TÉCNICO LISBOA

Projeto de Bases de Dados
Parte 3

Docentes
Leonardo Alexandre
João Aparício

Grupo 122 – Turno L20

Aluno	Horas de Trabalho
Carolina Coelho (99189)	25 (33.3%)
Gonçalo Nunes (99229)	25 (33.3%)
João Fonseca (95749)	25 (33.3%)

1 Base de Dados

Na implementação da nossa Base de Dados, considerámos que as cadeias de caracteres tinham todas, no máximo, 50 caracteres, à exceção da descrição do produto, que foi limitada a 255 caracteres.

Para o código **ean**, **num_serie** e **tin** optámos por utilizar o tipo de dados *integer*, por uma questão de otimização de espaço.

2 Restrições de Integridade

(RI-1) Uma Categoria não pode estar contida em si própria

Nesta restrição de integridade, considerámos que uma categoria não poder estar contida em si própria implica que, quando inserida uma entrada na relação **tem_outra**, deve existir garantia de que os nomes da **super categoria** e da **categoria simpes** sejam diferentes.

Uma vez que esta restrição de integridade consiste apenas em verificar que dois atributos inseridos numa relação são diferentes, esta poderia ser implementada com uma instrução "*check*" na criação da base de dados. No entanto, por uma questão de consistência com as soluções utilizadas nas restrições seguintes, optámos por utilizar um *trigger*.

(RI-4) O número de unidades repostas num Evento de Reposição não pode exceder o número de unidades especificado no planograma

Para implementarmos esta restrição de integridade, utilizámos um *trigger*. A solução consistiu em: procurar o **planograma** correspondente ao **evento de reposição** (o **planograma** que continha os mesmos **ean**, **nro**, **num_serie** e **fabricante**), obter o número de unidades especificado no referido **planograma** e comparar o número obtido com o número de unidades contido no **evento de reposição**.

(RI-5) Um Produto só pode ser reposto numa Prateleira que apresente (pelo menos) uma das Categorias desse Produto

Na implementação desta restrição considerámos que, para a **prateleira** apresentar a categoria do **produto**, tem obrigatoriamente de apresentar a **categoria** diretamente associada ao **produto** na relação **tem_categoria**, não podendo o **produto** ser reposto se a **categoria** exposta na **prateleira** for uma **super categoria** da categoria do **produto**.

Tendo o referido anteriormente em consideração, a nossa solução passou, mais uma vez, por um *trigger* com o seguinte comportamento: seleciona a categoria da **prateleira** referida no **planograma**, obtém todos os **eans** que pertençam a essa **categoria** e verifica se o ean do **planograma** está contido nessa lista de **eans**.

3 SQL

1. Qual o nome do retalhista (ou retalhistas) responsáveis pela reposição do maior número de categorias?

Nesta situação, considerámos que era pretendido o nome dos retalhistas com maior número de participações diretas na relação "responsavel_por", não sendo contabilizadas as sub categorias das possíveis super categorias da sua responsabilidade.

2. Qual o nome do ou dos retalhistas que são responsáveis por todas as categorias simpels?

Neste caso, considerámos que para um retalhista ser responsável por todas as categorias simples tinha que, para cada categoria simples existente na base de dados, existir uma entrada na tabela **responsavel_por** que associasse essa categoria ao retalhista.

3. Quais os produtos (ean) que nunca foram repostos?

Nesta *query*, selecionámos todos os produtos cujos eans não constavam em nenhum evento de reposição.

4 – Quais os produtos (ean) que foram repostos sempre pelo mesmo retalhista?

Nesta situação, selecionámos os produtos cujo ean, num evento de reposição estava sempre associado ao mesmo retalhista.

4 Vistas

```
CREATE view vendas(ean,cat,ano,trimestre,mes,dia_mes,
                dia_semana,distrito,concelho,unidades) AS
    SELECT t.ean,t.cat,
            EXTRACT(YEAR FROM e.instante),
            EXTRACT(QUARTER FROM e.instante),
            EXTRACT(MONTH FROM e.instante),
            EXTRACT(DAY FROM e.instante),
            EXTRACT(DOW FROM e.instante),
            p.distrito,p.concelho,e.unidades
    FROM ponto_de_retalho AS p
    INNER JOIN instalada_em AS i AS i.nome=p.nome
    INNER JOIN evento_reposicao AS e ON (e.num_serie=i.num_serie AND
                                        e.fabricante LIKE i.fabricante)
    INNER JOIN tem_categoria AS t ON (t.ean=e.ean)
```

Na solução apresentada optámos por utilizar **INNER JOIN** em vez de **NATURAL JOIN**, pois devido ao elevado número de tabelas e atributos envolvidos poderia existir um comportamento anómalo e a vista obtida não ser a pretendida.

5 Aplicação Web

Abaixo encontra-se uma descrição da arquitetura da Aplicação Web desenvolvida

5.1 Objetivos

O nosso objetivo era desenvolver uma aplicação funcional, simples e segura.

5.2 Ferramentas

Para esta aplicação, decidimos usar Python e HTML.

Com o Python, usámos a Driver Psycopg, para invocar comandos SQL de forma dinâmica, e Flask como Web Framework. Ao usarmos Flask, conseguimos inserir lógica e texto variável no HTML, através de delimitadores oferecidos pelos templates Jinja2 deste módulo.

5.3 Estrutura

5.3.1 Ficheiros

Todos os ficheiros desta aplicação encontram-se na pasta web/. Dentro desta temos:

- Ficheiro app.cgi - Interface entre o Servidor Web e a Aplicação, escrito em Python;
- Pasta templates/ - Conjunto dos ficheiros HTML usados e acedidos diretamente pela Framework Flask;
- Pasta queries/ - Conjunto de ficheiros .txt com algumas das *queries* usadas;
- Ficheiro aux_scripts.py - Conjunto de funções necessárias para o bom funcionamento da App.

5.3.2 Funcionamento

A página inicial, renderizada através do ficheiro index.html, apresenta 6 opções: **Inserir Categoria**, **Remover Categoria**, **Inserir Retalhista**, **Remover Retalhista**, **Ver Eventos de Reposição de uma IVM por Categoria**, **Ver Sub-Categorias de uma Categoria**. Quando o utilizador seleciona uma delas, envia um pedido à aplicação para mudar de página.

Cada uma destas 6 páginas, que sucedem à página inicial, apresenta formulários para o utilizador submeter certas informações. Para recolher estes inputs, recorremos às *tags* <**input**>, tendo utilizado **type="text"** para inputs textuais e numéricos e **type="checkbox"** para perguntas sim/não, que facilitam a análise do input.

Todas as páginas que sucedem à página inicial, excepto **Inserir Retalhista**, apresentam informações sobre a Base de Dados, que ajudam o utilizador na sua submissão. Nestes casos, a aplicação não envia logo o ficheiro HTML respetivo. Primeiro, tenta estabelecer uma ligação com a Base de Dados, com recurso à biblioteca **psycopg2**. Se tiver sucesso, executa uma *query* através de **cursor**, disponibilizado novamente pela biblioteca **psycopg2**, que também guarda os resultados da *query*, de maneira a poderem ser iteráveis.

Estes resultados são passados ao ficheiro HTML, que, com os *templates* Jinja2 disponibilizados pelo Flask, consegue interpretá-los. Só aqui é que a página pedida é enviada ao utilizador.

Após o cliente ter submetido as informações que achou necessárias, é gerado um novo pedido. Todos os pedidos resultantes de submissões de dados resultam em procedimentos da aplicação semelhantes. Em todos estes pedidos, a aplicação tenta conectar-se, novamente, com a Base de Dados e executa uma *query* usando o input do cliente.

Contudo existe uma diferença, que nos permite separar estes pedidos em dois grupos:

- Pedidos de Inserção/Deleção (1);
- Pediddos de Seleção (2).

Enquanto que o resultado das *queries* dos pedidos de (2) são para mostrar ao utilizador, as dos de (1) não. Para além disso, estes casos também se diferenciam de (2), pois a página enviada ao cliente apenas contém a mensagem Sucesso ou Erro, enquanto que em (1) é apresentada uma página com os resultados da *query*.

É importante salientar que as duas páginas que apresentam dados ao utilizador têm comportamentos distintos. Enquanto a página que apresenta as reposições por **categoria** de uma **IVM** mostra uma tabela vazia, quando a **IVM** inserida não existe, a página que apresenta as sub-categorias de uma **categoria** informa o utilizador quando este escolhe uma **categoria simples** e informa o utilizador que não foi possível obter dados da Base de Dados, quando a **categoria** não existe.

Para garantir a atomicidade das *queries* com "*sub-queries*" usámos o *statement* START TRANSACTION, no início das *queries*.

Dependendo do sucesso ou erro da *query*, era necessário enviar COMMIT ou ROLLBACK à Base de Dados, respetivamente. Esta funcionalidade está salvaguardada pelo método dbConn.commit(), disponibilizado pela biblioteca **psycopg2**.

Sendo assim, quando a aplicação recebe um pedido de tipo (1), executa uma *query* com várias "*sub-queries*", começando com START TRANSACTION. Se

essa *query* for sucedida, executa-se COMMIT, caso contrário executa-se ROLLBACK.

Em termos de segurança, para evitar **SQL Injection**, entre outros, sempre que a aplicação executa *queries* com inputs do utilizador, estes são passados como segundo argumento do método cursor.execute(). Além disso, todos os formulários usam o método "POST" e nenhuma mensagem de erro da Base de Dados é mostrada ao utilizador.

Por fim, indicar que todas as páginas apresentadas ao utilizador contêm um *link* para a página inicial, à exceção desta.

Abaixo encontra-se um link para a Web App.

https://web2.ist.utl.pt/ist195749/app.cgi/

5.3.3 Relações entre os arquivos

Passamos agora a apresentar, mais em específico, as relações entre os arquivos desta Aplicação.

A página inicial é modelada pelo ficheiro index.html que apresenta links para outras páginas, tal como explicado anteriormente. Abaixo encontra-se a relação entre as opções e os ficheiros utilizados caso sejam selecionadas:

- **Inserir Categoria**: após executar uma *query*[1], envia a página getInsertCateg.html ao utilizador. Nesta página:
  - Ao submeter uma **Categoria Simples**, a aplicação executa uma *query*[1] e apresenta a página sucesso.html ou error.html, dependendo do resultado;
  - Ao submeter uma **Super Categoria**, a aplicação executa uma *query*[1] e apresenta a página success.html ou error.html, dependendo do resultado;
- **Remover Categoria**: após executar uma *query*[1], envia a página getRemoveCateg.html ao utilizador. Nesta página
  - Ao submeter uma **Categoria**, a aplicação executa a *query* queries/deleteCat.txt e apresenta a página success.html ou error.html, dependendo do resultado;
- **Inserir Retalhista**: envia a página getInsertRet.html ao utilizador. Nesta página
  - Ao submeter os dados de um novo **Retalhista**, a aplicação executa a *query* queries/insertRet.txt e apresenta a página success.html ou error.html, dependendo do resultado;
- **Remover Retalhista**: após executar uma *query*[1], envia a página getRemoveRet.html ao utilizador. Nesta página:

[1] Não é usados nenhum ficheiro da pasta queries/

  - Ao submeter o **TIN** de um **Retalhista**, a aplicação executa a *query* queries/remRet.txt e apresenta a página success.html ou error.html, dependendo do resultado;

- **Ver Eventos de Reposição de uma IVM por Categoria**: após executar a *query* queries/getHighReplIVM.txt, envia a página getInsertCateg.html ao utilizador. Nesta página:
  - Ao submeter o **ID** de um **IVM**, a aplicação executa a *query* queries/QC.txt e apresenta a página seeRepByCateg.html.

- **Ver Sub-Categorias de uma Categoria**: após executar uma *query*[1], envia a página getInsertCateg.html ao utilizador. Nesta página:
  - Ao submeter uma **Categoria**, a aplicação executa várias *queries* em Loop e apresenta a página seeCatSubCats.html.

NOTA: Todas as páginas apresentam a opção de regressar à página inicial, que é logo apresentada.

6 OLAP

Exercício 1

Na primeira consulta, apresentamos o número de vendas entre duas datas (no caso, uma vez que não eram fornecidas datas, considerámos entre os anos 2016 e 2022), por dia da semana, por concelho e no total, ou seja, fizemos uma contagem para cada atributo pretendido isoladamente (1). Com o objetivo descrito, foram utilizados os seguintes *grouping sets*:

GROUPING SETS ((dia_semana), (concelho), ())

Dia da Semana	Concelho	COUNT
0	null	3
1	null	4
null	Lisboa	4
null	Porto	3
null	null	7

Figure 1: OLAP 1

Exercício 2

Na segunda consulta, apresentamos o número de vendas num determinado distrito, por concelho, categoria, dia da semana e no total, ou seja, fizemos uma

contagem para cada tuplo (concelho,categoria,dia da semana) e outra para o total (2). Foram, assim, utilizados os seguintes *grouping sets*:

GROUPING SETS ((concelho, cat, dia_semana), ())

Dia da Semana	Concelho	Categoria	COUNT
0	Lisboa	Pão	3
0	Porto	Pão	4
1	Lisboa	Pão	4
1	Porto	Pão	3
null	null	null	7

Figure 2: OLAP 2

7 Índices

Exercício 1

```
CREATE INDEX resp_nome ON responsavel_por USING HASH(nome_cat);
```

Criou-se um índice *hash* sobre a coluna **nome_cat** da tabela **responsavel_por**, uma vez que a condição na cláusula WHERE associada a esta coluna tratava-se de uma igualdade (a categoria tinha de ser 'Frutos'), permitindo fazer, na maioria dos casos, um *index scan*, em vez de um scan sequencial, reduzindo drasticamente o tempo de execução da *query*.

```
CREATE INDEX rtin ON retalhista USING HASH(tin);
```

Criou-se um índice, usando uma *hash table*, sobre a coluna **tin** da tabela **retalhista** (apesar desta já ser a *primary key* da tabela), para optimizar o *join* das tabelas.

Para se testar e confirmar algumas das conclusões tiradas, correu-se o seguinte trecho de código:

```
EXPLAIN ANALYZE
SELECT DISTINCT r.nome
FROM retalhista as r,
responsavel_por as p
WHERE r.tin = p.tin and p.nome_cat = 'Frutos';
```

Destes, o único em que se viu uma melhoria imediata foi com o indíce **resp_nome**, tendo-se passado a realizar um *index scan*, em vez de um *sequential scan*. O outro índice não resultou numa melhoria imediatamente visível, provavelmente, devido apenas a optimizações escolhidas pelo *query execution plan*.

Exercício 2

```
CREATE INDEX pdescr ON produto(descr);
```

Como sabemos como começam as descrições dos produtos procurados **(p.descr like 'A%')**, usa-se um índice, implemetado com uma *B-tree*, para optimizar essa procura.

```
CREATE INDEX tnome ON tem_categoria(nome);
```

Esta *query* pode ainda ser optimizada ao criar-se um índice sobre a coluna nome, da tabela **tem_categoria**, que beneficie operações de agrupamento, para tal escolheu-se uma *B-tree*.

```
CREATE INDEX pcat ON produto USING HASH(cat);
```

Optou-se ainda pela criação de um terceiro índice (um *hash*), desta vez para a coluna **cat** da tabela **produto**, para facilitar o *join* entre as tabelas. Aqui já não fazia sentido usar-se uma *B-tree*, uma vez que não se tratava de um agrupamento, a natureza da comparação beneficiaria mais das propriedades de uma tabela de dispersão.

Tal como anteriormente, testou-se a *query* com e sem os índices, mas desta vez não se notaram diferenças no tempo de execução, nem nas operações realizadas. Isto está, provavelmente, relacionado com uma escolha de optimização do *query execution plan* e não com os índices em si.

```
EXPLAIN ANALYZE
SELECT t.nome, count(t.ean)
FROM produto AS p, tem_categoria AS t
WHERE p.cat = t.nome and p.descr LIKE 'A%'
GROUP BY t.nome;
```