Laboratórios de Informática 2014/2015

DETI - Universidade de Aveiro

Docentes
João Paulo Barraca <jpbarraca@ua.pt>
Diogo Gomes <dgomes@ua.pt>
João Manuel Rodrigues <jmr@ua.pt>
Mário Antunes <mario.antunes@ua.pt>

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 uma peça fundamental para as aplicações atuais, pois agem como repositórios onde é possível armazenar e obter informação. Pode-se considerar que o mesmo pode ser feito utilizando ficheiros comuns, o que é verdade. Ou seja, é perfeitamente possível armazenar uma listagem de clientes num ficheiro de texto, que depois é lido por uma aplicação. No entanto isto nem sempre se faz [1] pois as bases de dados, em particular as relacionais, possuem características muito interessantes relacionadas com a manipulação de informação, que facilitam a sua utilização face a outros métodos.

Considere-se 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].

[1] Na realidade, raramente se faz em aplicações modernas!

```
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
```

Considerando que as bases de dados de clientes (e outras), podem conter milhares ou milhões de entradas, sendo necessário obter contactos específicos ou mesmo alterar o conteúdo, utilizar um simples ficheiro de texto rapidamente se torna um problema. Desde modo uma base de dados organiza informação de forma tabular (em tabelas) e permite que se pesquise/altere campos individuais. Em particular, estes sistemas suportam uma linguagem, denominada Structured Query Language (SQL) que permite efectuar operações muito detalhadas sobre a estrutura da informação ou os seus dados. Durante os próximos exercícios será explorada como a linguagem SQL pode ser utilizada para manipular bases de dados.

Assim, considerando uma tabela com os dados acima descritos, um servidor de base de dados necessita apenas que se identifique cada coluna com um identificador e um tipo de dados. O resultado de uma tabela com dados seria o apresentado de seguida:

first_name	**middle_name**	**last_name**	**email**	**phone**
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

Uma forma simples de criar esta base de dados será utilizar a ferramenta **`sqlite3`**, que permite criar bases de dados no sistema de ficheiros usando o formato *SQLite***sqlite3** Poderia-se utilizar em alternativa um servidor de bases de dados, o que tipicamente se utiliza em ambientes de produção. Do ponto de vista de utilização e criação de tabelas as diferenças são mínimas. Esta base de dados é muito comum sendo o sistema utilizado na plataforma *Android*.

Para criar uma base de dados é necessário executar a ferramenta **`sqlite3`** com a seguinte sintaxe[2]:

```
$ sqlite3 database.db
```

O resultado deverá ser um 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

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

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 tipo de dados a armazenar. São suportados vários tipos de dados, sendo os seguintes os mais relevantes para este trabalho:

- **TEXT**: Texto
- **INTEGER**: Números Inteiros
- **REAL**: Números reais
- **BLOB**: Um qualquer conteúdo (ex, uma imagem)

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**.

Depois da tabela existir, é agora necessário que se insiram dados. Isto faz uso do comando **INSERT INTO** no formato:

```
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 todos os comandos necessários para que se possam inserir os dados dos clientes. Mais uma vez, construa primeiro os comandos num ficheiro de texto e depois copie-os para a ferramenta **sqlite3**.

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

```
SELECT * FROM contacts;
```

Sendo que o resultado deverá ser o seguinte:

```
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 first_name="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 column_name 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

Uma aplicação terá várias tabelas, cada uma com um tipo de informação. Neste exemplo, existiriam tabelas com empresas parceiras, poderiam existir tabelas com produtos,

promoções, taxas de impostos, vendas realizadas, etc.... Tudo quando é necessário para armazenar toda a informação. Considerando o exemplo das empresas clientes, às quais pertencem os clientes anteriormente listados, poderia-se ter uma tabela como a seguinte:

name	**address**	**vat_number**
MaxiPlano	Aveiro	123123123123
Luis 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. Em particular porque uma empresa pode conter várias pessoas. Uma solução seria a de armazenar o nome da empresa junto de cada contacto. Mas então e o caso actual que a empresa possui contactos próprios e morada própria? Além disso estaria-se a replicar informação. Caso uma empresa mudasse a morada, teria de se pesquisar todos os contactos e alterar a morada a empresa em todos.

A solução para esta questão passa por considerar que a informação numa base de dados está relacionada e daí o termo Base de Dados Relacional. Para criar relações entre dados, considera-se assim que existem chaves que identificam cada linha, que podem ser depois utilizadas noutras tabelas. Pode considerar que são números inteiros.

No caso das empresas a tabela teria o seguinte formato:

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

Enquanto a tabela de contactos teria o seguinte formato:

id	**first_name**	**middle_name**	**last_name**	**email**	**phone**	**comp_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

A coluna **id** serve em ambos os casos para identificar de forma única uma dada entrada. No caso da tabela **contacts**, o campo **comp_id** pode ser utilizado para indicar a que empresa pertence uma pessoa específica. Neste caso o João Pertence à empresa ProDesign, enquanto a Catarina pertence à empresa MaxiPlano.

Os comandos **SELECT** podem ser construídos de forma a permitir pesquisar esta informação. Por exemplo, poderiam-se listar todos os contactos da empresa MaxiPano com o seguinte comando:

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

Analisando o comando verifica-se que ele lista todos as entradas da tabela **contacts** que correspondam possuam um valor na coluna **contacts.comp_id** igual ao valor **companies.id** sempre que **companies.name** seja igual a MaxiPlano. Neste caso, a informação entre contactos e empresas encontra-se relacionada. À coluna **id** dá-se o nome de chave primária, enquanto que a coluna **comp_id** denomina-se por chave estrangeira (por referir a uma outra tabela).

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 automaticamente incrementado a 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 **igualmente** uma nova coluna no final chamada **comp_id**. Esta coluna serve para indicar a que empresa pertence um dado contacto.

18.2 Acesso Programático

Descritos os comandos básicos que permitem acesso a uma base de dados relacional, falta descrever como é possível aceder à informação a partir de aplicações desenvolvidas. De uma forma geral considera-se que tem de existir um módulo que permita aceder à base de dados, podendo a base de dados encontrar-se ou não no mesmo sistema que executa o código. Por este motivo, existe a noção de ligação à base de dados, consulta dos seus dados, seguida de uma terminação da ligação. Mesmo no caso de se pretender aceder a uma base de dados num ficheiro local.

Em *Python* o acesso a bases de dados do tipo *sqlite* é fornecido pelo módulo **sqlite3** que necessita de ser importado. Este módulo permite estabelecer uma ligação à base de dados, devolvendo um objeto que representa essa ligação. Para instalar o módulo, pode recorrer ao gestor de pacotes do seu sistema, ou ao comando **pip** de acordo com as permissões que possuir.

```
$ apt-get install python-sqlite
```

ou:

```
$ pip install --user sqlite3
```

O exemplo seguinte considera o caso 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])

    # realizar operações sobre a db

    db.close()

main(sys.argv)
```

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 18.9

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

O acesso à informação da base de dados é realizado da forma indicada anteriormente, isto é, através de comandos SQL. As respostas podem depois ser obtidas sobre o formato de linhas da tabela. O exemplo seguinte demonstra como é possível obter todas as linhas da tabela:

```
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 caso, cada **row** é uma linha da tabela **contacts**, sendo que o objecto **row** possui a estrutura de um tuplo. É assim possível obter cada valor de forma individual como se de uma lista se tratasse. Também é possível obter os resultados um de cada vez, o que por vezes é obrigatório caso o seu tamanho seja muito grande[3]. Neste caso é possível utilizar o método **fetchone()**:

```
...
    while True:
        row = result.fetchone()
        if not row:
                break
```

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

```
        print row
...
```

ou pode-se tratar o resultado como um iterador:

```
...
    for row in result:
        print row
...
```

Exercício 18.10

Considere os métodos anteriors e implemente um programa que imprima a informação seguinte:

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

Frequentemente é necessário realizar pesquisas com argumentos variáveis. Por exemplo, poderia ser pretendido que se pesquisassem todas as linhas com endereços de correio **`@gmail.com`**. Neste sentido é necessário construir comandos de SQL que argumentos adicionais.

Uma solução seria a de construir uma *String* com a estrutura pretendida[4]:

```
...
   domain = '%@gmail.com'
   db = sql.execute('SELECT * FROM contacts WHERE email LIKE \'%s\'' % domain)
...
```

Embora esta seja a aproximação frequentemente seguida, **ela deve ser de todo evitada!**. Se o conteúdo da variável **`domain`** for dinâmico e introduzido pelo utilizador, poderiam existir problemas caso o utilizador especificasse **`"%@gmail.com'; DELETE FROM contacts -"`**. Neste caso, o resultado da *String* que compõe o comando seria **`SELECT`**

[4] A instrução **`LIKE`** significa semelhante e permite pesquisar valores que não são exactamente iguais.

`* FROM contacts WHERE email LIKE "%@gmail.com"; DELETE FROM contacts -'`, o que possui duas instruções: **SELECT** e **DELETE**. Efectivamente o resultado é que todos os dados seriam apagados.

A solução correcta é a de indicar explicitamente as partes variáveis do comando, sendo estes compostos pelo módulo de acesso à base de dados. Assim, embora o conteúdo a enviar seja o mesmo, nunca existe a possibilidade de injectar comandos adicionais.

O comando em causa deve assim ser construído da seguinte forma:

```
...
    domain = '%gmail.com'
    db = sql.execute('SELECT * FROM contacts WHERE email LIKE ?', (domain,) )
...
```

De notar que se utiliza um ponto de interrogação para indicar onde deve ser colocado o parâmetro, e que a *String* não é construída manualmente. Esta aproximação tem o nome de *Prepared Statements* e deve ser seguida em qualquer linguagem de programação.

Exercício 18.11

Construa um programa que permita pesquisar contactos por qualquer um dos nomes. Para isto pode utilizar o operador **OR** sendo possível conjugar diferentes condições de pesquisa. Por exemplo: `...WHERE first_name LIKE ? OR middle_name LIKE ? ....`

Exercício 18.12

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 18.13

Obtenha a base de dados *Chinook* acedendo ao endereço `http://chinookdatabase.codeplex.com/downloads/get/557773` e crie um programa que permita pesquisar por albums de música. O esquema da base de dados pode ser encontrado em `http://chinookdatabase.codeplex.com/wikipage?title=Chinook_Schema`.

Exercício 18.14

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

Exercício 18.15

Relembre um exercício anterior em que se capturava a taxa de ocupação de CPU. Replique o exercício mas capturando os valores para uma base de dados relacional.

Exercício 18.16

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.