

## **Projeto de BD - Parte 3**

24 de junho de 2022

| Nome                    | Número | Esforço Total |
|-------------------------|--------|---------------|
| Diogo Romão Cardoso     | 99209  | 15h (33.3%)   |
| Luís Humberto Fonseca   | 99266  | 15h (33.3%)   |
| Rafael Serra e Oliveira | 99311  | 15h (33.3%)   |

### **Grupo 9 — Lab 20**

Profs. João Aparício e Leonardo Alexandre

Bases de Dados

4º Período, 2º Semestre, Ano Letivo 2021/2022

Licenciatura em Engenharia Informática e de Computadores - Alameda

## 1. Base de Dados

Desenvolvemos instruções em SQL para criar o esquema de Base de Dados correspondente ao esquema relacional apresentado no enunciado e seguidamente populá-lo. Dividimo-las em dois ficheiros: **schema.sql** (criação de tabelas) e **populate.sql** (preenchimento de tabelas com dados diversos) de modo a segmentar em dois blocos lógicos e mais intuitivos os dois grupos de instruções.

Para além disso, esta divisão (vs. estar tudo no mesmo ficheiro) permite a definição e registo das Restrições de Integridade do domínio entre a definição das relações e a definição dos dados iniciais (isto é, podendo-se correr primeiro o **schema.sql**, seguidamente o **ICs.sql** e só depois o **populate.sql**), garantindo-se assim sempre a consistência dos dados, incluindo da população inicial. Se este não fosse o caso, os dados iniciais não seriam validados pelas Restrições de Integridade, sendo estas apenas registadas mais tarde.

Na definição do domínio, fizemos as seguintes interpretações (concretizadas no **schema.sql**) do modelo apresentado:

- O EAN de um Produto é composto necessariamente por 13 dígitos;
- Apenas se considerou o caso de Portugal, pelo que o TIN de um Retalhista é composto necessariamente por 9 dígitos representando um Número de Identificação Fiscal português.
- O número e a altura da Prateleira, o número de faces e de unidades do Planograma e o número de unidades do Evento de Reposição foram modelados como inteiros
- Exceto nos casos supramencionados, todos os outros campos foram modelados como cadeias de caracteres de tamanho variável até 255 caracteres, sendo este o tamanho convencional para campos deste tipo, visto não terem sido expressas quaisquer outras restrições.

Por seu lado, o **populate.sql** permite o preenchimento da Base de Dados de forma consistente, recorrendo a sintaxe de inserção multi-valor. Não é necessário a inserção de dados nas tabelas **categoria\_simples** nem **super\_categoria** devido ao mecanismo de promoção/demoção descrito abaixo, sendo estas automaticamente preenchidas aquando da inserção em **categoria** e **tem\_outra**.

De modo a carregar todos os ficheiros corretamente, a seguinte ordem deve ser respeitada:

- **schema.sql**
- **ICs.sql**
- **populate.sql**
- **view.sql**
- **analytics.sql**

O ficheiro **queries.sql** pode ser corrido a qualquer ponto depois do **populate.sql**.

## 2. Restrições de Integridade

Foram implementadas as Restrições de Integridade descritas no enunciado sob a forma de *Stored Procedures* e *Triggers* no ficheiro **ICs.sql**. As RI-1, RI-4 e RI-5 são concretizadas, respetivamente pelos *before triggers* **trigger\_cat\_owns\_self**, **trigger\_oversupply** e **trigger\_presentable**, que, respetivamente, invocam os *stored procedures* **chk\_cat\_owns\_self**, **chk\_oversupply** e **chk\_presentable** antes de uma inserção ou alteração na tabela **tem\_outra** (no caso da RI-1) e na tabela **evento\_reposicao** (no caso da RI-4 e da RI-5).

Apesar de a RI-1 poder ser facilmente, e discutivelmente de forma mais intuitiva, implementada através de uma *Check Constraint* na definição da tabela **tem\_outra** no ficheiro **schema.sql**, sob a forma **CHECK (super\_categoria <> categoria)**, foi decidido fazê-lo no ficheiro **ICs.sql** de modo a cumprir a especificação do enunciado.

Para além das implementações das Restrições de Integridade do enunciado, o ficheiro **ICs.sql** contém, numa secção à parte, pares de *Stored Procedures* e respetivos *Triggers* que promovem a manutenção da consistência de dados na Base de Dados, de acordo com a nossa interpretação do domínio:

- O atributo **cat** do Produto corresponde à sua Categoria principal, sendo esta uma Categoria do Produto de qualquer modo, portanto é obrigatório que exista uma relação **tem\_categoria** entre o Produto e a sua Categoria principal, i.e., **tem\_categoria(produto.ean, produto.cat)**.
- Todas as Categorias são, quando são adicionadas, Categorias Simples, sendo apenas promovidas a Super Categorias quando um filho lhes for adicionado (i.e., quando for criada uma nova relação **tem\_outra**). Analogamente, quando uma Super Categoria deixa de ter filhos, é automaticamente despromovida a Categoria Simples.
- Quando uma Categoria é eliminada, são também eliminados todos os Produtos que a tenham como Categoria principal, todas as relações de hierarquia **tem\_outra** que a envolvam, todas as responsabilidades de retalhistas sobre ela e todas as prateleiras a ela relacionadas.
- Quando um Produto é eliminado, são também eliminados todos os Planogramas sobre ele e todas as relações **tem\_categoria** que o envolvam.
- Quando uma Prateleira é eliminada, também são eliminados todos os Planogramas sobre ela.
- Quando um Retalhista é eliminado, também são eliminadas as suas responsabilidades e os eventos a si associados.

### 3. SQL

Para a segunda interrogação explícita no enunciado, dado a cláusula `DIVIDE` não existir em PostgreSQL, foi utilizada a abordagem da dupla negação de modo a obter os mesmos efeitos.

### 4. Vistas

Para o campo **cat** da vista **Vendas**, foi utilizado o atributo **cat** do produto, ou seja, a categoria principal do produto.

### 5. Protótipo de Aplicação Web

É possível encontrar um protótipo da aplicação web em <http://doppler.dcard.pt>. Na página inicial encontra-se um menu para consultar as diversas secções do mesmo. Através da página das Categorias é possível **inserir e remover categorias e sub-categorias** e adicionalmente **listar todas as sub-categorias de uma super-categoria, a todos os níveis de profundidade**. A relação super-categoria <-> sub-categoria deve ser indicada ao ser criada a sub-categoria, sendo impossível à posteriori adicionar uma super-categoria a uma categoria já criada (é apenas possível adicionar sub-categorias a outras categorias). Como explicado na secção 2, qualquer categoria simples se pode tornar numa super-categoria, tendo sido implementado um trigger na Base de Dados que permite a sua promoção quando é adicionada uma categoria nova e indicada que essa dada categoria simples é a sua super-categoria. O mesmo acontece quando todas as sub-categorias de uma super-categoria são removidas, havendo uma despromoção da super-categoria para categoria simples. Através da página das IVMs é possível **listar todos os eventos de reposição de uma IVM, apresentando o número de unidades repostas por categoria de produto**. Finalmente, na página dos retalhistas, é possível **inserir e remover um retalhista, com todos os seus produtos**. Foram adicionadas funcionalidades extra: listagem de responsabilidades de um dado retalhista, a adição de responsabilidades e a eliminação de responsabilidades. Estas funcionalidades podem ser encontradas ao navegar para a página dos Retalhistas e depois clicar em "Ver Responsabilidades" para um dado retalhista. De forma a mitigar error handling, em todos os momentos de inserção de dados é feita uma filtragem ao input na frontend que não permite a submissão de dados irregulares quando é impreterível estes serem inseridos pelo utilizador e apresentado um *drop-menu* com opções pré-definidas caso contrário. Ao serem apagadas ou inseridas entradas da Base de Dados, por ordem do utilizador, todas as dependências são tratadas na Base de Dados, através de *triggers*, mantendo a aplicação o mais simples possível.

### 6. Consultas OLAP

Para a análise do número total de artigos vendidos foram utilizados mecanismos `GROUPING SETS` e `CUBE` para obter os resultados especificados no enunciado, seguindo-se a convenção de que células vazias representam "ALL". Para a primeira *query*, foi utilizado, para fins de exemplificação, o intervalo de datas entre 2020-01-01 e 2021-12-31. Do mesmo modo, para a segunda *query*, foi utilizado o distrito de Lisboa.

## 7. Índices

### 7.1.

O **SELECT DISTINCT** beneficia de um índice para *name* em *retalhista*.

No entanto, o PostgreSQL já cria este índice devido à **UNIQUE** constraint da coluna.

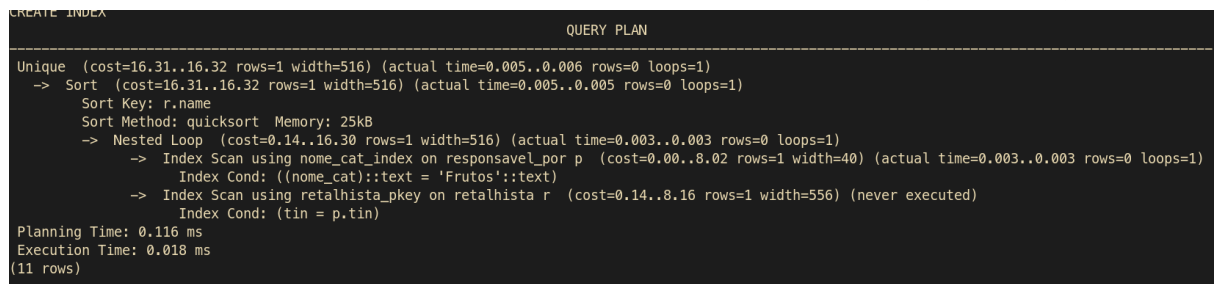
Na comparação **R.tin = P.tin**, **R.tin** é uma *primary key* pelo que não é necessário criar um índice.

**P.tin** é uma *foreign key*, pelo que criamos um índice do mesmo tipo que o usado por **R.tin**, **B-TREE**.

**P.nome\_cat = 'Frutos'** é uma comparação simples para a qual um índice do tipo **HASH** é o mais apropriado.

```
CREATE INDEX tin_index ON responsavel_por USING BTREE(tin);
```

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



```

CREATE INDEX                                QUERY PLAN
-----
Unique  (cost=16.31..16.32 rows=1 width=516) (actual time=0.005..0.006 rows=0 loops=1)
-> Sort  (cost=16.31..16.32 rows=1 width=516) (actual time=0.005..0.005 rows=0 loops=1)
    Sort Key: r.name
    Sort Method: quicksort  Memory: 25kB
-> Nested Loop  (cost=0.14..16.30 rows=1 width=516) (actual time=0.003..0.003 rows=0 loops=1)
    -> Index Scan using nome_cat_index on responsavel_por p  (cost=0.00..8.02 rows=1 width=40) (actual time=0.003..0.003 rows=0 loops=1)
        Index Cond: ((nome_cat)::text = 'Frutos'::text)
    -> Index Scan using retalhista_pkey on retalhista r  (cost=0.14..8.16 rows=1 width=556) (never executed)
        Index Cond: (tin = p.tin)
Planning Time: 0.116 ms
Execution Time: 0.018 ms
(11 rows)

```

Usando **EXPLAIN ANALYZE** é possível confirmar que os índices são usados pelo *planner* como previsto.

### 7.2.

O **GROUP BY T.nome** beneficia de um índice em *tem\_categoria* para *nome*.

No entanto, o PostgreSQL já cria um índice para (*ean*, *nome*) porque se trata da *primary key*.

O *planner* é capaz de usar este índice, pelo que não se justifica criar outro.

Na comparação **P.cat = T.nome**, **T.nome** já tem um índice utilizável.

Criamos para **P.cat** um índice do mesmo tipo que o usado por **T.nome**, **B-TREE**.

Na comparação **P.descr LIKE 'A%'**, criamos um índice do tipo **B-TREE** para *descr* em *produto*.

Índices do tipo **B-TREE** são capazes de agilizar operações de *pattern matching*.

```
CREATE INDEX cat_index ON produto USING BTREE(cat);
```

```
CREATE INDEX descr_index ON produto USING BTREE(descr);
```

```
QUERY PLAN
-----
GroupAggregate (cost=0.28..25.36 rows=1 width=524) (actual time=0.006..0.007 rows=0 loops=1)
  Group Key: t.nome
    -> Nested Loop (cost=0.28..25.34 rows=1 width=572) (actual time=0.006..0.006 rows=0 loops=1)
      -> Index Scan using cat_index on produto p (cost=0.13..12.20 rows=1 width=516) (actual time=0.006..0.006 rows=0 loops=1)
        Filter: ((descr)::text ~ 'A% '::text)
        Rows Removed by Filter: 4
      -> Index Only Scan using tem_categoria_pkey on tem_categoria t (cost=0.14..13.13 rows=1 width=572) (never executed)
        Index Cond: (nome = (p.cat)::text)
        Heap Fetches: 0
  Planning Time: 0.090 ms
  Execution Time: 0.018 ms
(11 rows)
```

Usando EXPLAIN ANALYZE é possível confirmar que os índices são usados pelo *planner* como previsto.