

Para programar em sql, precisamos primeiro de um servidor (pode ser local) para as nossas databases. A linguagem não é case sensitive; logo, podes escrever "select", "SELECT" ou "sEleCT" (e não esquecer do ponto e vírgula).

Para criar, usar e eliminar databases, é assim:

```
CREATE DATABASE Test;
USE DATABASE Test;
DROP DATABASE Test;
```

Para criar tabela com uma coluna chamada "test_col" para guardar números inteiros:

```
CREATE TABLE Test (test_col INT);
```

Para alterar essa tabela DEPOIS de já ter sido criada:

```
ALTER TABLE Test ADD another_col VARCHAR(255);
```

O comando "SELECT" é basicamente o "print()" ou assim. Para ir buscar toda a tabela "bands", por exemplo:

```
SELECT * FROM bands;
```

Podemos pôr mais restrições, por exemplo, dizer exatamente o que queremos dar select, dar nomes custom, limitar a 2 e ordenar:

```
SELECT id AS "ID", name AS "Nome da Banda" FROM bands LIMIT 2 ORDER BY name;
```

Para ignorar itens repetidos e dar mais restrições (nota: o comando "like" está a indicar "qualquer coisa que tenha 'er' no nome", pois o "%" significa "qualquer coisa"):

```
SELECT DISTINCT name FROM albums WHERE band_id < 10 AND name LIKE "%er%" AND release_year IS NOT NULL;
```

Para dar delete:

```
DELETE FROM albums WHERE id = 5;
```

Algumas notas básicas random sobre SQL:

1. existem vários tipos de "joins": left, right, inner, outter, ...
2. existem funções, tipo: SELECT SUM(release_year) FROM albums;
3. usando funções, podemos também usar "GROUP BY"; por exemplo, se tivermos tabela exemplo com continentes, países e população de cada país, fazer SELECT SUM(population) FROM exemplo GROUP BY continent, dá a população de cada continente
4. para além de "WHERE", existe também "HAVING", onde a única diferença é que enquanto o "WHERE" põe-se antes de um "GROUP BY", o "HAVING" é depois; logo, usamos "HAVING" se quisermos usar info de funções, em coisas que foram especificadas num "GROUP BY" atrás definido

```
69. SELECT * FROM bands
70 JOIN albums ON bands.id = albums.band_id;
71
```

Result Grid |  Filter Rows: | Export:  | Wrap Cell Contents: 

id	name	id	name	release_year	band_id
1	Iron Maiden	1	The Number of the Beasts	1982	1
1	Iron Maiden	2	Power Slave	1984	1
2	Deuce	3	Nightmare	2018	2
3	Avenged Sevenfold	4	Nightmare	2010	3

```
1 CREATE DATABASE record_company;
2 USE record_company;
3 CREATE TABLE bands (
4     id INT NOT NULL AUTO_INCREMENT,
5     name VARCHAR(255) NOT NULL,
6     PRIMARY KEY (id)
7 );
8 CREATE TABLE albums (
9     id INT NOT NULL AUTO_INCREMENT,
10    name VARCHAR(255) NOT NULL,
11    release_year INT,
12    band_id INT NOT NULL,
13    PRIMARY KEY (id),
14    FOREIGN KEY (band_id) REFERENCES bands(id)
15 );
```

O primeiro item da tabela "bands" será um número que aumenta sempre que adicionamos uma nova entrada

A única coisa que temos de adicionar é o nome da banda (não pode ser *vazio*)

Definir o id como primary key, para poder referenciar noutras tabelas

O id desta tabela vai referenciar o da tabela acima


```
INSERT INTO bands (name)
VALUES ('Iron Maiden');
```

Lembrete: podemos separar o comando em quantas linhas quisermos, isto só está assim para ser mais fácil de ler. O comando só acaba no ";"

Estas são duas formas como podemos inserir itens na tabela "bands"

```
INSERT INTO bands (name)
VALUES ('Deuce'), ('Avenged Sevenfold'), ('Ankor');
```

Para a tabela albums, temos três itens a inserir, sendo que o "release_year" pode ser nulo (não havia restrição)

```
INSERT INTO albums (name, release_year, band_id)
VALUES ('The Number of the Beasts', 1985, 1),
       ('Power Slave', 1984, 1),
       ('Nightmare', 2018, 2),
       ('Nightmare', 2010, 3),
       ('Test Album', NULL, 3);
```


Pandas Dataframes

`data_csv = pd.read_csv(location)` \Rightarrow transforma o ficheiro csv num dataframe

`data_csv["bgl_level"]` \Rightarrow retorna todos os valores da coluna que tem o nome "bgl_level"

`data_csv["bgl_level"][0]` \Rightarrow retorna a linha 0 da coluna

`len(data_csv)` \Rightarrow comprimento (nº de linhas), sendo que a "linha" do cabeçalho não conta

`data_csv = data_csv.sort_values(by=["bgl_level"])` \Rightarrow dá sort, por ordem crescente e, se em vez disto, se fizer `by=["bgl_level", "temperature"]`, 1º dá sort por bgl_level e, dentro do mesmo bgl_level, havendo diferentes temperaturas, dá-se sort por temperatura (dá para ter 2 argumentos).

`data_filtered = data_csv[(data_csv["bgl_level"] == 0) & (data_csv["temperature"] == 25.0)]` \Rightarrow retorna o dataframe depois de aplicar esta mask, tendo só estas linhas, **MAS ATENÇÃO:**

- os índices não são atualizados

- tentar retornar a linha [0], vai retornar a antiga, mesmo que não cumpra a mask

- para corrigir isto, temos de dar reset aos índices:

`data_filtered.reset_index()` \Rightarrow dar reset aos índices

`data_filtered.loc[:, "bgl_level"].values` \Rightarrow retorna todos os valores de "bgl_level", sendo que o values transforma numa lista, para ser lz manipular


```
In 19 1 looking_for = ["r420", "r840", "r270", "r135"]
      2 |
      3 from zipfile import ZipFile
      4 from zipfile import Path as ZipPath
      5
      6 path = ZipFile("../odb_example.zip")
      7 lista = path.namelist()
      8
      9 path = ZipPath("../odb_example.zip")
     10 for directory in lista:
     11     try:
     12         total_path = path / directory
     13         with total_path.open() as file:
     14             data_list = file.readlines()
     15             for element in data_list:
     16                 if any(record in str(element) for record in looking_for):
     17                     print(str(element))
     18                     print(directory)
     19             except:
     20                 pass
```

Executed at 2023.11.08 15:01:43 in 80ms

\$0 r840 M