FOCO NO MERCADO DE TRABALHO

APLICAÇÃO DE BANCO DE DADOS COM PYTHON

Vanessa Cadan Scheffer

SOLUÇÃO CRUD

CRUD é um acrônimo para as quatro operações de DML que podemos fazer em uma tabela no banco de dados, como inserir informações (CREATE), ler (READ), atualizar (UPDATE) e apagar (DELETE).

Fonte: Shutterstock.

Deseja ouvir este material?

Áudio disponível no material digital.

DESAFIO

Utilizar bancos de dados para persisti-los faz parte da realidade de toda empresa que tenha algum sistema computadorizado, mesmo o destinado somente para cadastro de cliente. Na edição do dia 6 de março de 2017, a revista *The Economist* publicou um artigo cujo título afirma "The world's most valuable resource is no longer oil, but data" (O recurso mais valioso do mundo não é mais petróleo, mas dados) (https://econ.st/3goh4Mj). Esse texto trouxe os holofotes para o que muitas empresas já sabiam: uma vez que os dados são preciosos, precisamos armazená-los e tratá-los para extrair informações.

Hoje é comum encontrar nas empresas equipes totalmente focadas em soluções que envolvem dados. Parte do time é responsável por construir componentes (softwares ou microsserviços) voltados às necessidades de armazenamento e recuperação de dados. Esses componentes precisam ser parametrizáveis, o que significa que o componente deve ser capaz de fazer a mesma tarefa para diferentes

tecnologias. Para os casos de recuperação de dados em banco de dados relacional, construir um componente parametrizável significa recuperar dados de diferentes RDBMS com o mesmo código mas com diferentes parâmetros.

Como desenvolvedor em uma empresa de consultoria de software, você foi alocado para construir uma solução parametrizável capaz de criar banco de dados na tecnologia SQLite, criar e apagar tabelas, o nome do banco, das tabelas e a DDL necessária para criar uma tabela (tudo deve ser parametrizável). Você também deve construir uma solução capaz de fazer um CRUD em um banco SQLite – veja que o componente deve funcionar para qualquer nome de banco e de tabela, ou seja, parâmetros. As regras que lhe foram passadas são as seguintes:

- Para criar uma nova base de dados, é necessário informar o nome.
- Para criar uma nova tabela, é necessário informar o nome do banco que receberá a tabela e a DDL de criação.
- Para excluir uma tabela, é necessário informar o nome do banco e da tabela.
- Para inserir um novo registro em uma tabela, é preciso informar: o nome do banco e da tabela e um dicionário contendo a chave e o valor a ser inserido. Por exemplo, {'nome': 'João', 'idade': 30}.
- Para recuperar, os dados é preciso informar o nome do banco e da tabela.
- Para atualizar um novo registro em uma tabela é preciso informar: o nome do banco e da tabela e dois dicionários, um contendo a chave e o valor a ser atualizado e outro contendo a chave e o valor da condição do registro que será atualizado.
- Para excluir um registro em uma tabela, é preciso informar: o nome do banco e da tabela e um dicionário contendo a chave e o valor da condição para localizar o registro a ser inserido. Por exemplo, {'id_cliente': 10}.

Então, mãos à obra!

RESOLUÇÃO

Chegou o momento de implementar a versão inicial da solução parametrizável que realiza funções tanto de DDL quanto de DML em um banco de dados SQLite. Certamente, existem muitas possibilidades de implementação. Vejamos uma proposta inicial de solução.

Uma possibilidade é construir duas classes, uma com os métodos para DDL e outra para o CRUD (DML). Vamos, então, começar pela classe capaz de criar um banco, criar e apagar uma tabela. Veja a seguir a classe *DDLSQLite*. Criamos um método privado (lembre-se de que, em Python, trata-se somente uma convenção de

nomenclatura) que retorna uma instância de conexão com um banco parametrizável. Também criamos três método públicos, um para criar um banco de dados, outro para criar uma tabela e o último para apagar uma tabela. Todos os métodos são parametrizáveis, conforme foi solicitado. Chamo a atenção para o método *criar_banco_de_dados()*, que cria o banco fazendo uma conexão e já fechando-a (lembre-se de que existem diferentes formas de fazer uma determinada implementação).

In [12]:

```
import sqlite3



class DDLSQLite:
    def _conectar(self, nome_banco):
        nome_banco += '.db'
        conn = sqlite3.connect(nome_banco)
        return conn

    def criar_banco_de_dados(self, nome_banco):
        nome_banco += '.db'
        sqlite3.connect(nome_banco).close()
        print(f"O banco de dados {nome_banco} foi criado com sucesso!")
        return None

    def criar_tabela(self, nome_banco, ddl_create):
        conn = self._conectar(nome_banco)
        cursor = conn.cursor()
        cursor.execute(ddl_create)
        cursor.close()
        conn.close()
        print(f"Tabela criada com sucesso!")
        return None

    def apagar_tabela(self, nome_banco, tabela):
        conn = self._conectar(nome_banco)
        cursor = conn.cursor()
        cursor.execute(f"DROP TABLE {tabela}")
        cursor.close()
        conn.close()
        print(f"A tabela {tabela} foi excluída com sucesso!")
        return None
```

O próximo passo é construir a classe que faz o CRUD. Observe a proposta da classe *CrudSQLite*. Como todas as operações precisam receber como parâmetro o nome do banco de dados, colocamos o parâmetro no método construtor, ou seja, para

instanciar essa classe, deve ser informado o nome da banco e, a partir do objeto, basta chamar os métodos e passar os demais parâmetros. O nome do banco será usado pelo método privado *_conectar(),* que retorna uma instância da conexão – esse método será chamado internamente pelos demais métodos. O método inserir_registro(), extrai como uma tupla as colunas (linha 13) e os valores (linha 14) a serem usados para construir o comando de inserção. O método ler_registros() seleciona todos os dados de uma tabela e os retorna, lembrando que o retorno será uma lista de tuplas. O método atualizar_registro(), precisa tando dos dados a serem atualizados (dicionário) quanto dos dados que serão usados para construir a condição. **Nessa primeira versão do componente, a atualização só pode ser feita desde que o valor da condição seja inteiro**. O método *apagar_registro()* também só pode ser executado desde que o valor da condição seja inteiro.

In [13]:

```
import sqlite3



class CrudSQLite:
    def __init__(self, nome_banco):
        self.nome_banco = nome_banco + '.db'

    def _conectar(self):
        conn = sqlite3.connect(self.nome_banco)
        return conn

    def inserir_registro(self, tabela, registro):
        colunas = tuple(registro.keys())
        valores = tuple(registro.values())

        conn = self._conectar()
        cursor = conn.cursor()
        query = f"""INSERT INTO {tabela} {colunas} VALUES {valores}"""
        cursor.execute(query)
        conn.commit()
        cursor.close()
        conn.close()
        print("Dados inseridos com sucesso!")
        return None

    def ler_registros(self, tabela):
        conn = self._conectar()
        cursor = conn.cursor()
        query = f"""SELECT * FROM {tabela}"""
        cursor.execute(query)
        resultado = cursor.fetchall()
        cursor.close()
        conn.close()
        return resultado

    def atualizar_registro(self, tabela, dado, condicao):
        campo_alterar = list(dado.keys())[0]
        valor_alterar = dado.get(campo_alterar)
        campo_condicao = list(condicao.keys())[0]
        valor_condicao = condicao.get(campo_condicao)

        conn = self._conectar()
        cursor = conn.cursor()
        query = f"""UPDATE {tabela} SET {campo_alterar} = '{valor_alterar}'
WHERE {campo_condicao} = {valor_condicao}"""
        cursor.execute(query)
        conn.commit()
```

```
        cursor.close()
        conn.close()
        print("Dado atualizado com sucesso!")
        return None

    def apagar_registro(self, tabela, condicao):
        campo_condicao = list(condicao.keys())[0]
        valor_condicao = condicao.get(campo_condicao)
        conn = self._conectar()
        cursor = conn.cursor()
        query = f"""DELETE FROM {tabela} WHERE {campo_condicao} =
{valor_condicao}"""
        cursor.execute(query)
        conn.commit()
        cursor.close()
        conn.close()
        print("Dado excluído com sucesso!")
        return None
```

Antes de entregar uma solução, é preciso testar muitas vezes. Portanto, vamos aos testes. Primeiro vamos testar a classe DDLSQLite. Como podemos observar no resultado, tudo ocorreu como esperado: tanto o método para criar o banco quanto a tabela funcionaram corretamente.

In [14]:

```
# instancia um objeto
objeto_ddl = DDLSQLite()

# Cria um banco de dados
objeto_ddl.criar_banco_de_dados('desafio')

# Cria uma tabela chamada cliente
ddl_create = """
CREATE TABLE cliente (
    id_cliente INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    nome_cliente TEXT NOT NULL,
    cpf VARCHAR(14) NOT NULL,
    email TEXT NOT NULL,
    telefone VARCHAR(15),
    cidade TEXT,
    estado VARCHAR(2) NOT NULL,
    data_cadastro DATE NOT NULL
);
"""
objeto_ddl.criar_tabela(nome_banco='desafio', ddl_create=ddl_create)

# Caso precise excluir a tabela, o comando a seguir deverá ser usado
# objeto_ddl.apagar_tabela(nome_banco='desafio', tabela='cliente')
```

```
O banco de dados desafio.db foi criado com sucesso!
Tabela criada com sucesso!
```

Vamos fazer os testes do CRUD em duas etapas. Na primeira, vamos inserir registros e consultarmos para checar se a inserção deu certo.

In [15]:

```
objeto_dml = CrudSQLite(nome_banco='desafio')
# Inserir registros
dados = [
    {
        'nome_cliente': 'João',
        'cpf': '111.111.111-11',
        'email': 'joao@servidor',
        'cidade': 'São Paulo',
        'estado': 'SP',
        'data_cadastro': '2020-01-01'
    },
    {
        'nome_cliente': 'Maria',
        'cpf': '222.222.222-22',
        'email': 'maria@servidor',
        'cidade': 'São Paulo',
        'estado': 'SP',
        'data_cadastro': '2020-01-01'
    },
]
# Para cada dicionário na lista de dados, invoca o método de inserção
for valor in dados:
    objeto_dml.inserir_registro(tabela='cliente', registro=valor)

# Carrega dados salvos
dados_carregados = objeto_dml.ler_registros(tabela='cliente')
for dado in dados_carregados:
    print(dado)
```

```
Dados inseridos com sucesso!
Dados inseridos com sucesso!
(1, 'João', '111.111.111-11', 'joao@servidor', None, 'São Paulo', 'SP',
'2020-01-01')
(2, 'Maria', '222.222.222-22', 'maria@servidor', None, 'São Paulo', 'SP',
'2020-01-01')
```

Agora vamos atualizar um registro e excluir outro.

In [16]:

```
# Atualiza registro
dado_atualizar = {'telefone': '(11)1.1111-1111'}
condicao = {'id_cliente': 1}

objeto_dml.atualizar_registro(tabela='cliente', dado=dado_atualizar,
condicao=condicao)

dados_carregados = objeto_dml.ler_registros(tabela='cliente')
for dado in dados_carregados:
    print(dado)

# Apaga registro
condicao = {'id_cliente': 1}

objeto_dml.apagar_registro(tabela='cliente', condicao=condicao)
```

```
Dado atualizado com sucesso!
(1, 'João', '111.111.111-11', 'joao@servidor', '(11)1.1111-1111', 'São
Paulo', 'SP', '2020-01-01')
(2, 'Maria', '222.222.222-22', 'maria@servidor', None, 'São Paulo', 'SP',
'2020-01-01')
Dado excluído com sucesso!
```

Felizmente nosso componente funcionou conforme esperado. Gostaríamos de encerrar o desafio lembrando que são muitas as formas de implementação. Tratanddo-se de classes que se comunicam com banco de dados, o padrão de projeto singleton é o mais adequado para determinados casos, uma vez que permite instanciar somente uma conexão com o banco.

DESAFIO DA INTERNET

Ganhar habilidade em programação exige estudo e treino (muito treino). Acesse a biblioteca virtual no endereço: (http://biblioteca-virtual.com/), e busque pelo livro a seguir referenciado. Nas páginas 202 e 203 da obra, você encontra o exemplo 8.5, que faz a atualização de dados a partir de um arquivo no disco. Que tal implementar essa solução?