

SQL

O SQL (Structured Query Language) é uma linguagem de programação usada para gerenciar dados em bancos de dados relacionais. O SQL é usado para executar operações como inserir, atualizar, selecionar e excluir dados de um banco de dados.

O SQL funciona através de comandos que são escritos em uma sintaxe específica. Os comandos do SQL são executados em um servidor de banco de dados que recebe as consultas, processa as instruções e retorna os resultados.

Existem várias instruções SQL que podem ser usadas para manipular dados em um banco de dados. Algumas das instruções mais comuns incluem :

- **SELECT:** é usada para recuperar dados de uma ou mais tabelas no banco de dados;
- **INSERT:** é usada para inserir dados em uma tabela no banco de dados;
- **UPDATE:** é usada para atualizar dados existentes em uma tabela;
- **DELETE:** é usada para excluir dados de uma tabela;
- **WHERE:** é usado para filtrar os dados a serem buscados em uma tabela;

- **CREATE:** é usado para manipular as criações dos objetos do banco de dados;
- **DROP:** é usada para excluir os objetos do banco de dados;
- **ALTER:** é usada para modificar a estrutura de um objeto existente;
- **JOIN:** é usada para combinar dados de duas ou mais tabelas em uma única consulta;

- **GROUP BY:** é usada para agrupar dados com base em uma ou mais colunas;
- **ORDER BY:** é usada para classificar os dados com base em uma ou mais colunas;

Para executar essas instruções, é necessário conectar-se ao servidor do banco de dados usando um software cliente que permita a entrada de comandos SQL. É possível conectar-se a um banco de dados local ou remoto usando um software cliente SQL como MySQL Workbench, pgAdmin, Oracle SQL Developer, entre outros.

Os comandos SQL são enviados do software cliente para o servidor do banco de dados para serem processados e executados. O servidor responde ao cliente com os resultados da consulta.

SQLite3

O [SQLite](#) é um sistema de gerenciamento de banco de dados relacional que suporta vários tipos de dados. Alguns dos tipos de dados suportados pelo SQLite são :

- **NULL:** usado para representar valores nulos ou ausentes;
- **INTEGER:** usado para armazenar números inteiros. O SQLite tem suporte para diferentes tipos de inteiros, como TINYINT, SMALLINT, MEDIUMINT, INT e BIGINT;
- **REAL:** usado para armazenar números de ponto flutuante, como números decimais;
- **TEXT:** usado para armazenar dados de texto, como nomes, descrições, mensagens etc. O SQLite suporta várias codificações de caracteres, como ASCII, UTF-8, UTF-16, entre outras;
- **BLOB:** usado para armazenar dados binários, como imagens, arquivos de áudio e vídeo;

O SQLite é um banco de dados relacional que é usado em muitos aplicativos para armazenar e acessar dados. É uma escolha popular para projetos de pequena escala, pois é leve e fácil de usar.

O Python possui uma biblioteca nativa para trabalhar com bancos de dados SQLite chamada `sqlite3`, presente desde a versão 2.5 (lançada em setembro de 2006).

Básico

Vamos passar pelo passo a passo de como usar o SQLite3 junto com o Python.

Sendo o sqlite3 uma biblioteca padrão do Python, não tem necessidade instalar nada. Basta chamar ele importando no começo do módulo.

```
C:\Users\gutoh\OneDrive\Área de Trabalho\main.py
1 import sqlite3
2
```

Uma vez importado, temos que criar uma conexão com o banco. Para isso temos que especificar o nome do arquivo de banco de dados. Caso o arquivo não exista, ele será criado (assim como quando abrimos um arquivo no modo 'w'). Depois de criado, uma conexão será estabelecida com o novo arquivo.

```
C:\Users\gutoh\OneDrive\Área de Trabalho\sql\main.py
3 import sqlite3
2
1 conn = sqlite3.connect('meu_banco.db')
4
```

Agora, ao executar o código, um arquivo no formato db será criado.

```
~\OneDrive\Área de Trabalho\
  ✚ main.py
  📄 meu_banco.db
```

Agora que temos nosso banco criado, temos que criar a estrutura que vai armazenar os nossos dados. A estrutura usada é a tabela, que será criada com o comando CREATE TABLE.

```
C:\Users\gutoh\OneDrive\Área de Trabalho\sql\main.py
1 import sqlite3
2
3 conn = sqlite3.connect('meu_banco.db')
4
5 cur = conn.cursor()
6
7 query = """
8 CREATE TABLE USUARIOS
9 (
10     id INTEGER PRIMARY KEY,
11     nome TEXT,
12     idade INTEGER
13 )
14 """
15
16 cur.execute(query)
17
```

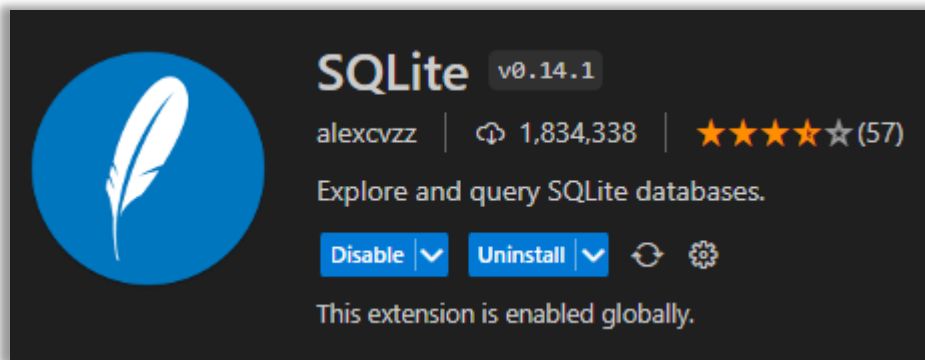
Repare no código acima.

Na linha 5 temos a criação do [cursor](#), que é uma variável iterativa definida que permite a interação com o banco de dados, seja para realizar alterações em sua estrutura quanto para buscar dados em uma tabela.

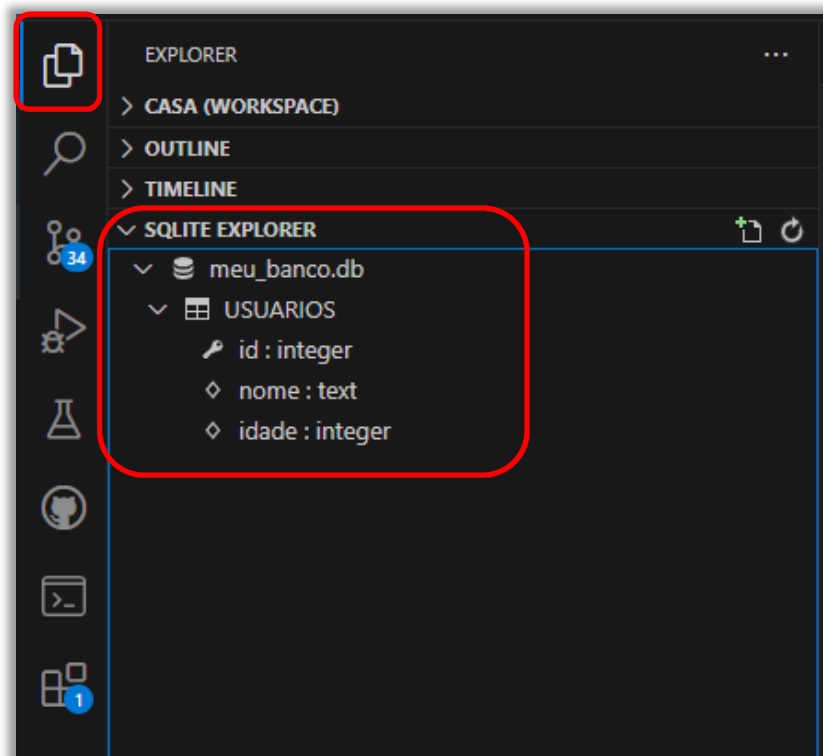
Na linha 7, temos a query de criação de uma tabela (como uma string do Python). O CREATE TABLE é seguido pelo nome da tabela a ser criada. Dentro dos parênteses, temos a criação das colunas e seus respectivos tipos. Repare que temos um comando PRIMARY KEY junto da criação do campo id, ele serve para indicar que aquele campo não aceita valores duplicados.

Na linha 16, temos a chamada da nossa variável cursor (cur) e chamamos a função execute passando como parâmetro a variável query que criamos.

Agora, temos nossa tabela criada. Podemos conferir sua estrutura usando a extensão SQLite, de alexcvzz.



Ela vai ativar uma lista na aba Explorer do VS Code.



Agora que nossa tabela já está criada, vamos inserir alguns dados dentro dela. Para isso, usamos o comando INSERT.

```

C:\Users\gutoh\OneDrive\Área de Trabalho\sql\main.py
1 import sqlite3
2
3 conn = sqlite3.connect('meu_banco.db')
4
5 cur = conn.cursor()
6
7 query = """
8 CREATE TABLE USUARIOS
9 (
10     id INTEGER PRIMARY KEY,
11     nome TEXT,
12     idade INTEGER
13 )
14 """
15
16 cur.execute(query)
17
18 query = """
19 INSERT INTO USUARIOS (nome, idade)
20     VALUES ('Tutankamon', 15)
21 """
22 cur.execute(query)
23

```

Para inserir, temos que especificar o nome da tabela, os campos a serem preenchidos e depois os valores a serem inseridos. Repare que não especificamos o campo id. Isso porque esse campo será preenchido automaticamente pelo SQLite.

Depois de criada a query, a executamos com o comando execute.

Mas, se olharmos o banco de dados pelo VS Code, não veremos qualquer dado. Por que? Porque não foi dado commit e nem fechado a conexão com o banco de dados.

```

22 cur.execute(query)
23
24 conn.commit()
25 conn.close()
26

```





(Trecho final do código acima)

Todas as alterações realizadas até o momento foram feitas apenas na memória RAM. Se ocorrer uma queda de luz, todos os dados serão perdidos. Aí que entram os comandos commit e close.

O comando commit é responsável por consolidar o que foi alterado até o momento no banco de dados em disco.

O comando close é responsável por fechar a conexão com o banco, salvando tudo em definitivo no disco. Se o banco de dados não vai mais ser usado, é preciso fechar o banco para evitar que haja corrupção dos dados internos.

Uma vez executados, agora sim podemos conferir o dado salvo na tabela usando o VS Code.

SQL ▾ < 1 / 1 > 1 - 1 of 1    

id	nome	idade
1	Tutankamon	15

Agora, para buscar esses dados da tabela com o Python, usamos o comando SELECT.

```
C:\Users\gutoh\OneDrive\Área de Trabalho\sql\main.py
7 query = """
8 CREATE TABLE USUARIOS
9 (
10     id INTEGER PRIMARY KEY,
11     nome TEXT,
12     idade INTEGER
13 )
14 """
15
16 cur.execute(query)
17
18 query = """
19 INSERT INTO USUARIOS (nome, idade)
20     VALUES ('Tutankamon', 15)
21 """
22 cur.execute(query)
23
24 cur.execute('SELECT * FROM USUARIOS')
25 linhas = cur.fetchall()
26
27 for linha in linhas:
28     print(linha)
29
30 conn.commit()
31 conn.close()
32
```

Com esse trecho adicional do código, podemos realizar a consulta da tabela no banco. Veja como vai ficar a saída do dado buscado:

```
C:\Users\gutoh\OneDrive\Área de Trabalho\sql>python main.py
(1, 'Tutankamon', 15)
C:\Users\gutoh\OneDrive\Área de Trabalho\sql>
```

O comando `fetchall()` é usado para recuperar tudo o que foi buscado na tabela. Ele sempre irá retornar uma lista. Se não foi encontrado qualquer registro, vai retornar uma lista vazia. Se encontrar um ou mais, vai retornar uma lista com um ou mais registros.

Cada item buscado será, também, uma lista.

Veja um exemplo de código completo de uma agenda eletrônica:

```
C:\Users\gutoh\OneDrive\Área de Trabalho\sql\main.py
1 import sqlite3
2
3 # se conecta ao banco de dados
4 conn = sqlite3.connect('agenda.db')
5
6 # cria um cursor para interagir com o banco de dados
7 c = conn.cursor()
8
9 # query para criar a tabela
10 query = '''
11 CREATE TABLE CONTATOS
12 (
13     id INTEGER PRIMARY KEY,
14     nome TEXT,
15     telefone TEXT
16 )'''
17 c.execute(query)
18
19 # inserindo os contatos na tabela
20 c.execute("INSERT INTO CONTATOS (nome, telefone) VALUES (?, ?)", ('João Silva', '555-1234'))
21 c.execute("INSERT INTO CONTATOS (nome, telefone) VALUES (?, ?)", ('Maria Santos', '555-9876'))
22 c.execute("INSERT INTO CONTATOS (nome, telefone) VALUES (?, ?)", ('Pedro Souza', '555-5555'))
23
24 # salvar as alterações no banco de dados
25 conn.commit()
26
27 # executa uma consulta para recuperar todos os contatos da tabela
28 c.execute("SELECT * FROM CONTATOS")
29 linhas = c.fetchall()
30
31 # exibe os resultados na tela
32 print("Contatos:")
33 for linha in linhas:
34     print(f"{linha[0]} - {linha[1]}: {linha[2]}")
35
36 # fecha a conexão com o banco de dados
37 conn.close()
38
```

Veja que agora temos uma leve diferença na parte de INSERT do código. Ali, nós estamos usando pontos de interrogação nos parênteses depois do VALUES e, no final, temos uma tupla com dois valores. Isso é uma forma segura de enviar dados do Python para o banco de dados. Cada um dos pontos de interrogação será substituído por um item da tupla, respectivamente.

Isso é o básico do SQLite e Python. Agora, vamos nos aprofundar mais em cada comando.

Comandos SELECT e WHERE

Anteriormente, vimos como buscar todos os dados de uma só vez, mas podemos usar comando WHERE para realizar filtros a serem mostrados.

```
# solicita o ID do contato a ser recuperado
contato_id = int(input("Digite o ID do contato que deseja visualizar: "))

# executa uma consulta para recuperar o contato especificado pelo usuário
c.execute("SELECT * FROM CONTATOS WHERE id = ?", (contato_id,))
linha = c.fetchone()

# exibe o resultado na tela
if linha:
    print(f"{linha[0]} - {linha[1]}: {linha[2]}")
else:
    print("Contato não encontrado")
```

Nesta trecho do código, pedimos para o usuário digitar um ID do contato que quer visualizar. A função **fetchone()**, diferente da **fetchall()**, retorna apenas um único registro.

Observe que, desta vez, usamos uma vírgula para indicar que a tupla de parâmetros contém apenas um valor, em vez de usar um único sinal de interrogação como na inserção de múltiplos valores. Isso ocorre porque estamos passando apenas um valor, o ID do contato.

Se nenhum registro corresponder ao ID especificado, exibimos uma mensagem informando que o contato não foi encontrado.

Especificando Campos

Para buscar apenas os nomes dos contatos registrados na tabela `contatos` do exemplo da agenda, você pode executar uma consulta SQL SELECT que seleciona apenas a coluna `nome`.

```
# executa uma consulta para buscar apenas os nomes dos contatos
c.execute("SELECT nome FROM contatos")

# recupera os resultados da consulta
resultados = c.fetchall()
for nome in resultados:
    print(nome[0])
```

Neste exemplo, executamos uma consulta SQL SELECT que seleciona apenas a coluna **nome** da tabela CONTATOS. Em seguida, usamos o método `fetchall()` para recuperar os resultados da consulta e usamos um loop `for` para exibir cada nome na tela.

Observe que, ao recuperar os resultados da consulta usando o método `fetchall()`, obtemos uma lista de tuplas, onde cada tupla representa um registro na tabela CONTATOS contendo apenas um valor (o nome). Por isso, ao exibir os nomes na tela no loop `for`, usamos a expressão **nome[0]** para acessar o primeiro valor da tupla (o nome).

Comando IN

A instrução SQL **IN** é usada para verificar se um valor está presente em uma lista de valores. Podemos usar a cláusula **IN** em uma consulta SQL para selecionar apenas os contatos cujos nomes estejam em uma lista específica.

```
# seleciona apenas os contatos cujos nomes estão na lista
nomes = ['Alice', 'Bob', 'Carol']
c.execute("SELECT * FROM CONTATOS WHERE nome IN {}".format(', '.join(['?']*len(nomes))), nomes)
resultados = c.fetchall()

# exibe os resultados na tela
for resultado in resultados:
    print(f"{resultado[1]}: {resultado[2]}")
```

Observe como usamos a cláusula **WHERE** para especificar que queremos apenas os contatos cujos nomes estão na lista **nomes**. Usamos o método `join` para juntar as strings na lista com vírgulas e, em seguida, passamos a lista de nomes como um segundo argumento para o método **execute**. Isso garante que os valores na lista sejam escapados corretamente antes de serem usados na consulta.

Comando LIKE

Para recuperar apenas os contatos que começam com a letra **A** (ou qualquer outra letra), podemos usar a cláusula **WHERE** em sua consulta SQL com uma expressão de comparação.

```
# insere mais contatos na tabela
c.execute("INSERT INTO CONTATOS (nome, telefone) VALUES (?, ?)", ('Ana Paula', '555-7777'))
c.execute("INSERT INTO CONTATOS (nome, telefone) VALUES (?, ?)", ('Alberto Santos', '555-8888'))

# executa uma consulta para recuperar apenas os contatos que começam com a letra "A"
c.execute("SELECT * FROM CONTATOS WHERE nome LIKE 'A%'")

# recupera os resultados da consulta
resultados = c.fetchall()
for contato in resultados:
    print(contato)
```

Neste exemplo, após inserir os contatos na tabela, executamos uma consulta SQL **SELECT** para recuperar apenas os contatos cujo nome começa com a letra **A**. A cláusula **WHERE** é usada para especificar uma condição de filtragem que compara o valor da coluna **nome** com a expressão 'A%', que significa qualquer valor que comece com a letra A.

Observe que, neste exemplo, usamos o operador **LIKE** para comparar os valores da coluna **nome**. Este operador permite que você use caracteres curinga (como % e _) para corresponder a qualquer sequência de caracteres ou um único caractere, respectivamente. Por exemplo, a expressão 'A%' corresponde a qualquer valor que comece com a letra A, enquanto a expressão %Paula corresponde a qualquer valor que termine com a sequência **Paula**.

Comando LIMIT

Podemos usar a condição **LIMIT** para limitar a quantidade de registros que queremos buscar. Veja abaixo como faríamos isso na agenda :

```
# executa uma consulta para buscar pelos 2 primeiros registros
c.execute("SELECT * FROM CONTATOS LIMIT 2")
linhas = c.fetchall()
```

Neste exemplo, após inserir os contatos na tabela, executamos uma consulta SQL **SELECT** para selecionar os 2 primeiros registros da tabela, usando a cláusula **LIMIT 2**.

Observe que, neste exemplo, não solicitamos a entrada do usuário, mas sim buscamos diretamente pelos 2 primeiros registros da tabela. Se você quiser permitir que o usuário especifique o número de registros a serem exibidos, basta modificar a consulta SQL para incluir uma variável que contenha o número de registros desejados, por exemplo: **LIMIT ?**, e passar o número de registros como um parâmetro na chamada de execute().

Comando ORDER BY

A instrução SQL **ORDER BY** é usada para classificar as linhas de uma consulta em ordem crescente ou decrescente com base no valor de uma ou mais colunas. No exemplo da agenda, podemos usar **ORDER BY** para classificar os contatos em ordem alfabética com base em seus nomes.

```
# seleciona todos os contatos, classificados em ordem alfabética por nome
c.execute("SELECT * FROM CONTATOS ORDER BY nome ASC")
resultados = c.fetchall()

# Exibir os resultados na tela
for resultado in resultados:
    print(f"{resultado[1]}: {resultado[2]}")
```

Observe como usamos **ASC** para classificar em ordem crescente. Se quisermos classificar em ordem decrescente, podemos usar **DESC** em vez de **ASC**.

Exercícios para Praticar

1. Crie uma tabela chamada "Clientes" com as colunas "ID", "Nome" e "Email".
 - a. Insira um novo cliente na tabela "Clientes" com ID 1, nome "João" e email "joao@example.com".
 - b. Insira mais dois clientes na tabela "Clientes" com dados de sua escolha.
 - c. Selecione todos os registros da tabela "Clientes".
 - d. Selecione apenas o nome e o email de todos os clientes da tabela "Clientes".
2. Crie uma tabela chamada "Produtos" com as colunas "ID", "Nome" e "Preço".
 - a. Insira alguns produtos na tabela "Produtos" com diferentes IDs, nomes e preços.
 - b. Selecione todos os registros da tabela "Produtos" em ordem decrescente de ID.
 - c. Selecione apenas os produtos que possuem um valor acima de R\$100 na tabela "Produtos".
 - d. Selecione todos os registros da tabela "Produtos" cuja quantidade seja maior que 10.
 - e. Selecione apenas o nome dos produtos da tabela "Produtos" que possuam a palavra "chocolate" em seu nome.
3. Crie uma nova tabela chamada "Funcionarios" com as colunas "ID", "Nome" e "Cargo".
 - a. Insira alguns funcionários na tabela "Funcionarios" com diferentes IDs, nomes e cargos.
 - b. Consulte todos os funcionários na tabela "Funcionarios".
4. Crie uma tabela chamada "Livros" com as colunas "ID", "Título" e "Autor".
 - a. Insira um novo livro na tabela "Livros" com ID 1, título "Dom Casmurro" e autor "Machado de Assis".
 - b. Insira mais dois livros na tabela "Livros" com dados de sua escolha.
 - c. Selecione todos os registros da tabela "Livros" cujo autor seja "Machado de Assis".
 - d. Selecione apenas o título dos livros da tabela "Livros" que tenham mais de 200 páginas.
5. Crie uma tabela chamada "Cidades" com as colunas "ID" e "Nome".
 - a. Insira algumas cidades na tabela "Cidades" com diferentes IDs e nomes.
 - b. Selecione todos os registros da tabela "Cidades" cujo email contenha a palavra "gmail".
 - c. Selecione apenas o nome dos clientes da tabela "Clientes" cujo nome comece com a letra "A".
6. Crie uma tabela chamada "Estudantes" com as colunas "ID", "Nome" e "Idade".
 - a. Insira alguns estudantes na tabela "Estudantes" com diferentes IDs, nomes e idades.
7. Crie uma tabela chamada "Pedidos" com as colunas "ID", "Cliente" e "Total".
 - a. Insira um novo pedido na tabela "Pedidos" com ID 1, cliente "Maria" e total R\$50.00.
 - b. Insira mais dois pedidos na tabela "Pedidos" com dados de sua escolha.
8. Crie uma tabela chamada "Funcionarios" com as colunas "ID", "Nome" e "Salario".
 - a. Insira alguns funcionários na tabela "Funcionarios" com diferentes IDs, nomes e salários.
 - b. Selecione todos os registros da tabela "Funcionarios" com salário acima de R\$5000.
 - c. Selecione apenas os nomes dos funcionários da tabela "Funcionarios" cujo salário seja maior que R\$7000.

9. Crie uma tabela chamada "Eventos" com as colunas "ID", "Nome" e "Data".
 - a. Insira alguns eventos na tabela "Eventos" com diferentes IDs, nomes e datas.
 - b. Selecione todos os registros da tabela "Eventos" cuja data seja posterior a "2023-01-01".
 - c. Selecione apenas o nome dos eventos da tabela "Eventos" que ocorrerão no mês de julho.
10. Crie uma tabela chamada "Veiculos" com as colunas "ID", "Marca" e "Ano".
 - a. Insira alguns veículos na tabela "Veiculos" com diferentes IDs, marcas e anos.
 - b. Selecione todos os registros da tabela "Veiculos" cuja marca seja "Toyota" e o ano seja 2022.
 - c. Selecione apenas a marca e o ano dos veículos da tabela "Veiculos" cujo ID seja ímpar.
11. Crie uma tabela chamada "Pacientes" com as colunas "ID", "Nome" e "DataNascimento".
 - a. Insira alguns pacientes na tabela "Pacientes" com diferentes IDs, nomes e datas de nascimento.
 - b. Selecione todos os registros da tabela "Pacientes" cuja data de nascimento seja anterior a "1990-01-01".
 - c. Selecione apenas o nome e a data de nascimento dos pacientes da tabela "Pacientes" com mais de 30 anos.