		4	eh colocar a app na internet
4	Renata Alonso		4	depende da tecnologia, mas geralmente eh levar pra um servidor q
ue ta na internet				
5	Paulo da Silva		NULL	NULL
6	Carlos Cunha		NULL	NULL
7	Paulo José		NULL	NULL
8	Manoel Santos		NULL	NULL
9	Renata Ferreira		NULL	NULL
10	Paula Soares		NULL	NULL
11	Jose da Silva		NULL	NULL
12	Danilo Cunha		NULL	NULL
13	Zilmira José		NULL	NULL
14	Cristaldo Santos		NULL	NULL
15	Osmir Ferreira		NULL	NULL
16	Claudio Soares		NULL	NULL
+-----+			+-----+	
-----+				

```
SELECT a.nome, COUNT(r.id) AS respostas
FROM aluno a
LEFT JOIN resposta r ON r.aluno_id = a.id
GROUP BY a.nome;
```



Alberto Santos		9	
Carlos Cunha		0	
Claudio Soares		0	
Cristaldo Santos		0	
Danilo Cunha		0	
Frederico José		4	
João da Silva		7	
Jose da Silva		0	
Manoel Santos		0	
Osmir Ferreira		0	
Paula Soares		0	
Paulo da Silva		0	
Paulo José		0	
Renata Alonso		7	
Renata Ferreira		0	
Zilmira José		0	

+-----+-----+

Agora conseguimos retornar todos os alunos e a quantidade de respostas, mesmo que o aluno não tenha respondido pelo menos uma resposta.

## 12.1 RIGHT JOIN

Vamos supor que ao invés de retornar todos os alunos e suas respostas, mesmo que o aluno não tenha nenhuma resposta, queremos fazer o contrário, ou seja, retornar todos as respostas que foram respondidas e as que não foram respondidas. Vamos verificar se existe alguma resposta que não foi respondida por um aluno:

```
SELECT r.id FROM resposta r
WHERE r.aluno_id IS NULL;
```

Empty set (0,00 sec)

Não existe exercício sem resposta, então vamos inserir uma resposta sem associar a um aluno:

```
INSERT INTO resposta (resposta_dada) VALUES ('x vale 15.');
```

Query OK, 1 row affected (0,01 sec)

Se verificarmos novamente se existe uma resposta que não foi respondida por um aluno:

```
SELECT r.id FROM resposta r
WHERE r.aluno_id IS NULL;
```

```
+----+
| id |
+----+
| 28 |
+----+
```

Agora existe uma resposta que não foi associada a um aluno. Da mesma forma que utilizamos um JOIN diferente para pegar todos os dados da tabela da **esquerda (LEFT)** mesmo que não tenha associação com a tabela que está sendo juntada, existe também o JOIN que fará o procedimento, porém para a tabela da **direita (RIGHT)**, que é o tal do RIGHT JOIN :

```
SELECT a.nome, r.resposta_dada
FROM aluno a
RIGHT JOIN resposta r ON r.aluno_id = a.id;
```

```
+-----+-----+
-+
| nome          | resposta_dada
|
+-----+-----+
-+
| João da Silva | uma selecao
|
| João da Silva | ixi, nao sei
|
| João da Silva | alterar dados
|
| João da Silva | esquecer o where e alterar tudo
|
| João da Silva | apagar coisas
|
| João da Silva | tb nao pode esquecer o where
|
| João da Silva | inserir dados
|
| Frederico José | buscar dados
|
| Frederico José | select campos from tabela
|
| Frederico José | alterar coisas
|
| Frederico José | ixi, nao sei
|
| Alberto Santos | tempo pra fazer algo
|
| Alberto Santos | 1 a 4 semanas
|
| Alberto Santos | melhoria do processo
|
| Alberto Santos | todo dia
|
| Alberto Santos | reuniao de status
|
| Alberto Santos | todo dia
|
| Alberto Santos | o quadro branco
|
| Alberto Santos | um metodo agil
|
| Alberto Santos | tem varios outros
|
| Renata Alonso  | eh a internet
|
| Renata Alonso  | browser faz requisicao, servidor manda resposta
|
| Renata Alonso  | eh o servidor que lida com http
|
| Renata Alonso  | nao sei
|
| Renata Alonso  | banco de dados!
|
| Renata Alonso  | eh colocar a app na internet
|
| Renata Alonso  | depende da tecnologia, mas geralmente eh levar pra um servidor que ta na internet
```

	NULL	x vale 15
+-----+		
-+		

Observe que foi retornada a resposta em que não foi respondida por um aluno.

Quando utilizamos apenas o `JOIN` significa que queremos retornar todos os registros que tenham uma associação, ou seja, que exista tanto na tabela da esquerda quanto na tabela da direita, esse `JOIN` também é conhecido como `INNER JOIN`. Vamos verificar o resultado utilizando o `INNER JOIN`:

```
SELECT a.nome, COUNT(r.id) AS respostas
FROM aluno a
INNER JOIN resposta r ON r.aluno_id = a.id
GROUP BY a.nome;
```

+-----+	
nome	respostas
+-----+	
Alberto Santos	9
Frederico José	4
João da Silva	7
Renata Alonso	7
+-----+	

Ele trouxe apenas os alunos que possuem ao menos uma resposta, ou seja, que exista a associação entre a tabela da esquerda ( `aluno` ) e a tabela da direita ( `resposta` ).

## 12.2 JOIN OU SUBQUERY?

No capítulo anterior tivemos que fazer uma *query* para retornar todos os alunos e a quantidade de matrículas, porém utilizamos *subqueries* para resolver o nosso problema. Podemos também, construir essa *query* apenas com `JOIN` s. Vamos tentar:

```
SELECT a.nome, COUNT(m.id) AS qtd_matricula FROM aluno a
JOIN matricula m ON m.aluno_id = a.id
GROUP BY a.nome;
```

+-----+	
nome	qtd_matricula
+-----+	
Alberto Santos	2
Frederico José	3
João da Silva	2
Manoel Santos	2
Paula Soares	1
Paulo José	1
Renata Alonso	2
Renata Ferreira	1
+-----+	

Aparentemente não retornou os alunos que não possuem matrícula, porém utilizamos apenas o `JOIN`, ou seja, o `INNER JOIN`. Ao invés do `INNER JOIN` que retorna apenas se existir a associação entre as tabelas da esquerda ( `aluno` ) e a da direita( `matricula` ), nós queremos retornar todos os

alunos, mesmo que não possuam matrículas, ou seja, tabela da esquerda. Então vamos tentar agora com o `LEFT JOIN` :

```
SELECT a.nome, COUNT(m.id) AS qtd_matricula FROM aluno a
LEFT JOIN matricula m ON m.aluno_id = a.id
GROUP BY a.nome;
```

nome	qtd_matricula
Alberto Santos	2
Carlos Cunha	0
Claudio Soares	0
Cristaldo Santos	0
Danilo Cunha	0
Frederico José	3
João da Silva	2
Jose da Silva	0
Manoel Santos	2
Osmir Ferreira	0
Paula Soares	1
Paulo da Silva	0
Paulo José	1
Renata Alonso	2
Renata Ferreira	1
Zilmira José	0

Da mesma forma que conseguimos pegar todos os alunos e a quantidade de respostas mesmo que o aluno não tenha respondido nenhuma resposta utilizando o `LEFT JOIN` poderíamos também resolver utilizando uma *subquery* parecida com qual retornava todos os alunos e a quantidade de matrículas:

```
SELECT a.nome,
(SELECT COUNT(r.id) FROM resposta r WHERE r.aluno_id = a.id) AS respostas
FROM aluno a;
```

nome	respostas
João da Silva	7
Frederico José	4
Alberto Santos	9
Renata Alonso	7
Paulo da Silva	0
Carlos Cunha	0
Paulo José	0
Manoel Santos	0
Renata Ferreira	0
Paula Soares	0
Jose da Silva	0
Danilo Cunha	0
Zilmira José	0
Cristaldo Santos	0
Osmir Ferreira	0
Claudio Soares	0

O resultado é o mesmo! Se o resultado é o mesmo, quando eu devo utilizar o `JOIN` ou as *subqueries*? Aparentemente a *subquery* é mais enxuta e mais fácil de ser escrita, porém os SGBDs sempre terão um

desempenho melhor para `JOIN` em relação a *subqueries*, então prefira o uso de `JOIN` s ao invés de *subquery*.

Vamos tentar juntar as *queries* que fizemos agora pouco, porém em uma única *query*. Veja o exemplo em uma planilha.

aluno	qtd_respostas	qtd_matricula
Alex	2	2
Paulo	1	1
João	0	3

Figura 12.1: Planilha exemplo

Primeiro vamos fazer com *subqueries*. Então começaremos retornando o nome do aluno:

```
SELECT a.nome FROM aluno a;
```

Agora vamos contar todas as respostas e testar o resultado:

```
SELECT a.nome,
(SELECT COUNT(r.id) FROM resposta r WHERE r.aluno_id = a.id) AS qtd_respostas
FROM aluno a;
```

nome	qtd_respostas
João da Silva	7
Frederico José	4
Alberto Santos	9
Renata Alonso	7
Paulo da Silva	0
Carlos Cunha	0
Paulo José	0
Manoel Santos	0
Renata Ferreira	0
Paula Soares	0
Jose da Silva	0
Danilo Cunha	0
Zilmira José	0
Cristaldo Santos	0
Osmir Ferreira	0
Claudio Soares	0

Por fim, vamos contar as matrículas e retornar a contagem:

```
SELECT a.nome,
(SELECT COUNT(r.id) FROM resposta r WHERE r.aluno_id = a.id) AS qtd_respostas,
(SELECT COUNT(m.id) FROM matricula m WHERE m.aluno_id = a.id) AS qtd_matriculas
FROM aluno a;
```

nome	qtd_respostas	qtd_matriculas
João da Silva	7	2
Frederico José	4	3
Alberto Santos	9	2
Renata Alonso	7	2

Paulo da Silva		0		0	
Carlos Cunha		0		0	
Paulo José		0		1	
Manoel Santos		0		2	
Renata Ferreira		0		1	
Paula Soares		0		1	
Jose da Silva		0		0	
Danilo Cunha		0		0	
Zilmira José		0		0	
Cristaldo Santos		0		0	
Osmir Ferreira		0		0	
Claudio Soares		0		0	
+-----+					

Conseguimos o resultado esperado utilizando as *subqueries*, vamos tentar com o `LEFT JOIN` ? Da mesma forma que fizemos anteriormente, começaremos retornando os alunos:

```
SELECT a.nome FROM aluno a;
```

Agora vamos juntar as tabela `aluno` com as tabelas `resposta` e `matricula` :

```
SELECT a.nome, r.id AS qtd_respostas,
m.id AS qtd_matriculas
FROM aluno a
LEFT JOIN resposta r ON r.aluno_id = a.id
LEFT JOIN matricula m ON m.aluno_id = a.id;
```

+-----+					
nome		qtd_respostas		qtd_matriculas	
+-----+					
João da Silva		1		1	
João da Silva		2		1	
João da Silva		3		1	
João da Silva		4		1	
João da Silva		5		1	
João da Silva		6		1	
João da Silva		7		1	
Frederico José		8		2	
Frederico José		9		2	
Frederico José		10		2	
Frederico José		11		2	
Alberto Santos		12		3	
Alberto Santos		13		3	
Alberto Santos		14		3	
Alberto Santos		15		3	
Alberto Santos		16		3	
Alberto Santos		17		3	
Alberto Santos		18		3	
Alberto Santos		19		3	
Alberto Santos		20		3	
Renata Alonso		21		4	
Renata Alonso		22		4	
Renata Alonso		23		4	
Renata Alonso		24		4	
Renata Alonso		25		4	
Renata Alonso		26		4	
Renata Alonso		27		4	
Renata Alonso		21		9	
Renata Alonso		22		9	
Renata Alonso		23		9	
Renata Alonso		24		9	

Renata Alonso		25		9	
Renata Alonso		26		9	
Renata Alonso		27		9	
João da Silva		1		11	
João da Silva		2		11	
João da Silva		3		11	
João da Silva		4		11	
João da Silva		5		11	
João da Silva		6		11	
João da Silva		7		11	
Frederico José		8		12	
Frederico José		9		12	
Frederico José		10		12	
Frederico José		11		12	
Alberto Santos		12		13	
Alberto Santos		13		13	
Alberto Santos		14		13	
Alberto Santos		15		13	
Alberto Santos		16		13	
Alberto Santos		17		13	
Alberto Santos		18		13	
Alberto Santos		19		13	
Alberto Santos		20		13	
Frederico José		8		14	
Frederico José		9		14	
Frederico José		10		14	
Frederico José		11		14	
Paulo José		NULL		5	
Manoel Santos		NULL		6	
Renata Ferreira		NULL		7	
Paula Soares		NULL		8	
Manoel Santos		NULL		10	
Paulo da Silva		NULL		NULL	
Carlos Cunha		NULL		NULL	
Jose da Silva		NULL		NULL	
Danilo Cunha		NULL		NULL	
Zilmira José		NULL		NULL	
Cristaldo Santos		NULL		NULL	
Osmir Ferreira		NULL		NULL	
Claudio Soares		NULL		NULL	
+-----+-----+-----+-----+					

Antes de contarmos as colunas de qtd\_resposas e qtd\_matricula, vamos analisar um pouco esse resultado. Note que o aluno João da Silva retornou 14 vezes, parece que tem alguma coisa estranha. Vamos pegar o id do João da Silva e vamos verificar os registros dele na tabela resposta e na tabela matricula :

```
SELECT a.id FROM aluno a WHERE a.nome = 'João da Silva';
```

```
+-----+
| id |
+-----+
| 1 |
+-----+
```

Agora vamos verificar todos os registros dele na tabela resposta :

```
SELECT r.id FROM resposta r
WHERE r.aluno_id = 1;
```

```
+-----+
| id |
+-----+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
+-----+
```

Foram retornados 7 registros, agora vamos verificar na tabela `matricula` :

```
SELECT m.id FROM matricula m
WHERE m.aluno_id = 1;
```

```
+-----+
| id |
+-----+
| 1 |
| 11 |
+-----+
```

Foram retornados 2 registros. Se analisarmos um pouco esses três resultados chegamos aos seguintes números:

aluno = 1 respostas = 7 matrículas = 2

Vamos executar novamente a nossa *query* que retorna o aluno e a contagem de respostas e matrículas:

```
SELECT a.nome, r.id AS qtd_respostas,
m.id AS qtd_matriculas
FROM aluno a
LEFT JOIN resposta r ON r.aluno_id = a.id
LEFT JOIN matricula m ON m.aluno_id = a.id;
```

```
+-----+-----+-----+
| nome          | qtd_respostas | qtd_matriculas |
+-----+-----+-----+
| João da Silva | 1 | 1 |
| João da Silva | 2 | 1 |
| João da Silva | 3 | 1 |
| João da Silva | 4 | 1 |
| João da Silva | 5 | 1 |
| João da Silva | 6 | 1 |
| João da Silva | 7 | 1 |
| Frederico José | 8 | 2 |
| Frederico José | 9 | 2 |
| Frederico José | 10 | 2 |
| Frederico José | 11 | 2 |
| Alberto Santos | 12 | 3 |
| Alberto Santos | 13 | 3 |
| Alberto Santos | 14 | 3 |
| Alberto Santos | 15 | 3 |
| Alberto Santos | 16 | 3 |
| Alberto Santos | 17 | 3 |
| Alberto Santos | 18 | 3 |
| Alberto Santos | 19 | 3 |
```



Alberto Santos		20		3	
Renata Alonso		21		4	
Renata Alonso		22		4	
Renata Alonso		23		4	
Renata Alonso		24		4	
Renata Alonso		25		4	
Renata Alonso		26		4	
Renata Alonso		27		4	
Renata Alonso		21		9	
Renata Alonso		22		9	
Renata Alonso		23		9	
Renata Alonso		24		9	
Renata Alonso		25		9	
Renata Alonso		26		9	
Renata Alonso		27		9	
João da Silva		1		11	
João da Silva		2		11	
João da Silva		3		11	
João da Silva		4		11	
João da Silva		5		11	
João da Silva		6		11	
João da Silva		7		11	
Frederico José		8		12	
Frederico José		9		12	
Frederico José		10		12	
Frederico José		11		12	
Alberto Santos		12		13	
Alberto Santos		13		13	
Alberto Santos		14		13	
Alberto Santos		15		13	
Alberto Santos		16		13	
Alberto Santos		17		13	
Alberto Santos		18		13	
Alberto Santos		19		13	
Alberto Santos		20		13	
Frederico José		8		14	
Frederico José		9		14	
Frederico José		10		14	
Frederico José		11		14	
Paulo José		NULL		5	
Manoel Santos		NULL		6	
Renata Ferreira		NULL		7	
Paula Soares		NULL		8	
Manoel Santos		NULL		10	
Paulo da Silva		NULL		NULL	
Carlos Cunha		NULL		NULL	
Jose da Silva		NULL		NULL	
Danilo Cunha		NULL		NULL	
Zilmira José		NULL		NULL	
Cristaldo Santos		NULL		NULL	
Osmir Ferreira		NULL		NULL	
Claudio Soares		NULL		NULL	

Repare que a nossa *query* associando um aluno 1, com a resposta 1 e matrícula 1, o mesmo aluno 1, com resposta 1 e matrícula 11 e assim sucessivamente... Isso significa que essa query está multiplicando o aluno(1) x respostas(7) x matrículas(2)... Com certeza essa contagem não funcionará! Precisamos de resultados distintos, ou seja, iremos utilizar o `DISTINCT` para evitar esse problema. Agora podemos contar as respostas e as matrículas:

```

SELECT a.nome, COUNT(DISTINCT r.id) AS qtd_respostas,
COUNT(DISTINCT m.id) AS qtd_matriculas
FROM aluno a
LEFT JOIN resposta r ON r.aluno_id = a.id
LEFT JOIN matricula m ON m.aluno_id = a.id
GROUP BY a.nome;

```

nome	qtd_respostas	qtd_matriculas
Alberto Santos	9	2
Carlos Cunha	0	0
Claudio Soares	0	0
Cristaldo Santos	0	0
Danilo Cunha	0	0
Frederico José	4	3
João da Silva	7	2
Jose da Silva	0	0
Manoel Santos	0	2
Osmir Ferreira	0	0
Paula Soares	0	1
Paulo da Silva	0	0
Paulo José	0	1
Renata Alonso	7	2
Renata Ferreira	0	1
Zilmira José	0	0

E se não adicionássemos a instrução `DISTINCT` ? O que aconteceria? Vamos testar:

```

SELECT a.nome, COUNT(r.id) AS qtd_respostas,
COUNT(m.id) AS qtd_matriculas
FROM aluno a
LEFT JOIN resposta r ON r.aluno_id = a.id
LEFT JOIN matricula m ON m.aluno_id = a.id
GROUP BY a.nome;

```

nome	qtd_respostas	qtd_matriculas
Alberto Santos	18	18
Carlos Cunha	0	0
Claudio Soares	0	0
Cristaldo Santos	0	0
Danilo Cunha	0	0
Frederico José	12	12
João da Silva	14	14
Jose da Silva	0	0
Manoel Santos	0	2
Osmir Ferreira	0	0
Paula Soares	0	1
Paulo da Silva	0	0
Paulo José	0	1
Renata Alonso	14	14
Renata Ferreira	0	1
Zilmira José	0	0

O resultado além de ser bem maior que o esperado, repete nas duas colunas , pois está acontecendo aquele problema da multiplicação das linhas! Perceba que só conseguimos verificar de uma forma rápida o problema que aconteceu, pois fizemos a *query* **passo-a-passo**, verificando cada resultado e, ao mesmo

tempo, corrigindo os problemas que surgiam.

## 12.3 RESUMINDO

Neste capítulo aprendemos como utilizar os diferentes tipos de `JOIN`s, como por exemplo o `LEFT JOIN` que retorna os registros da tabela a esquerda e o `RIGHT JOIN` que retorna os da direita mesmo que não tenham associações. Vimos também que o `JOIN` também é conhecido como `INNER JOIN` que retorna apenas os registros que estão associados. Além disso, vimos que algumas *queries* podem ser resolvidas utilizando *subqueries* ou `LEFT/RIGHT JOIN`, porém é importante lembrar que os SGBDs sempre terão melhor desempenho com o os `JOIN`s, por isso é recomendado que utilize os `JOIN`s. Vamos para os exercícios?

## EXERCÍCIOS

1. Exiba todos os alunos e suas possíveis respostas. Exiba todos os alunos, mesmo que eles não tenham respondido nenhuma pergunta.
2. Exiba agora todos os alunos e suas possíveis respostas para o exercício com `ID = 1`. Exiba todos os alunos mesmo que ele não tenha respondido o exercício.

Lembre-se de usar a condição no `JOIN`.

1. Qual a diferença entre o `JOIN` convencional (muitas vezes chamado também de `INNER JOIN`) para o `LEFT JOIN`?

# MUITOS ALUNOS E O LIMIT

Precisamos de um relatório que retorne todos os alunos, algo como selecionar o nome de todos eles, com um `SELECT` simples:

```
SELECT a.nome FROM aluno a;
```

```
+-----+
| nome   |
+-----+
| João da Silva |
| Frederico José |
| Alberto Santos |
| Renata Alonso |
| Paulo da Silva |
| Carlos Cunha |
| Paulo José |
| Manoel Santos |
| Renata Ferreira |
| Paula Soares |
| Jose da Silva |
| Danilo Cunha |
| Zilmira José |
| Cristaldo Santos |
| Osmir Ferreira |
| Claudio Soares |
+-----+
```

Para melhorar o resultado podemos ordenar a *query* por ordem alfabética do nome:

```
SELECT a.nome FROM aluno a ORDER BY a.nome;
```

```
+-----+
| nome   |
+-----+
| Alberto Santos |
| Carlos Cunha |
| Claudio Soares |
| Cristaldo Santos |
| Danilo Cunha |
| Frederico José |
| João da Silva |
| Jose da Silva |
| Manoel Santos |
| Osmir Ferreira |
| Paula Soares |
| Paulo da Silva |
| Paulo José |
| Renata Alonso |
| Renata Ferreira |
| Zilmira José |
+-----+
```

+-----+

Vamos verificar agora quantos alunos estão cadastrados:

```
SELECT count(*) FROM aluno;
```

```
+-----+
| count(*) |
+-----+
|      16  |
+-----+
```

Como podemos ver, é uma quantidade relativamente baixa, pois quando estamos trabalhando em uma aplicação real, geralmente o volume de informações é muito maior.

No facebook, por exemplo, quantos amigos você tem? Quando você entra no facebook, aparece todas as atualizações dos seus amigos de uma vez? Todas as milhares de atualizações de uma única vez? Imagine a loucura que é trazer milhares de dados de uma única vez para você ver apenas 5, 10 notificações. Provavelmente vai aparecendo aos poucos, certo? Então que tal mostrarmos os alunos aos poucos também? Ou seja, fazermos uma **paginação** no nosso relatório, algo como 5 alunos "por página". Mas como podemos fazer isso? No MySQL, podemos **limitar** em 5 a quantidade de registros que desejamos retornar:

```
SELECT a.nome FROM aluno a
ORDER BY a.nome
LIMIT 5;
```

```
+-----+
| nome          |
+-----+
| Alberto Santos |
| Carlos Cunha  |
| Claudio Soares |
| Cristaldo Santos |
| Danilo Cunha  |
+-----+
```

Nesse caso retornamos os primeiros 5 alunos em ordem alfabética. O ato de limitar é extremamente importante a medida que os dados crescem. Se você tem mil mensagens antigas, não vai querer ver as mil de uma vez só, traga somente as 10 primeiras e, se tiver interesse, mais 10, mais 10 etc.

## 13.1 LIMITANDO E BUSCANDO A PARTIR DE UMA QUANTIDADE ESPECÍFICA

Agora vamos pegar os próximos 5 alunos, isto é, senhor MySQL, ignore os 5 primeiros, e depois pegue para mim os próximos 5:

```
SELECT a.nome FROM aluno a
ORDER BY a.nome
LIMIT 5,5;
```

```
+-----+
| nome          |
```

```

+-----+
| Frederico José |
| João da Silva  |
| Jose da Silva  |
| Manoel Santos  |
| Osmir Ferreira |
+-----+

```

LIMIT 5 ? LIMIT 5,5 ? Parece um pouco estranho, o que será que isso significa? Quando utilizamos o LIMIT funciona da seguinte maneira: LIMIT linha\_inicial, qtd\_de\_linhas\_para\_avançar, ou seja, quando fizemos LIMIT 5, informamos ao MySQL que avance 5 linhas apenas, pois por padrão ele iniciará pela primeira linha, chamada de linha 0. Se fizéssemos LIMIT 0,5, por exemplo:

```

SELECT a.nome FROM aluno a
ORDER BY a.nome
LIMIT 0,5;

```

```

+-----+
| nome          |
+-----+
| Alberto Santos |
| Carlos Cunha   |
| Claudio Soares |
| Cristaldo Santos |
| Danilo Cunha   |
+-----+

```

Perceba que o resultado é o mesmo que LIMIT 5 ! Se pedimos LIMIT 5,10 o que ele nos trás?

```

SELECT a.nome FROM aluno a
ORDER BY a.nome
LIMIT 5,10;

```

```

+-----+
| nome          |
+-----+
| Frederico José |
| João da Silva  |
| Jose da Silva  |
| Manoel Santos  |
| Osmir Ferreira |
| Paula Soares   |
| Paulo da Silva  |
| Paulo José     |
| Renata Alonso   |
| Renata Ferreira |
+-----+

```

O resultado iniciará após a linha 5, ou seja, linha 6 e avançará 10 linhas. Vamos demonstrar os exemplos todos de uma única vez:

Todos os alunos:

```

SELECT a.nome FROM aluno a;

```

```

+-----+
| nome          |
+-----+

```

```

+-----+
| Alberto Santos |
| Carlos Cunha  |
| Claudio Soares |
| Cristaldo Santos |
| Danilo Cunha  |
| Frederico José |
| João da Silva |
| Jose da Silva  |
| Manoel Santos |
| Osmir Ferreira |
| Paula Soares  |
| Paulo da Silva |
| Paulo José    |
| Renata Alonso  |
| Renata Ferreira |
| Zilmira José   |
+-----+

```

Pegando os 5 primeiros:

```

SELECT a.nome FROM aluno a
ORDER BY a.nome
LIMIT 5;

```

```

+-----+
| nome      |
+-----+
| Alberto Santos |
| Carlos Cunha  |
| Claudio Soares |
| Cristaldo Santos |
| Danilo Cunha  |
+-----+

```

Ignorando os 5 primeiros, pegando os próximos 5 alunos:

```

SELECT a.nome FROM aluno a
ORDER BY a.nome
LIMIT 5,5;

```

```

+-----+
| nome      |
+-----+
| Frederico José |
| João da Silva  |
| Jose da Silva  |
| Manoel Santos  |
| Osmir Ferreira |
+-----+

```

## 13.2 RESUMINDO

Nesse capítulo vimos como nem sempre retornar todos os registros das tabelas são necessários, algumas vezes, precisamos filtrar a quantidade de linhas, pois em uma aplicação real, podemos lidar com uma quantidade bem grande de dados. Justamente por esse caso, podemos limitar as nossas *queries* utilizando a instrução `LIMIT`. Vamos para os exercícios?

# EXERCÍCIOS

1. Escreva uma query que traga apenas os dois primeiros alunos da tabela.
2. Escreva uma SQL que devolva os 3 primeiros alunos que o e-mail termine com o domínio ".com".
3. Devolva os 2 primeiros alunos que o e-mail termine com ".com", ordenando por nome.
4. Devolva todos os alunos que tenham Silva em algum lugar no seu nome.