Laboratório de Banco de Dados

SQL Avançado

Acesso via Linguagens de Programação de Propósito Geral

UFMT

Julho de 2025

Agenda

1. **MySQL Connector/Python.**
2. Conectando ao MySQL usando Connector/Python.
3. Cursores.
4. Comandos parametrizados e pré-preparados.
5. Tratamento de erros.

Introdução

- O MySQL Connector/Python é um driver de conexão com banco de dados MySQL. Ou seja, é a "cola" usada para integrar um programa Python com um banco de dados em um SGBD MySQL.

- Pode ser usado para manipular o esquema do banco de dados com comandos DDL (Data Definition Language), bem como para consultar e alterar os dados por meio de comandos DML (Data Manipulation Language).

- É o conector oficial para Python, desenvolvido e mantido pela equipe de desenvolvimento do MySQL da Oracle Corporation.

Instalação

- **Community Edition**: Versão gratuita e open-source, mas com suporte limitado oferecido principalmente por fóruns e comunidades na Internet.
- **Enterprise Edition**: Versão que inclui suporte técnico profissional, ferramentas adicionais de segurança e gerenciamento, otimizações de desempenho, e conformidade com padrões empresariais.
- Usaremos a edição Community do conector. A instalação pode ser feita usando o gerenciador de pacotes pip:

```
pip install mysql-connector-python
```

Instalação

- Detalhes adicionais relacionados à instalação podem ser encontrados no seguinte link: https://dev.mysql.com/doc/connector-python/en/connector-python-installation.html

APIs

- MySQL Connector/Python basicamente incorpora três APIs:
  - **API Connector/Python**: É a API padrão para acessar o MySQL a partir de programas Python, aderindo à especificação DB-API (PEP 249).
  - **C Extension API**: Fornece uma interface de nível mais baixo, baseada em C, que pode ser usada para melhorar o desempenho em comparação com as implementações puramente em Python.
  - **API X Dev**: Usada para interações com documentos JSON e coleções, adequada para aplicações NoSQL.

Agenda

1. MySQL Connector/Python.
2. **Conectando ao MySQL usando Connector/Python.**
3. Cursores.
4. Comandos parametrizados e pré-preparados.
5. Tratamento de erros.

Conectando ao MySQL

- O módulo `mysql.connector` inclui a implementação da API Connector/Python, definida na PEP249 (https://www.python.org/dev/peps/pep-0249/).

- Há vários caminhos para uma aplicação Python criar uma conexão com um banco de dados MySQL, mas o mais comum é usar a função `mysql.connector.connect()`, por ser poderosa e flexível.

- A função `mysql.connector.connect()` retorna um objeto do tipo `MySQLConnection` que representa todos os recursos relacionados à conexão.

Conectando ao MySQL

- A tabela abaixo mostra os parâmetros mais comuns, necessários para o estabelecimento de uma conexão usando `mysql.connector.connect()`.
- Existem vários parâmetros que não cobriremos aqui e, dependendo de sua necessidade, você precisa checar o manual da API.

Argumento	Default	Descrição
host	127.0.0.1 (host local).	host onde está executando o servidor MySQL ao qual você deseja se conectar.
port	3306	A porta **default** na qual o servidor MySQL aguarda conexões.
user	--------	O nome de usuário com acesso ao banco de dados.
password	--------	A senha para fazer a autenticação do usuário.

Conectando ao MySQL

```
1. import mysql.connector # importa o módulo do conector
```

```
2. conn = mysql.connector.connect(
3.   host = “localhost”,
4.   user = “seu_usuário”,
5.   password = “sua_senha”,
6.   database = “nome_do_banco”
7. )
8.
9.  ...  # faça algo com a conexão
10.
11.print(f‘ID da conexao : {conn.connection_id}’)
12.conn.close()
```

- Se a conexão for estabelecida com sucesso, a linha 11 exibirá o ID. Caso contrário, `None` será impresso.

Conectando ao MySQL usando Dicionário Python

```
1. import mysql.connector # importa o módulo do conector
```

```
2. # cria um dicionário com os parâmetros da conexão
3. connect_args = {
4.   “host”: “localhost”,
5.   “port”: 3306,
6.   “user”: “seu_usuário”,
7.   “password”: “sua_senha”,
8.   “database”: “nome_do_banco”
9. }
```

```
9. conn = mysql.connector.connect(**connect_args)
10.          ...  # faça algo com a conexão
11.print(f‘ID da conexao : {conn.connection_id}’)
12.conn.close()
```

Reconfiguração e Reconexão

- Embora pouco utilizado, é possível reconfigurar uma conexão existente e reconectar com os novos parâmetros.

```
1. new_args = {
2.   # descreva os parâmetros a serem alterados
3.   # na forma de um dicionário
4. }
```

```
4. con.config(**new_args) #redefine parâmetros
5. con.reconnect()        #reconecta
6.
7.  ... #faça algo com a conexão
8.
9. print(f'ID da conexao : {con.connection_id}')
10.con.close()
```

Práticas Recomendadas

- Em primeiro lugar, é importante gerenciar as conexões, **fechando-as** sempre que não forem mais necessárias. Isso evita vazamento de recursos.
- Além disso, nos exemplos anteriores, os parâmetros da conexão (IP, porta, nome de usuário e a senha) foram codificados diretamente na aplicação. Isso torna o código mais difícil de manter e também é uma preocupação de segurança porque a senha fica visível a quem tenha acesso ao código-fonte.
- Para melhorar esse aspecto de segurança, veremos a seguir como usar **arquivos de configuração**!

Arquivos de Configuração MySQL

- Como vimos, há várias formas de fornecer os parâmetros de conexão. Particularmente, há suporte nativo do MySQL Connector/Python para leitura de arquivos de configuração da conexão.
- MySQL usa o formato de arquivo **INI** (Inicialization File) para seus arquivos de configuração. Trata-se de um formato textual simples, que contém seções, propriedades e valores. Segue um exemplo.

```
[connector_python]
user     = usuário
host     = 127.0.0.1
port     = 3306
password = senha
```

Arquivos de Configuração MySQL

- `mysql.connector.conect()` fornece dois argumentos para controlar o uso dos arquivos de configuração de conexão MySQL.
- `option_files`: String ou lista de Strings que especifica o caminho para um ou mais arquivos de configuração. Não há valor padrão.
- `option_groups`: especifica de quais grupos de opções (especificado entre colchetes no arquivo) ler os parâmetros. O padrão é ler dos grupos `client` e `connector_python`.

```
[connector_python]
user     = usuário
host     = 127.0.0.1
port     = 3306
password = senha
```

Exemplo de Arquivos de Configuração MySQL

```
[connector_python]
user     = usuário
host     = 127.0.0.1
port     = 3306
password = senha
```

my.cnf

Abaixo, trecho de código para estabelecer conexão usando o arquivo my.cnf

```
1. import mysql.connector
2. conn = mysql.connector.connect(option_files="my.cnf")
3. print("arquivo de configuracao unico")
4. print(f‘ID da conexao con: {conn.connection_id}’)
5. con.close()
```

Exemplos de Arquivos de Configuração MySQL

```
[client]
host     = 127.0.0.1
port     = 3306
```

my_shared.cnf

```
[connector_python]
user     = pyuser
password = Py@pp4Demo
```

specific.cnf

```
1. import mysql.connector
2. conn = mysql.connector.connect(option_files=[“my_shared.cnf”,”specific.cnf”],
                                  option_groups=[“client”, “connector_python”])
3. print(“dois arquivos de configuracao”)
4. print(f‘ID da conexao con: {conn.connection_id}’)
5. conn.close()
```

Exemplo de Arquivos de Configuração MySQL

Utilizando a biblioteca `configparser` para ler o arquivo INI

```
1.  import mysql.connector
2.  import configparser
3.  # Lê o arquivo INI
4.  config = configparser.ConfigParser()
5.  config.read('my.cnf')
6.  # Obtem as informações de configuração
7.  db_config = {
8.      'host': config['mysql']['host'],
9.      'user': config['mysql']['user'],
10.     'password': config['mysql']['password'],
11.     'database': config['mysql']['database'],
12.     'port': int(config['mysql']['port']),
13. }
```

```
14. # Estabelecer a conexão
15. conn =
    mysql.connector.connect(**db_config)
16.
17. ... # faça algo com a conexão
18.
19. # Fechar a conexão
20. conn.close()
```

Exemplos de Programas para Criação de Conexão

- Os programas `exemplo1.py` e `exemplo2.py`, ambos disponíveis na pasta fazConexao, no site da disciplina, ilustram a criação de conexões.

- `exemplo1.py`: estabelece uma conexão usando um dicionário de dados com os parâmetros da conexão.

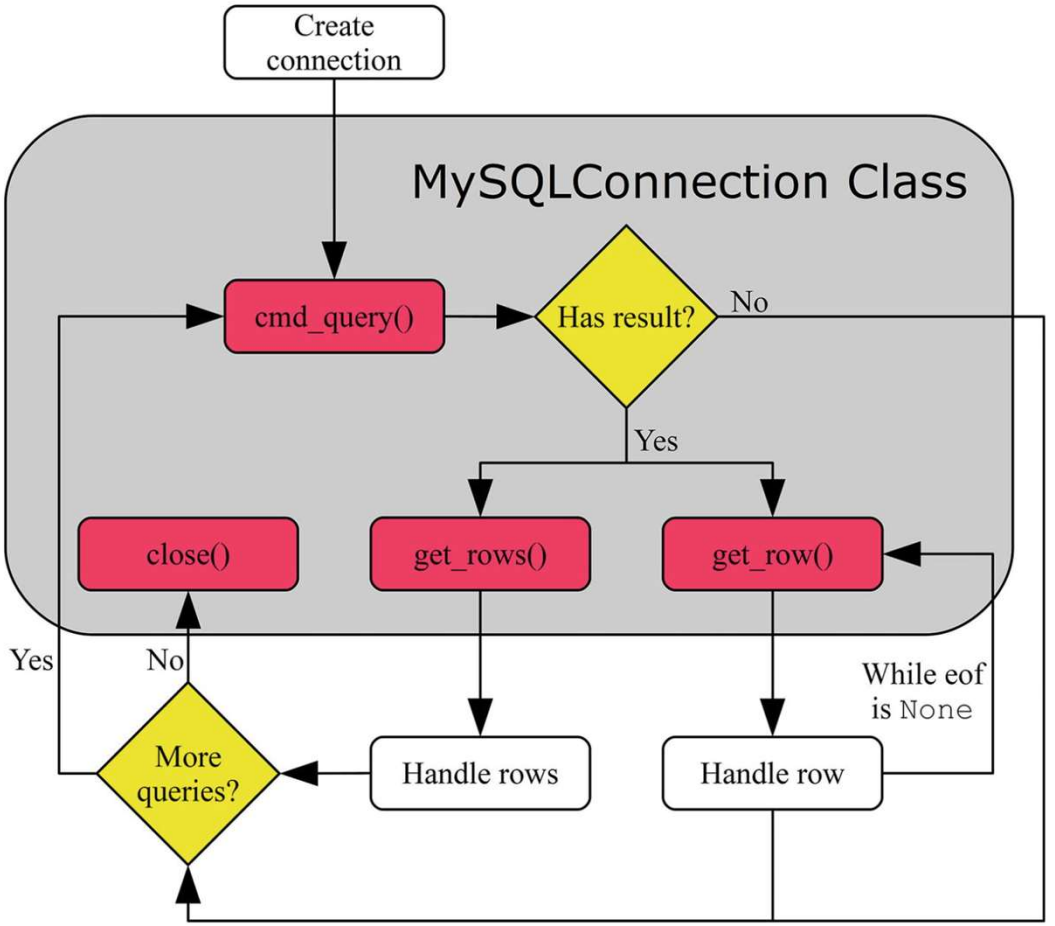
- `exemplo2.py`: estabelece conexão usando um arquivo de configuração.

Agenda

1. MySQL Connector/Python.
2. Conectando ao MySQL usando Connector/Python.
3. **Cursores.**
4. Comandos parametrizados e pré-preparados.
5. Tratamento de erros.

Métodos para Execução de Comandos SQL

- É possível usar o método `cmd_query()` do objeto de conexão para realizar consultas, inserções, atualizações e deleções.
- No caso de consultas, o resultado é recuperado com os métodos `get_rows()` e `get_row()`.
- Entretanto, em geral, é melhor e mais fácil usar um objeto `cursor` para consultar e modificar o banco de dados.



Executando comandos SQL com cmd_query()

- O método `cmd_query()` é muito simples: ele recebe como argumento a string que define o comando a ser executado e retorna um dicionário python com informações sobre o resultado do comando.

- O conteúdo exato do dicionário retornado depende do comando realizado.
  - Por exemplo, para uma consulta SELECT, o dicionário incluirá informações sobre as colunas selecionadas.
  - Para consultas sem um conjunto de resultados (ou seja, que não retornam tuplas), a informação eof é um "pacote OK", que inclui informações sobre o comando.
  - Para todos os comandos, o status também é incluído.

- Um irmão de `cmd_query()` é o método `cmd_query_iter()`, que pode ser usado para enviar de uma só vez várias consultas ao servidor MySQL.

Executando consulta com cmd_query()

```
1. import mysql.connector
2. from pprint
3.
4. printer = pprint.PrettyPrinter()
5. # faz a conexão
6. con = mysql.connector.connect(option_files=”config.cnf")
3. result = con.cmd_query("""SELECT *
4.                           FROM livro
5.                           WHERE livro_id = 135""")
6. # mostra o dicionario retornado
7. print("Dicionario retornado\n" +"="*17)
8. printer.pprint(result)
9. con.close() # fecha a conexão
```

Executando consulta com cmd_query()

- O exemplo do slide anterior faz uma consulta com `cmd_query()` para recuperar as informações sobre o livro com chave 135. Para esta consulta, cmd_query() retorna o seguinte dicionário de dados:

```
{'columns': [('livro_id', 3, None, None, None, None, 0, 53251, 63),
             ('titulo', 253, None, None, None, None, 1, 0, 255),
             ('isbn13', 253, None, None, None, None, 1, 0, 255),
             ('linguagem_id', 3, None, None, None, None, 1, 49160, 63),
             ('num_paginas', 3, None, None, None, None, 1, 32768, 63),
             ('data_publicacao', 10, None, None, None, None, 1, 128, 63),
             ('editora_id', 3, None, None, None, None, 1, 49160, 63)],
 'eof': {'status_flag': 16385, 'warning_count': 0}}
```

Executando consulta com cmd_query() (Cont.)

- A parte `columns` do dicionário será melhor discutida mais adiante; por enquanto, apenas saiba que o primeiro elemento da tupla para uma coluna é o nome da coluna.

- A segunda parte do dicionário de resultados, o elemento `eof`, inclui alguns detalhes sobre a consulta, mas os campos incluídos dependem da consulta. Os campos comuns do elemento `eof` são `status_flag` e `warning_count`.

- `status_flag` não é tão útil quanto possa parecer; na verdade, o valor não está documentado e nenhum significado deve ser tirado de seu valor. `warning_count`, por outro lado, mostra o número de avisos ocorridos durante a consulta.

Executando comandos que não retornam tuplas (Cont.)

- O exemplo abaixo faz um update na tabela livro. Neste caso, tal como discutivo no próximo slide, o campo `eof` inclui informações relevantes sobre o comando.

```
1. import mysql.connector
2. from pprint
3.
4. printer = pprint.PrettyPrinter()
5. # faz a conexão
6. con = mysql.connector.connect(option_files=”config.cnf")
3. result = con.cmd_query("""UPDATE livro
4.                           SET num_paginas = num_paginas-1
5.                           WHERE livro_id < 10""")
6. # mostra o dicionario retornado
7. print("Dicionario retornado\n" +"="*17)
8. printer.pprint(result)
9. con.close() # fecha a conexão
```

Executando comandos que não retornam tuplas (Cont.)

- Para o exemplo do slide anterior, o retorno de `cmd_query()` é o seguinte:

```
{'affected_rows': 9,
 'field_count': 0,
 'insert_id': 0,
 'server_status': 1,
 'warning_count': 0}
```

- `affected_rows`: mostra o número de linhas afetadas. Nesse caso, 9 linhas foram atualizadas.
- `insert_id`: para comandos INSERT inserindo dados em tabelas com colunas autoincrementáveis
  - `insert_id` terá o id da última tupla inserida pelo comando.

Recuperando tuplas: `get_rows()`

- Alguns comandos SQL como CREATE TABLE, ALTER TABLE, INSERT, UPDATE e DELETE, não retornam tuplas e verificar se o comando foi bem-sucedida é tudo o que precisa ser feito.

- No entanto, em geral, a maioria dos comandos SQL submetidos por aplicações são consultas (SELECT) que retornam um conjunto de resultados. Nesse caso, as tuplas devem ser buscadas.

- Quando a consulta é submetida com `cmd_query()`, buscamos as tuplas usando `get_row()` e `get_rows()`:
  - `get_rows()` retorna uma lista contendo todas as tuplas resultantes da consulta.
  - `get_row()` retorna uma tupla de cada vez.

Executando Consulta: get_rows() (Cont.)

```
1. import mysql.connector
2. import pprint
3. printer = pprint.PrettyPrinter(indent=1)
4. # estabelece a conexão
5. con = mysql.connector.connect(option_files=”config.cnf")
3. result = con.cmd_query("""SELECT  titulo, num_paginas
4.                           FROM livro
5.                           WHERE livro_id < 10
6.                           order by titulo""")
7. resultset = con.get_rows() # recupera a lista de tuplas
8. print("Dicionario retornado\n" +"="*20)
9. printer.pprint(result)
10.print(“Tuplas retornadas\n" +"="*20)
11.printer.pprint(resultset)
12.con.close() # fecha a conexão
```

`get_rows()` com limite

- O exemplo anterior busca todas as tuplas do conjunto de resultados e, em seguida as usa. Isso funciona bem para um resultados pequenos, mas não é eficiente para um grande número de tuplas.

- Uma opção é limitar o número de tuplas, especificando o número a ser buscado como um argumento para get_rows().
  - Por exemplo: (tup, eof) = con.get_rows(4) retorna as tuplas em tup e um indicador de final de arquivo em eof.

- O número de tuplas especificado é o número máximo de tuplas a serem lidas por lote. Enquanto houver mais tuplas a serem lidas, `eof` será definido como None.
- Se houver menos linhas disponíveis do que o solicitado, get_rows() retornará o que resta e definirá `eof` para indicar fim de arquivo.

`get_rows()` com limite (Cont.)

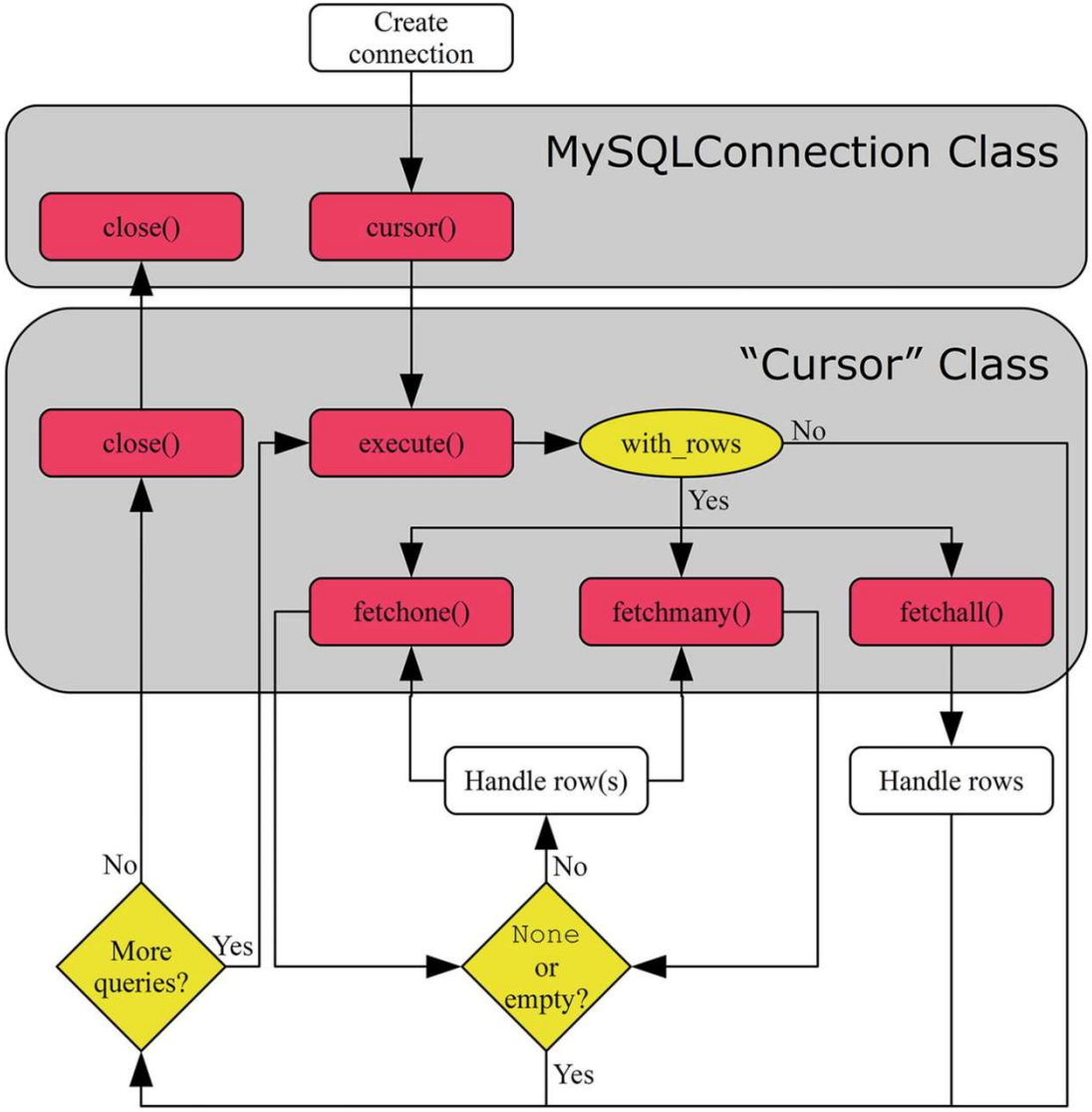
- Os programas `exemplo1.py` e `exemplo2.py`, ambos disponíveis na pasta `consultasBasicas`, no site da disciplina ilustram a utilização de get_rows().

- `exemplo1.py`: estabelece uma conexão e usa get_rows() para recuperar todas as tuplas de uma única vez.

- `exemplo2.py`: estabelece uma conexão e usa get_rows() para recuperar as tuplas iterativamente, de 4 em 4.

Cursores: Introdução

- Objetos cursores fornecem uma maneira amigável (de mais alto nível) de executar comandos SQL:
  - SELECT
  - INSERT
  - UPDATE
  - DELETE
  - CREATE
  - etc.
- Ao lado temos um diagrama que ilustra a operação com cursores.



Instanciação de Cursores

- Para instanciar um objeto cursor usamos o método `cursor()` do objeto de conexão, tal como indicado na linha 3 do trecho do código abaixo:

```
1. import mysql.connector
2. # Estabelece a conexão
3. con = mysql.connector.connect(option_files= “config.cnf”)
3. cur = con.cursor() . #instancia o cursor
4. # faça algo com o cursor
5. cur.close() #fecha o cursor
6. con.close() # fecha a conexão
```

- O cursor é fechado chamando o método `close()` do objeto cursor. Fechar o cursor garante que a referência que aponta de volta ao objeto de conexão seja excluída, evitando vazamentos de memória.

Instanciação de Cursores

- Existem classes cursor com propriedades diferentes, cujo uso depende dos requisitos da aplicação. A tabela abaixo ilustra argumentos do método `cursor()` e a classe resultante que é instanciada:

buffered	raw	prepared	dict	tuple	class
				True	MySQLCursor
True					MySQLCursorBuffered
	True				MySQLCursorRaw
True	True				MySQLCursorBufferedRaw
			True		MySQLCursorDict
True			True		MySQLCursorBufferedDict
				True	MySQLCursorNamedTuple
True				True	MySQLCursorBufferedNamedTuple
		True			MySQLCursorPrepared

Instanciação de Cursores

- Todas as classes cursoras disponíveis são subclasses da classe MySQLCursor. Isso significa que o comportamento em geral é o mesmo para todas as classes cursoras; a diferença são os detalhes de como eles lidam com o resultado dos comandos SELECT e, particularmente, para a classe `MySQLCursorPrepared`, como a consulta é executada.
- Após uma consulta, as classes criadas com o argumento `Buffered` automaticamente buscam do servidor todo o conjunto de resultados e armazena no cliente. São adequadas para situações onde a rapidez de acesso aos dados é mais importante do que o consumo de memória, ou quando é necessário acessar o conjunto completo de resultados múltiplas vezes.

Instanciação de Cursores

- Resultados de consultas com cursores **não buferizados** só são buscados do servidor quando um método de busca de tuplas é chamado, conforme necessário. São úteis para minimizar o uso de memória do cliente, ou quando a latência de acesso aos dados não é uma preocupação.
- `MySQLCursor` retorna resultados na forma de tuplas. `MySQLCursorRaw` de forma "crua" (strings e bytes). `MySQLCursorDict` retorna resultados na forma de um dicionário e `MySQLCursorNamedTuple` na forma de tuplas nomeadas.
- `MySQLCursorPrepared` são usados para realização de consultas pré-preparadas. Em vez de criar uma consulta SQL estática diretamente em uma string, a consulta pré-preparada é parametrizada. Definimos uma vez e depois fornecemos os valores dos parâmetros separados a cada vez que precisamos executá-la. Isso melhora o desempenho de consultas repetitivas.

Execução de comandos SQL com Cursores

- Para realizar consultas ou modificações no BD, o método `execute()` é usado. Isso inclui suporte para executar várias consultas diferentes em uma única chamada. O método `executemany()` é uma variação que pode ser usado para executar a **mesma consulta** com **diferentes** conjuntos de parâmetros.
- O método `execute(cmd, [params, multi])` possui um argumento obrigatório e dois argumentos opcionais:
  - cmd: A string com a consulta a ser executada (obrigatório!).
  - params: Um dicionário, lista ou tupla com os parâmetros a serem usados na consulta. O padrão é None.
  - multi: Quando True, cmd é considerado como sendo várias consultas separadas por ponto e vírgula. Nesse caso execute() retorna um iterador para possibilitar a iteração sobre o resultado de cada consulta.

Execução de consultas com Cursores

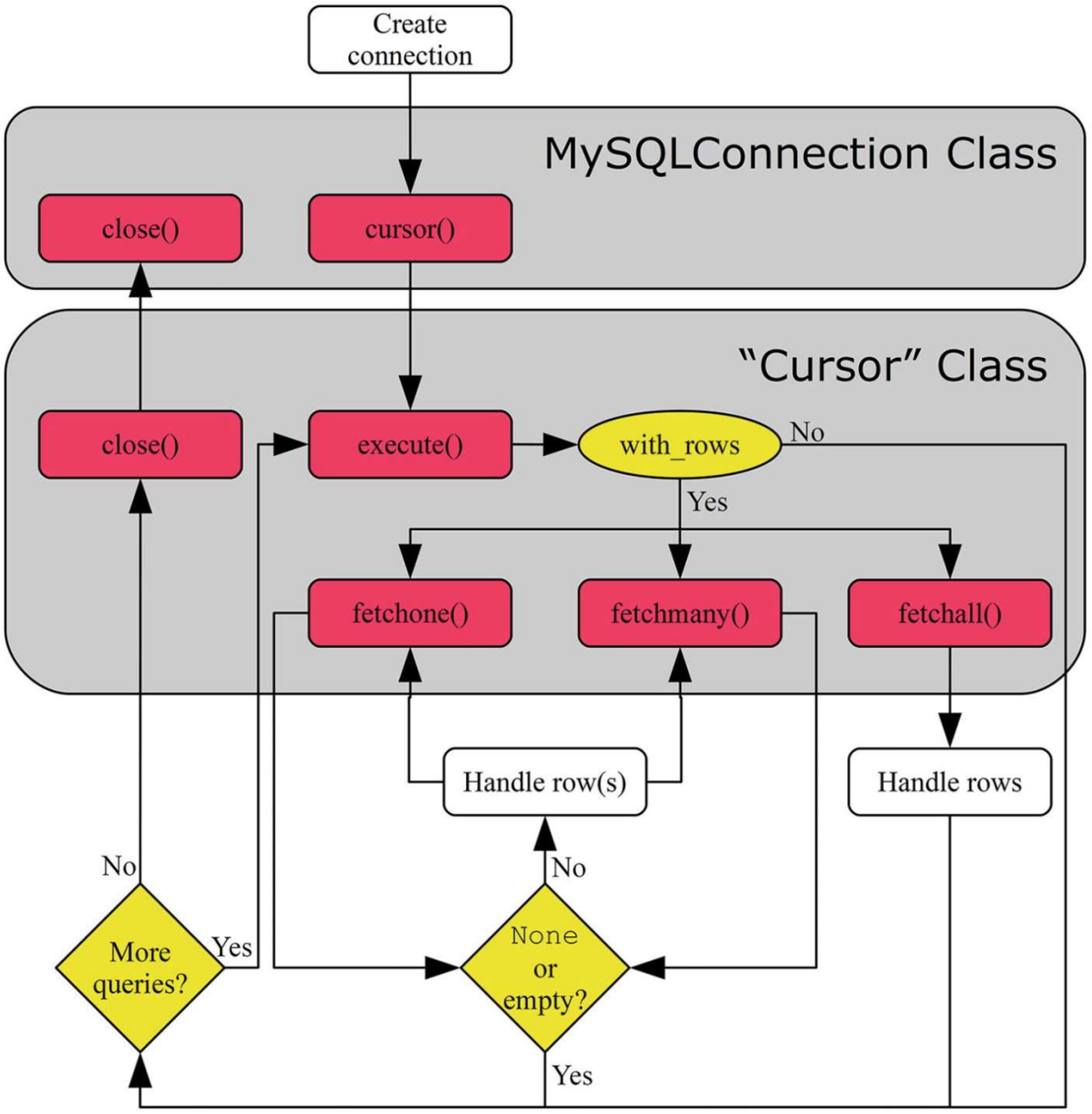
- O conjunto de resultados pode ser buscado usando um dos seguintes métodos:

- `fetchall()`: Busca todas as tuplas restantes. fetchall() usa get_rows() para obter todas as tuplas em uma chamada com cursores não buferizado.

- `fetchmany()`: Obtém um lote de tuplas com a possibilidade de definir o número máximo a serem incluídas no lote. O método `fetchmany()` é implementado usando `fetchone()`. O padrão é ler uma tupla por vez.

- `fetchone()`: Lê uma tupla por vez, o que é equivalente ao método `get_row()`.

Cursores: fluxo de execução



Execução de consultas com Cursores

- Exemplo disponível no site da disciplina no arquivo exemplo1.py da pasta cursores. O exemplo2.py e exemplo3.py são variações que usam outros cursores.

```
1.  #cria uma conexao
2.  conn = mysql.connector.connect(option_files = 'config.cnf')
3.  cur = conn.cursor() #cria um cursor default
4.  sql = """select editora_nome, count(*) as numero_livros
5.          from livro natural join editora
6.          group by editora_nome
7.          having numero_livros > 10"""
8.  cur.execute(sql)       # Realiza a consulta estática
9.  tuplas = cur.fetchall() #busca as tuplas do servidor
10. for tup in tuplas:
11.   print(f'Editora = {tup[0]}, tot. livros = {tup[1]}')
12. cur.close()
13. conn.close()
```

Propriedade dos Cursores

- A seguir estão listadas as principais propriedades disponíveis em um cursor, todas as quais são read-only e se referem ao último comando SQL executado.
  - column_names
  - description
  - lastrowid
  - rowcount
  - with_rows

Propriedade `column_names`

- A propriedade `column_names` inclui o nome de cada coluna na mesma ordem de seus valores. Os nomes das colunas podem, por exemplo, ser úteis se uma tupla precisa ser convertida em um dicionário usando os nomes das colunas como chaves, tal como no exemplo abaixo:

```
1. row = cursor.fetchone()
2. row_dict = dict(zip(cursor.column_names, row))
```

Propriedade `description`

- A propriedade `description` descreve propriedades das colunas de tuplas, como a seguinte formato (impresso usando o módulo pprint):

```
[('Name', 254, None, None, None, None, 0, 1),
 ('CountryCode', 254, None, None, None, None, 0, 16393),
 ('Population', 3, None, None, None, None, 0, 1)]
```

Propriedades `lastrowid`, `rowcount`, `with_rows`

- `lastrowid` pode ser usado para obter o último ID atribuído após inserção em uma tabela com uma coluna auto incrementável.

- O significado da propriedade `rowcount` depende do comando SQL executado:
  - Para instruções SELECT, é o número de tuplas retornadas.
  - Para instruções de modificação de dados (DML), como INSERT, UPDATE e DELETE, é o número de linhas afetadas.

- A propriedade `with_rows` é um booleano que é True quando o comando retorna um conjunto de resultados.
  - Obs: `with_rows` não é definido como False quando todas as linhas tiverem sido lidas.

Agenda

1. MySQL Connector/Python.
2. Conectando ao MySQL usando Connector/Python.
3. Cursores.
4. **Comandos parametrizados e pré-preparados.**
5. Tratamento de erros.

Comandos Parametrizados e Pré-Preparados

- Muitas vezes, consultas são geradas com base em entrada de usuários ou outras fontes externas. Afinal, um programa com todas as consultas estáticas raramente é de muito interesse.
- É fundamental tratar essas entradas de forma apropriada. A manipulação inadequada pode, na melhor das hipóteses, resultar em erros misteriosos e, na pior das hipóteses, pode resultar em inconsistências e ataques ao banco de dados.
- Independente da linguagem utilizada, é sempre importante validar as entradas. Além disso, a API MySQL Connector/Python permite a especificação de consultas parametrizadas e pré-preparadas.

Comandos SQL Parametrizados

- Uma ótima maneira de defender o banco de dados contra tentativas de ataques por injeção de SQL é usar consultas parametrizadas. Isso passará a tarefa para o MySQL Connector/Python.
- Existem duas maneiras de executar consultas parametrizadas com o método `execute()` do cursor.
  1. Fornecer uma lista ou tupla com os valores, na mesma ordem em que aparecem na consulta. Nesse caso, cada parâmetro é representado por `%s` na consulta.
  2. Fornecer um dicionário onde cada parâmetro recebe um nome (a chave do dicionário com o valor sendo o valor do parâmetro). Isso facilita a leitura do código-fonte.

Comandos SQL Parametrizados

```python
#executa uma consulta
sql = """select editora_nome, count(*) as numero_livros
        from livro natural join editora
        group by editora_nome
        having numero_livros > %s"""

params = 10
cur.execute(sql, params=(params,)) #usando tupla
```

Pasta cursor, exemplo4.py

```python
#executa uma consulta
sql = """select editora_nome, count(*) as numero_livros
        from livro natural join editora
        group by editora_nome
        having numero_livros > %(qtd)s"""

params = {'qtd': 10}
cur.execute(sql, params=params) #usando dicionário
```

Pasta cursor, exemplo5.py

Comandos SQL Pré-Preparados

- Comandos pré-preparados apresentam algumas vantagens sobre as formas mais diretas de execução de consultas usadas até agora.
- Duas das vantagens são o desempenho aprimorado quando uma consulta é reutilizada e a proteção contra injeção de SQL.
- Do ponto de vista da aplicação, há pouca diferença entre usar parametrização ou instruções pré-preparadas. Na verdade, a diferença está apenas na instanciação subclasse cursora.
- Nos bastidores, porém, existem diferenças sutis. A primeira vez que a consulta é executada, a instrução é preparada; ou seja, a instrução é enviada ao SGBD MySQL que prepara o comando para uso futuro. Em seguida, o cursor envia um comando para dizer ao servidor MySQL para executar a instrução preparada junto com os parâmetros.

Comandos SQL Pré-Preparados

Pasta cursor, exemplo6.py

```
#cria uma conexao
con = mysql.connector.connect(option_files = 'config.cnf')

#cria um cursor
cur = con.cursor(prepared=True)
sql = """select editora_nome, count(*) as numero_livros
        from livro natural join editora
        group by editora_nome
        having numero_livros > %s"""

params = 10
cur.execute(sql, params=(params,)) #usando tupla

if cur.with_rows:
    tuplas = cur.fetchall()
    for tup in tuplas:
        print("Editora = {0}, Numero_livros = {1}".format(tup[0], tup[1]))

print("\nRepetindo a consulta com novos parâmetros:")
params = 20
cur.execute(sql, params=(params,)) #usando tupla

if cur.with_rows:
    tuplas = cur.fetchall()
    for tup in tuplas:
        print("Editora = {0}, Numero_livros = {1}".format(tup[0], tup[1]))


cur.close()
con.close()
```

Agenda

1. MySQL Connector/Python.
2. Conectando ao MySQL usando Connector/Python.
3. Cursores.
4. Comandos parametrizados e pré-preparados.
5. **Tratamento de erros.**

Erros em Comandos SQL

- O módulo `mysql.connector.errors` define classes de exceção para erros e warnings gerados pelo MySQL Connector/Python.
- A maioria das classes definidas neste módulo estão disponíveis quando você importa `mysql.connector`.
- Os erros do servidor MySQL são mapeados para exceção Python com base no valor de SQLSTATE (consulte Referência de mensagem de erro do servidor). No link a seguir você encontra o SQLSTATE e exceção do Connector/Python correspondentes:
- https://dev.mysql.com/doc/connector-python/en/connector-python-api-errors.html

mysql.connector.Error

- Essa exceção é a classe base para todas as outras exceções do módulo `errors`.
- Ela pode ser usada para detectar todos os erros em uma única instrução `except`. O exemplo a seguir mostra como podemos detectar erros de sintaxe:

```
1. import mysql.connector
2. try: cnx = mysql.connector.connect(user='scott',database='employees’)
3.   cursor = cnx.cursor()
4.   cursor.execute("SELECT * FORM employees") # erro de sintaxe
5.   cnx.close()
6. except mysql.connector.Error as err:
7.   print(f'Alguma coisa de errado aconteceu: {err}')
```