Tema 18

Bases de Dados

Objetivos:

- Bases de dados relacionais
- SQL
- Acesso programático a bases de dados

18.1 Bases de Dados

As bases de dados são um componente fundamental em muitas aplicações, pois agem como repositórios onde é possível armazenar e consultar informação. É certo que isto pode ser feito utilizando ficheiros comuns, por exemplo é perfeitamente possível armazenar uma listagem de clientes num ficheiro de texto que depois é lido pela aplicação. No entanto,

· se a quantidade de dados é grande;

· se se pretende consultar os dados de forma não sequencial;

· se se pretende pesquisar ou ordenar dados segundo múltiplos critérios;

· se se pretende cruzar dados de várias naturezas;

· se ocasionalmente se acrescentam dados;

· se várias das anteriores são verdade;

então uma base de dados é uma solução muito mais poderosa e eficiente do que um ficheiro tradicional. As bases de dados, em particular as **relacionais**, possuem características de manipulação de informação muito interessantes, que facilitam a sua utilização face a outros métodos.

Considere uma lista de contactos de clientes em que é necessário armazenar o nome, endereço de correio electrónico e contacto telefónico. Um ficheiro simples poderia armazenar esta informação, por exemplo utilizando o formato Comma Separated Values (CSV)[1].

```
João, Manuel, Fonseca, jmf@gmail.com, 912345654
Pedro, Albuquerque, Silva, pedro23@gmail.com, 932454349
Maria, Carreira, Dinis, mariadi@ua.pt, 234958673
Catarina, Alexandra, Rodrigo, calexro@sapo.pt, 963343386
```

Se precisarmos de procurar um contacto de um cliente específico, temos de percorrer o ficheiro até o encontrar. Se precisarmos de alterar os dados de algum cliente, temos de reescrever o resto do ficheiro. Considerando que uma base de dados de clientes (ou outra) pode conter milhares ou milhões de entradas, usar um simples ficheiro de texto rapidamente se torna um problema.

Uma *base de dados relacional* organiza os dados na forma de *tabelas*. Cada tabela contém várias linhas, cada uma contendo um *registo* (ou *tuplo*) de dados de uma entidade de um certo tipo, e cada linha contém várias colunas, correspondendo aos diferentes *atributos* (ou *campos*) dessa entidade. Por exemplo, os dados de contactos descritos acima poderiam ser guardados numa base de dados relacional na forma de uma tabela com 4 registos com 5 atributos cada, como representado abaixo.

Table: contacts

firstname (TEXT)	**middlename** (TEXT)	**lastname** (TEXT)	**email** (TEXT)	**phone** (TEXT)
João	Manuel	Fonseca	jmf@gmail.com	912345654
Pedro	Albuquerque	Silva	pedro23@gmail.com	932454349
Maria	Carreira	Dinis	mariadi@ua.pt	234958673
Catarina	Alexandra	Rodrigo	calexro@sapo.pt	963343386

Na base de dados, a tabela é caracterizada por um nome e pelo identificador e tipo de dados de cada atributo.

Os *sistemas de gestão de bases de dados*(SGBD) são programas que permitem fazer pesquisas e alterações de registos numa base de dados, segundo critérios complexos, de forma versátil e eficiente. Um ponto importante é que os dados possuem relações entre si, sendo que base de dados segue um dado modelo relacional. Do exemplo anterior podemos assumir que o modelo possui pessoas e contactos. As pessoas podem ter um ou mais contactos e os contactos pode ser de dois tipos: email e telefone. Com base neste tipo de organização, durante os exercícios seguintes, irá ser explorado como se pode aceder e armazenar informação.

Para isto, existem diversos sistemas de gestão de bases de dados relacionais, mas todos suportam interação com programas clientes através de uma linguagem comum: a linguagem Structured Query Language (SQL). Esta permite efectuar operações muito detalhadas sobre a estrutura da informação ou os seus dados, o que iremos explorar de seguida.

Para criar uma base de dados vamos usar o **`sqlite3`**, que é um SGBD simples que permite criar bases de dados num sistema de ficheiros tradicional, usando o formato *SQLite*. Esta base de dados é muito usada e encontra-se por exemplo nas plataformas Android e iOS. Em alternativa, poderíamos utilizar um servidor de bases de dados, como se faz tipicamente em ambientes de produção, mas do ponto de vista de utilização as diferenças são mínimas.

Para criar uma base de dados é necessário executar a ferramenta **`sqlite3`** com a seguinte sintaxe:[1]

```
sqlite3 database.db
```

Deve aparecer novo *Prompt* que nos permite invocar comandos sobre a base de dados. Um exemplo de um comando é a criação de uma tabela, que se faz através da linguagem SQL:

```
CREATE TABLE tablename(
    field1 TYPE1,
    field2 TYPE2
);
```

Em que **`tablename`** representa o nome da tabela, **`field1`** representa o nome da primeira coluna e **`TYPE1`** o seu tipo de dados. São suportados vários tipos de dados. Os mais relevantes para este trabalho são:

TEXT: Texto.

INTEGER: Números inteiros.

REAL: Números reais.

BLOB: Um qualquer conteúdo (ex, uma imagem).

[1] Esta ferramenta deve existir nos repositórios da maioria dos sistemas.

Exercício 18.1

Crie uma tabela chamada **contacts** contendo os campos identificados anteriormente. Para facilitar a criação da base de dados, crie os comandos num ficheiro de texto e depois copie e cole o texto para o *Prompt* da ferramenta.

Pode verificar o conteúdo da tabela se executar o comando **.tables**. Pode sair desta ferramenta executando o comando **.quit**. (Estes comandos são específicos do **sqlite**. Não são SQL.)

Depois de criada a tabela, é necessário inserir dados. Para isso, usa-se o comando **INSERT INTO**.

```
INSERT INTO tablename
VALUES (
    "value1", "value2"
    );
```

A directiva **VALUES** indica os valores a colocar nas colunas existentes. Valores do tipo **TEXT** devem possuir aspas. É possível inserir informação em apenas alguns campos, bastando para isso que se especifiquem quais os campos de destino da informação.

```
INSERT INTO tablename(field1)
VALUES (
    "value1"
    );
```

Exercício 18.2

Construa os comandos necessários para inserir os dados de todos os clientes. Mais uma vez, crie os comandos num ficheiro de texto e depois copie-os para o **sqlite3**.

De forma consultar os dados inseridos em tabelas utiliza-se o comando **SELECT**. Com este comando é possível discriminar de forma muito detalhada que informação deve ser obtida e em que formato. Na sua forma básica o comando **SELECT** pode obter toda a informação de uma tabela:

```
SELECT * FROM contacts;
```

O que deverá produzir o seguinte resultado:

```
João|Manuel|Fonseca|jmf@gmail.com|912345654
Pedro|Albuquerque|Silva|pedro23@gmail.com|932454349
Maria|Carreira|Dinis|mariadi@ua.pt|234958673
Catarina|Alexandra|Rodrigo|calexro@sapo.pt|963343386
```

Também é possível obter apenas algumas colunas, e/ou apenas algumas linhas. Por exemplo, poderemos obter apenas o email e número de telefone dos contactos com nome Pedro:

```
SELECT email,phone FROM contacts WHERE firstname="Pedro";
```

Exercício 18.3

Construa vários comandos que permitam obter informação específica sobre os utilizadores e teste-os.

Exercício 18.4

Adicione a directiva **`ORDER BY columnname ASC`** aos seus comandos e compare o resultado. Em vez de **`ASC`**, também pode utilizar **`DESC`**

Por vezes é necessário actualizar informação, nomeadamente mudar o valor de células específicas ou mesmo apagar linhas inteiras. O comando **`UPDATE`** permite actualizar uma ou mais células. Por exemplo, podemos mudar o número de telefone do João Fonseca através do comando:

```
UPDATE contacts SET phone = 912345653 WHERE email="jmf@gmail.com";
```

Tem de se ter em consideração que o comando **`UPDATE`** altera todas as linhas, excepto se for especificada uma regra que restrinja o número de linhas a considerar. Neste caso isto é feito usando **`WHERE email="jmf@gmail.com"`**. Sem esta especificação o comando **`UPDATE`** iria colocar todas as linhas com o mesmo número de telefone.

Exercício 18.5

Construa um comando que altere o último nome do utilizador com telefone 963343386 para "Sousa".

Finalmente, é possível apagar linhas inteiras utilizando o comando **DELETE**. Deve-se ter o mesmo cuidado que com o comando **UPDATE** no sentido em que o comando **DELETE** pode apagar uma ou mais linhas. O exemplo seguinte apaga uma linha específica da tabela:

```
DELETE FROM contacts WHERE phone = 912345653;
```

18.1.1 Modelo Relacional

Em geral, uma aplicação terá várias tabelas, cada uma com um tipo de dados. Por exemplo, à nossa base de dados podemos acrescentar uma tabela das empresas clientes, às quais pertencem os clientes anteriormente listados.

Table: companies

name	address	vatnumber
MaxiPlano	Aveiro	123123123123
Luís Manuel & filhos	Águeda	54534343435
ProDesign	Porto	54534343435

Exercício 18.6

Crie uma tabela chamada **companies** e insira a informação anterior.

Levanta-se agora a questão de como indicar que uma dada pessoa pertence a uma empresa. Uma solução passaria por armazenar o nome da empresa junto de cada contacto. E a morada e número fiscal da empresa? Também se acrescentam à tabela de contactos?

Mas isso implicaria replicar informação, visto que várias pessoas pertencem à mesma empresa. Isso é um problema porque caso uma empresa mudasse a morada, teríamos de pesquisar todos os contactos e alterar esse atributo, ou senão ficaríamos com informações incoerentes!

A solução para esta questão passa por estabelecer uma *relação* entre as tabelas da base de dados.[2] Para criar relações entre dados, cada registo de uma tabela deve ter uma *chave*, que pode depois ser utilizada noutra tabela para o referir. Uma chave é um qualquer atributo que permita identificar univocamente o registo. As chaves podem ser de qualquer tipo, mas são tipicamente números inteiros, que até podem ser atribuídos de forma automática quando o registo é criado.

Para estabelecer a relação no nosso exemplo, a tabela das empresas passaria a ser:

Table: companies

id	name	address	vatnumber
1	MaxiPlano	Aveiro	123123123123
2	Luis Manuel & filhos	Águeda	54534343435
3	ProDesign	Porto	54534343435

Enquanto que a tabela de contactos ficaria:

Table: contacts

id	firstname	middlename	lastname	email	phone	company_id
1	João	Manuel	Fonseca	jmf@gmail.com	912345654	3
2	Pedro	Albuquerque	Silva	pedro23@gmail.com	932454349	2
3	Maria	Carreira	Dinis	mariadi@ua.pt	234958673	1
4	Catarina	Alexandra	Rodrigo	calexro@sapo.pt	963343386	1

Em ambas as tabelas, a coluna `id` é uma chave: serve para identificar de forma única cada registo. Na tabela `contacts`, o campo `company_id` contém a chave da empresa a que pertence cada pessoa. Neste caso o João pertence à empresa ProDesign, enquanto a Maria e a Catarina pertencem ambas à empresa MaxiPlano.

Assim, a informação entre contactos e empresas encontra-se relacionada. A relação é estabelecida pelo campo `company_id` da tabela contactos. A este tipo de campo chama-se uma *chave estrangeira* por conter valores que são chaves (ditas *chaves primárias*) noutra tabela.

Os comandos `SELECT` podem ser construídos de forma a permitir pesquisar informação relacionada distribuída por diversas tabelas. Por exemplo, poderíamos listar todos os contactos da empresa MaxiPlano com o seguinte comando:

[2] É esta característica que está na origem do termo Base de Dados Relacional.

```
SELECT contacts.*
    FROM contacts,companies
    WHERE contacts.company_id = companies.id
    AND companies.name = "MaxiPlano"
```

Analisando o comando verifica-se que ele lista todos os campos dos registos da tabela **contacts** que possuam no atributo **company_id** dessa tabela um valor igual ao do atributo **id** da tabela **companies** sempre que **name** nessa mesma tabela seja igual a MaxiPlano.

Exercício 18.7

Volte a criar uma tabela de empresas com uma nova coluna chamada **id**. Especifique o tipo como sendo **INTEGER PRIMARY KEY AUTOINCREMENT**. Isto irá criar uma coluna que armazena um contador inteiro, único e incrementado automaticamente quando novas linhas são inseridas. Recomenda-se que se crie uma base de dados diferente da anterior.

Exercício 18.8

Aplique o mesmo processo à tabela de contactos mas neste caso adicione também uma nova coluna no final chamada **company_id**. Esta coluna serve para indicar a que empresa pertence um dado contacto, pelo que deve atualizar a base de dados de forma a que os contactos pertençam a uma empresa.

Exercício 18.9

Construa instruções que permitam listar todas os contactos das empresas de Aveiro.

18.2 Acesso Programático

Na secção anterior vimos os comandos básicos que permitem criar, aceder e modificar uma base de dados relacional. Nesta secção veremos como fazer o mesmo dentro de um programa.

Como em geral as bases de dados podem residir num sistema servidor, distinto do sistema que executa o programa cliente, o procedimento a seguir envolve três fases:

1. estabelecer a ligação à base de dados;
2. fazer as consultas e/ou alterações aos dados;
3. terminar a ligação.

Estas fases também são seguidas mesmo no caso da base de dados residir num ficheiro local. Em *Python* o acesso a bases de dados do tipo *sqlite* é fornecido pelo módulo **sqlite3**, que tem de ser importado. Para instalar o módulo, pode recorrer ao gestor de pacotes do seu sistema, ou ao comando **pip3** de acordo com as permissões que possuir.

```
apt-get install sqlite3
```

ou:

```
pip3 install --user pysqlite
```

O exemplo seguinte mostra o esqueleto de um programa que abre um ficheiro fornecido no primeiro argumento (p.ex, **database.db**).

```
import sqlite3 as sql
import sys

def  main(argv):
    db = sql.connect(argv[1])    # estabelecer ligação à BD

    ...                          # realizar operações sobre a BD

    db.close()                   # terminar ligação

main(sys.argv)
```

O método **connect** estabelece a ligação à base de dados e devolve um objeto que representa essa ligação. É através desse objeto que serão realizadas as operações na base de dados e deve-se depois terminar a ligação. Se o argumento do método **connect** for igual a **":memory:"**, a base de dados é criada em memória, sendo apagada no final do programa.

Isto é útil caso de pretenda armazenar informação apenas durante a execução do programa, não sendo ela válida numa futura execução.

Exercício 18.10

Construa um programa como indicado anteriormente e verifique que pode aceder à base de dados que criou anteriormente.

Tal como anteriormente, as operações sobre a base de dados são codificadas através de comandos SQL. As respostas podem depois ser obtidas sob o formato de linhas da tabela. O exemplo seguinte demonstra como é possível obter todos os registos da tabela de contactos.

```
import sqlite3 as sql
import sys

def  main(argv):
    db = sql.connect(argv[1])

    result = db.execute("SELECT * FROM contacts")
    rows = result.fetchall()
    for row in rows:
        print(row)

    db.close()

main(sys.argv)
```

Neste exemplo o ciclo percorre os registos devolvidos pelo comando **`SELECT`** e, em cada iteração, a variável **`row`** recebe um tuplo com os atributos de um desses registos. Também é possível obter os resultados um de cada vez, o que por vezes é obrigatório caso o seu tamanho seja muito grande.[3] Para isso podemos utilizar o método **`fetchone()`**:

```
    ...
    result = db.execute("SELECT * FROM contacts")
    while True:
        row = result.fetchone()
        if not row:
                break
        print(row)
    ...
```

[3] As bases de dados possuem sempre um limite (por exemplo, 4MB) para o tamanho de cada resultado.

Ou pode-se simplesmente tratar o resultado como um iterador:

```
...
result = db.execute("SELECT * FROM contacts")
for row in result:
    print(row)
...
```

Exercício 18.11

Considere os métodos anteriores e implemente um programa que imprima os nomes próprios de todos os contactos e indique quantos são, como no exemplo abaixo.

```
Catarina
João
Maria
Pedro
4 contactos
```

Frequentemente é necessário realizar pesquisas com argumentos variáveis. Por exemplo, podemos querer pesquisar todos os contactos cujo email pertença a um domínio indicado pelo utilizador. Assim, se o utilizador introduzir `%@gmail.com`, serão encontrados todos os contactos com email naquele servidor.

Uma solução ingénua seria construir uma *String* com a estrutura do comando pretendido:[4]

```
...
domain = raw_input("Domínio de email? ")
result = db.execute("SELECT * FROM contacts WHERE email LIKE '%s'" % domain)
# ATENÇÃO: ESTA SOLUÇÃO DEVE SER EVITADA!
...
```

Esta abordagem, embora frequente, **deve ser absolutamente evitada**! Caso contrário, correm-se sérios riscos de segurança. Por exemplo, se o utilizador introduzir

```
%@gmail.com'; DELETE FROM contacts --
```

[4] O operador `LIKE` permite pesquisar valores com um certo padrão, que pode incluir partes desconhecidas.

o comando SQL construído seria

```
SELECT * FROM contacts WHERE email LIKE '%@gmail.com'; DELETE FROM contacts --'
```

o que inclui duas instruções: **SELECT** e **DELETE**. Efectivamente o resultado é que todos os dados seriam apagados.

A solução correcta é a seguinte.

```
    ...
    domain = raw_input("Domínio de email? ")
    result = db.execute("SELECT * FROM contacts WHERE email LIKE ?", (domain,) )
    # ESTA É A SOLUÇÃO CORRETA!
    ...
```

Note que se utiliza um ponto de interrogação para indicar explicitamente onde deve ser colocado o parâmetro variável e o seu valor é passado separadamente. Desta forma é o próprio módulo que constrói internamente o comando SQL e assim é impossível injetar comandos adicionais. Esta abordagem tem o nome de *Prepared Statements* e deve ser seguida independentemente da linguagem de programação.

Exercício 18.12

Construa um programa que permita pesquisar contactos por qualquer um dos nomes. Para isto pode utilizar o operador **OR** para conjugar diferentes condições de pesquisa. Por exemplo: **...WHERE firstname LIKE ? OR middlename LIKE ? ....**

Exercício 18.13

Construa um programa que aceite um argumento, usando-o para localizar uma pessoa através das partes do seu nome, imprimindo a que empresa pertence.

18.3 Para Aprofundar

Exercício 18.14

Obtenha a base de dados *Chinook* acedendo ao endereço `https://github.com/lerocha/chinook-database` e crie um programa que permita pesquisar por albums de música. O esquema da base de dados pode ser encontrado em `https://github.com/lerocha/chinook-database/wiki/Chinook-Schema`.

Exercício 18.15

Obtenha a base de dados *Chinook* e crie uma aplicação que demonstre quais os 10 clientes que efectuaram o maior volume de compras.

Exercício 18.16

Relembre um exercício anterior em que se capturava a taxa de ocupação de CPU. Replique o exercício, mas registando os valores numa base de dados relacional.

Exercício 18.17

Implemente um programa que, tendo em consideração a base de dados criada no exercício anterior, imprima o valor médio de ocupação de processador entre duas datas.

Glossário

CPU Central Processing Unit

CSV Comma Separated Values

SQL Structured Query Language

Referências

[1] Y. Shafranovich, *Common Format and MIME Type for Comma-Separated Values (CSV) Files*, RFC 4180 (Informational), Internet Engineering Task Force, out. de 2005.