

# **SQL - STRUCTURED QUERY LANGUAGE**

Antonio Sergio Ferreira Bonato

Como citar este material:

BONATO, Antonio Sergio Ferreira. **SQL - Structured Query Language**. Rio de Janeiro: FGV, 2024.

Todos os direitos reservados. Textos, vídeos, sons, imagens, gráficos e demais componentes deste material são protegidos por direitos autorais e outros direitos de propriedade intelectual, de forma que é proibida a reprodução no todo ou em parte, sem a devida autorização.

# APRESENTAÇÃO DA LINGUAGEM SQL

SQL, ou *Structured Query Language*, é uma linguagem de programação utilizada para gerenciar e manipular dados em sistemas de gerenciamento de banco de dados relacionais (RDBMS). Desde a sua criação na década de 1970, a SQL se tornou uma ferramenta essencial para qualquer pessoa envolvida no desenvolvimento ou administração de bancos de dados.

O desenvolvimento recente da ciência de dados tem sido marcado pela explosão de dados disponíveis e pela necessidade crescente de extrair informações significativas desses volumes massivos de informações. Nesse cenário, a SQL desempenha um papel crucial como uma linguagem fundamental para manipular e analisar dados estruturados. Embora novas ferramentas e linguagens tenham surgido para lidar com dados não estruturados e complexos, a estrutura e a eficiência da SQL continuam sendo inigualáveis para consultas precisas, transformações de dados e preparação para análises avançadas. A sua capacidade de realizar operações complexas de junção, agregação e filtragem em bancos de dados relacionais e não relacionais não apenas facilita a exploração inicial de dados mas também garante a integridade e a consistência dos conjuntos de dados, sendo essencial para a criação de insights confiáveis na ciência de dados moderna.

Com a SQL, os usuários podem executar uma variedade de operações em um banco de dados, incluindo inserção, consulta, atualização e exclusão de dados. A sua sintaxe simples e poderosa permite que até mesmo iniciantes possam começar a trabalhar com bancos de dados com relativa facilidade.

Existem diferentes dialetos de SQL, cada um adaptado para um RDBMS específico, como MySQL, PostgreSQL, SQL Server, Oracle, entre outros. Embora haja diferenças entre esses dialetos, os conceitos fundamentais de SQL permanecem consistentes em todos eles.

As principais operações de SQL incluem:

1. **consulta de dados (SELECT)** – essa é uma das operações mais comuns na SQL, usada para recuperar dados de uma ou mais tabelas em um banco de dados. Os usuários podem especificar os critérios de seleção para filtrar e sumarizar os resultados conforme necessário. O comando **SELECT** e todas as suas variações são um subconjunto da SQL denominado DQL<sup>1</sup>, ou linguagem de consulta de dados;
2. **inserção de dados (INSERT)** – essa operação permite adicionar novas linhas de dados a uma tabela existente em um banco de dados;
3. **atualização de dados (UPDATE)** – a SQL oferece a capacidade de modificar os dados existentes em uma tabela com base em critérios específicos;
4. **exclusão de dados (DELETE)** – essa operação permite remover linhas de dados de uma tabela com base em determinados critérios;

O conjunto de operações **INSERT**, **UPDATE** e **DELETE** é denominado DML<sup>2</sup>, ou linguagem de manipulação de dados.

Além dessas operações básicas, a SQL também suporta uma variedade de outras funcionalidades avançadas, agrupadas sob a denominação DDL<sup>3</sup>, ou linguagem de definição de dados, como:

- **criação e modificação de esquemas** – os usuários podem criar e modificar a estrutura (ou esquema) de um banco de dados, incluindo tabelas, índices, visões e procedimentos armazenados;
- **restrições de integridade** – a SQL permite impor regras de integridade referencial para garantir a consistência dos dados;
- **transações** – a SQL suporta transações, que permitem agrupar um conjunto de operações de banco de dados em uma única unidade lógica e garantir a atomicidade, consistência, isolamento e durabilidade (Acid) dos dados;
- **controle de acesso** – os usuários podem controlar quem pode acessar e modificar os dados em um banco de dados, definindo permissões de acesso granulares.

Em resumo, a SQL é uma ferramenta poderosa e versátil para gerenciar dados em bancos de dados relacionais. Com uma compreensão básica da sua sintaxe e funcionalidades principais, os usuários podem realizar uma ampla variedade de tarefas relacionadas a dados de forma eficiente e eficaz.

---

<sup>1</sup> Data Query Language.

<sup>2</sup> Data Manipulation Language.

<sup>3</sup> Data Definition Language.

## BREVE INTRODUÇÃO AOS BANCOS DE DADOS RELACIONAIS

Bancos de dados relacionais são uma ferramenta fundamental na organização e gestão de informações estruturadas de maneira lógica e eficiente. Essa tecnologia tem suas raízes no desenvolvimento do modelo relacional por Edgar F. Codd, na década de 1970, um marco importante na história da computação e gestão de dados.

Desde então, sistemas de gerenciamento de banco de dados (SGBD) relacionais como o Oracle, MySQL, PostgreSQL e Microsoft SQL Server têm sido desenvolvidos e aprimorados continuamente. Eles se tornaram fundamentais não apenas na informática mas também em áreas como Economia, Administração, Biologia e muitas outras disciplinas acadêmicas e profissionais.

O modelo relacional propôs uma abordagem inovadora para armazenar dados em **tabelas**, de modo que cada tabela representa uma entidade específica, como clientes, produtos ou pedidos. Além disso, cada uma delas é composta por **colunas** que definem os tipos de informações que podem ser armazenadas, como nomes, datas ou valores. Por exemplo, em uma tabela de clientes, podemos ter colunas para nome, endereço e telefone.

Além da estrutura tabular, os bancos de dados relacionais introduzem o conceito de **relacionamento** entre tabelas. Isso permite conectar informações entre diferentes tabelas por meio de chaves. A **chave primária** é um campo (ou conjunto de campos) único em cada tabela, que identifica exclusivamente cada registro. Por exemplo, em uma tabela de clientes, a chave primária poderia ser um número de identificação único para cada cliente.

Já a **chave estrangeira** é um campo em uma tabela que estabelece uma relação com a chave primária de outra tabela. Isso possibilita criar conexões entre tabelas, como associar pedidos a clientes por meio das suas chaves primárias e estrangeiras correspondentes.

Esses conceitos são essenciais para garantir a integridade e consistência dos dados dentro de um banco de dados relacional, facilitando operações como consultas complexas e análises detalhadas, e garantindo que as informações sejam armazenadas de forma organizada e acessível.



# SUMÁRIO

<b>MÓDULO I – CONSULTANDO DADOS.....</b>	<b>9</b>
UNIDADE 1 – O ESSENCIAL SOBRE CONSULTAS.....	9
Comando SELECT.....	9
Tipos de dados .....	12
Operadores relacionais.....	13
UNIDADE 2 – OPERADORES ESPECIAIS E OPERADORES LÓGICOS.....	15
Operador IS NULL.....	15
Operador BETWEEN .....	17
Operadores lógicos.....	19
Operador IN.....	24
Operador LIKE .....	26
UNIDADE 3 – ORDENAÇÃO, LIMITAÇÃO E UNICIDADE .....	28
Ordenando dados.....	28
Limitando os resultados .....	32
Trazendo resultados únicos .....	33
<b>MÓDULO II – JUNTANDO DADOS .....</b>	<b>37</b>
UNIDADE 1 – JUNÇÕES INTERNAS E JUNÇÕES NATURAIS.....	38
Junção interna .....	38
Junção natural .....	42
UNIDADE 2 – JUNÇÕES EXTERNAS.....	43
Juntando dados à esquerda .....	44
Juntando dados à direita.....	46
Juntando dados em ambos os lados.....	48
Apelidos de tabelas e colunas.....	52
UNIDADE 3 – OPERAÇÕES DE CONJUNTOS.....	53
Unindo dados .....	53
Subtraindo dados .....	57
Intersecção de dados .....	59
<b>MÓDULO III – AGRUPANDO OS DADOS .....</b>	<b>61</b>
UNIDADE 1 – FUNÇÕES SUMARIZADORAS .....	61
Operadores aritméticos .....	61
Sumarizando dados.....	65
UNIDADE 2 – AGRUPANDO DADOS .....	67
Comando GROUP BY.....	67
Selecionando por valor agrupado .....	69
Contando linhas .....	71

UNIDADE 3 – <i>SUBQUERIES</i> .....	75
Cláusula EXISTS .....	78
<b>MÓDULO IV – MANIPULANDO DADOS.....</b>	<b>81</b>
UNIDADE 1 – COMANDOS DML .....	81
Inserindo novas linhas .....	81
Alterando linhas .....	84
Removendo linhas .....	85
UNIDADE 2 – TRANSAÇÕES.....	86
UNIDADE 3 – ALGUNS COMANDOS DDL .....	87
Criando tabelas .....	87
Alterando tabelas.....	89
Removendo tabelas.....	90
Limpendo tabelas .....	91
<b>REFERÊNCIAS BIBLIOGRÁFICAS .....</b>	<b>92</b>
<b>APÊNDICE I – DICIONÁRIO DE DADOS.....</b>	<b>94</b>
<b>PROFESSOR-AUTOR .....</b>	<b>97</b>

```
22 FROM
23 OrderHeader AS OrderH
24 INNER JOIN Customer AS Cust ON OrderH.custId = Cust.id
25 | INNER JOIN OrderDetail AS OrderD ON OrderH.orderId = OrderD.orderId
26 | INNER JOIN Item AS I ON OrderD.itemId = I.itemCode
27 WHERE OrderD.netPrice > 1000
```

## MÓDULO I – CONSULTANDO DADOS

Neste módulo, veremos o essencial para a realização de consultas em uma tabela única. Para isso, precisamos conhecer o comando **SELECT**, as cláusulas **FROM** e **WHERE**, bem como todos os operadores envolvidos na comparação de resultados para a realização de filtros de consulta. Veremos os tipos de dados e como ordenar os resultados, como eliminar linhas duplicadas e como limitar a quantidade de linhas retornada em uma coluna.

### UNIDADE 1 – O ESSENCIAL SOBRE CONSULTAS

Esta unidade apresenta o básico do comando **SELECT** e como filtrar linhas e colunas. Veremos também brevemente os tipos de dados que podem ser armazenados em tabelas de bancos de dados. Além disso, iremos conhecer os operadores relacionais, que servem para fazer a comparação entre campos e valores literais, ou seja, números, cadeias de caracteres, etc.

#### Comando **SELECT**

O comando **SELECT** é utilizado para a leitura dos dados gravados nas tabelas do banco de dados. Denominado **consulta** ou, mais habitualmente na sua versão inglesa **query**, tem, na sua forma mais simples, a seguinte sintaxe:

```
SELECT * FROM tabela
```

Os comandos SQL podem ser escritos em letras maiúsculas ou minúsculas. A linguagem não faz diferenciação de caso, isso é, não é *case sensitive*. Porém, é comum escrevermos as *queries* como no exemplo anterior: nomes de campos e tabelas em letras minúsculas, os comandos SQL em letras maiúsculas. Mas isso é apenas uma convenção.

O comando a seguir seleciona todas as linhas e todas as colunas de uma determinada tabela. Por exemplo, observe a *query* apresentada a seguir:

```
SELECT *
FROM co2_emissions_pc;
```

**Bloco de código 1.**

Veja que ela retorna todas as linhas e colunas da tabela chamada *co2\_emissions\_pc*, que contém as emissões de CO<sub>2</sub> *per capita* por país em um determinado ano (veja o dicionário de dados do banco utilizado como exemplo no Apêndice A).

Figura 1 – Resultado do bloco de código 1 (apenas as 10 primeiras linhas)

	RBC country	123 ref_year	123 co2_pc
1	Afghanistan	1,800	0.001
2	Afghanistan	1,801	0.001
3	Afghanistan	1,802	0.001
4	Afghanistan	1,803	0.001
5	Afghanistan	1,804	0.001
6	Afghanistan	1,805	0.001
7	Afghanistan	1,806	0.001
8	Afghanistan	1,807	0.001
9	Afghanistan	1,808	0.001
10	Afghanistan	1,809	0.001

É possível selecionar apenas algumas colunas da tabela, listando o nome das colunas no comando **SELECT**.

```
SELECT coluna_1, coluna_2, coluna_n FROM tabela
```

Por exemplo, veja a *query* seguinte:

```
SELECT co2_pc, country
FROM co2_emissions_pc;
```

**Bloco de código 2.**

Observe que ela retorna todas as linhas, mas apenas as colunas *co2\_pc* e *country* da tabela *co2\_emissions\_pc*. Note que a ordem das colunas pode ser especificada em uma ordem diferente da qual elas se encontram na tabela.

Figura 2 – Resultado do bloco de código 2 (apenas as 10 primeiras linhas)

	123 co2_pc	RBC country
1	0.001	Afghanistan
2	0.001	Afghanistan
3	0.001	Afghanistan
4	0.001	Afghanistan
5	0.001	Afghanistan
6	0.001	Afghanistan
7	0.001	Afghanistan
8	0.001	Afghanistan
9	0.001	Afghanistan
10	0.001	Afghanistan

Podemos aplicar filtros ao comando **SELECT** usando a cláusula **WHERE**, na qual especificamos uma condição lógica para ser aplicada como filtro e trazer apenas as linhas nas quais essa condição é verdadeira. Podemos, por exemplo, selecionar apenas os dados do Brasil na tabela *co2\_emissions\_pc*:

```
SELECT *
FROM co2_emissions_pc
WHERE country = 'Brazil';
```

#### Bloco de código 3.

Que tem como resultado:

Figura 3 – Resultado do bloco de código 3 (apenas as 10 primeiras linhas)

	RBC country	123 ref_year	123 co2_pc
1	Brazil	1,800	0.026
2	Brazil	1,801	0.027
3	Brazil	1,802	0.027
4	Brazil	1,803	0.027
5	Brazil	1,804	0.028
6	Brazil	1,805	0.028
7	Brazil	1,806	0.028
8	Brazil	1,807	0.029
9	Brazil	1,808	0.029
10	Brazil	1,809	0.029

Finalmente, podemos combinar a seleção de linhas e colunas em um único comando:

```
SELECT ref_year, co2_pc
FROM co2_emissions_pc
WHERE country = 'Brazil';
```

#### Bloco de código 4.

O resultado é o seguinte:

Figura 4 – Resultado do bloco de código 4 (apenas as 10 primeiras linhas)

	123 ref_year	123 co2_pc
1	1,800	0.026
2	1,801	0.027
3	1,802	0.027
4	1,803	0.027
5	1,804	0.028
6	1,805	0.028
7	1,806	0.028
8	1,807	0.029
9	1,808	0.029
10	1,809	0.029

## Tipos de dados

Um aspecto importante em qualquer linguagem de programação são os tipos de dados que ela consegue manipular. A gama de tipos é ampla e varia de uma implementação de bancos de dados para outra implementação. Todavia, existe um conjunto básico de tipos de dados com os quais é possível realizar quaisquer consultas em um banco de dados.

Os tipos, a sua descrição e como utilizá-los pode ser visto na tabela a seguir.

Tabela 1 – Tipos de dados em SQL

tipo	significado	como utilizar
caracter	Campo, sequência de caracteres ou <i>string</i> .	A <i>string</i> deve vir sempre entre aspas simples, nunca entre aspas duplas, como na frase ' <a href="#">um texto qualquer</a> '; nunca pule linhas "dentro" de uma <i>string</i> *; o conteúdo de uma sequência de caracteres é <i>case sensitive</i> **.
real ou numeric	Número decimal, positivo ou negativo.	O separador de decimais é sempre o . (ponto); nunca utilize separador de milhar; coloque um hífen na frente caso deseje digitar um número negativo, ex.: <a href="#">1.23</a> e <a href="#">-58.003</a> ; notação científica também é aceita: <a href="#">8.05e4</a> para o valor <a href="#">80500.0</a>
inteiro	Número inteiro, positivo ou negativo.	Apenas digite o número; coloque um hífen na frente caso deseje digitar um número negativo; exemplos: <a href="#">1000</a> e <a href="#">-12</a> .

tipo	significado	como utilizar
boolean	Tipo lógico, que assume valores verdadeiro ou falso.	Utilize a palavra-chave <code>true</code> para verdadeiro ou a palavra-chave <code>false</code> para falso; não faz diferença se as letras são maiúsculas ou minúsculas; como deve ter notado, não utilize aspas, senão o valor será interpretado como uma sequência de caracteres.
data	Tipo data de calendário.	Digite a data como se fosse uma <i>string</i> , utilizando ano com quatro dígitos, mês e dia, separando por hífen; por exemplo, <code>'2023-12-01'</code> .
tempo	Horas, minutos, segundos e milissegundos.	Digite a data como se fosse uma <i>string</i> , utilizando dois pontos para separar cada elemento do horário e ponto para separar os milissegundos , <code>'23:12:15.123'</code> .
timestamp	Campo com data e hora.	Digite a data e a hora juntas, como se fosse uma <i>string</i> , utilizando as regras da data e do tempo: <code>'2023-12-01 23:12:15.123'</code> .

\* Às vezes, o editor vai pular linha se a *string* for muito grande, mas isso não causa problemas. O problema ocorre quando se pressiona a tecla *enter* dentro da *string*.

\*\* Embora a linguagem SQL não diferencie maiúsculas de minúsculas, isso faz diferença dentro de uma *string*; se o valor `'Um texTo QualQuer'` estiver escrito desta forma no conteúdo de uma coluna, a consulta precisa ser escrita exatamente deste modo, caso contrário a linha desejada não será encontrada; no bloco de código 4, nenhuma linha teria sido retornada caso o valor digitado tivesse sido `'brazil'`.

## Operadores relacionais

Para criarmos uma regra para um filtro, que chamamos de expressão relacional ou condição lógica, utilizamos um operador relacional para comparar os dois elementos dessa condição, como o operador `=` (igual). Esses operadores são semelhantes aos que estamos acostumados a usar na matemática e tem o mesmo significado, como podemos ver na tabela a seguir:

Tabela 2 – Operadores relacionais

operador	significado
=	igual
>	maior
>=	maior ou igual
<	menor
<=	menor ou igual
!= ou <>	diferente (ambos formatos aceitos)

Já usamos o operador **=** nos exemplos anteriores, mas podemos usar os outros operadores para extrair diferentes informações do banco de dados. Por exemplo, se quisermos saber quais os países que atingiram população superior a 1 bilhão de habitantes e em que ano isso aconteceu, podemos usar o seguinte bloco de código:

```
SELECT country, ref_year, tot_pop
FROM population
WHERE tot_pop > 10000000000;
```

#### Bloco de código 5.

Como resultado, teremos:

Figura 5 – Resultado do bloco de código 5 (apenas as 5 primeiras linhas de cada país)

	ABC country	123 ref_year	123 tot_pop
1	China	1,982	1,010,000,000
2	China	1,983	1,030,000,000
3	China	1,984	1,040,000,000
4	China	1,985	1,060,000,000
5	China	1,986	1,080,000,000

	RBC country	123 ref_year	123 tot_pop
97	India	1,998	1,020,000,000
98	India	1,999	1,040,000,000
99	India	2,000	1,060,000,000
100	India	2,001	1,080,000,000
101	India	2,002	1,100,000,000

**Obs.:** a *query* do bloco de código 5 não garante que os dados venham ordenados por país e ano; para termos certeza que essa situação é verdadeira, é necessário indicar explicitamente no comando de consulta que queremos os dados ordenados. Veremos essa cláusula de ordenação da SQL mais adiante nesta apostila.

## UNIDADE 2 – OPERADORES ESPECIAIS E OPERADORES LÓGICOS

Na unidade 2, apresentaremos os operadores lógicos, utilizados para unir sentenças lógicas. Também veremos os operadores especiais da SQL: **IS NULL**, **BETWEEN**, **IN** e **LIKE**.

### Operador IS NULL

Os campos que não contêm nenhum valor são tratados, pelos bancos de dados, como campos nulos, ou seja, **NULL**. Não podemos comparar nada a **NULL**, de modo que os operadores **=** (igual) e **!=** (diferente) não funcionam com campos nulos.

Por exemplo, observe os comandos seguintes:

```
SELECT *
FROM population
WHERE tot_pop = NULL;
```

**Bloco de código 6.**

```
SELECT *
FROM population
WHERE tot_pop != NULL;
```

**Bloco de código 7.**

Veja que eles não retornam nenhuma linha. Ambas as expressões lógicas da cláusula **WHERE** dos blocos de código 6 e 7 são avaliadas como falsas.

Para realizarmos esse tipo de consulta, precisamos utilizar um operador relacional especial chamado **IS NULL**. Por exemplo, observe a consulta realizada com o bloco de código 8, cujo resultado é apresentado na tabela da figura 6:

```
SELECT *
FROM population
WHERE tot_pop IS NULL;
```

**Bloco de código 8.**

Figura 6 – Resultado do bloco de código 8 (apenas as 10 primeiras linhas)

	ABC country	123 ref_year	123 tot_pop
1	Holy See	2,001	[NULL]
2	Holy See	2,002	[NULL]
3	Holy See	2,003	[NULL]
4	Holy See	2,004	[NULL]
5	Holy See	2,005	[NULL]
6	Holy See	2,006	[NULL]
7	Holy See	2,007	[NULL]
8	Holy See	2,008	[NULL]
9	Holy See	2,009	[NULL]
10	Holy See	2,010	[NULL]

Observe ainda a consulta realizada com o bloco de código 9:

```
SELECT *
FROM population
WHERE tot_pop IS NOT NULL;
```

**Bloco de código 9.**

Veja que ela resulta em todos os países e anos, exceto a Santa Sé nos anos em que o valor do campo *tot\_pop* é nulo, como apresentado na tabela da figura 7:

Figura 7 – Resultado do bloco de código 9 (apenas as 10 primeiras linhas)

	RBC country	123 ref_year	123 tot_pop
1	Afghanistan	1,800	3,280,000
2	Afghanistan	1,801	3,280,000
3	Afghanistan	1,802	3,280,000
4	Afghanistan	1,803	3,280,000
5	Afghanistan	1,804	3,280,000
6	Afghanistan	1,805	3,280,000
7	Afghanistan	1,806	3,280,000
8	Afghanistan	1,807	3,280,000
9	Afghanistan	1,808	3,280,000
10	Afghanistan	1,809	3,280,000

## Operador BETWEEN

Quando queremos fazer o filtro não por um valor, mas por uma faixa de valores, podemos utilizar o operador **BETWEEN**.

Por exemplo, se for necessário consultar quais são os países e em quais anos a mortalidade infantil se encontra no intervalo  $600 \pm 5\%$ , isto é, entre 570 e 630 mortes de crianças entre 0 e 5 anos a cada 1000 nascidos vivos. A consulta é esta:

```
SELECT country, ref_year, tot_deaths
FROM child_mortality
WHERE tot_deaths BETWEEN 570 AND 630;
```

**Bloco de código 10.**

Obtemos como resultado que Cuba, Irlanda, Islândia, no século XIX, e Barbados, Paquistão e Ucrânia, no século XX, possuíam a elevada taxa de 60% de mortalidade infantil, como mostra a tabela da figura a seguir:

Figura 8 – Resultado do bloco de código 10 (apenas as 10 primeiras linhas)

	RBC country	123 ref_year	123 tot_deaths
30	Barbados	1,919	572
31	Barbados	1,923	598
32	Cuba	1,896	575
33	Cuba	1,897	589
34	Cuba	1,898	604
35	Cuba	1,899	621
36	Ireland	1,848	587
37	Ireland	1,849	598
38	Iceland	1,846	608
39	Iceland	1,882	588
40	Pakistan	1,921	570
41	Ukraine	1,943	577

O operador **BETWEEN** pode ser utilizado com qualquer tipo de dado: caractere, data e numérico (inteiro e real). Esse operador também funciona com tipos de dados lógicos, aqueles que assumem valores TRUE (verdadeiro) e FALSE (falso). Porém não faz sentido utilizá-lo para essa operação, pois a avaliação da expressão retorna sempre falso, uma vez que não há nada entre TRUE e FALSE.

Veja, a seguir, um exemplo de uso com campos do tipo data:

```
SELECT "date", open, high, low, close, "adj close", volume
FROM petrobras
WHERE "date" BETWEEN '2020-02-20' AND '2020-03-20';
```

#### Bloco de código 11.

Observe que o campo **"adj close"** está entre aspas duplas. Elas são necessárias por causa do espaço em branco dentro do nome do campo, que seria interpretado como um erro pela linguagem SQL se não estivesse entre aspas. As aspas também devem ser utilizadas caso o editor de SQL reconheça o nome do campo como sendo uma palavra reservada da SQL, como o campo **"date"**, pois essa é a palavra reservada para indicar o tipo data de calendário.

O resultado da consulta são todas as linhas que estão entre as datas da expressão lógica, inclusive as próprias datas. Observe a queda do valor em USD das ações da Petrobrás na Nasdaq neste período devido ao início dos *lockdowns* decorrentes da pandemia de Covid-19.

Figura 9 – Resultado do bloco de código 11 (todas as linhas)

	RBC Date	123 Open	123 High	123 Low	123 Close	123 Adj Close	123 Volume
1	2020-02-20	14.77	14.93	14.37	14.4	5.303012	21,453,600
2	2020-02-21	14.17	14.18	13.89	14.03	5.166753	22,637,200
3	2020-02-24	13.14	13.28	12.71	13.08	4.816903	38,318,000
4	2020-02-25	13.2	13.3	12.62	12.82	4.721154	26,389,700
5	2020-02-26	12.84	13.1	12.45	12.55	4.621723	41,263,500
6	2020-02-27	12	12.55	11.71	12.08	4.448638	41,795,900
7	2020-02-28	11.56	12.11	11.51	12.1	4.456003	46,881,600
8	2020-03-02	12.2	12.57	12.02	12.49	4.599626	29,931,200
9	2020-03-03	12.51	12.91	12	12.16	4.478099	37,360,900
10	2020-03-04	12.45	12.45	12.1	12.33	4.540704	23,901,800
11	2020-03-05	12.01	12.05	11.31	11.74	4.323427	42,527,900
12	2020-03-06	10.7	10.87	10.27	10.52	3.874145	52,078,200
13	2020-03-09	7.55	8.26	7.04	7.26	2.673602	95,745,000
14	2020-03-10	8.5	8.57	7.35	8.01	2.9498	54,755,700
15	2020-03-11	7.52	7.77	6.64	6.85	2.522614	55,715,500
16	2020-03-12	5.75	5.89	4.83	5.45	2.007043	86,571,700
17	2020-03-13	6.69	6.74	5.57	6.58	2.423182	59,809,100
18	2020-03-16	5.4	5.96	5.13	5.25	1.93339	35,275,300
19	2020-03-17	5.41	5.66	5.14	5.4	1.988629	51,896,200
20	2020-03-18	4.56	4.82	4.01	4.31	1.587221	49,017,000
21	2020-03-19	4.4	5.12	4.1	5.06	1.863419	56,339,600
22	2020-03-20	5.34	5.39	4.72	4.85	1.786084	38,227,800

## Operadores lógicos

Os operadores lógicos unem duas expressões relacionais, e, assim como estas, retornam sempre um valor booleano (verdadeiro ou falso). Os operadores lógicos estão na tabela a seguir.

Tabela 3 – Operadores lógicos

operador lógico	significado
AND	e
OR	ou
NOT	negação

O operador **AND** retorna verdadeiro se ambas as expressões lógicas forem verdadeiras. Caso uma delas seja falsa, ou ambas, a expressão resultante retorna falso. Veja a tabela-verdade do **AND**:

Tabela 4 – Tabela-verdade do operador AND

AND	VERDADEIRO	FALSO
VERDADEIRO	verdadeiro	falso
FALSO	falso	falso

Por exemplo, queremos selecionar todos os países que emitiram mais de 20 toneladas de CO<sub>2</sub> *per capita* no ano de 2020:

```
SELECT country, co2_pc
FROM co2_emissions_pc
WHERE ref_year = 2020
AND co2_pc > 20.0;
```

Bloco de código 12.

Na figura 10, vemos que Brunei, Kuwait, Catar, três economias baseadas na extração de petróleo, e Cingapura, foram os maiores emissores de CO<sub>2</sub> no ano pesquisado.

Figura 10 – Resultado do bloco de código 12 (todas as linhas)

	ABC country	123 co2_pc
1	Brunei	22.4
2	Kuwait	22.1
3	Qatar	25.6
4	Singapore	24.1

Já o operador **OR** atua de maneira oposta ao **AND**. Ele retorna falso apenas se ambas as expressões lógicas forem falsas. Caso contrário, retorna verdadeiro, como pode ser visto na tabela-verdade a seguir:

Tabela 5 – Tabela-verdade do operador OR

OR	VERDADEIRO	FALSO
VERDADEIRO	verdadeiro	verdadeiro
FALSO	verdadeiro	falso

Suponha que precisemos selecionar os países das Américas segundo a classificação de regiões do Banco Mundial:

```
SELECT country, wb_regions
FROM country
WHERE wb_regions = 'North America'
OR wb_regions = 'Latin America & Caribbean';
```

Bloco de código 13.

O resultado é o seguinte:

Figura 11 – Resultado do bloco de código 13 (10 últimas linhas)

	ABC country	ABC wb_regions
25	Paraguay	Latin America & Caribbean
26	Peru	Latin America & Caribbean
27	St. Kitts and Nevis	Latin America & Caribbean
28	St. Lucia	Latin America & Caribbean
29	St. Vincent and the Gr	Latin America & Caribbean
30	Suriname	Latin America & Caribbean
31	Trinidad and Tobago	Latin America & Caribbean
32	Uruguay	Latin America & Caribbean
33	Venezuela	Latin America & Caribbean
34	Canada	North America
35	USA	North America

Por sua vez, o operador **NOT** inverte o resultado da expressão lógica. Se ela for verdadeira, esse operador a torna false, e vice-versa. Veja a tabela-verdade:

Tabela 6 – Tabela-verdade do operador NOT

NOT	VERDADEIRO	FALSO
	falso	verdadeiro

Se quisermos, agora, todos os países do mundo com exceção dos países das Américas, segundo a mesma classificação do Banco Mundial, podemos fazer da seguinte forma:

```
SELECT country, wb_regions
FROM country
WHERE NOT (wb_regions = 'North America'
OR wb_regions = 'Latin America & Caribbean');
```

Bloco de código 14.

Fazemos a mesma *query*, mas adicionamos **NOT** antes da expressão lógica. Observe que é necessário colocar toda a expressão lógica entre parênteses, caso contrário apenas a parte `wb_regions = 'North America'` seria negada e os países da América Latina e do Caribe seriam incluídos na consulta.

Figura 12 – Resultado do bloco de código 14 (10 primeiras linhas)

	RBC country	RBC wb_regions
1	Afghanistan	South Asia
2	Albania	Europe & Central Asia
3	Algeria	Middle East & North Africa
4	Andorra	Europe & Central Asia
5	Angola	Sub-Saharan Africa
6	Armenia	Europe & Central Asia
7	Australia	East Asia & Pacific
8	Austria	Europe & Central Asia
9	Azerbaijan	Europe & Central Asia
10	Bahrain	Middle East & North Africa

Uma consulta SQL pode apresentar não apenas duas expressões lógicas mas uma série delas conectadas por operadores lógicos. Nesse caso, a expressão é interpretada da esquerda para direita, a não ser que o uso de parênteses altere essa ordem de interpretação, pois o que está entre parênteses tem precedência. O bloco de código 14, apresentado anteriormente, é um exemplo disso.

Outro exemplo seria retomarmos a consulta do bloco de código 10, aquela sobre mortalidade infantil, mas agora reduzindo o intervalo para  $60 \pm 5\%$ , isto é, entre 57 e 63 mortes de crianças entre 0 e 5 anos a cada 1000 nascidos vivos e restringindo os países a China e Brasil. Queremos, então, saber em quais anos esses países estiveram nessa faixa de mortalidade infantil.

Observe, como exemplo, o bloco de código a seguir:

```
SELECT country, ref_year, tot_deaths
FROM child_mortality
WHERE tot_deaths BETWEEN 57 AND 63
AND country = 'China' OR country = 'Brazil'
```

**Bloco de código 15.**

Se fizermos essa consulta, iremos obter o resultado a seguir, que está claramente incorreto, pois *tot\_deaths* é 417, totalmente fora da faixa que pesquisamos. Veja:

Figura 13 – Resultado do bloco de código 15 (13 últimas linhas)

	<code>RBC country</code>	<code>123 ref_year</code>	<code>123 tot_deaths</code>
292	Brazil	2,091	3.23
293	Brazil	2,092	3.19
294	Brazil	2,093	3.15
295	Brazil	2,094	3.11
296	Brazil	2,095	3.07
297	Brazil	2,096	3.03
298	Brazil	2,097	3
299	Brazil	2,098	2.96
300	Brazil	2,099	2.92
301	Brazil	2,100	2.92
302	China	1,980	62.5
303	China	1,981	59.7
304	China	1,982	57.5

Então, o que pode ter acontecido de errado? É simples. Como a expressão é analisada da esquerda para a direita, todas as linhas nas quais `tot_deaths` estava entre 57 e 63, e o país era China, apareceram dentro do que esperávamos. Mas a outra expressão lógica foi analisada em separado, e todas as linhas cujo país era Brasil apareceram também, mesmo com `tot_deaths` estando fora da faixa.

Mas não é isso que queremos, e sim as linhas em que o país é Brasil ou China e `tot_deaths` está entre 57 e 63. Para isso, temos que usar parênteses para mudar a ordem de avaliação e priorizar a cláusula `country = 'China' OR country = 'Brazil'` e, para as linhas onde ela for verdadeira, trazer somente as que estiverem dentro da faixa de mortes procurada, como ilustrado a seguir:

```
SELECT country, ref_year, tot_deaths
FROM child_mortality
WHERE tot_deaths BETWEEN 57 AND 63
AND (country = 'China' OR country = 'Brazil')
```

**Bloco de código 16.**

O resultado, então, passa a ser:

Figura 14 – Resultado do bloco de código 16 (todas as linhas)

	RBC country	123 ref_year	123 tot_deaths
1	Brazil	1,990	63
2	Brazil	1,991	60.2
3	Brazil	1,992	57.2
4	China	1,980	62.5
5	China	1,981	59.7
6	China	1,982	57.5

Desse modo, vemos que a China atingiu o valor de  $60 \pm 5\%$  mortes de crianças de 0 e 5 anos a cada 1000 nascidos vivos 10 anos antes do Brasil.

## Operador IN

O operador **IN** equivale ao operador  $\in$ , utilizado para denotar que um elemento pertence a um conjunto.

Sendo assim, ele pode ser utilizado para perguntar se o valor de uma coluna da tabela está em uma lista de elementos. Por exemplo, a consulta do bloco 16 poderia ser escrita assim:

```
SELECT country, ref_year, tot_deaths
FROM child_mortality
WHERE tot_deaths BETWEEN 57 AND 63
AND country IN ('China', 'Brazil')
```

Bloco de código 17.

O resultado seria o mesmo da figura 14, mas dessa vez não foi necessário o uso de parênteses para juntar as expressões lógicas, apenas para descrever a lista de países. Desse modo, o código SQL fica muito mais legível, principalmente quando temos muitas condições lógicas na cláusula **WHERE**.

Outro uso comum do operador **IN** é para substituir uma sequência de **OR**. Por exemplo, se quisermos saber a emissão de CO<sub>2</sub> *per capita* no ano de 2022 nos países do G7 podemos fazer assim:

```

SELECT country, co2_pc
FROM co2_emissions_pc
WHERE ref_year = 2022
AND (country = 'Canada'
OR country = 'France'
OR country = 'Germany'
OR country = 'Italy'
OR country = 'Japan'
OR country = 'UK'
OR country = 'USA');

```

**Bloco de código 18.**

No entanto, isso pode-se tornar uma tarefa tediosa se a lista for muito grande dada a repetição de *country* e de **OR**. Nesse caso, podemos fazer da seguinte forma:

```

SELECT country, co2_pc
FROM co2_emissions_pc
WHERE ref_year = 2022
AND country IN ('Canada', 'France', 'Germany',
'Italy', 'Japan', 'UK', 'USA');

```

**Bloco de código 19.**

Para ambas as *queries* apresentadas, o resultado é o seguinte:

Figura 15 – Resultado dos blocos de código 18 e 19 (todas as linhas)

	ABC country ▾	123 co2_pc ▾
1	Canada	13.2
2	Germany	9.3
3	France	6.07
4	UK	6.93
5	Italy	6.92
6	Japan	9.99
7	USA	16.4

Observe que o código do bloco 19 fica mais simples e mais legível.

## Operador LIKE

O operador **LIKE** procura por ocorrências de cadeias de caracteres dentro de campos do tipo caractere (*string*). Associado ao operador **LIKE**, utilizamos o símbolo, também chamado *token*, **%**, para indicar se queremos que a cadeia de caracteres seja procurada no início, no final ou em qualquer posição do campo caractere-alvo.

Veja os exemplos:

- a) Busca pela cadeia de caracteres '**Congo%**' no início do nome do país na tabela de fertilidade, que contém a média de filhos por mulher em um determinado ano e país:

```
SELECT country, mean_babies
FROM fertility
WHERE ref_year = 2021
AND country LIKE 'Congo%';
```

Bloco de código 20.

Queremos saber a taxa de fertilidade nos dois Congos em 2021. O resultado é o seguinte:

Figura 16 – Resultado do bloco de código 20 (todas as linhas)

	ABC country	123 mean_babies
1	Congo, Dem. Rep.	5.62
2	Congo, Rep.	4.37

- b) Agora estamos interessados na taxa de fertilidade das Guinés em 2021. Procuramos, então, a sequência '**%Guinea%**' em qualquer lugar da *string*.

```
SELECT country, mean_babies
FROM fertility
WHERE ref_year = 2021
AND country LIKE '%Guinea%';
```

Bloco de código 21.

Temos quatro países no mundo com Guiné no nome:

Figura 17 – Resultado do bloco de código 21 (todas as linhas)

	RBC country	123 mean_babies
1	Guinea	4.48
2	Guinea-Bissau	4.27
3	Equatorial Guinea	4.26
4	Papua New Guinea	3.43

- c) Finalmente, queremos a taxa de fertilidade de todos os países terminados em "lândia" em 2021. Procuramos, então, a sequência '**%land**' no final da *string*.

```
SELECT country, mean_babies
FROM fertility
WHERE ref_year = 2021
AND country LIKE '%land';
```

Bloco de código 22.

Temos oito países terminados em "lândia" no mundo:

Figura 18 – Resultado do bloco de código 22 (todas as linhas)

	RBC country	123 mean_babies
1	Switzerland	1.56
2	Finland	1.79
3	Greenland	[NULL]
4	Ireland	1.97
5	Iceland	1.89
6	New Zealand	1.94
7	Poland	1.29
8	Thailand	1.42

Observe que, ao contrário dos outros operadores, o **LIKE** é *case insensitive*, ou seja, não faz diferença de letras maiúsculas e minúsculas. Por exemplo, na consulta do bloco de código 20, as mesmas linhas seriam retornadas se tivéssemos usado '**congo%**' em vez de '**Congo%**'.

**Dica de performance:** evite utilizar o operador **LIKE** quando puder substituí-lo por um **OR** ou um **IN**, pois além da busca linha a linha que é feita na tabela também é feita uma busca em cada *string* para encontrar a sequência de caracteres procurada. Claro que para tabelas pequenas não há impacto, mas quando existem milhões de linhas envolvidas, e a busca é repetitiva, o custo computacional se torna proibitivo.

Na verdade, vale a pena usar o **LIKE** quando não conhecemos bem os valores do campo de busca, como no exemplo que procuramos países terminados em '**%land**'. Seria bastante trabalhoso escrever todos eles em uma lista para usar o **IN** e teríamos que sabê-los *a priori*. Porém as buscas solicitadas nos exercícios a seguir são mais fáceis e efetivas usando o operador **IN**, como você irá perceber ao fazê-los.

## UNIDADE 3 – ORDENAÇÃO, LIMITAÇÃO E UNICIDADE

Na unidade 3, veremos como garantir a ordem dos dados nos resultados da consultas, aprenderemos a limitar a quantidade de linhas retornadas em uma consulta e veremos como remover linhas duplicadas quando realizamos uma consulta.

### Ordenando dados

Como vimos na observação do bloco de código 5, o comando **SELECT** não garante a ordem em que os dados são exibidos, a não ser que declaremos isso explicitamente por meio da cláusula **ORDER BY**.

O **ORDER BY** ordena os dados em ordem crescente: números do menor para o maior, caracteres em ordem alfabética do A para o Z, datas da mais antiga para a mais nova, booleano do falso para o verdadeiro.

Retomando a consulta do bloco 5, podemos querer saber quais os países menos populosos em um determinado ano, por exemplo, no ano 2000.

```
SELECT country, tot_pop
FROM population
WHERE ref_year = 2000
ORDER BY tot_pop;
```

Bloco de código 23.

Vemos que o Vaticano é o país menos populoso, seguido de ilhas da Oceania, como Tuvalu, Nauru e Palau. Mas dessa vez sabemos com certeza, pois o **ORDER BY** garante que os dados vem ordenados em ordem crescente.

Figura 19 – Resultado do bloco de código 23 (10 primeiras linhas)

	RBC country	123 tot_pop
1	Holy See	779
2	Tuvalu	9,640
3	Nauru	10,400
4	Palau	19,700
5	San Marino	26,800
6	Monaco	32,500
7	Liechtenstein	33,000
8	St. Kitts and Nevis	45,500
9	Marshall Islands	54,200
10	Andorra	66,100

Podemos ainda querer saber os países mais populosos. A cláusula **DESC** inverte a ordem, ordenando os números do maior para o menor, as *strings* de Z para A, as datas da mais nova para a mais antiga e os booleanos do verdadeiro para o falso:

```
SELECT country, tot_pop
FROM population
WHERE ref_year = 2000
ORDER BY tot_pop DESC;
```

#### Bloco de código 24.

Como esperado, China e Índia ocupam as primeiras posições.

Figura 20 – Resultado do bloco de código 24 (10 primeiras linhas)

	RBC country	123 tot_pop
1	China	1,260,000,000
2	India	1,060,000,000
3	USA	282,000,000
4	Indonesia	214,000,000
5	Brazil	176,000,000
6	Pakistan	154,000,000
7	Russia	147,000,000
8	Bangladesh	129,000,000
9	Japan	127,000,000
10	Nigeria	123,000,000

Retomemos agora a pergunta que originou a *query* do bloco 5: Quais os países que atingiram população superior a 1 bilhão de habitantes e em que ano isso aconteceu?

```
SELECT country, ref_year, tot_pop
FROM population
WHERE tot_pop > 1000000000
ORDER BY country, ref_year;
```

**Bloco de código 25.**

Fazendo uma rolagem da tabela resultante, vemos que a China atingiu essa população em 1982, e a Índia em 1998. Note que agora utilizamos dois campos na cláusula **ORDER BY**. Podemos usar quantos precisarmos, separando um do outro por vírgulas.

Figura 21 – Resultado do bloco de código 25 (excerto das 5 primeiras linhas da China e das 5 primeiras linhas da Índia)

	abc country	123 ref_year	123 tot_pop
1	China	1,982	1,010,000,000
2	China	1,983	1,030,000,000
3	China	1,984	1,040,000,000
4	China	1,985	1,060,000,000
5	China	1,986	1,080,000,000
97	India	1,998	1,020,000,000
98	India	1,999	1,040,000,000
99	India	2,000	1,060,000,000
100	India	2,001	1,080,000,000
101	India	2,002	1,100,000,000

**Obs.:** essa consulta ainda não é a ideal, pois temos que rolar a tabela para procurar as informações, mas pelo menos estamos certos de que o primeiro ano que aparece é realmente o primeiro; mais adiante, nesta apostila, veremos como filtrar somente a primeira linha de cada país.

Podemos utilizar a cláusula **DESC** para inverter a ordem de apenas um campo:

```
SELECT country, ref_year, tot_pop
FROM population
WHERE tot_pop > 1000000000
ORDER BY country DESC, ref_year;
```

**Bloco de código 26.**

O resultado agora traz primeiro a Índia e depois a China.

Figura 22 – Resultado do bloco de código 26 (excerto das 5 primeiras linhas da Índia e das 5 primeiras linhas da China)

	RBC country	123 ref_year	123 tot_pop
1	India	1,998	1,020,000,000
2	India	1,999	1,040,000,000
3	India	2,000	1,060,000,000
4	India	2,001	1,080,000,000
5	India	2,002	1,100,000,000
...			
104	China	1,982	1,010,000,000
105	China	1,983	1,030,000,000
106	China	1,984	1,040,000,000
107	China	1,985	1,060,000,000
108	China	1,986	1,080,000,000

Se quisermos inverter a ordem dos dois campos, temos de utilizar **DESC** em cada um deles.

```
SELECT country, ref_year, tot_pop
FROM population
WHERE tot_pop > 10000000000
ORDER BY country DESC, ref_year DESC;
```

#### Bloco de código 27.

O resultado agora traz primeiro a Índia e depois a China.

Figura 23 – Resultado do bloco de código 27 (excerto das 5 primeiras linhas da Índia e das 5 primeiras linhas da China)

	RBC country	123 ref_year	123 tot_pop
1	India	2,100	1,530,000,000
2	India	2,099	1,540,000,000
3	India	2,098	1,540,000,000
4	India	2,097	1,550,000,000
5	India	2,096	1,560,000,000
...			
104	China	2,077	1,010,000,000
105	China	2,076	1,020,000,000
106	China	2,075	1,030,000,000
107	China	2,074	1,040,000,000
108	China	2,073	1,050,000,000

## Limitando os resultados

Nos exemplos e exercícios anteriores, quase sempre mostramos as 10 primeiras linhas ou as cinco últimas linhas dos resultados obtidos. Mas isso foi feito usando-se o editor de imagens com a intenção de mostrar os resultados nesta apostila, pois a consulta retornou milhares de linhas. A tabela *population*, por exemplo, tem 59.297 linhas.

Entretanto, é possível limitar o número de linhas retornadas na consulta SQL por meio da cláusula **LIMIT**. Por exemplo, se quisermos trazer apenas os 20 países mais populosos no ano 2000, fazemos a seguinte *query*:

```
SELECT country, tot_pop
FROM population
WHERE ref_year = 2000
ORDER BY tot_pop DESC
LIMIT 10;
```

Bloco de código 28.

Para esse bloco de código, o resultado é:

Figura 24 – Resultado do bloco de código 28 (todas as linhas)

	RBC country ▾	123 tot_pop ▾
1	China	1,260,000,000
2	India	1,060,000,000
3	USA	282,000,000
4	Indonesia	214,000,000
5	Brazil	176,000,000
6	Pakistan	154,000,000
7	Russia	147,000,000
8	Bangladesh	129,000,000
9	Japan	127,000,000
10	Nigeria	123,000,000

O **LIMIT** também é usado quando estamos explorando os dados e queremos dar uma olhada no conteúdo de uma tabela muito grande. Nesse caso, para economizar tempo, limitamos o resultado às cinco primeiras linhas, por exemplo:

```
SELECT *
FROM petrobras
LIMIT 5;
```

#### Bloco de código 29.

O resultado é:

Figura 25 – Resultado do bloco de código 29 (todas as linhas)

	RBC Date	123 Open	123 High	123 Low	123 Close	123 Adj Close	123 Volume
1	2000-08-11	6.84375	7.28125	6.75	7.140625	1.527205	25,139,200
2	2000-08-14	7.03125	7.21875	7	7.109375	1.520521	6,935,200
3	2000-08-15	7.109375	7.359375	7.015625	7.140625	1.527205	9,078,800
4	2000-08-16	7.203125	7.5	7.125	7.328125	1.567306	11,728,000
5	2000-08-17	7.390625	7.78125	7.359375	7.6875	1.644167	7,148,000

## Trazendo resultados únicos

Às vezes, dependendo de como os dados estão organizados ou como a nossa consulta é montada, o resultado que obtemos traz linhas repetidas. Por exemplo, se queremos saber quais são as oito classificações geográficas da Gapminder utilizadas no campo *wb\_regions* da tabela *country*, fazemos a seguinte *query*:

```
SELECT DISTINCT eight_regions
FROM country
ORDER BY eight_regions;
```

#### Bloco de código 30.

Vemos que há duas para a África, Norte e Subsaariana; duas para as Américas, do Norte e do Sul; duas para a Ásia, Ocidental e Oriental/Pacífico e duas para a Europa, Oeste e Leste.

Figura 26 – Resultado do bloco de código 30 (todas as linhas)

	RBC eight_regions
1	africa_north
2	africa_sub_saharan
3	america_north
4	america_south
5	asia_west
6	east_asia_pacific
7	europe_east
8	europe_west

Para resolver a questão 7 e a questão 10 do módulo 1 da unidade, provavelmente, você ficou rolando a tabela para encontrar quais as diferentes classificações da Europa e da Ásia na coluna *eight\_regions*. Com **DISTINCT** fica bem mais fácil.

De modo análogo, se queremos saber quais e quantas são as classificações geográficas do Banco Mundial, no campo *wb\_regions* da tabela *country*, fazemos:

```
SELECT DISTINCT wb_regions
FROM country
ORDER BY wb_regions;
```

#### Bloco de código 31.

Desse modo, vemos que há sete classificações: três para a Ásia – Leste e Pacífico, Central e Oriente Médio; mas a da Ásia Central também engloba a Europa, e a do Oriente Médio, o Norte da África. As Américas também são divididas de maneira diferente, entre América do Norte e América Latina e Caribe. Vemos também uma linha nula, que, se verificarmos, refere-se ao Vaticano (Holy See), que o Banco Mundial não categorizou.

Figura 27 – Resultado do bloco de código 31 (todas as linhas)

	RBC wb_regions
1	[NULL]
2	East Asia & Pacific
3	Europe & Central Asia
4	Latin America & Caribbean
5	Middle East & North Africa
6	North America
7	South Asia
8	Sub-Saharan Africa

A cláusula DISTINCT se aplica à linha inteira retornada pela *query*, não é possível aplicá-la a uma coluna específica. Veja o resultado da consulta que traz as combinações distintas entre *eight\_regions* e *wb\_regions* na tabela *country*:

```
SELECT DISTINCT eight_regions, wb_regions
FROM country
ORDER BY eight_regions, wb_regions;
```

**Bloco de código 32.**

O resultado são 13 linhas, nas quais se pode notar diferentes combinações de *eight\_regions* e *wb\_regions*.

Figura 27 – Resultado do bloco de código 32 (todas as linhas)

	RBC eight_regions	RBC wb_regions
1	africa_north	Middle East & North Africa
2	africa_sub_saharan	Sub-Saharan Africa
3	america_north	Latin America & Caribbean
4	america_north	North America
5	america_south	Latin America & Caribbean
6	asia_west	Europe & Central Asia
7	asia_west	Middle East & North Africa
8	asia_west	South Asia
9	east_asia_pacific	East Asia & Pacific
10	europe_east	Europe & Central Asia
11	europe_west	[NULL]
12	europe_west	Europe & Central Asia
13	europe_west	Middle East & North Africa

Há, por exemplo, repetições no campo *eight\_regions* que correspondem a diferentes valores do campo *wb\_regions*. Isso significa que há países classificados como *asia\_west* pelo critério de *eight\_regions*, mas que podem pertencer a três diferentes regiões segundo o critério *wb\_regions*. Outro exemplo é o México, que pelo critério *eight\_regions* pertence à América do Norte, mas pelo critério *wb\_regions* está na América Latina e Caribe.





## MÓDULO II – JUNTANDO DADOS

Como vimos na Introdução, os bancos de dados relacionais são compostos de tabelas, e essas tabelas se relacionam entre si por meio de chaves primárias e estrangeiras. Portanto, é necessário haver um meio de juntá-las para extrair informações mais relevantes dos bancos de dados.

Uma consulta com junção une duas ou mais tabelas no sentido das colunas, trazendo as colunas selecionadas de uma e da outra tabela, colocando-as lado a lado no resultado.

Ao fazermos junções, é necessário ter em mente a natureza das chaves dos bancos de dados para que a junção seja correta:

- as chaves primárias são identificadores únicos de cada linha da tabela; podem ser compostas de um ou mais campos;
- as chaves estrangeiras de uma tabela estabelecem uma relação com a chave primária de outra tabela; neste caso costumamos chamar, à luz de um determinado relacionamento, a tabela que contém a chave primária de "tabela-pai" e a que contém a chave estrangeira de "tabela-filha";
- os relacionamentos devem ser sempre de 1 para  $n$ , isto é, uma linha na tabela-pai tem zero, uma ou várias linhas relacionadas na tabela-filha; mas sob o ponto de vista da tabela-filha, cada linha se relaciona com apenas uma linha da tabela-pai.

Tudo isso é importante para que não ocorra um **produto cartesiano** entre as linhas das tabelas pai e filho. Pois se ao executar uma consulta, o banco encontra duas linhas na tabela-pai relacionadas com uma mesma linha na tabela-filha, ele duplica o resultado, gerando uma linha que junta as chaves de uma das linhas da tabela-pai com as chaves estrangeiras correspondentes na tabela-filha. Depois gera outras linhas no resultado, juntando as chaves dessa segunda linha da tabela-pai com as suas correspondentes na tabela-filha.

Vejamos cada tipo de junção na unidade a seguir.

# UNIDADE 1 – JUNÇÕES INTERNAS E JUNÇÕES NATURAIS

As junções internas e as naturais são os tipos de junção mais utilizados na SQL. Com elas é possível unir as colunas de duas ou mais tabelas e realizar consultas com maior complexidade e riqueza de detalhes.

## Junção interna

Suponha que nós queiramos saber os 10 países com maior PIB em 2019 e também a taxa de natalidade desses 10 países. Fizemos isso nas questões da unidade 3 do módulo 1, mas ali nós fizemos cinco consultas em separado e depois juntamos os dados manualmente. Agora queremos trazer os resultados em uma única consulta. Vamos ver o exemplo e depois detalhamos a sintaxe.

```
SELECT gp.country,  
       gp.gdp_pc,  
       f.mean_babies  
  FROM gdp_pc AS gp  
INNER JOIN fertility AS f ON gp.country = f.country  
          AND gp.ref_year = f.ref_year  
 WHERE gp.ref_year = 2019  
 ORDER BY gp.gdp_pc DESC  
 LIMIT 10;
```

Bloco de código 33.

As 10 linhas resultantes trazem os 10 países com maior PIB *per capita*, mas não os 10 países com menor taxa de natalidade, em vez disso, mostram a taxa de natalidade dos referidos países. Ocorre que esses países estão entre os de menor taxa de natalidade, que aliás está abaixo da taxa de reposição, que é de 2.1 filhos por mulher (Ministério da Saúde, 2000).

Figura 28 – Resultado do bloco de código 33 (todas as linhas)

	abc country	123 gdp_pc	123 mean_babies
1	Luxembourg	115,000	1.61
2	Singapore	98,500	1.27
3	Qatar	90,800	1.85
4	Ireland	86,900	1.98
5	UAE	71,800	1.7
6	Switzerland	69,900	1.56
7	Norway	65,000	1.83
8	USA	62,500	1.9
9	Brunei	61,400	1.84
10	Hong Kong, China	59,600	1.36

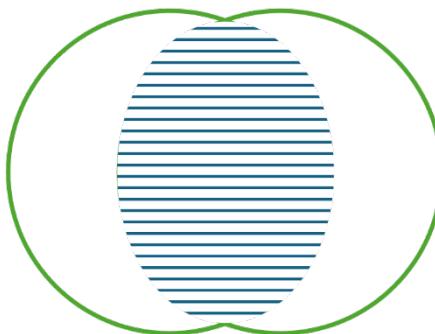
No entanto, vamos dissecar o bloco de código 33. Veja que agora mencionamos duas tabelas, *gdp\_pc* e *fertility*. Para juntá-las, usamos a cláusula **INNER JOIN**, que é a junção interna, e estabelecemos que os campos de união são *country* e *ref\_year* em ambas as tabelas. Isso porque esses campos são chaves primárias de ambas as tabelas. A chave primária não é só *country*, pois cada país se repete várias vezes, e também não é *ref\_year*, que também é uma informação que se repete várias vezes. Mas as combinações de *country* e *ref\_year* são únicas, isto é, para cada país existe apenas uma linha que se refere a um determinado ano.

Veja que não soubemos isso de antemão a partir do dicionário de dados; somente após termos analisado as tabelas, podemos chegar a essa conclusão. Aliás, a maioria das tabelas do nosso banco de dados de exemplos tem *country* e *ref\_year* como chave primária. Esses dois campos também servem como chave estrangeira, pois eles são equivalentes (mas não totalmente iguais) em quase todas as tabelas do banco de exemplos.

E como funciona o **INNER JOIN**? Ele une as duas tabelas de acordo com valores comuns a ambas. Por exemplo, a linha Brazil-1970 da tabela *gdp\_pc* vai se unir à linha Brazil-1970 da tabela *fertility*. Se houver algum par *country-ref\_year* na tabela *gdp\_pc* que não exista na tabela *fertility*, a linha que contém esse par será descartada. E a recíproca é verdadeira. Uma linha com um par *country-ref\_year* na tabela *fertility* sem equivalente na tabela *gdp\_pc* será igualmente descartada.

Desse modo, podemos enxergar o **INNER JOIN** como uma operação de **intersecção** entre dois conjuntos, ilustrada pela figura 29.

Figura 29 – Diagrama de Venn representando um INNER JOIN. A área hachurada mostra as linhas que serão recuperadas na consulta, ou seja, as linhas comuns entre as duas tabelas



Note agora que estamos usando apelidos para as tabelas, ou *table aliases*. Esses apelidos foram estabelecidos quando escrevemos *gdp\_pc AS pc* e *fertility AS f* – caso você omita o AS nos apelidos de tabelas, não tem problema, ele é opcional. E para que usamos os apelidos? Para explicitar, no comando SQL, de onde vem cada campo. Por exemplo, quando estamos selecionando quais colunas, escrevemos *pc.country*, *pc.gdp\_pc*, *f.mean\_babies*. Com isso dizemos que o campo *country* vem da tabela *gdp\_pc*. Se não informássemos o apelido, a query não iria rodar porque existe um campo com o mesmo nome na tabela *fertility* o que ocasionaria um erro de ambiguidade. Já nas

outras duas colunas, o apelido é opcional, pois os nomes só existem em uma ou outra tabela. Mas usamos os apelidos para melhorar a legibilidade do comando. Imagine uma consulta que traga 50 colunas espalhadas em cinco tabelas que estão sendo juntadas. Fica difícil para quem está lendo a *query* (mesmo para quem a escreveu) entender de onde está vindo o quê. O apelido ajuda com isso.

O uso de apelidos é feito em todas as cláusulas da consulta, não só na seleção das colunas. Usamos também no **INNER JOIN**, no **WHERE** e até no **ORDER BY**.

Finalmente, na cláusula **ON** do **INNER JOIN**, especificamos exatamente quais os critérios para fazer a junção: *gp.country* = *f.country* **AND** *gp.ref\_year* = *f.ref\_year*. Por fim, arrematando a nossa consulta, utilizamos a cláusula **WHERE** para filtrar pelo ano de 2019, o **ORDER BY** para ordenar o PIB em ordem crescente, porque queremos os maiores, e o **LIMIT** para trazer apenas 10.

Para continuar a resolver as consultas da unidade 3 do módulo 1, vamos juntar a tabela *woman\_years\_at\_school* para listar o tempo médio na escola das mulheres de 25 anos em 2019. A *query* agora fica assim:

```
SELECT gp.country,
       gp.gdp_pc,
       f.mean_babies,
       ws.mean_years
  FROM gdp_pc AS gp
    JOIN fertility AS f ON gp.country = f.country
                           AND gp.ref_year = f.ref_year
    JOIN women_years_at_school AS ws ON gp.country = ws.country
                           AND gp.ref_year = ws.ref_year
 WHERE gp.ref_year = 2019
 ORDER BY gp.gdp_pc DESC
 LIMIT 10;
```

#### Bloco de código 34.

Só que não obtemos nenhuma linha como resultado. Isso acontece porque não há dados na tabela *woman\_years\_at\_school* além do ano de 2009. Como não há *match* das chaves das três tabelas envolvidas, nenhuma linha é retornada.

Figura 30 – Resultado do bloco de código 34 (todas as linhas)

lock	RBC country	123 gdp_pc	123 mean_babies	123 mean_years

O uso da palavra **INNER** é opcional. Pode-se utilizar apenas **JOIN** que o banco de dados entende que é um **INNER JOIN**.

Por exemplo, queremos trazer os cinco países com as maiores emissões de CO<sub>2</sub> em 1970 e as suas respectivas classificações geográficas segundo o critério do Banco Mundial.

```
SELECT co2.country,  
       c.wb_regions,  
       co2.co2_pc  
  FROM co2_emissions_pc co2  
  JOIN country c ON co2.country = c.country  
 WHERE co2.ref_year = 1970  
 ORDER BY co2.co2_pc DESC  
 LIMIT 5;
```

#### Bloco de código 35.

Note que também não utilizamos o **AS** para definir os apelidos das tabelas. O resultado é:

Figura 31 – Resultado do bloco de código 35 (todas as linhas)

lock	RBC country	RBC wb_regions	123 co2_pc
1	Luxembourg	Europe & Central Asia	42.4
2	Kuwait	Middle East & North Africa	41.4
3	USA	North America	20.8
4	Canada	North America	16.1
5	Czech Republic	Europe & Central Asia	15.5

Existe um outro formato para fazer os **INNER JOINS**, no qual não se utiliza nem a palavra-chave **JOIN**, apenas listam-se as tabelas e colocam-se os critérios de junção na cláusula **WHERE**.

Por exemplo, queremos saber os cinco países mais populosos da Europa no ano 2000. A *query* é:

```
SELECT p.country,  
       p.tot_pop  
  FROM population p,  
       country c  
 WHERE p.country = c.country  
   AND p.ref_year = 2000  
   AND c.four_regions = 'europe'  
 ORDER BY p.tot_pop DESC  
 LIMIT 5;
```

#### Bloco de código 36.

O resultado é:

Figura 32 – Resultado do bloco de código 36 (todas as linhas)

	ABC country	123 tot_pop
1	Russia	147,000,000
2	Germany	81,600,000
3	Turkey	64,099,999
4	UK	58,900,000
5	France	58,700,000

O problema desse tipo de construção é que não distinguimos facilmente os critérios de junção dos critérios de filtragem, pois todos ficam na cláusula **WHERE**. Portanto, prefira a construção com a palavra **JOIN**.

## Junção natural

A junção natural, ou **NATURAL JOIN**, é similar ao **INNER JOIN**, mas sem especificar quais as colunas que irão fazer o *match*. Então, para que a junção ocorra, é necessário que os campos de junção tenham o mesmo nome e tipo em ambas as tabelas; além disso, é importante que nenhum campo que esteja fora da lista de campos de junção tenha o mesmo nome em ambas as tabelas, pois, neste caso, ele também será usado para fazer a junção, e o resultado estará incorreto.

Suponha que queiramos saber os cinco países de maior taxa de mortalidade infantil relacionada com a renda média diária *per capita* das famílias em 2019. A *query* pode ser:

```
SELECT cm.country,  
       cm.tot_deaths,  
       inc.mean_usd  
  FROM child_mortality AS cm  
 NATURAL JOIN avg_income AS inc  
 WHERE cm.ref_year = 2019  
 ORDER BY cm.tot_deaths  
 LIMIT 5;
```

#### Bloco de código 37.

O resultado dessa consulta é:

Figura 33 – Resultado do bloco de código 37 (todas as linhas)

	RBC country	123 tot_deaths	123 mean_usd
1	Finland	1.63	58.8
2	Iceland	1.92	64
3	San Marino	1.93	60.3
4	Hong Kong, China	2.02	62.1
5	Slovenia	2.1	43.4

O resultado seria o mesmo se utilizássemos **INNER JOIN** e os campos *country* e *ref\_year* para fazer as junções. Faça o teste!

**Dica:** prefira o **INNER JOIN** ao **NATURAL JOIN**, pois, ao listar as colunas que fazem parte do critério de junção, você terá certeza de que nenhuma coluna indesejada será incluída nesse critério.

## UNIDADE 2 – JUNÇÕES EXTERNAS

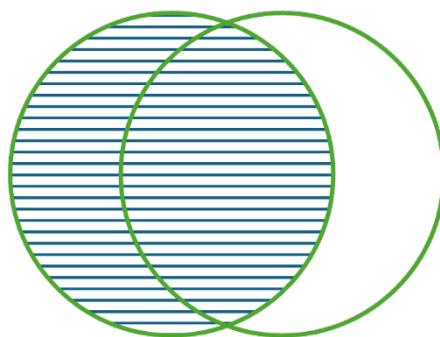
Como vimos, o **INNER JOIN** traz apenas os dados que são comuns nas duas tabelas juntadas. Porém, pode ser o caso de que queiramos trazer todos os dados de uma das tabelas, quer ela tenha correspondência na outra tabela ou não. Nesses casos, usamos uma operação chamada **OUTER JOIN**, ou junção externa. Quando trazemos todos os dados da tabela da esquerda, ou seja, a primeira tabela a ser listada na junção, temos um **LEFT JOIN**. Caso seja a tabela da direita, temos um **RIGHT JOIN**. Caso queiramos trazer todas as linhas tanto da tabela da esquerda quanto da direita, temos um **FULL JOIN**.

Em todos os casos, os campos da tabela não correspondida assumem valores nulos nos resultados.

## Juntando dados à esquerda

O **LEFT JOIN** traz todos os dados da tabela da esquerda, independentemente se há ou não correspondência na tabela da direita. As linhas da esquerda que tiverem correspondência com as da direita serão juntadas. As linhas da esquerda sem correspondência na direita irão trazer valores nulos nos campos pertencentes à tabela da direita. O diagrama de Venn ilustra esquematicamente um **LEFT JOIN**.

Figura 34 – Diagrama de Venn representando um LEFT JOIN. A área hachurada mostra as linhas que serão recuperadas na consulta, ou seja, todas as linhas da tabela da esquerda e somente as da tabela da direita que tiverem equivalência na tabela da esquerda



Considere o seguinte exemplo: queremos trazer a taxa de fecundidade e o tempo de escolarização materna, no Brasil, nos anos de 2001 a 2020. Se fizermos a *query* com um **INNER JOIN**:

```
SELECT f.ref_year,  
       f.mean_babies,  
       wy.mean_years  
  FROM fertility f  
 INNER JOIN women_years_at_school wy ON f.country = wy.country  
          AND f.ref_year = wy.ref_year  
 WHERE f.country = 'Brazil'  
          AND f.ref_year BETWEEN 2001 AND 2020  
 ORDER BY f.ref_year;
```

Bloco de código 38.

O resultado dessa consulta traz apenas o período de 2001 a 2009, pois não há dados posteriores a 2009 na tabela *woman\_years\_at\_school*, como já vimos. Também se nota uma clara correlação negativa entre as duas colunas.

Figura 35 – Resultado do bloco de código 38 (todas as linhas)

	123 ref_year ▾	123 mean_babies ▾	123 mean_years ▾
1	2,001	2.23	5.9
2	2,002	2.16	6.1
3	2,003	2.1	6.2
4	2,004	2.03	6.4
5	2,005	1.98	6.5
6	2,006	1.93	6.7
7	2,007	1.88	6.8
8	2,008	1.85	7
9	2,009	1.82	7.2

Agora, se queremos trazer todas as linhas da tabela *fertility* que atenderem ao filtro de data da consulta, fazemos um **LEFT JOIN** em vez de um **INNER JOIN**:

```

SELECT f.ref_year,
       f.mean_babies,
       wy.mean_years
  FROM fertility f
 LEFT JOIN women_years_at_school wy ON f.country = wy.country
          AND f.ref_year = wy.ref_year
 WHERE f.country = 'Brazil'
          AND f.ref_year BETWEEN 2001 AND 2020
 ORDER BY f.ref_year;

```

#### Bloco de código 39.

Agora, além das linhas de 2001 a 2009, que vêm com os valores de *mean\_years* preenchidos, temos também as linhas de 2010 a 2020, que vem com os valores de *mean\_years* nulos.

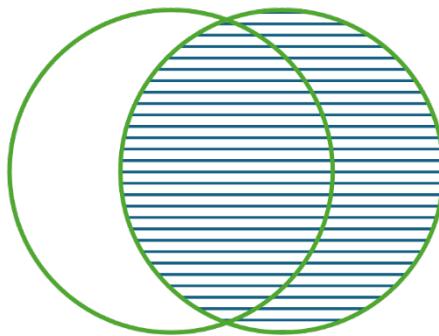
Figura 36 – Resultado do bloco de código 39 (todas as linhas)

🔒	123 ref_year ▼	123 mean_babies ▼	123 mean_years ▼
1	2,001	2.23	5.9
2	2,002	2.16	6.1
3	2,003	2.1	6.2
4	2,004	2.03	6.4
5	2,005	1.98	6.5
6	2,006	1.93	6.7
7	2,007	1.88	6.8
8	2,008	1.85	7
9	2,009	1.82	7.2
10	2,010	1.81	[NULL]
11	2,011	1.79	[NULL]
12	2,012	1.78	[NULL]
13	2,013	1.77	[NULL]
14	2,014	1.75	[NULL]
15	2,015	1.74	[NULL]
16	2,016	1.73	[NULL]
17	2,017	1.71	[NULL]
18	2,018	1.7	[NULL]
19	2,019	1.69	[NULL]
20	2,020	1.67	[NULL]

## Juntando dados à direita

De modo análogo ao **LEFT JOIN**, o **RIGHT JOIN** traz todas as linhas da tabela da direita. As da tabela da esquerda que tiverem correspondência com as da direita virão com valores preenchidos. Já as da esquerda sem correspondência virão com valores nulos. A figura 37 representa o **RIGHT JOIN** esquematicamente.

Figura 37 – Diagrama de Venn representando um RIGHT JOIN. A área hachurada mostra as linhas que serão recuperadas na consulta, ou seja, todas as linhas da tabela da direita e somente as da tabela da esquerda que tiverem equivalência na tabela da direita



Suponha que queiramos investigar a emissão de CO<sub>2</sub> *per capita* e o PIB *per capita* do Brasil na última década. Utilizando-se **INNER JOIN**, a consulta é:

```
SELECT gp.ref_year,
       cep.co2_pc,
       gp.gdp_pc
FROM co2_emissions_pc cep
INNER JOIN gdp_pc gp ON gp.country = cep.country
              AND gp.ref_year = cep.ref_year
WHERE gp.country = 'Brazil'
              AND gp.ref_year BETWEEN 2014 AND 2023
ORDER BY gp.ref_year;
```

#### Bloco de código 40.

Nota-se que o ano de 2023 não aparece nos resultados. Se avaliarmos as tabelas, notamos que o PIB vai até 2100, mas as emissões de CO<sub>2</sub> vão apenas até 2022.

Figura 36 – Resultado do bloco de código 40 (todas as linhas)

	123 ref_year	123 co2_pc	123 gdp_pc
1	2,014	3	15,700
2	2,015	2.66	15,000
3	2,016	2.35	14,400
4	2,017	2.38	14,500
5	2,018	2.26	14,600
6	2,019	2.24	14,700
7	2,020	1.99	14,100
8	2,021	2.07	14,700
9	2,022	2.11	15,100

Se refizermos a consulta utilizando um **RIGHT JOIN**, pois é a da tabela *gdp\_pc*, que vem em segundo na lista da junção, queremos trazer todas as linhas.

```
SELECT gp.ref_year,
       cep.co2_pc,
       gp.gdp_pc
FROM co2_emissions_pc cep
RIGHT JOIN gdp_pc gp ON gp.country = cep.country
              AND gp.ref_year = cep.ref_year
WHERE gp.country = 'Brazil'
```

```

AND gp.ref_year BETWEEN 2014 AND 2023
ORDER BY gp.ref_year;

```

#### Bloco de código 41.

Desse modo, a linha referente a 2023 aparece, trazendo o valor do PIB *per capita* e nulo no lugar do valor do CO<sub>2</sub> *per capita*.

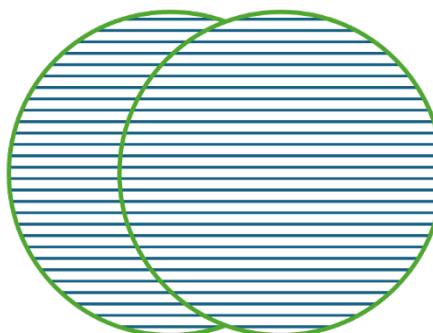
Figura 37 – Resultado do bloco de código 41 (todas as linhas)

	123 ref_year	123 co2_pc	123 gdp_pc
1	2,014	3	15,700
2	2,015	2.66	15,000
3	2,016	2.35	14,400
4	2,017	2.38	14,500
5	2,018	2.26	14,600
6	2,019	2.24	14,700
7	2,020	1.99	14,100
8	2,021	2.07	14,700
9	2,022	2.11	15,100
10	2,023	[NULL]	15,500

### Juntando dados em ambos os lados

O **FULL JOIN** é uma mistura do **LEFT JOIN** e do **RIGHT JOIN**. É pouco usado, mas é útil para trazer todos os dados da tabela da esquerda e da tabela da direita, fazendo a junção onde há correspondência e trazendo nulo onde não há. O diagrama de Venn da figura 38 representa esquematicamente um **FULL JOIN**.

Figura 38 – Diagrama de Venn representando um FULL JOIN. A área hachurada mostra que todas as linhas, em ambas as tabelas, serão recuperadas na consulta. As que não tiverem equivalência terão os seus valores preenchidos com NULL, em um lado ou no outro



Queremos investigar a população e a taxa de natalidade de algumas ilhas da Oceania: Fiji, Guam, Nauru, Palau e Polinésia Francesa. Vamos começar fazendo um **INNER JOIN**:

```
SELECT f.country,
       p.tot_pop,
       f.mean_babies
  FROM fertility f
 INNER JOIN population p ON p.country = f.country
                      AND p.ref_year = f.ref_year
 WHERE f.country IN ('Fiji', 'Nauru', 'Palau',
                     'French Polynesia', 'Guam')
   AND f.ref_year = 2018;
```

Bloco de código 42.

O resultado é o único país em comum, nas duas tabelas, entre os listados na cláusula **WHERE**.

Figura 39 – Resultado do bloco de código 42 (todas as linhas)

	RBC country	123 tot_pop	123 mean_babies
1	Fiji	919,000	2.47

Vamos agora usar um **LEFT JOIN** para tentar trazer os dados existentes dos cinco países-alvo:

```
SELECT f.country,
       p.tot_pop,
       f.mean_babies
  FROM fertility f
 LEFT JOIN population p ON p.country = f.country
                      AND p.ref_year = f.ref_year
 WHERE f.country IN ('Fiji', 'Nauru', 'Palau',
                     'French Polynesia', 'Guam')
   AND f.ref_year = 2018;
```

Bloco de código 43.

Agora temos três países: Fiji, Guam e Polinésia Francesa. Eles não apareceram na consulta anterior porque não existem dados referentes a eles na tabela *population*, pelo menos no ano de 2018. Mas existem na tabela *fertility*.

Figura 40 – Resultado do bloco de código 43 (todas as linhas)

	RBC country	123 tot_pop	123 mean_babies
1	Fiji	919,000	2.47
2	Guam	[NULL]	2.31
3	French Polynesia	[NULL]	1.98

Mas ainda faltam dois países da lista. Experimentemos um **RIGHT JOIN**:

```
SELECT p.country,
       p.tot_pop,
       f.mean_babies
  FROM fertility f
RIGHT JOIN population p ON p.country = f.country
                      AND p.ref_year = f.ref_year
 WHERE p.country IN ('Fiji', 'Nauru', 'Palau',
                      'French Polynesia', 'Guam')
   AND p.ref_year = 2018;
```

Bloco de código 44.

Nessa consulta, aparecem Nauru e Palau, que não haviam aparecido na consulta anterior. Isso acontece porque eles estão na tabela *population*, mas não na tabela *fertility*.

Figura 41 – Resultado do bloco de código 44 (todas as linhas)

	RBC country	123 tot_pop	123 mean_babies
1	Fiji	919,000	2.47
2	Nauru	11,900	[NULL]
3	Palau	17,900	[NULL]

Vamos então fazer um **FULL JOIN** para que os cinco países apareçam na nossa consulta com os dados existentes para cada um:

```
SELECT f.country,
       p.country,
       p.tot_pop,
       f.mean_babies
FROM fertility f
FULL JOIN population p ON p.country = f.country
                  AND p.ref_year = f.ref_year
WHERE (f.country IN ('Fiji', 'Nauru', 'Palau',
                    'French Polynesia', 'Guam')
OR p.country IN ('Fiji', 'Nauru', 'Palau',
                    'French Polynesia', 'Guam'))
AND (f.ref_year = 2018
OR p.ref_year = 2018);
```

Bloco de código 45.

Pronto. Todos os países aparecem, como vemos na figura 42. Mas note que mudar a junção para **FULL JOIN** não foi a única alteração que fizemos. Precisamos também incluir mais uma coluna *country* no **SELECT**, pois temos que usar os apelidos das tabelas qualificando os campos com nomes iguais para evitar a ambiguidade. No entanto, Guam e Polinésia Francesa não existem na tabela *population*, então não tem como eles aparecerem na *query* se fazemos referência a *p.country*. Por outro lado, Nauru e Palau não existem em *fertility*, então eles não são retornados em *f.country*.

Além disso, alteramos também a cláusula **WHERE**, consultando a lista de países nas duas tabelas e unindo o resultado por meio de um **OR**. O mesmo foi feito para o ano de 2018.

Figura 42 – Resultado do bloco de código 45 (todas as linhas)

	ABC country	ABC country	123 tot_pop	123 mean_babies
1	Fiji	Fiji	919,000	2.47
2	Guam	[NULL]	[NULL]	2.31
3	French Polynesia	[NULL]	[NULL]	1.98
4	[NULL]	Nauru	11,900	[NULL]
5	[NULL]	Palau	17,900	[NULL]

Note que esses ajustes nas consultas vieram sendo feitos conforme mudávamos os **JOINS**. Suponha duas tabelas, com apelidos **A** e **B**. No **INNER JOIN**, tanto faz o apelido da tabela que usamos, o da tabela **A** ou o da tabela **B**, pois os valores retornados são valores comuns entre as duas tabelas, eles aparecem dos dois lados. Mas sempre privilegiamos a tabela que vai retornar todos os valores nos casos de **OUTER JOINs**. No caso do **LEFT JOIN**, usamos o apelido da tabela **A**, supondo que ela seja listada primeiro na junção, ou seja, é a tabela que está no **FROM**. Se é um **RIGHT JOIN**, usamos o apelido da tabela **B**. E, se for um **FULL JOIN**, temos que usar o apelido das duas, fazendo a *query* de modo a listar os campos-chave das duas no **SELECT** e adequando o filtro no **WHERE** usando **OR** para consultar os valores nas duas tabelas.

## Apelidos de tabelas e colunas

Antes de passar para as operações de conjuntos, é útil voltarmos ao tema dos apelidos (*aliases*). Já vimos que, para fazermos as junções, é necessário utilizar apelidos para as tabelas de modos a qualificar as colunas que tenham o mesmo nome em ambas as tabelas da junção. Na verdade, você não precisa usar apelidos, pode utilizar diretamente o nome da tabela. Mas fica meio longo escrever o nome da tabela antes de cada campo.

Outro tipo de apelido (*alias*) bastante útil é o utilizado para renomear colunas. Por exemplo, se quisermos traduzir o nome das colunas nas consultas que temos feito até agora, podemos escrever a *query* assim:

```
SELECT gdp.country AS pais,
       gdp.ref_year AS ano,
       gdp.gdp_pc AS pib_pc
  FROM gdp_pc gdp
 WHERE pais = 'Brazil'
   AND ano BETWEEN 2013 AND 2022
 ORDER BY pais, ano;
```

**Bloco de código 46.**

As colunas aparecem no resultado com os seus nomes novos. Aliás, não apenas nos resultados, mas já podemos utilizar os novos nomes nas cláusulas seguintes da consulta, como fizemos no **WHERE** e no **ORDER BY**.

Figura 43 – Resultado do bloco de código 46 (todas as linhas)

	ABC pais	123 ano	123 pib_pc
1	Brazil	2,013	15,800
2	Brazil	2,014	15,700
3	Brazil	2,015	15,000
4	Brazil	2,016	14,400
5	Brazil	2,017	14,500
6	Brazil	2,018	14,600
7	Brazil	2,019	14,700
8	Brazil	2,020	14,100
9	Brazil	2,021	14,700
10	Brazil	2,022	15,100

Note que o **AS**, que é opcional no apelido de tabelas, é obrigatório no apelido de colunas. Se quisermos dar um nome composto para a coluna, separado por um espaço, é necessário utilizar aspas. Caso contrário, estas são dispensáveis. No entanto, recomendamos que não se utilize nomes separados por espaços e nem caracteres acentuados no nome das colunas. Às vezes, pode dar problema de interpretação de codificação de caracteres<sup>4</sup>.

## UNIDADE 3 – OPERAÇÕES DE CONJUNTOS

As operações de conjuntos têm como objetivo juntar tabelas no sentido das linhas, e não no das colunas, como ocorre nos **JOINs**.

### Unindo dados

As operações **UNION** e **UNION ALL** têm por objetivo juntar linhas de duas ou mais tabelas. Por exemplo, temos duas tabelas de tempo médio na escola, uma para homens e outra para mulheres. Podemos juntar as duas tabelas para realizar, posteriormente, consultas nelas:

```
SELECT m.country, m.ref_year, m.mean_years
FROM men_years_at_school m
UNION
SELECT w.country, w.ref_year, w.mean_years
FROM women_years_at_school w;
```

**Bloco de código 47.**

<sup>4</sup> Veja mais sobre codificação de caracteres em [https://pt.wikipedia.org/wiki/Codificação\\_de\\_caracteres](https://pt.wikipedia.org/wiki/Codificação_de_caracteres).

O resultado é:

Figura 44 – Resultado do bloco de código 47 (10 primeiras linhas)

	RBC country	123 ref_year	123 mean_years
1	Afghanistan	1,970	0
2	Afghanistan	1,970	0.7
3	Afghanistan	1,971	0.1
4	Afghanistan	1,971	0.7
5	Afghanistan	1,972	0.1
6	Afghanistan	1,972	0.8
7	Afghanistan	1,973	0.1
8	Afghanistan	1,973	0.8
9	Afghanistan	1,974	0.1
10	Afghanistan	1,974	0.8

O comando UNION juntou as linhas das duas tabelas, *men\_years\_at\_school* e *women\_years\_at\_school*. Note que as linhas do Afeganistão aparecem duplicadas em cada ano, mas com médias diferentes. Isso porque agora temos linhas dos dois gêneros. Para que a união aconteça, é necessário que as colunas das duas tabelas a serem juntadas tenham o mesmo nome e o mesmo tipo, além de a quantidade de colunas ter de ser a mesma.

Mas, voltando ao resultado da figura 44, ele não é muito útil, pois não é possível distinguir o gênero ao qual a linha se refere. Para resolver isso, vamos utilizar um pequeno truque:

```
SELECT m.country, m.ref_year, m.mean_years, 'male' as gender
FROM men_years_at_school m
UNION
SELECT w.country, w.ref_year, w.mean_years, 'female' as gender
FROM women_years_at_school w,
```

#### Bloco de código 48.

Veja que criamos uma nova coluna, com um valor constante: *male* para os homens, e *female* para mulheres. E batizamos a coluna de *gender* por meio de um apelido.

Figura 45 – Resultado do bloco de código 48 (10 primeiras linhas)

	RBC country	123 ref_year	123 mean_years	RBC gender
1	Afghanistan	1,970	0	female
2	Afghanistan	1,970	0.7	male
3	Afghanistan	1,971	0.1	female
4	Afghanistan	1,971	0.7	male
5	Afghanistan	1,972	0.1	female
6	Afghanistan	1,972	0.8	male
7	Afghanistan	1,973	0.1	female
8	Afghanistan	1,973	0.8	male
9	Afghanistan	1,974	0.1	female
10	Afghanistan	1,974	0.8	male

O UNION permite que façamos filtros de linhas:

```
SELECT m.country, m.ref_year, m.mean_years, 'male' as gender
FROM men_years_at_school m
WHERE m.country = 'Brazil'
AND m.ref_year = 2005
UNION
SELECT w.country, w.ref_year, w.mean_years, 'female' as gender
FROM women_years_at_school w
WHERE w.country = 'Brazil'
AND w.ref_year = 2005;
```

#### Bloco de código 49.

O resultado é:

Figura 46 – Resultado do bloco de código 49 (todas as linhas)

	RBC country	123 ref_year	123 mean_years	RBC gender
1	Brazil	2,005	6.3	male
2	Brazil	2,005	6.5	female

A diferença entre o UNION e o UNION ALL é que o UNION faz um DISTINCT das linhas iguais. O UNION ALL mantém as linhas repetidas.

Veja o resultado da *query* com UNION:

```
SELECT m.country, m.ref_year
FROM men_years_at_school m
WHERE m.country = 'Brazil'
UNION
SELECT w.country, w.ref_year
FROM women_years_at_school w
WHERE w.country = 'Brazil';
```

Bloco de código 50.

O resultado são 40 linhas referentes ao Brasil, de 1970 a 2009:

Figura 47 – Resultado do bloco de código 50 (5 primeiras linhas)

	ABC country	123 ref_year
1	Brazil	1,970
2	Brazil	1,971
3	Brazil	1,972
4	Brazil	1,973
5	Brazil	1,974

A mesma consulta, agora com UNION ALL:

```
SELECT m.country, m.ref_year
FROM men_years_at_school m
WHERE m.country = 'Brazil'
UNION ALL
SELECT w.country, w.ref_year
FROM women_years_at_school w
WHERE w.country = 'Brazil';
```

Bloco de código 51.

O resultado são 80 linhas referentes ao Brasil, de 1970 a 2009, com cada ano aparecendo duas vezes:

Figura 48 – Resultado do bloco de código 51 (5 primeiras linhas e mais 5 linhas quando estas começam a se repetir: linha 41 a 45)

	RBC country	123 ref_year
1	Brazil	1,970
2	Brazil	1,971
3	Brazil	1,972
4	Brazil	1,973
5	Brazil	1,974
41	Brazil	1,970
42	Brazil	1,971
43	Brazil	1,972
44	Brazil	1,973
45	Brazil	1,974

## Subtraindo dados

Outra operação de conjuntos que podemos realizar sobre as linhas de duas tabelas é a **diferença**, por meio do operador **EXCEPT**. Com ele, conseguimos descobrir linhas que existem em uma determinada tabela mas não existem em outras; além disso, as operações de diferença de conjuntos trazem os elementos que estão em um conjunto e não estão em outro. Assim como a operação de diferença entre conjuntos, o **EXCEPT** não é transitivo, ou seja, a consulta "*tabela alfa EXCEPT tabela beta*" traz resultados diferentes da consulta "*tabela beta EXCEPT tabela alfa*". A não ser que as tabelas sejam iguais, situação em que a consulta com **EXCEPT** irá retornar zero linhas em ambos os casos.

Queremos, por exemplo, saber todos os países que estão na tabela *population* e que não estão na tabela *fertility*. Nesse caso, a consulta deverá ser feita assim:

```
SELECT p.country
FROM population p
EXCEPT
SELECT f.country
FROM fertility f,
```

**Bloco de código 52.**

O resultado é a lista de países da figura 49:

Figura 49 – Resultado do bloco de código 52 (todas as linhas)

	RBC country
1	Andorra
2	Dominica
3	Holy See
4	Liechtenstein
5	Marshall Islands
6	Monaco
7	Nauru
8	Palau
9	San Marino
10	St. Kitts and Nevis
11	Tuvalu

Por outro lado, se quisermos saber todos os países que estão na tabela *fertility* e que não estão na tabela *population*, fazemos a consulta da seguinte forma:

```
SELECT f.country
FROM fertility f
EXCEPT
SELECT p.country
FROM population p;
```

Bloco de código 53.

O resultado é a lista de países da figura 50:

Figura 50 – Resultado do bloco de código 53 (todas as linhas)

	RBC country
1	Aruba
2	Channel Islands
3	French Guiana
4	French Polynesia
5	Greenland
6	Guadeloupe
7	Guam
8	Macao, China
9	Martinique
10	Mayotte
11	Netherlands Antilles
12	New Caledonia
13	Puerto Rico
14	Reunion
15	Virgin Islands (U.S.)
16	Western Sahara

Uma observação importante é que o **EXCEPT** compara as linhas da tabela. Então, para fazer o **EXCEPT**, é necessário que as colunas dos dois **SELECTs** tenham o mesmo nome e o mesmo tipo. Se utilizarmos várias colunas, a comparação será feita a partir da combinação dos valores de todas as colunas. Dessa forma, se compararmos *country* e *ref\_year* das tabelas *population* e *fertility*, e se houver uma combinação de país e ano que existe em uma, mas não na outra, essa linha irá aparecer no resultado.

## Intersecção de dados

A última operação de conjuntos que podemos fazer sobre as linhas de duas tabelas é a **intersecção**. Nesse caso, o operador é o **INTERSECT**, que traz as linhas em comum de duas tabelas. Muito parecido com o **INNER JOIN**, exceto pelo fato de que atua sobre as linhas, e não sobre as colunas, como as junções.

Assim como no **EXCEPT**, é necessário que as colunas dos dois **SELECTs** tenham as mesmas colunas –mesmo nome e mesmo tipo –; a comparação é feita a partir da combinação dos valores de todas as colunas. Entretanto, ao contrário do **EXCEPT**, tanto faz a ordem das tabelas. O resultado será o mesmo, como acontece nas operações de intersecção de conjuntos.

Verificando agora quais são os países que estão nas tabelas *population* e *fertility*, fazemos:

```
SELECT p.country
FROM population p
INTERSECT
SELECT f.country
FROM fertility f,
```

Bloco de código 54.

O resultado é a lista de países da figura 51:

Figura 51 – Resultado do bloco de código 54 (10 primeiras linhas)

	ABC country
1	Afghanistan
2	Albania
3	Algeria
4	Angola
5	Antigua and Barbuda
6	Argentina
7	Armenia
8	Australia
9	Austria
10	Azerbaijan





## MÓDULO III – AGRUPANDO OS DADOS

Neste módulo, iremos explorar as capacidades de consulta da linguagem SQL em todo o seu potencial. Nele veremos como realizar operações matemáticas entre colunas e utilizar funções summarizadoras para fazer uma estatística descritiva das colunas.

Aprenderemos a agrupar os dados para sermos capazes de aplicar essas funções summarizadoras de acordo com diferentes critérios.

Veremos também todo o poder das *subqueries*, com as quais conseguimos relacionar dinamicamente o resultado de duas ou mais consultas.

### UNIDADE 1 – FUNÇÕES SUMARIZADORAS

As funções summarizadoras trazem algum poder estatístico para a SQL, permitindo que calculemos a média de colunas, que façamos somatórios e contagens e que encontremos os valores máximos e mínimos de uma coluna.

Além disso, veremos que é possível criar novas colunas a partir de operações aritméticas entre colunas existentes.

#### Operadores aritméticos

Os operadores aritméticos comuns da matemática podem ser usados na cláusula **SELECT**. Eles estão listados na tabela a seguir:

Tabela 7 – Operadores aritméticos

operador	significado
+	soma
-	subtração
*	multiplicação
/	divisão
%	resto da divisão ou módulo

Utilizando esses operadores, podemos realizar operações matemáticas entre valores literais (1 ou 0.75), entre colunas e entre valores literais e colunas.

Por exemplo, queremos saber, em termos percentuais, a taxa de mortalidade infantil do Brasil, de 1910 a 2020, de 10 em 10 anos. Podemos fazer:

```
SELECT cm.ref_year,
    cm.tot_deaths,
    100 * cm.tot_deaths / 1000 AS percent_deaths
FROM child_mortality cm
WHERE cm.ref_year IN (1910, 1920, 1930, 1940,
    1950, 1960, 1970, 1980,
    1990, 2000, 2010, 2020)
AND cm.country = 'Brazil'
ORDER BY cm.ref_year;
```

#### Bloco de código 55.

Notamos, no resultado da figura 52, que a taxa de mortalidade infantil caiu de 41% dos nascidos vivos em 1910 para 1,35% em 2020.

Figura 52 – Resultado do bloco de código 55 (todas as linhas)

	123 ref_year	123 tot_deaths	123 percent_deaths
1	1,910	410	41
2	1,920	400	40
3	1,930	391	39
4	1,940	315	31
5	1,950	212	21
6	1,960	169	16
7	1,970	133	13
8	1,980	96.3	9.63
9	1,990	63	6
10	2,000	34.8	3.48
11	2,010	18.7	1.87
12	2,020	13.5	1.35

Podemos também utilizar colunas de diferentes tabelas para fazer o cálculo. Vamos estimar o PIB total dos 10 países mais ricos em trilhões de USD, usando para isso o PIB *per capita* da tabela *gdp\_pc* multiplicado pela população total do país da tabela *population*. Vamos fazer essa estimativa para o ano de 2022:

```

SELECT gp.country,
       gp.gdp_pc,
       p.tot_pop,
       (gp.gdp_pc * p.tot_pop)/1E12 AS gdp_usd_tri
FROM gdp_pc gp
JOIN population p ON gp.country = p.country
      AND gp.ref_year = p.ref_year
WHERE gp.ref_year = 2022
ORDER BY gdp_usd_tri DESC
LIMIT 10;

```

#### Bloco de código 56.

Veja que, em termos de PIB *per capita*, a ordem dos países seria diferente. Os três primeiros seriam EUA, Alemanha e Reino Unido, e não China, EUA e Índia. Mas o tamanho da população fez bastante diferença. A Índia é o segundo colocado de acordo com o PIB total, mas seria o décimo pelo critério PIB *per capita*. Os dados estão na figura 53.

Figura 53 – Resultado do bloco de código 56 (todas as linhas)

	RBC country	123 gdp_pc	123 tot_pop	123 gdp_usd_tri
1	China	18,200	1,430,000,000	26.026
2	USA	64,700	338,000,000	21.8686
3	India	7,100	1,420,000,000	10.082
4	Japan	41,600	124,000,000	5.1584
5	Germany	53,600	83,400,000	4.47024
6	Russia	27,600	145,000,000	4.002
7	Indonesia	12,400	276,000,000	3.4224
8	Brazil	15,100	215,000,000	3.2465
9	UK	46,800	67,500,000	3.159
10	France	46,000	64,599,999	2.971599954

Também é possível utilizar uma operação aritmética na cláusula **WHERE**, desde que essa operação seja comparada com um resultado e, dessa forma, retorne um valor lógico. Por exemplo, queremos saber as emissões de CO<sub>2</sub> *per capita* de Portugal em ano de Jogos Olímpicos de Verão, de 1972 (Munique) até 2016 (Rio de Janeiro):

```

SELECT cep.ref_year,
       cep.co2_pc
  FROM co2_emissions_pc cep
 WHERE cep.ref_year BETWEEN 1972 AND 2016
   AND cep.ref_year % 4 = 0
   AND cep.country = 'Portugal'
 ORDER BY cep.ref_year;
    
```

**Bloco de código 57.**

Nós pegamos os anos cujo resto da divisão por quatro é igual a zero, expressão que retorna verdadeiro ou falso, como condição de filtro.

Figura 54 – Resultado do bloco de código 57 (todas as linhas)

	123 ref_year	123 co2_pc
3	1,980	3.43
4	1,984	3.77
5	1,988	3.93
6	1,992	5.99
7	1,996	5.96
8	2,000	10.7
9	2,004	10.5
10	2,008	10.8
11	2,012	4.92
12	2,016	5.27

As emissões aumentam até 2008 e depois caem abruptamente, mostrando que não há correlação entre Jogos Olímpicos e emissões de CO<sub>2</sub>, lembrando que Portugal nunca sediou uma olimpíada de verão.

## Sumarizando dados

Uma ferramenta importante para a compreensão de uma base de dados é a estatística descritiva. Com ela, por meio das medidas de posição e de dispersão, podemos sumarizar cada uma das colunas numéricas da base de dados.

A linguagem SQL possui algumas funções de sumarização<sup>5</sup> de modo nativo, como a soma e a média. Porém, não possui, nativamente, funções para medidas de dispersão, que podem ser calculadas ou fornecidas pelo fabricante do banco de dados que estamos usando. Mas, nesse caso, é necessário consultar a documentação elaborada pelo fabricante e ter em mente que a nossa consulta pode não rodar em outro banco de dados, uma vez que estamos usando uma função não padrão.

A lista das funções de sumarização padrão está na tabela 8.

Tabela 8 – Funções de sumarização

operador	significado
SUM()	soma de uma coluna
COUNT()	contagem de linhas
MAX()	valor máximo de uma coluna
MIN()	valor mínimo de uma coluna
AVG()	média aritmética de uma coluna

Todas as funções, exceto COUNT(), ignoram valores nulos.

Podemos agora calcular o PIB *per capita* médio dos países das Américas no ano de 2020.

---

<sup>5</sup> Alguns autores se referem às funções de sumarização como funções de agregação.

```

SELECT AVG(gp.gdp_pc) AS avg_gdp_pc
FROM gdp_pc gp
JOIN country c ON gp.country = c.country
WHERE gp.ref_year = 2020
AND c.four_regions = 'americas';

```

#### Bloco de código 58.

O valor é 15.199,12 USD, na média, para cada habitante do continente americano no ano de 2020.

Figura 55 – Resultado do bloco de código 58 (todas as linhas)

	123 avg_gdp_pc
1	16,484.8571428571

Podemos também aplicar todas as funções-sumário em uma tabela. Por exemplo, calcular, em relação à renda *per capita* média dos habitantes da Europa, a soma, a média, o valor máximo, o valor mínimo e a contagem de países no ano de 2020:

```

SELECT SUM(ai.mean_usd) AS "sum",
       AVG(ai.mean_usd) AS mean,
       MAX(ai.mean_usd),
       MIN(ai.mean_usd) AS "min",
       COUNT(ai.mean_usd) AS "count"
FROM avg_income ai
JOIN country c ON ai.country = c.country
WHERE ai.ref_year = 2020
AND c.four_regions = 'europe';

```

#### Bloco de código 59.

Veja que é bastante comum usarmos apelidos para as novas colunas que estamos criando com a função de sumarização. Propositalmente, deixamos a coluna de **MAX()** sem apelido para você ver como fica o nome da tabela. É usada a própria função e o nome da coluna.

Figura 56 – Resultado do bloco de código 59 (todas as linhas)

	123 sum	123 mean	123 MAX(ai.mean_usd)	123 min	123 count
1	1,935.73	42.0810869565	93.3	7.26	46

A soma das rendas médias diárias *per capita* dos países europeus é 1.880,93 USD; a média é 41,8 USD; o país com maior renda média tem um valor diário de 93,3 USD *per capita*, e o de menor renda, 7,26 USD. Fizeram parte do cálculo 45 países, que é o resultado de **COUNT()**. Se você fizer a conta, irá perceber que **AVG()** é **SUM()/COUNT()**.

## UNIDADE 2 – AGRUPANDO DADOS

Até o momento, utilizamos as funções de sumarização sobre os dados completos das tabelas ou sobre um recorte dos dados. Mas uma funcionalidade muito útil da SQL é a de agrupar dados sob determinados critérios de modo que possamos realizar a sumarização mediante esses critérios.

É isto que veremos nesta unidade.

### Comando GROUP BY

Na seção anterior vimos o PIB médio *per capita* das Américas no ano de 2020. Mas, e se quisermos investigar a variação desse PIB ao longo de cinco anos, dois anos antes de 2020 e dois anos depois? Nesse caso, utilizamos a cláusula **GROUP BY**:

```
SELECT gp.ref_year,  
       AVG(gp.gdp_pc) AS avg_gdp_pc  
  FROM gdp_pc gp  
  JOIN country c ON gp.country = c.country  
 WHERE gp.ref_year BETWEEN 2018 AND 2022  
   AND c.four_regions = 'americas'  
 GROUP BY gp.ref_year;
```

Bloco de código 60.

Sempre colocamos o **GROUP BY** no final da *query*, mas não precisa ser necessariamente o último. Podemos utilizar um **ORDER BY** para garantir que os dados apareçam em uma certa ordem. No caso da nossa consulta, eles apareceram por ordem de ano, que é o que queríamos, mas nem sempre o **GROUP BY** traz os dados ordenados. Além disso, há outras cláusulas SQL que podemos usar depois do **GROUP BY**, como logo veremos.

Geralmente, os campos que colocamos no **GROUP BY** são os mesmos que colocamos no **SELECT**, com exceção das funções de sumarização. Então, se queremos saber a média por ano, colocamos *ref\_year* no **SELECT** e *ref\_year* no **GROUP BY**.

Figura 57 – Resultado do bloco de código 60 (todas as linhas)

	123 ref_year	123 avg_gdp_pc
1	2,018	18,167.4285714286
2	2,019	18,220.2857142857
3	2,020	16,484.8571428571
4	2,021	17,536.2857142857
5	2,022	18,682

Notamos, nos resultados, o "efeito Covid-19" em 2020, que derrubou o PIB médio *per capita*.

Podemos colocar mais de uma coluna para fazer o agrupamento. Investiguemos agora se esse "efeito Covid-19" acontece também nos demais continentes no mesmo período:

```

SELECT c.four_regions,
    gp.ref_year,
    ROUND(AVG(gp.gdp_pc), 2) AS avg_gdp_pc
FROM gdp_pc gp
JOIN country c ON gp.country = c.country
WHERE gp.ref_year BETWEEN 2018 AND 2022
GROUP BY c.four_regions,
    gp.ref_year
ORDER BY c.four_regions,
    gp.ref_year;

```

Bloco de código 61.

Agora, temos dois campos no **SELECT**, então repetimos esses mesmos dois campos no **GROUP BY**. Para garantir que os dados venham ordenados do jeito que queremos, usamos um **ORDER BY** com os mesmos dois campos. Note que utilizamos a função **ROUND(valor, n)**, em que *valor* é o valor que queremos arredondar e *n* é o número de casas decimais para tornar os números mais fáceis de visualizar. **ROUND()** não é uma função-padrão da SQL, mas está presente praticamente em todos os bancos de dados.

Figura 58 – Resultado do bloco de código 61 (todas as linhas)

	RBC four_regions	123 ref_year	123 avg_gdp_pc
1	africa	2,018	5,765.09
2	africa	2,019	5,746.19
3	africa	2,020	5,259.36
4	africa	2,021	5,473.36
5	africa	2,022	5,632.51
6	americas	2,018	18,167.43
7	americas	2,019	18,220.29
8	americas	2,020	16,484.86
9	americas	2,021	17,536.29
10	americas	2,022	18,682
11	asia	2,018	20,645.76
12	asia	2,019	20,860.17
13	asia	2,020	19,924.92
14	asia	2,021	20,745.25
15	asia	2,022	21,306.44
16	europe	2,018	42,595.65
17	europe	2,019	43,450
18	europe	2,020	41,680.43
19	europe	2,021	44,113.04
20	europe	2,022	45,610.87

Notamos nos resultados que o efeito Covid-19 acontece nas quatro regiões da classificação da Gapminder. Além disso, podemos notar um *ranking* em termos de PIB médio *per capita*. Os valores mais altos estão na Europa, em seguida vem Ásia, América e África. O PIB médio *per capita* europeu é quase nove vezes maior que o da África.

## Selecionando por valor agrupado

Quando fazemos um agrupamento podemos selecionar somente as linhas que atinjam um determinado valor na função de agrupamento. Para isso, utilizamos a cláusula **HAVING**. Por exemplo, na *query* do PIB *per capita* médio, podemos querer trazer apenas aqueles que ultrapassam o valor de 17 mil USD por dia. Para isso fazemos:

```

SELECT c.four_regions,
       gp.ref_year,
       ROUND(AVG(gp.gdp_pc), 2) AS avg_gdp_pc
FROM gdp_pc gp
JOIN country c ON gp.country = c.country
WHERE gp.ref_year BETWEEN 2018 AND 2022
GROUP BY c.four_regions,
         gp.ref_year
HAVING avg_gdp_pc > 17000
ORDER BY c.four_regions,
         gp.ref_year;
    
```

Bloco de código 62.

Na cláusula **HAVING**, colocamos o critério que queremos que as linhas resultantes tenham.

Figura 59 – Resultado do bloco de código 62 (todas as linhas)

	RBC four_regions	123 ref_year	123 avg_gdp_pc
1	americas	2,018	18,167.43
2	americas	2,019	18,220.29
3	americas	2,021	17,536.29
4	americas	2,022	18,682
5	asia	2,018	20,645.76
6	asia	2,019	20,860.17
7	asia	2,020	19,924.92
8	asia	2,021	20,745.25
9	asia	2,022	21,306.44
10	europe	2,018	42,595.65
11	europe	2,019	43,450
12	europe	2,020	41,680.43
13	europe	2,021	44,113.04
14	europe	2,022	45,610.87

Podemos também colocar uma faixa de valores. Por exemplo, PIB médio *per capita* entre 17 mil e 20 mil USD por dia:

```
SELECT c.four_regions,
       gp.ref_year,
       ROUND(AVG(gp.gdp_pc), 2) AS avg_gdp_pc
  FROM gdp_pc gp
 JOIN country c ON gp.country = c.country
 WHERE gp.ref_year BETWEEN 2018 AND 2022
 GROUP BY c.four_regions,
          gp.ref_year
 HAVING avg_gdp_pc BETWEEN 17000 AND 20000
 ORDER BY c.four_regions,
          gp.ref_year;
```

Bloco de código 63.

Na cláusula **HAVING**, podemos utilizar os mesmos operadores que usamos na cláusula **WHERE**.

Figura 60 – Resultado do bloco de código 63 (todas as linhas)

	<b>ABC four_regions</b>	<b>123 ref_year</b>	<b>123 avg_gdp_pc</b>
1	americas	2,018	18,167.43
2	americas	2,019	18,220.29
3	americas	2,021	17,536.29
4	americas	2,022	18,682
5	asia	2,020	19,924.92

Além disso, também podemos escolher as linhas que não queremos que venham na consulta. Por exemplo, valores de PIB maiores do que 17 mil USD por dia:

```
SELECT c.four_regions,
       gp.ref_year,
       ROUND(AVG(gp.gdp_pc), 2) AS avg_gdp_pc
  FROM gdp_pc gp
 JOIN country c ON gp.country = c.country
 WHERE gp.ref_year BETWEEN 2018 AND 2022
 GROUP BY c.four_regions,
           gp.ref_year
 HAVING NOT avg_gdp_pc > 17000
 ORDER BY c.four_regions,
           gp.ref_year;
```

Bloco de código 64.

Nesse caso, utilizamos **HAVING NOT** para negar a expressão lógica seguinte.

Figura 61 – Resultado do bloco de código 64 (todas as linhas)

	<b>ABC four_regions</b>	<b>123 ref_year</b>	<b>123 avg_gdp_pc</b>
1	africa	2,018	5,765.09
2	africa	2,019	5,746.19
3	africa	2,020	5,259.36
4	africa	2,021	5,473.36
5	africa	2,022	5,632.51
6	americas	2,020	16,484.86

## Contando linhas

É bem comum contarmos a quantidade de linhas em uma tabela. Para isso, utilizamos a função de sumarização COUNT(). Nesse caso, as contagens podem ser de várias formas diferentes.

Começamos contando o total de linhas da tabela *women\_years\_at\_school*.

```
SELECT count(*)  
FROM women_years_at_school;
```

**Bloco de código 65.**

Existem 7000 mil linhas na tabela, como mostra a figura 62:

Figura 62 – Resultado do bloco de código 65 (todas as linhas)

	123 count(*)
1	7,000

Agora vamos ver quantos países distintos temos nesta tabela:

```
SELECT count(DISTINCT country)  
FROM women_years_at_school;
```

**Bloco de código 66.**

Existem 175 países diferentes, como mostra a seguir:

Figura 63 – Resultado do bloco de código 66 (todas as linhas)

	123 count(DISTINCT country)
1	175

E quantos anos distintos temos? Para saber a resposta, usamos:

```
SELECT count(DISTINCT ref_year)  
FROM women_years_at_school;
```

**Bloco de código 67.**

Existem 40 anos diferentes, como mostra a figura 64:

Figura 64 – Resultado do bloco de código 67 (todas as linhas)

	123 count(DISTINCT ref_year)
1	40

Poderíamos ter buscado nessa consulta também o ano mais antigo e o mais recente para os quais temos dados:

```
SELECT count(DISTINCT ref_year), min(ref_year), max(ref_year)
FROM women_years_at_school wyas;
```

#### Bloco de código 68.

Agora sabemos que a faixa de 40 anos de dados vai de 1970 a 2009.

Figura 65 – Resultado do bloco de código 68 (todas as linhas)

	123 count(DISTINCT ref_year)	123 min(ref_year)	123 max(ref_year)
1	40	1,970	2,009

Vamos agora contar quantas linhas temos por país:

```
SELECT country, count(*)
FROM women_years_at_school wyas
GROUP BY country;
```

#### Bloco de código 69.

Rolando a tabela toda, vemos que temos 40 linhas para cada um dos países. Isso nos mostra que todos os países têm linhas para os 40 anos, que é a faixa de anos. Não podemos ainda falar que temos dados para todos os anos e todos os países porque não contamos se há linhas nulas.

Figura 66 – Resultado do bloco de código 69 (10 primeiras linhas)

	RBC country	123 count(*)
1	Afghanistan	40
2	Albania	40
3	Algeria	40
4	Angola	40
5	Antigua and Barbuda	40
6	Argentina	40
7	Armenia	40
8	Australia	40
9	Austria	40
10	Azerbaijan	40

Vamos então contar as linhas nulas para o campo *mean\_years*:

```
SELECT count(*)
FROM women_years_at_school
WHERE mean_years IS NULL;
```

#### Bloco de código 70.

Como o resultado foi zero, sabemos que não temos nenhuma linha com o valor de *mean\_years* nulo e podemos afirmar que temos dados para todos os anos em todos os países.

Figura 67 – Resultado do bloco de código 70 (todas as linhas)

	123 count(*)
1	0

Podemos olhar a mesma coisa de outra perspectiva, contando o número de linhas por ano:

```
SELECT ref_year, count(*)
FROM women_years_at_school wyas
GROUP BY ref_year;
```

#### Bloco de código 71

Se visualizarmos todas as linhas retornadas na consulta, percebemos que há 175 linhas para cada ano, que é o número de diferentes países. Portanto, vindo por outro caminho, chegamos à conclusão de que temos dados para todos os 40 anos e para os 175 países.

Figura 68 – Resultado do bloco de código 71 (todas as linhas)

1	123 count(*)	0

## UNIDADE 3 – *SUBQUERIES*

Uma ferramenta que expande bastante o leque de consultas que podemos realizar são as *subqueries*, que significa executar uma consulta baseado no resultado de outra consulta.

Por exemplo, queremos saber qual é o país que tem a menor expectativa de vida no ano 2000. Podemos fazer uma consulta na tabela *life\_expectancy* filtrando por *ref\_year* = 2000, usar a função de sumarização **MIN()** para pegar o menor valor de *tot\_years*, anotar o resultado em um papel e depois fazer outra consulta para pegar a lista de *country* cujo valor de *tot\_years* é igual ao que anotamos.

Ou podemos fazer direto usando uma *subquery*:

```
SELECT le.country, le.tot_years
FROM life_expectancy le
WHERE le.ref_year = 2000
AND le.tot_years =
    SELECT MIN(le2.tot_years)
    FROM life_expectancy le2
    WHERE le2.ref_year = 2000
);
```

Bloco de código 72.

Primeiramente, o banco de dados executa a *query* que está entre parênteses, que chamamos de *subquery*, e recebe um único resultado. Em seguida, o banco executa a *query* mais externa usando esse valor que recebeu na sua cláusula **WHERE**.

Veja que tomamos o cuidado de utilizar apelidos diferentes para as duas *queries*, pois estamos executando-as sobre a mesma tabela. Os apelidos diferentes evitam ambiguidades. Veja também que filtramos por *ref\_year* = 2000 nas duas consultas. Se não fizéssemos isso na *subquery*, iríamos receber o valor mínimo global da tabela, independentemente do ano. Além disso, se não fizéssemos isso na *query* principal, correríamos o risco de trazer também países que têm a expectativa de vida igual à expectativa de vida mínima não só no ano 2000 mas também em outros anos. Os testes das *queries* sem o filtro você pode fazer nos exercícios para ver o resultado.

Figura 69 – Resultado do bloco de código 72 (todas as linhas)

	RBC country	123 ref_year	123 tot_years
1	Central African Republic	2,000	43.8

Na figura 69, temos o resultado da consulta como queríamos. Podemos ver não só a expectativa de vida mínima mas também a máxima em uma mesma consulta fazendo duas *subqueries*:

```

SELECT /e.country,
       /e.ref_year,
       /e.tot_years
  FROM life_expectancy /e
 WHERE /e.ref_year = 2000
   AND (
        /e.tot_years = (
            SELECT MIN(/e2.tot_years)
              FROM life_expectancy /e2
             WHERE /e2.ref_year = 2000
        )
      OR /e.tot_years = (
            SELECT MAX(/e3.tot_years)
              FROM life_expectancy /e3
             WHERE /e3.ref_year = 2000
        )
    );

```

#### Bloco de código 73.

A consulta fica um pouco mais complicada. Veja que tomamos novamente o cuidado com os apelidos, agora com as três *queries*, e tivemos que colocar o OR entre parênteses para agrupar esses resultados, caso contrário a *query* seria executada da esquerda para a direita e o resultado estaria errado.

Figura 70 – Resultado do bloco de código 73 (todas as linhas)

	RBC country	123 ref_year	123 tot_years
1	Central African Republic	2,000	43.8
2	Japan	2,000	81.6

Outra forma de usarmos *subqueries* é criando tabelas fictícias, denominadas visões (*views*), para depois realizarmos consultas sobre essas visões. Por exemplo, as tabelas de tempo na escola que temos estão separadas por gênero e já fizemos alguns exemplos juntando essas tabelas na seção sobre **UNION**.

Podemos agora juntá-las novamente e calcular a escolaridade média dos homens e das mulheres, de 2000 a 2009, nos países da região africana. Nesse caso, queremos saber o país com menor tempo na escola para os homens e o com menor tempo na escola para as mulheres. A consulta fica assim:

```
SELECT tab.country,
       tab.gender,
       ROUND(AVG(tab.mean_years), 2) as avg_mean_years
  FROM (
    SELECT m.country,
           m.ref_year,
           'male' as gender,
           m.mean_years
      FROM men_years_at_school m
     UNION ALL
    SELECT w.country,
           w.ref_year,
           'female' as gender,
           w.mean_years
      FROM women_years_at_school w
  ) AS tab
  JOIN country c ON tab.country = c.country
 WHERE c.four_regions = 'africa'
   AND tab.ref_year BETWEEN 2000 AND 2009
 GROUP BY tab.country, tab.gender
 ORDER BY tab.gender, avg_mean_years;
```

#### Bloco de código 74.

Veja que realizamos a união das duas tabelas, colocamos essa união entre parênteses e a apelidamos de *tab*. Depois fazemos a consulta em cima de *tab*, por meio de um **INNER JOIN** com *country*, agrupando, calculando a média e ordenando os dados, como fazemos com uma tabela normal.

Figura 71 – Resultado do bloco de código 74 (5 primeiras linhas do bloco feminino e 5 primeiras linhas do bloco masculino)

	RBC country	RBC gender	123 avg_mean_years
1	Niger	female	0.5
2	Chad	female	0.59
3	Burkina Faso	female	0.62
4	Mali	female	0.71
5	Ethiopia	female	0.76
52	Niger	male	1.22
53	Burkina Faso	male	1.47
54	Mali	male	1.75
55	Ethiopia	male	1.97
56	Chad	male	2.35

A figura 71 mostra o resultado. Notamos que são os mesmos países, em diferente ordem, e que os homens ficam mais tempo na escola que as mulheres.

## Cláusula EXISTS

As *subqueries* podem ser correlacionadas, de modo que utilizamos colunas da *query* no corpo da *subquery*. Nesse caso, empregamos a cláusula **EXISTS**, que retorna verdadeiro se a *subquery* trouxer alguma linha ou falso se a *subquery* voltar vazia.

Por exemplo, se quisermos encontrar, no ano 2000, todos os países com emissões de CO<sub>2</sub> *per capita* maiores do que as emissões *per capita* dos Estados Unidos, a consulta é realizada da seguinte forma:

```

SELECT *
FROM co2_emissions_pc cep
WHERE cep.ref_year = 2000
AND EXISTS
(SELECT *
FROM co2_emissions_pc cep2
WHERE cep2.country = 'USA'
AND cep2.ref_year = 2000
AND cep.co2_pc > cep2.co2_pc
)

```

Bloco de código 75.

Observe que fazemos, na *subquery*, *co2\_pc* da tabela fora da *subquery* maior do que *co2\_pc* da tabela da *subquery*.

Figura 72 – Resultado do bloco de código 75 (todas as linhas)

	RBC country	123 ref_year	123 co2_pc
1	UAE	2,000	28.3
2	Qatar	2,000	24.1
3	Singapore	2,000	30.9

Poderíamos ter feito a pergunta "ao contrário" e iríamos obter o mesmo resultado:

```
SELECT *
FROM co2_emissions_pc cep
WHERE cep.ref_year = 2000
AND NOT EXISTS
(SELECT *
FROM co2_emissions_pc cep2
WHERE cep2.country = 'USA'
AND cep2.ref_year = 2000
AND cep.co2_pc <= cep2.co2_pc
)
```

#### Bloco de código 76.

Dessa vez, nós negamos o EXISTS e fazemos, na *subquery*, *co2\_pc* da tabela fora da *subquery* menor ou igual do que *co2\_pc* da tabela da *subquery*.

Figura 73 – Resultado do bloco de código 76 (todas as linhas)

	RBC country	123 ref_year	123 co2_pc
1	UAE	2,000	28.3
2	Qatar	2,000	24.1
3	Singapore	2,000	30.9





# MÓDULO IV – MANIPULANDO DADOS

Neste módulo, veremos os comandos da linguagem de manipulação de dados (DML<sup>6</sup>) da SQL, cuja finalidade é o de alterar o conteúdo de uma tabela, ou seja, os dados armazenados. Mais especificamente, os comandos se destinam a inserir novas linhas, remover e alterar linhas existentes.

Outro aspecto da SQL que iremos tratar é a linguagem de definição de dado (DDL<sup>7</sup>), que tem como propósito alterar a estrutura do banco de dados, criando novas tabelas e removendo e alterando tabelas existentes.

## UNIDADE 1 – COMANDOS DML

Os comandos DML são aqueles utilizados para alterar o conteúdo de uma tabela, quer seja pela inserção de novas linhas – o comando **INSERT** –, quer seja pela remoção de linhas existentes – o **DELETE** –, quer seja pela alteração de valores de algumas linhas – o comando **UPDATE**.

### Inserindo novas linhas

Geralmente, conseguimos as nossas bases de dados a partir de bancos de dados conhecidos, como o Ipeadata (Ipea, 2024), o Sistema Gerenciador de Séries Temporais (Banco Central do Brasil, 2024), o Sidra (IBGE, 2024), o Fred Economic Data (Federal Reserve Bank of St. Louis, 2024), o OECD Data (OCDE, 2024), o IMF Data (Fundo Monetário Internacional, 2024), o World Bank Open Data (Banco Mundial, 2024), o Our World in Data (Oxford University, 2024), o Portal Brasileiro de Dados Aberto (Governo Federal do Brasil, 2024) e o Gapminder (Gapminder, 2024), os quais são as principais fontes dos dados que temos utilizado nos nossos exemplos.

---

<sup>6</sup> *Data Manipulation Language*

<sup>7</sup> *Data Definition Language*

Entretanto, pode ser o caso de termos que incluir manualmente novas linhas nas tabelas. Para isso, utilizamos o comando **INSERT**.

A nossa base de cotações de ações da Petrobrás na Nasdaq vai até 29/12/2023. Suponha que tenhamos obtido os dados dos primeiros cinco dias úteis de 2024, dos dias 2 a 5/1 e queiramos inserir manualmente esses dias na base.

Figura 74 – Cotações das ações da Petrobrás (BR) em USD na Nyse de 2/1/2024 a 8/1/2024

Date	Open	High	Low	Close ⓘ	Adj Close ⓘ
Jan 8, 2024	16.23	16.28	15.97	16.28	14.80
Jan 5, 2024	16.57	16.65	16.37	16.54	15.04
Jan 4, 2024	16.54	16.79	16.35	16.35	14.87
Jan 3, 2024	16.04	16.64	16.00	16.58	15.08
Jan 2, 2024	16.09	16.22	15.92	16.00	14.55

Fonte: Yahoo! Finance<sup>8</sup>.

O comando para inserir uma linha, a referente ao dia 2/1, fica assim:

```
INSERT INTO petrobras ("Date", "Open", High, Low, "Close",
                      "Adj Close", Volume)
VALUES ('2024-01-02', 16.09, 16.22, 15.92, 16.00, 14.55,
       10100.0);
```

#### Bloco de código 76.

Note que, após o nome da tabela, listamos todos os campos. A ordem não importa. Mas, na cláusula VALUES, temos que colocar valores para todos os campos na ordem que os listamos entre parênteses. O valor '2024-01-02' vai para o campo "*Date*", e o valor 16.00 vai para o campo "*Close*" no exemplo do bloco de código 76.

Caso não saibamos algum valor, devemos escrever **NULL** no lugar dele. Porém, o campo deve ter sido configurado para aceitar **NULL**, caso contrário ocorrerá um erro, e a nova linha não será inserida.

No caso do nosso exemplo, o banco de dados retorna apenas que uma linha foi inserida. Mas se fizermos uma *query* para conferir o resultado, teremos:

---

<sup>8</sup> Disponível em: <https://finance.yahoo.com/quote/PBR/history/?period1=1704153600&period2=1704758400>. Acesso em: 5 jul. 2024.

```
SELECT *
FROM petrobras
WHERE "Date" > '2023-12-29';
```

**Bloco de código 77.**

Vemos realmente que a linha foi inserida.

Figura 74 – Resultado do bloco de código 77 (todas as linhas)

	RBC Date	123 Open	123 High	123 Low	123 Close	123 Adj Close	123 Volume
1	2024-01-02	16.09	16.22	15.92	16	14.55	10,100

Podemos também inserir as linhas em lote:

```
INSERT INTO petrobras ("Date", "Open", High, Low, "Close",
"Adj Close", Volume)
VALUES
('2024-01-03', NULL, NULL, NULL, NULL, NULL, NULL),
('2024-01-04', 16.54, 16.79, 16.35, 16.36, 14.87, 13954.7),
('2024-01-05', 16.57, 16.65, 16.07, 16.54, 15.04, 9872.0),
('2024-01-08', 16.23, 16.28, 15.97, 16.28, 14.80, 14011.8);
```

**Bloco de código 78.**

Nesse caso, listamos os campos entre parênteses logo após o nome da tabela e, depois de VALUES, colocamos várias sequências de valores entre parênteses, de modo análogo ao comando com uma só linha (bloco de código 76). Veja que propositalmente inserimos NULL nos valores da linha de 3/1/2024, mas logo consertaremos isso.

Rodando novamente a consulta do bloco de código 77, obtemos:

Figura 75 – Novo resultado do bloco de código 77 (todas as linhas)

	A BC Date	123 Open	123 High	123 Low	123 Close	123 Adj Close	123 Volume
1	2024-01-02	16.09	16.22	15.92	16	14.55	10,100
2	2024-01-03	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]
3	2024-01-04	16.54	16.79	16.35	16.36	14.87	13,954.7
4	2024-01-05	16.57	16.65	16.07	16.54	15.04	9,872
5	2024-01-08	16.23	16.28	15.97	16.28	14.8	14,011.8

## Alterando linhas

A atualização de valores em tabelas é feita por meio do comando **UPDATE**. É muito importante que utilizemos a cláusula **WHERE** para atualizarmos somente as linhas de interesse. O comando **UPDATE** sem **WHERE** executa normalmente, porém **atualiza todas as linhas** da tabela.

Para corrigirmos a linha referente à data de 3/1/2024 que inserimos com valores nulos no comando **INSERT** anterior, fazemos o seguinte:

```
UPDATE petrobras
SET High = 16.64
WHERE "Date" = '2024-01-03';
```

Bloco de código 79.

Na cláusula **SET**, listamos as colunas que queremos alterar e os seus respectivos valores. Rodando novamente a consulta do bloco de código 77, obtemos:

Figura 76 – Novo resultado do bloco de código 77 (todas as linhas)

	A BC Date	123 Open	123 High	123 Low	123 Close	123 Adj Close	123 Volume
1	2024-01-02	16.09	16.22	15.92	16	14.55	10,100
2	2024-01-03	[NULL]	16.64	[NULL]	[NULL]	[NULL]	[NULL]
3	2024-01-04	16.54	16.79	16.35	16.36	14.87	13,954.7
4	2024-01-05	16.57	16.65	16.07	16.54	15.04	9,872
5	2024-01-08	16.23	16.28	15.97	16.28	14.8	14,011.8

Note que o valor da coluna *High* para a linha de 3/1/2024 foi alterado conforme queríamos. Para mudar várias colunas de uma vez fazemos da seguinte forma:

```
UPDATE petrobras
SET "Open" = 16.04,
Low = 16.00,
"Close" = 16.58,
"Adj Close" = 15.08,
Volume = 18366.8
WHERE "Date" = '2024-01-03';
```

#### Bloco de código 80.

Na cláusula **SET**, listamos as colunas que queremos alterar e os seus respectivos valores. Rodando novamente a consulta do bloco de código 77, obtemos:

Figura 76 – Novo resultado do bloco de código 77 (todas as linhas)

	RBC Date	123 Open	123 High	123 Low	123 Close	123 Adj Close	123 Volume
1	2024-01-02	16.09	16.22	15.92	16	14.55	10,100
2	2024-01-03	16.04	16.64	16	16.58	15.08	18,366.8
3	2024-01-04	16.54	16.79	16.35	16.36	14.87	13,954.7
4	2024-01-05	16.57	16.65	16.07	16.54	15.04	9,872
5	2024-01-08	16.23	16.28	15.97	16.28	14.8	14,011.8

## Removendo linhas

O comando para apagar linhas de uma tabela é o **DELETE**. Nesse comando, não é preciso listar colunas, pois ele apaga a linha toda. Além disso, a mesma advertência feita com relação ao **UPDATE** é feita para o **DELETE**: não execute esse comando sem a cláusula **WHERE**. Caso contrário todas as linhas da tabela serão apagadas.

Nos exemplos anteriores inserimos as cotações da Petrobrás de 2/1 a 8/1/2024. Porém, queríamos apenas ter inserido os dados da primeira semana de janeiro. O dia 8/1 então está sobrando, e vamos apagá-lo:

```
DELETE
FROM petrobras
WHERE "Date" = '2024-01-08';
```

#### Bloco de código 81.

Rodando novamente a consulta do bloco de código 77, obtemos:

Figura 77 – Novo resultado do bloco de código 77 (todas as linhas)

	AAC Date	123 Open	123 High	123 Low	123 Close	123 Adj Close	123 Volume
1	2024-01-02	16.09	16.22	15.92	16	14.55	10,100
2	2024-01-03	16.04	16.64	16	16.58	15.08	18,366.8
3	2024-01-04	16.54	16.79	16.35	16.36	14.87	13,954.7
4	2024-01-05	16.57	16.65	16.07	16.54	15.04	9,872

## UNIDADE 2 – TRANSAÇÕES

O conceito de transação em SQL refere-se a uma sequência de comandos de manipulação de dados que só devem existir no banco se todos eles ocorrerem com sucesso. Caso um deles falhe, todos os outros devem ser desfeitos. Esse é apenas um dos aspectos de uma transação, mas é o principal.

Alguns bancos, para iniciar uma transação, exigem que você execute o comando **BEGIN TRANSACTION**. Outros bancos já estão com o modo de transação ligado, ou seja, você não precisa iniciar a transação com o **BEGIN TRANSACTION**, mas somente o fato de ter executado um comando **DML** é o suficiente para iniciar a transação.

O importante é que, se a sua sessão no banco está no modo de transação, os comandos **DML** que você executar somente serão efetivados se você executar um **COMMIT**. Isso irá salvar tudo que você fez no banco e encerrar a transação. Caso você tenha cometido algum erro ou algum comando tenha dado errado, você pode desfazer tudo desde o **BEGIN TRANSACTION** com o comando **ROLLBACK**. Esse comando irá apagar tudo o que você alterou, inclusive linhas deletadas, e vai encerrar a transação.

Caso o banco com o qual você está trabalhando exija o **BEGIN TRANSACTION** para começar uma transação, se você não o executar, todos os comandos que você executar serão imediatamente efetivados, dispensando o uso do **COMMIT**. E, neste caso, o **ROLLBACK** não irá funcionar. Aliás, é fundamental salientar que, depois de um **COMMIT** não é mais possível fazer **ROLLBACK**. E vice-versa.

**Importante:** apenas comandos **DML** têm transação. Comandos **DQL**, isto é, os **SELECTs** não tem controle de transação.

## UNIDADE 3 – ALGUNS COMANDOS DDL

Os comandos DDL são utilizados para a definição e manutenção das estruturas de dados. Servem para criar tabelas, para alterar a estrutura de tabelas, adicionando campos ou mudando os seus nomes, e para remover tabelas que não são mais necessárias.

Esta unidade também apresenta um modo simples de limpar o conteúdo de uma tabela.

### Criando tabelas

Uma das formas de se criar uma tabela é a partir da importação de um arquivo CSV<sup>9</sup>. Porém, você pode criar tabelas manualmente utilizando o comando **CREATE TABLE**.

Por exemplo, se quiséssemos ter criado a tabela *child\_mortality* manualmente, o comando seria:

```
CREATE TABLE "child_mortality" (
    "country" TEXT NULL,
    "ref_year" INTEGER NULL,
    "tot_deaths" INTEGER NULL
);
```

Bloco de código 82.

Verificando a estrutura da tabela, vemos:

Figura 78 – Estrutura da tabela *child\_mortality* resultante do bloco de código 82

Column Name	#	Data Type	Length	Not Null	Auto Increment
ABC country	1	TEXT	[ ]	[ ]	[ ]
123 ref_year	2	INTEGER	[ ]	[ ]	[ ]
123 tot_deaths	3	INTEGER	[ ]	[ ]	[ ]

Vamos criar agora duas tabelas com chaves primárias e um relacionamento entre elas. As tabelas serão *uf* e *cidade*. Primeiramente, iremos criar a tabela *uf*, pois ela será necessária na hora de criar *cidade*, pois *uffará* referência à *cidade*.

<sup>9</sup> Comma separated values, ou valores separados por vírgula (no formato brasileiro geralmente é ponto e vírgula).

```

CREATE TABLE "uf" (
    "cd_uf" CHAR(2) NOT NULL PRIMARY KEY,
    "descr_uf" VARCHAR(50) NOT NULL
);

```

Bloco de código 83.

Verificando a estrutura da tabela, vemos os campos com os seus diferentes tipos de caracteres e observamos que ambos não aceitam valores nulos:

Figura 78 – Estrutura da tabela *uf* resultante do bloco de código 83

Column Name	#	Data Type	Length	Not Null
cd_uf	1	CHAR		[v]
descr_uf	2	VARCHAR		[v]
Name	Owner	Type		
UF_PK	uf	PRIMARY KEY		
cd_uf	—	—		

O campo **CHAR** tem tamanho fixo. No caso de *cd\_uf*, irá ocupar o espaço de dois caracteres mesmo que só seja inserido um. O campo **VARCHAR** também é caractere, mas é de tamanho variável. Nós o criamos com tamanho 50, que é o tamanho máximo. Mas se inserirmos quatro caracteres, o campo irá ocupar apenas o espaço de quatro caracteres.

Vamos criar agora a tabela *cidade* que terá um relacionamento com *uf*, isto é, receberá a chave primária de *uf* como chave estrangeira.

```

CREATE TABLE "cidade" (
    "cd_ibge" NUMERIC(8) NOT NULL PRIMARY KEY,
    "descr_municipio" VARCHAR(100) NOT NULL,
    "cd_uf" CHAR(2) NOT NULL,
    FOREIGN KEY ("cd_uf")
    REFERENCES "uf"("cd_uf")
);

```

Bloco de código 84.

Verificando a estrutura da tabela, vemos os campos com os seus diferentes tipos caracteres e observamos que ambos não aceitam valores nulos:

Figura 79 – Estrutura da tabela *cidade* resultante do bloco de código 84

Column Name	#	Data Type	Length	Not Null
123 cd_ibge	1	NUMERIC		[v]
ABC descr_munic	2	VARCHAR		[v]
ABC cd_uf	3	CHAR		[v]
Name	Owner	Type		
▼ - CIDADE_PK cidade		PRIMARY KEY		
123 cd_ibge	-	-		
Name	Owner	Ref Table	Type	Ref Object
▼ FK_cidade_uf cidade	uf		FOREIGN KEY	UF_PK
ABC cd_uf	-	-	-	cd_uf

Quando as tabelas estão relacionadas entre si, o banco de dados nos presta alguns serviços de validação. Ele não deixa que cadastremos na tabela *cidade* uma cidade com *cd\_uf* que não esteja previamente cadastrado na tabela *uf*. Outro serviço importante para a manutenção da integridade dos dados é não permitir que se apague uma UF da tabela *uf* caso alguma cidade esteja usando o seu código.

E o uso de chaves primárias evita que sejam cadastradas UFs com o mesmo *cd\_uf* na tabela *uf* e cidades com o mesmo código IBGE (*cd\_ibge*) na tabela *cidade*.

## Alterando tabelas

Com o comando **ALTER TABLE**, é possível fazer uma série de alterações na estrutura de uma tabela, como alterar o nome, adicionar ou remover colunas, alterar o nome e o tipo de colunas, transformar um campo em chave primária, mudar uma coluna que aceita nulos para **NOT NULL**, e assim por diante.

Existem algumas limitações importantes, entretanto, se já houver linhas cadastradas na tabela. Por exemplo, se alguma coluna tiver um valor nulo cadastrado, tentar transformá-la em **NOT NULL** irá causar um erro. Tentar transformar uma coluna em chave primária também irá causar erro se ela contiver valores repetidos. Além disso, a adição de uma coluna **NOT NULL** em uma tabela que possui linhas cadastradas tem de ser feita em três etapas: primeiro adicionar a coluna como **NULL**, depois fazer **UPDATE** nas linhas preenchendo essa coluna com valores e, finalmente, fazer outro **ALTER TABLE** para transformar a coluna em **NOT NULL**.

As operações relacionadas no parágrafo anterior não estão disponíveis em todos os bancos de dados. Alguns não permitem que haja a remoção de colunas, por exemplo, e então é necessário criar uma nova tabela sem a coluna indesejada e transferir os dados da antiga para a nova para depois remover a antiga e renomear a nova. Bastante trabalhoso.

Vamos fazer um exemplo simples mas frequente, que é adicionar colunas. Se quisermos adicionar na tabela *cidade* um campo para o número de habitantes e outro para a área em km<sup>2</sup>, podemos fazer assim:

```
ALTER TABLE cidade
ADD COLUMN "nr_habitantes" INTEGER NULL;

ALTER TABLE cidade
ADD COLUMN "area_km2" NUMERIC(15,2) NULL;
```

Bloco de código 85.

Verificando a estrutura da tabela, podemos ver as duas novas colunas:

Figura 80 – Estrutura da tabela *cidade* resultante do bloco de código 85

Column Name	#	Data Type	Length	Not Null
cd_ibge	1	NUMERIC		[v]
descr_municipio	2	VARCHAR		[v]
cd_uf	3	CHAR		[v]
nr_habitantes	4	INTEGER		[ ]
area_km2	5	NUMERIC		[ ]

## Removendo tabelas

Quando não queremos mais uma tabela em um banco de dados podemos simplesmente apagá-la. O comando para isso é o **DROP TABLE**.

Por exemplo, vamos apagar as duas tabelas que criamos, *cidade* e *uf*. Os comandos para isso são:

```
DROP TABLE uf;

DROP TABLE cidade;
```

Bloco de código 86.

O banco simplesmente apaga as duas tabelas, sem fazer perguntas, eliminando todos os dados e relacionamentos.

**Cuidado:** comandos DDL, como **DROP TABLE**, **CREATE TABLE** e **ALTER TABLE** não são passíveis de **ROLLBACK**. Uma vez executados não é possível desfazê-los.

## Limpando tabelas

Um meio rápido de se apagar todos os dados de uma tabela é o comando **TRUNCATE TABLE**. Ele é melhor que o **DELETE**, pois é praticamente instantâneo, e libera o espaço alocado pela tabela, coisa que o **DELETE** não faz. Mas ele tem um problema, pois como é um comando DDL, ele não é passível de **ROLLBACK**. Já o **DELETE** é, se estiver sendo executado no contexto de uma transação.

Para limpar a tabela *petrobras* poderíamos rodar:

```
TRUNCATE TABLE petrobras;
```

**Bloco de código 87.**

Nesse caso, dizemos que poderíamos porque o SQLite, que estamos utilizando como SGBD, não implementa o comando **TRUNCATE**, mas vários outros bancos de dados o implementam.

# REFERÊNCIAS BIBLIOGRÁFICAS

CARDOSO, G. C.; CARDOSO, V. M. **Linguagem SQL, fundamentos e práticas**. São Paulo: SRV Editora LTDA, 2013. ISBN 9788502200463.

DATE, C. J. **Introdução a sistemas de bancos de dados**. Rio de Janeiro: Grupo GEN, 2004. ISBN 8535212736.

ELMASRI, R.; NAVATHE, S. B. **Sistemas de bancos de dados**. São Paulo: Editora Pearson Universidades, 2019. ISBN 8543025001.

MINISTÉRIO DA SAÚDE (Brasil). DATASUS. **Qualificação de Indicadores do IDB-1997: taxa de mortalidade infantil**. [s. l.], 1997. Disponível em: <http://tabnet.datasus.gov.br/cgi/idb1997/mort/fqc02.htm>. Acesso em: 19 jun. 2024.

MINISTÉRIO DA SAÚDE (Brasil). DATASUS. **Qualificação de Indicadores do IDB-2000: taxa de fecundidade total**. [s. l.], 2000. Disponível em: <http://tabnet.datasus.gov.br/cgi/idb2000/fqa05.htm>. Acesso em: 25 jun. 2024.

NIELD, Thomas. **Introdução à linguagem SQL: abordagem prática para iniciantes**. São Paulo: Novatec Editora, 2016. ISBN 8575225014.

RAMAKRISHNAN, R.; GEHRKE, J. **Sistemas de gerenciamento de bancos de dados**. [s.l.]: Editora Grupo A, 2008. E-book. ISBN 8577260275.

ZHAO, A. **SQL – Guia Prático: um guia para o uso de SQL**. 4ª edição. São Paulo: Novatec Editora LTDA, 2023. ISBN 8575228315.

## Bases de dados

BANCO CENTRAL DO BRASIL. **Sistema Gerenciador de Séries Temporais**. Disponível em: <https://www3.bcb.gov.br/sgspub>. Acesso em 5. jul. 2024.

BANCO MUNDIAL. **World Bank Open Data**. Disponível em: <https://data.worldbank.org>. Acesso em 5. jul. 2024.

BRASIL. Governo Federal. **Portal Brasileiro de Dados Abertos**. Disponível em: <https://dados.gov.br/home>. Acesso em 5. jul. 2024.

FMI. Fundo Monetário Internacional. **IMF Data**. Disponível em: <https://www.imf.org/en/Data>. Acesso em 5. jul. 2024.

FEDERAL RESERVE BANK OF ST. LOUIS. **FRED Economic Data**. Disponível em: <https://fred.stlouisfed.org>. Acesso em 5. jul. 2024.

GAPMINDER. **Download the data**. Disponível em: <https://www.gapminder.org/data/>. Acesso em 23. jun. 2024.

IBGE. **SIDRA – Sistema IBGE de Recuperação Automática**. Disponível em: <https://sidra.ibge.gov.br>. Acesso em 5. jul. 2024.

IPEA. Ipeadata: coletânea de dados macroeconômicos, regionais e sociais. Disponível em: <http://www.ipeadata.gov.br/Default.aspx>. Acesso em 5. jul. 2024.

OCDE. OECD Data. Disponível em: <https://www.oecd.org/en/data.html>. Acesso em 5. jul. 2024.

OXFORD UNIVERSITY. Our World in Data. Disponível em: <https://ourworldindata.org>. Acesso em 5. jul. 2024.

# APÊNDICE I – DICIONÁRIO DE DADOS

Em todos os exemplos deste curso, utilizamos um conjunto de tabelas obtidas da base de dados do Gapminder<sup>10</sup>, uma organização sueca sem fins lucrativos que declara ter como missão "lutar contra a ignorância devastadora com uma visão de mundo baseada em fatos que qualquer um possa entender".

Para cumprir essa missão, a Gapminder reúne dados estatísticos, coletados das mais diversas fontes, e os torna disponíveis para *download* gratuito no seu *website*.

Segue, na sequência, a lista das tabelas utilizadas.

Nota importante: a maioria as tabelas está organizada em um formato pouco adequado para a consulta por SQL, por isso foi feito um tratamento desses dados antes da criação do banco de dados de exemplo.

- *avg\_income*: renda média diária das famílias, USD/pessoa/dia, ajustados pela inflação (paridade do poder de compra de 2017, em dólares internacionais constantes. Possui as colunas:
  - *country* (texto): país a que se refere a métrica;
  - *ref\_year* (inteiro): ano de referência da métrica;
  - *mean\_usd* (real): valor da renda média;
  - *link* de acesso: [http://gapm.io/dminccpcap\\_capp](http://gapm.io/dminccpcap_capp).
- *child\_mortality*: mortalidade das crianças de 0 a 5 anos de idade para cada 1000 nascidos vivos. Possui as colunas:
  - *country* (texto): país a que se refere a métrica;
  - *ref\_year* (inteiro): ano de referência da métrica;
  - *tot\_deaths* (inteiro): número de crianças mortas;
  - *link* de acesso: <http://gapm.io/du5mr>.
- *co2\_emissions\_pc*: emissão de CO<sub>2</sub> *per capita* por ano em toneladas. Possui as colunas:
  - *country* (texto): país a que se refere a métrica;
  - *ref\_year* (inteiro): ano de referência da métrica;
  - *co2\_pc* (real): toneladas de CO<sub>2</sub>;
  - *link* de acesso: [http://gapm.io/dco2\\_consumption\\_historic](http://gapm.io/dco2_consumption_historic).
- *country*: lista de países e as suas diversas classificações utilizadas pela Gapminder. Possui as colunas:
  - *country* (texto): país ao qual os dados se referem;

---

<sup>10</sup> [www.gapminder.org](http://www.gapminder.org)

- *four\_regions* (texto): classificação de quatro regiões (*europe*, *americas*, *asia* e *africa*) da Gapminder;
- *six\_regions* (texto): classificação de seis regiões da Gapminder;
- *eight\_regions* (texto): classificação de oito regiões da Gapminder;
- *wb\_regions* (texto): classificação de regiões utilizada pelo Banco Mundial;
- *wb4income* (texto): classificação de quatro perfis de renda utilizada pelo Banco Mundial;
- *wb3income* (texto): classificação de três perfis de renda utilizada pelo Banco Mundial;
- *link* de acesso: <http://gapm.io/datageo>.
- *fertility*: média do número de filhos por mulher em idade fértil. Possui as colunas:
  - *country* (texto): país a que se refere a métrica;
  - *ref\_year* (inteiro): ano de referência da métrica;
  - *mean\_babies* (real): quantidade média de filhos;
  - *link* de acesso: <http://www.healthmetricsandevaluation.org>.
- *gdp\_pc*: PIB per capita, em USD/pessoa, ajustados pela inflação (paridade do poder de compra de 2017, em dólares internacionais constantes). Possui as colunas:
  - *country* (texto): país a que se refere a métrica;
  - *ref\_year* (inteiro): ano de referência da métrica;
  - *gdp\_pc* (real): valor do PIB *per capita*;
  - *link* de acesso: [http://gapm.io/dgdp\\_cap\\_cppp](http://gapm.io/dgdp_cap_cppp).
- *life\_expectancy*: expectativa de vida do nascituro, em anos, se as taxas de mortalidade das diferentes idades se mantiverem constantes ao longo da sua vida. Possui as colunas:
  - *country* (texto): país a que se refere a métrica;
  - *ref\_year* (inteiro): ano de referência da métrica;
  - *tot\_years* (inteiro): anos de vida;
  - *link* de acesso: <http://gapm.io/dlex>.
- *men\_years\_at\_school*: o número médio de anos que um homem de 25 ou mais anos de idade passa na escola, incluindo ensino fundamental, médio e superior. Possui as colunas:
  - *country* (texto): país a que se refere a métrica;
  - *ref\_year* (inteiro): ano de referência da métrica;
  - *mean\_years* (real): média de anos na escola;
  - *link* de acesso: <http://www.healthmetricsandevaluation.org>.
- *petrobras*: valor das cotações das ações da Petrobrás na Nyse Nasdaq em USD. Essa é a única tabela do banco de dados que não foi obtida na Gapminder, mas sim no Yahoo! Finance. Possui as colunas:
  - *Date* (tipo data): indica a data específica para a qual os dados de preço, volume e outras métricas são apresentados;

- *Open* (real): o preço da ação no momento em que o mercado abre naquela data específica;
- *High* (real): o preço mais alto alcançado pela ação durante o período de negociação naquele dia;
- *Low* (real): o preço mais baixo alcançado pela ação durante o período de negociação naquele dia;
- *Close* (real): o preço da ação no momento em que o mercado fecha naquela data específica;
- *Adj Close* (real): o preço de fechamento ajustado é o preço de fechamento da ação ajustado para refletir eventos corporativos que afetam o preço da ação, como dividendos, desdobramentos (*splits*) e agrupamentos (*reverse splits*);
- *Volume* (inteiro): o número total de ações negociadas durante o dia de negociação;
- *link* de acesso: <https://finance.yahoo.com/quote/PBR/>.
- *population*: contagem do número total de habitantes vivendo em determinado território em um determinado ano. Possui as colunas:
  - *country* (texto): país a que se refere a métrica;
  - *ref\_year* (inteiro): ano de referência da métrica;
  - *tot\_years* (inteiro): anos de vida;
  - *link* de acesso: <http://gapm.io/dpop>.
- *women\_years\_at\_school*: o número médio de anos que uma mulher de 25 ou mais anos de idade passa na escola, incluindo ensino fundamental, médio e superior. Possui as colunas:
  - *country* (texto): país a que se refere a métrica;
  - *ref\_year* (inteiro): ano de referência da métrica;
  - *mean\_years* (real): média de anos na escola;
  - *link* de acesso: <http://www.healthmetricsandevaluation.org>.

# PROFESSOR-AUTOR

## ANTONIO SERGIO FERREIRA BONATO

- Bacharelado em Ciência de Computação pela Unicamp (1999).
- Mestrado em Engenharia Elétrica pela Poli/USP (2005).
- Mestrado profissional em Economia pela FGV/EESP (2023).
- Professor de graduação nos cursos de Ciência da Computação, Análise e Desenvolvimento de Sistemas e Sistemas de Informação, lecionando disciplinas de programação em Python, Java, Javascript e R, de bancos de dados e de análise de dados.
- Auditor fiscal da Receita Estadual na Secretaria da Fazenda de SP desde 2014, onde já atuou como Diretor do Centro de Desenvolvimento de Sistemas e Diretor do Departamento de Tecnologia da Informação. Atualmente, é Assessor Econômico no Gabinete do Secretário da Fazenda.
- Possui mais de 30 anos de experiência em tecnologia da informação em empresas como o Jornal Estadão, onde foi DBA (administrador de bancos de dados), Vivo, Fleury, Klabin Segall, Votorantim Cimentos e Roche Farmacêutica.









Siga as nossas redes sociais!

