Desenvolvimento de Aplicações de Banco de Dados

Python

Prof. Alexandre L. Gonçalves

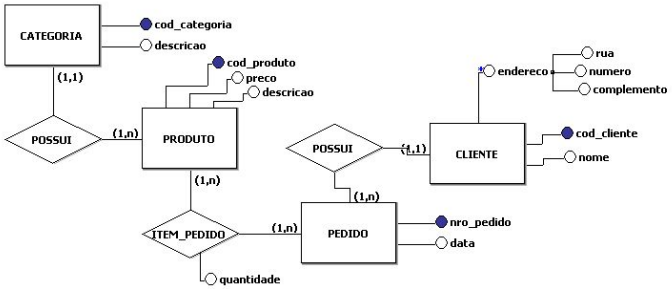
E-mail: a.l.goncalves@ufsc.br

Mapa

Descrição do Problema

Uma empresa vende produtos de limpeza, e deseja melhor controlar os produtos que vende, seus clientes e os pedidos. Cada produto é caracterizado por um código único, nome do produto, categoria (ex. detergente, sabão em pó, sabonete, etc), e seu preço. A categoria é uma classificação criada pela própria empresa. A empresa possui informações sobre todos seus clientes. Cada cliente é identificado por um código único (interno à firma), o nome do cliente, endereço (rua, nro, complemento, cidade, cep, UF), telefone, o status do cliente ("bom", "médio", "ruim"), e o seu limite de crédito. Guarda-se igualmente a informação dos pedidos feitos pelos clientes. Cada pedido possui um número (único), e guarda-se a data de elaboração do pedido. Cada pedido pode conter de 1 a vários produtos, e para cada produto, indica-se a quantidade deste pedida.

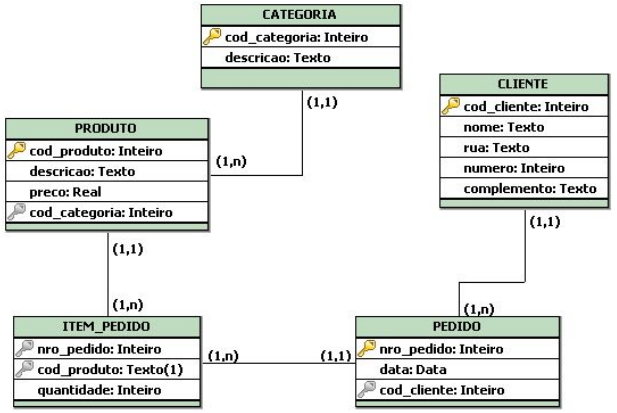
Modelagem



Modelo Conceitual

Projeto Lógico

Modelo Lógico



DDL/SQL

Banco de Dados

DML/SQL

Projeto Físico

Introdução

- O componente SQL dinâmica da SQL permite que programas construam e submetam consultas SQL em tempo de execução.

- Usando SQL dinâmica, programas podem criar consultas SQL como *strings* em tempo de execução (talvez baseados na entrada do usuário) e executá-las imediatamente ou mesmo prepará-las para uso subsequente.

Introdução

- Preparar uma instrução SQL dinâmica significa pré-compilar a instrução para que os usos subsequentes sejam sobre a versão compilada.

- Se uma declaração SQL é executada repetidamente com uma consulta pré-compilada (*Prepared Statement*) o custo de execução é reduzido.

- Pode-se dizer que existem dois modelos de SQL dinâmica: a) incorporação de chamadas SQL em uma linguagem hospedeira, como C ou Java; e b) utilização de uma API.

Introdução

- Independente do modelo, é necessário o conceito de sessão SQL. Para tal, os seguintes passos são executados:
  - O usuário (via aplicação) ou determinada aplicação se conecta a um servidor SQL;
  - Desse modo, é estabelecida uma sessão SQL;
  - Com a sessão é possível realizar uma série de instruções SQL;
  - Após isso é necessário desconectar a sessão.

Introdução

- Sendo assim, todas as atividades do usuário ou aplicação estão no contexto de uma sessão SQL;

- Além dos comandos SQL normais, uma sessão também pode conter comandos para confirmar o trabalho realizado (commit), ou reverter o trabalho realizado (rollback) na sessão.

Python

- Características Principais
- Lançada em 1991 por Guido van Rossum (Stichting Mathematisch Centrum, na Holanda);
- **Imperativa**, **interpretada**, de **alto nível** e com **tipagem forte** e **dinâmica;**
- Não utiliza delimitadores formados por chaves, usados em linguagens como C, C++, C# e Java;
- Curva de aprendizado menor em relação a outras linguagens tradicionais.

Whitespace Formatting

- Python utiliza indentação para delimitar blocos de código

```
for i in [1, 2, 3, 4, 5]:
    print(i)
    for j in [1, 2, 3, 4, 5]:
        print(j)
        print(i + j)
    print(i)
print(“Done!!")
```

Modules

- Algumas funcionalidades do Python não são carregadas por padrão. Inclui tanto características da própria linguagem quanto bibliotecas de terceiros.

```
#importação com renomeação
import re as regex
my_regex = regex.compile("[0-9]+", regex.I)
print(my_regex)

name_check = regex.compile(r"[^A-Za-zs.]")
name = input(“Informe o seu nome: ")
while name_check.search(name):
    print(“Por favor, entre com seu nome corretamente!")
    name = input(“Informe o seu nome: ")
```

Modules

```
#importação explícita de determinados módulos
from collections import defaultdict, Counter
food_list =
'macarronada macarronada macarronada macarronada macarronada pizza pizza pizz
a'.split()
food_count = defaultdict(int)
for food in food_list:
    food_count[food] += 1 # incrementa os elementos em 1
print(food_count)

#Counter é utilizado para contar objetos em uma estrutura
lang_list = ['Python', 'C++', 'C', 'Java', 'Python', 'C', 'Python', 'C++', 'Python', 'C']
print(Counter(lang_list))
occurrence = Counter(lang_list)
print(“A ocorrência da palavra Python é:",occurrence['Python'])
```

Functions

- Uma função é uma regra que obtém zero ou mais parâmetros retornando uma saída.
- Uma função é definida usando **def:**

```
"""Comentário sobre a função"""
def double(x):
    return x * 2

res = double(2.5)
print(res)

"""Comentário sobre a função"""
def apply_to_one(f):
    return f(1)

x = apply_to_one(double)
print(x)
```

Functions

```
"""Comentário sobre a função"""
def my_print(message="mensagem padrão"):
    print(message)
my_print("olá")
my_print()

"""Comentário sobre a função"""
def subtract(a=0, b=0):
    return a - b
print(subtract(10, 5))
print(subtract(0,5))
print(subtract(b=5))
```

Strings

- Strings podem ser delimitadas por apóstrofos ou aspas.

```
single_quoted_string = 'ciência de dados'
double_quoted_string = "ciência de dados"
print(single_quoted_string)
print(double_quoted_string)

tab_string = "\t"  #representa o caracter tab
print(len(tab_string))
print(tab_string)

multi_line_string = """Esta é a primeira linha
e esta é a segunda linha"""
print(multi_line_string)
```

Exceptions

- Quando algo errado ocorre no código uma exceção é lançada. Caso não seja tratada adequadamente, o programa será finalizado. Para tal, deve utilizar **try** e **except**.

```
try:
    print(0 / 0)
except ZeroDivisionError:
    print("Não é possível dividir por zero")
```

Lists

- Listas são uma das estruturas mais fundamentais em Python. Uma lista é simplesmente uma coleção ordenada.

```
integer_list = [1, 2, 3]
heterogeneous_list = ["string", 0.1, True]
print(len(integer_list))
print(len(heterogeneous_list))

#obtendo e atribuindo valores para
determinado elemento
x = [1, 2, 3, 4, 5]
print(x)
print(x[0])
x[0] = -1
print(x)
```

Lists

```
#verificação de pertinência
print(2 in x)

#acrescentando elementos em uma lista
x.extend([6, 7, 8])
print(x)

#acrescentando um elemento para uma
lista
x.append(9)
print(x)
```

Tuples

- Tuplas são listas imutáveis. Uma tupla é especificada utilizando parênteses ao invés de colchetes.

```
my_list = [1, 2]
my_tuple = (1, 2)
other_tuple = 3, 4
my_list[1] = 3
print(my_list)
print(my_tuple)

try:
    my_tuple[1] = 3
except TypeError:
    print("Não é possível modificar uma tupla")
```

Dictionaries

- Outra estrutura de dados fundamental é o dicionário, que associa valores com chaves permitindo uma rápida recuperação do valor a partir da chave.

```
grades = {"Aluno 1" : 8.5, "Aluno 2" : 9.0, "Aluno 3": 7.0}
print(grades)
grade = grades["Aluno 2"]
print(grade)

#Um erro será retornado se a chave não for localizada no
dicionário
try:
    grade = grades["Aluno 4"]
except:
    print("Nenhuma nota para o Aluno 4")
```

Dictionaries

```
#Pode-se utilizar o método get para obter os valores ao invés de tratar em uma exceção
std_1 = grades.get("Aluno 1", 0)
print(std_1)
std_4 = grades.get("Aluno 4", 0)
print(std_4)
print(len(grades))

#Utilizado para representar dados estruturados
tweet = {
    "user" : "Usuário 1",
    "text" : "Ciência de Dados é uma área promissora",
    "retweet_count" : 100,
    "hashtags" : ["#data", "#science"]
}
print(tweet.keys())
print(tweet.values())
print(tweet.items())
```

Control Flow

- Na maioria das linguagens de programação é possível executar ações condicionalmente utilizando **if** e realizar iterações através dos comandos **while** e **for**.

```
#Comando if
if (1 > 2):
    message = "se 1 for maior que 2."
elif (1 > 3):
    message = "caso contrário testa elsif."
else:
    message = "quando todos os outros
casos falham."
print(message)
```

```
#while loop
x = 0
while (x < 10):
    print(x, "é menor que 10.")
    x += 1

#for
for x in range(10):
    print(x, "é menor que 10.")
```

Sorting

- Uma lista em Python possui um método para classificação dos elementos.

```
x = [4, 1, 2, 3]
y = sorted(x)
print(y)
x.sort()
print(x)

#Ordenação do maior para o menor
x = sorted([-4, 1, -2, 3], key=abs, reverse=True)
print(x)
```

List Comprehensions

- Frequentemente é necessário transformar uma lista em outra lista, ou escolhendo somente certos elementos ou transformando os elementos.

```
#Transformando uma lista em outra lista
even_numbers  = [x for x in range(5) if x % 2 == 0]
print(even_numbers)
squares      = [x * x for x in range(5)]
print(squares)
even_squares = [x * x for x in even_numbers]
print(even_squares)
```

List Comprehensions

```
#Tonando listas em dicionários ou conjuntos
square_dict = {x : x * x for x in range(5)}
print(square_dict)
square_set = {x * x for x in [1, 2]}
print(square_set)

#Pode incluir múltiplos fors
pairs = [(x, y)
        for x in range(10)
        for y in range(10)]
print(pairs)
```

Randomness

- Na análise de dados, frequentemente existe a necessidade de gerar números randômicos, o que pode ser realizado utilizando o módulo **random**.

```
import random

four_uniform_randoms = [random.random() for _ in range(4)]
print(four_uniform_randoms)

#Obtém um valor de uma lista de maneira randômica
print(random.randrange(10))
print(random.randrange(2, 8))
```

Randomness

```
#Embaralhando
up_to_ten = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
random.shuffle(up_to_ten)
print(up_to_ten)

#Gerando uma amostra randômica
lottery_numbers = range(1, 60)
winning_numbers = random.sample(lottery_numbers, 6)
print(winning_numbers)
```

Efetuando uma Conexão

- Primeiramente, é necessário abrir uma conexão com um banco de dados para que se possa executar instruções SQL, ou seja, é necessário obter a sessão SQL.

- Cada provedor de banco de dados fornece o seu driver ou conector possibilitando a realização de conexões.

Efetuando uma Conexão

```
# Importação de bibliotecas
import mysql.connector
from mysql.connector import errorcode
from datetime import date, datetime
import pandas as pd

# Estabelece a conexão com o banco de dados
try:
    conn = mysql.connector.connect(user='root', password='master',
                                   host='localhost',
                                   database='bd',
                                   autocommit=False)
except mysql.connector.Error as error:
    print("Não foi possível realizar a conexão com o banco de dados {}.".format(error))

# Apresenta o valor da propriedade autocommit.
print(conn.autocommit)
```

Criando um Cursor

- Um objeto do tipo Cursor é utilizado para executar declarações (comandos) que se comunicam com um banco de dados relacional.

- Utilizando os métodos de um objeto Cursor é possível criar objetos no banco de dados, consultar dados, iterar sobre conjuntos de dados, chamar procedimentos, executar comandos que modificam o estado do banco de dados.

- Um objeto Cursor é criado utilizando o método cursor() a partir do objeto de conexão previamente instanciado.

Criando um Cursor

```
# Criando as tabelas
tables = {}
tables['departamento'] = (
    "CREATE TABLE `departamento` ("
    "  `id_depto` integer NOT NULL AUTO_INCREMENT,"
    "  `nome` varchar(100) NOT NULL,"
    "  PRIMARY KEY (`id_depto`)"
    ")")
tables['empregado'] = (
    "CREATE TABLE `empregado` ("
    "  `id_emp` INTEGER NOT NULL AUTO_INCREMENT,"
    "  `nome` VARCHAR(100) NOT NULL,"
    "  `data_nascimento` timestamp,"
    "  `data_contratacao` timestamp,"
    "  `id_depto` integer,"
    "  PRIMARY KEY (`id_emp`),"
    "  FOREIGN KEY(`id_depto`) REFERENCES `departamento`
(`id_depto`)"
    ")")
```

Criando um Cursor

```
# Cria um cursor
cursor = conn.cursor()

# Itera sobre o dicionário e cria uma tabela para cada entrada utilizando o cursor
for table_name in tables:
    table_description = tables[table_name]
    try:
        print("Criando tabela {}: ".format(table_name), end='')
        cursor.execute(table_description)
    except mysql.connector.Error as err:
        if err.errno == errorcode.ER_TABLE_EXISTS_ERROR:
            print("Tabela já existe!!!")
        else:
            print(err.msg)
    else:
        print("OK")
# Fecha o cursor
cursor.close()
```

Criando Declarações SQL

- Uma declaração SQL é executada utilizando-se um cursor, podendo ser de consulta ou atualização do banco de dados.

- Caso seja de atualização, após a execução do comando e com autocommit=False, deve-se executar explicitamente o método commit().

Criando Declarações SQL - INSERT

```
# Insere um empregado

cursor = conn.cursor()
insert = ("INSERT INTO empregado (nome, data_nascimento, "
            " data_contratacao, id_depto)"
           " VALUES ('Funcionário 1','1997-6-14','2022-10-10', 1)")
cursor.execute(insert)
conn.commit()
print("Empregado inserido com sucesso!!!")
```

Criando Declarações SQL - SELECT

```
# Seleciona todos os empregados

cursor = conn.cursor()

select = ("SELECT id_emp, nome, data_nascimento, data_contratacao, "
          "id_depto FROM empregado")

cursor.execute(select)

for (id_emp, nome, data_nascimento, data_contratacao, id_depto) in cursor:
  print("{}, {}, nascido em {:%d/%m/%Y}, foi contratado em {:%d/%m/%Y} para "
        "trabalhar no departamento {}.".format(id_emp, nome, data_nascimento,
        data_contratacao, id_depto))
```

Criando Declarações SQL - DELETE

```
# Elimina uma ou mais linhas da tabela empregado

cursor = conn.cursor()
delete = ("DELETE FROM empregado WHERE nome = 'Funcionário 1'")
cursor.execute(delete)
conn.commit()
print("Exclusão ocorrida com sucesso!!!")
```

Criando Declarações Parametrizadas

- Uma declaração SQL pode ser parametrizada, o que garante flexibilidade para o desenvolvimento de aplicações.

- Para garantir esta flexibilidade deve-se utilizar a expressão ‘%s’.

- A partir disso, basta:
  a) Criar uma declaração SQL (insert, update, delete ou select);
  b) Criar uma tupla com os dados requeridos;
  c) Utilizar o método execute().

Criando Declarações Parametrizadas - INSERT

```
cursor = conn.cursor()

datetime = datetime.now()

insert_employee = ("INSERT INTO empregado "
            "(nome, data_nascimento, data_contratacao, id_depto) "
            "VALUES (%s, %s, %s, %s)")

data_employee = ('Empregado 1', date(1977, 6, 14), datetime, 1)

many_data_employee = [
  ('Empregado 2', date(1980, 2, 12), datetime, 1),
  ('Empregado 3', date(1999, 5, 23), datetime, 1),
  ('Empregado 4', date(1987, 10, 3), datetime, 2),
  ('Empregado 5', date(1993, 7, 11), datetime, 2)
]
```

Criando Declarações Parametrizadas - INSERT

```
# Insere um empregado
cursor.execute(insert_employee, data_employee)
print("Empregado inserido com sucesso!!!")

# Insere vários empregados
cursor.executemany(insert_employee, many_data_employee)
print("Empregados inseridos com sucesso!!!")

conn.commit()
```

Criando Declarações Parametrizadas - SELECT

```
# Seleciona empregado considerando um intervalo de data

cursor = conn.cursor()

query = ("SELECT id_emp, nome, data_nascimento, data_contratacao,"
         " id_depto FROM empregado"
         " WHERE data_nascimento BETWEEN %s AND %s")

hire_start = date(1977, 1, 1)
hire_end = date(2000, 12, 31)

cursor.execute(query, (hire_start, hire_end))

for (id_emp, nome, data_nascimento, data_contratacao, id_depto) in cursor:
  print("{}, {} foi contratado em {:%d/%m/%Y}".format(
    id_emp, nome, data_contratacao))
```

Criando Declarações Parametrizadas - UPDATE

```
#Atualiza departamento de determinado empregado considerando o nome
cursor = conn.cursor()

query = ("UPDATE empregado set id_depto = %s "
        " WHERE nome = %s")

cursor.execute(query, (2, 'Empregado 1'))

print("Atualização realizado com sucesso!!!")

conn.commit()
```

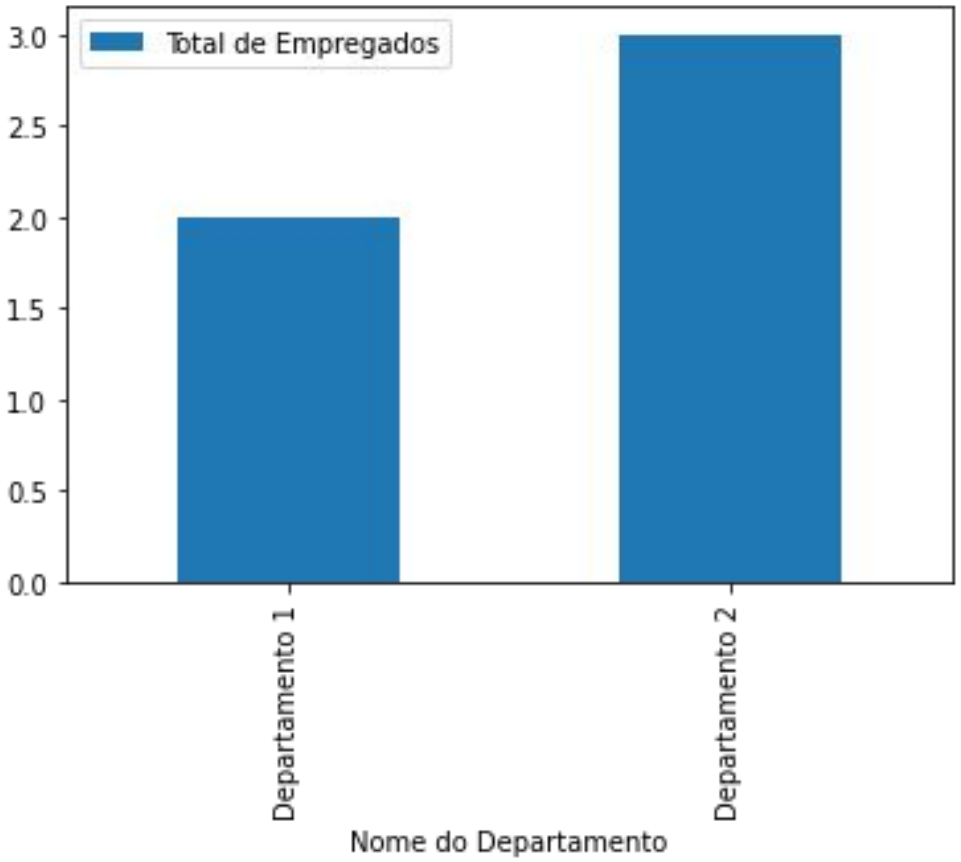
Consultando com Funções de Agregação

 # Executa uma consulta com agregação e armazena o resultado em um DataFrame

```
query = ("select d.nome as 'Nome do Departamento', "
         " count(*) as 'Total de Empregados' "
         " from empregado as e inner join departamento as d "
         " on e.id_depto = d.id_depto "
         " group by 1")
df = pd.read_sql(query, con = conn)
df
```

Apresentando o Resultado em um Gráfico

df.plot(kind="bar", x="Nome do Departamento")



Bons Estudos!