# Projeto de BD – Parte 3

## Professor Flávio Martins

## Grupo 201 – Turno L18

| Aluno | Esforço (horas) |
|---|---|
| Bernardo Prata (99184) | 15 horas (65%) |
| Duarte Gonçalves (99141) | 9 horas (35%) |
| André Matos (92420) | 0 horas (0%) |

Neste relatório apresentamos o nosso projeto da forma mais ampla e resumida possível. Explicamos em detalhe os motivos das implementações e as dúvidas geradas ao longo do processo.

# Instruções para criar esquema da BD

***create table categoria***
```
  (nome varchar(80)         not null unique,
   constraint pk_categoria primary key(nome));
```

***create table categoria_simples***
```
  (nome varchar(80)         not null unique,
   constraint pk_categoria_simples primary key(nome),
   constraint pk_categoria foreign key(nome) references categoria(nome));
```

***create table super_categoria***
```
  (nome varchar(80)         not null unique,
   constraint pk_super_categoria primary key(nome),
   constraint pk_categoria foreign key(nome) references categoria(nome));
```

***create table tem_outra***
```
  (super_categoria          varchar(80)         not null,
   categoria        varchar(80)         not null unique,
   constraint pk_tem_outra primary key(categoria),
   constraint fk_tem_outra_super foreign key(super_categoria) references super_categoria(nome),
   constraint fk_tem_outra_categoria foreign key(categoria) references categoria(nome));
```

***create table produto***
```
  (ean varchar(80)          not null unique,
   cat varchar(80) not null ,
   descr  varchar(80)        not null ,
   constraint pk_ean primary key(ean),
   constraint fk_produto_categoria foreign key(cat) references categoria(nome));
```

***create table tem_categoria***
```
  (ean varchar(80) not null,
   nome             varchar(80) not null,
   constraint pk_tem_categoria primary key(nome,ean),
   constraint fk_tem_categoria_produto foreign key(ean) references produto(ean),
   constraint fk_tem_categoria_categoria foreign key(nome) references categoria(nome));
```

***create table ivm***
```
  (  num_serie                integer    not null,
   fabricante       varchar(80) not null,
   constraint pk_borrower primary key(num_serie,fabricante));
```

***create table ponto_de_retalho***
```
  (   nome_retalho                      varchar(80)         not null unique,
   distrito         varchar(80) not null,
   concelho         varchar(80) not null,
   constraint pk_ponto_de_retalho primary key(nome_retalho));
```

***create table instalada_em***
```
  (  num_serie                integer    not null,
   fabricante       varchar(80) not null,
   local_nome       varchar(80) not null,
   constraint pk_instalada_em primary key(num_serie,fabricante),
   constraint fk_instalada_em_ivm foreign key(num_serie,fabricante) references ivm(num_serie,fabricante),
   constraint fk_instalada_em_local foreign key(local_nome) references ponto_de_retalho(nome_retalho));
```

***create table prateleira***
```
  (  nro   integer not null,
   num_serie                integer    not null,
   fabricante       varchar(80) not null,
   altura           numeric(3,2)   not null,
   nome varchar(80)         not null ,
   constraint pk_prateleira primary key(nro,num_serie,fabricante),
   constraint fk_ivm foreign key(num_serie,fabricante) references ivm(num_serie,fabricante),
   constraint fk_nome foreign key(nome) references categoria(nome));
```

***create table planograma***
```
  (ean varchar(80)            not null ,
   nro     integer not null,
   num_serie                  integer    not null,
   fabricante        varchar(80) not null,
   faces  integer   not null ,
   unidades                   integer   not null,
   loc               integer  not null,
   constraint pk_planograma primary key(ean,nro,num_serie,fabricante),
   constraint fk_planograma_prateleira foreign key(nro,num_serie,fabricante) references
prateleira(nro,num_serie,fabricante),
   constraint fk_planograma_produto foreign key(ean) references produto(ean));
```

***create table retalhista***
```
  ( tin     varchar(80)         not null unique ,
   retalhista_name           varchar(80)         not null unique,
   constraint pk_retalhista primary key(tin));
```

***create table responsavel_por***
```
  ( nome_cat       varchar(80)         not null  ,
   tin      varchar(80)         not null  ,
   num_serie                  integer    not null,
   fabricante        varchar(80) not null,
   constraint pk_responsavel_por primary key(num_serie,fabricante),
   constraint fk_responsavel_por_ivm foreign key(num_serie,fabricante) references ivm(num_serie,fabricante),
   constraint fk_responsavel_por_retalhista foreign key(tin) references retalhista(tin),
   constraint fk_responsavel_por_categoria foreign key(nome_cat) references categoria(nome));
```

***create table evento_reposicao***
```
  (  ean varchar(80)          not null ,
     nro   integer not null,
   num_serie                  integer    not null,
   fabricante        varchar(80) not null,
   instante timestamp not null,
   unidades                   integer   not null,
   tin      varchar(80)         not null  ,
   constraint pk_reposicao primary key(ean,nro,num_serie,fabricante,instante),
   constraint fk_reposicao_planograma foreign key(ean,nro,num_serie,fabricante) references
planograma(ean,nro,num_serie,fabricante),
   constraint fk_reposicao_retalhista foreign key(tin) references retalhista(tin));
```

# Restrições de Integridade

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

```
create or replace function não_contida_proc() returns Trigger as
$$
BEGIN
   IF New.categoria = New.super_categoria THEN
      RAISE EXCEPTION 'Uma Categoria não pode estar contida em si própria.';
   END IF;
   RETURN New;
END;
$$ LANGUAGE plpgsql;

create  trigger nao_contida_trigger
before update or insert on tem_outra
for each row execute procedure não_contida_proc();
```

# Restrições de Integridade

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

```
create or replace function max_unidades_respostas_proc() returns Trigger as
$$
DECLARE max_units INTEGER:=0;
BEGIN
        SELECT unidades INTO max_units FROM planograma p WHERE p.ean = NEW.ean AND
        p.nro=NEW.nro AND p.num_serie= NEW.num_serie AND p.fabricante=NEW.fabricante;
        IF New.unidades > max_units THEN
                RAISE EXCEPTION 'O número de unidades repostas num Evento de Reposição não pode
                exceder o número de unidades especificado no Planograma';
        END IF;
   RETURN New;
END;
$$ LANGUAGE plpgsql;

create  trigger max_unidades_respostas_trigger
before update or insert on evento_reposicao
for each row execute procedure max_unidades_respostas_proc();
```

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

```
create or replace function prateleira_produto_categoria_proc() returns Trigger as
$$
BEGIN
   IF NOT EXISTS (
        (SELECT DISTINCT p.nome FROM prateleira p WHERE p.nro = NEW.nro AND p.num_serie=
NEW.num_serie AND p.fabricante=NEW.fabricante)
        INTERSECT
        (SELECT DISTINCT t.nome FROM tem_categoria t WHERE t.ean = NEW.ean)
   ) THEN
      RAISE EXCEPTION 'Um Produto só pode ser reposto numa Prateleira que apresente (pelo menos) uma
das Categorias desse produto';
   END IF;
   RETURN New;
END;
$$ LANGUAGE plpgsql;

create  trigger prateleira_produto_categoria_trigger
before update or insert on evento_reposicao
for each row execute procedure prateleira_produto_categoria_proc();
```

# SQL

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

```
SELECT distinct tin
FROM responsavel_por
GROUP BY tin
HAVING count(*) >= ALL(
   SELECT count(*)
   FROM (SELECT DISTINCT tin,nome_cat FROM responsavel_por) AS a
   GROUP BY  tin
);
```

- ***Qual o nome do ou dos retalhistas que são responsáveis por todas as categorias simples?***

```
SELECT DISTINCT T.tin
FROM responsavel_por AS t
WHERE NOT EXISTS (
        SELECT c.nome
        FROM categoria_simples AS c
        EXCEPT
        SELECT t2.nome_cat
        FROM responsavel_por AS t2
        WHERE t2.tin = T.tin
);
```

- ***Quais os produtos (ean) que nunca foram repostos?***

```
SELECT ean
FROM produto
WHERE ean NOT IN (
   SELECT ean
   FROM evento_reposicao
)
```

- ***Quais os produtos (ean) que foram repostos sempre pelo mesmo retalhista?***

```
SELECT distinct ean
FROM evento_reposicao AS p
WHERE tin = ALL(
   SELECT tin
   FROM evento_reposicao AS e
   WHERE e.ean = p.ean
)
```

# View

```
CREATE VIEW Vendas
AS
SELECT e.ean as ean,
   c.cat as cat,
   EXTRACT(YEAR FROM e.instante) as ano,
   EXTRACT(QUARTER FROM e.instante) as trimestre,
   EXTRACT(MONTH FROM e.instante) as mes,
   EXTRACT(DAY FROM e.instante) as dia_mes,
   EXTRACT(DOW FROM e.instante) as dia_semana,
   p.distrito as distrito,p.concelho as concelho,ne.unidades as unidades
FROM evento_reposicao e  JOIN produto c
        ON e.ean = c.ean
      JOIN instalada_em i
        ON e.num_serie = i.num_serie AND e.fabricante = i.fabricante
      JOIN ponto_de_retalho p
        ON  i.local_nome = p.nome_retalho
```

Dado que cada produto pode ter várias categorias, mas cada evento de reposição corresponde a apenas uma única venda, decidimos assumir como categoria da venda a categoria associada na relação produto. Caso tivéssemos feito JOIN com tem_categoria, iriamos ter para um único produto várias rows, que significaram várias vendas para um único evento de reposição, o que não é desejado.

Não é  MATERIALIZED VIEW, dado que não queremos guardar a view em memória física, mas queremos que a view esteja sempre atualizada ( ou seja, que a query seja executada todas as vezes que a view é chamada) com os novos eventos de reposição.

Nota: por eg,SELECT p.distrito as distrito -> o uso de AS apenas deixa mais explicito qual é o nome  da chave na view.

# APP

A nossa aplicação encontra-se divida em três páginas principais:

https://web2.ist.utl.pt/ist199184/test.cgi/categorias (web/categoria.html)
https://web2.ist.utl.pt/ist199184/test.cgi/retalhistas (web/retalhista)
https://web2.ist.utl.pt/ist199184/test.cgi/ivms (web/ivm.html)

O design de todas as páginas é muito simples e intuitivo. Não há qualquer botão no design da app que permita navegar facilmente entre estas páginas principais.
Cada cursor do psycopg2 implementa uma transação, e por isso o Rollback está omisso dentro da implementação do cursor. Desse modo, sempre que ocorre um erro numa query ocorre rollback, caso contrário commit, assegurando-se assim a atomicidade das transações. A nossa solução também preza pela segurança.

### *a) Inserir e remover categorias e sub-categorias; d) Listar todas as sub-categorias*

https://web2.ist.utl.pt/ist199184/test.cgi/categorias

Nestas páginas temos dois formulários:
- para se inserir uma nova categoria na BD. Ao clicar no submit é chamada a função insert_categoria, que vai alterar a base de dados e inserir uma nova categoria e uma nova categoria simples.
- outro para inserir uma subcategoria e a sua super-categoria. É necessário que ambas as categorias existam e caso a super-categoria seja uma categoria simples até então, é feita essa alteração. Tudo isto é executado em insert_sub_categoria().
A nossa solução trata os dados recebidos do input do utilizador, prevenindo ataques por SQL INJECTION.

Na tabela, que mostra todas as categorias presentes na BD temos duas ações: ***Listar Sub-Categorias*** e ***Remover***.

- Ao clicar em ***Listar Sub-Categorias*** é direcionado para uma nova página https://web2.ist.utl.pt/ist199184/test.cgi/sub_categorias?categoria_name=Alimentos onde será apresentada uma tabela com todas as subcategorias da categoria da linha selecionada, neste caso Alimentos. Os dados de input do formulário são passados por url à função sub_categorias() em test.cgi, que vai comunicar com a BD selecionando a função sub_categorias(cat_name), que está escrita em plpgsql em ICs.sql e que calcula as todas subcategorias de uma dada categoria.Depois é renderizado o código html desta página que se encontra em web/subcategoria.html.

- Ao clicar ***Remover*** , a categoria presente vai ser removida assim como todas as suas subcategorias, e qualquer outro tuplo de outra relação que tenha a categoria atual como FK (produtos, responsável_por, tem_outra etc). É executado o método remover_categoria() em test.cgi, que por sua vez vai comunicar com a BD numa única transação e apagar todas as relações dependentes de todas as categorias e subcategorias a eliminar. Após a transação ter terminado, o utilizador é redirecionado para a página atual.

### *b) Inserir e remover um retalhista, com todas suas as responsabilidades de reposições de produtos*

https://web2.ist.utl.pt/ist199184/test.cgi/retalhistas

Tal como no caso anterior, temos um formulário bastante simples com dois dados de entrada:TIN e nome. Ao clicar no botão Submit é redirecionado para uma página test.cgi/insert_retalhista, que vai executar a respetiva função insert_retalhista() em test.cgi. Os parâmetros de input são passados no URL.
Para remover um retalhista, basta na tabela que apresenta todos os retalhistas clicar no botão ***Remover***. De forma a manter a boa gestão da base de dados, são também eliminados todos os tuplos de outras relações que dependam da chave do retalhista eliminado.

Tudo muito simples, questionando e alterando a base de dados com queries básicas (Insert e Delete).

***c) Listar todos os eventos de reposição de uma IVM, apresentando o número de unidades repostas por categoria de produto;***
https://web2.ist.utl.pt/ist199184/test.cgi/ivms

Apresentamos uma tabela, que é obtida a partir de uma query SELECT * FROM ivms; Ao lado de cada IVM temos uma ação ***Listar Eventos***.

Ao clicar em ***Listar Eventos*** é redirecionado para uma nova página http://localhost:8000/cgi-bin/test.cgi/ivm_events?num_Serie=1&fabricante=LIDL onde se encontram todos os eventos de reposição da IVM selecionada numa tabela, e no fim desta uma nova tabela com o número de unidades repostas por categoria de produto.
Assumimos que o numero de unidades repostas por categoria é para todas as categorias às quais pertence o produto em tem_categoria. Por eg, se um produto pertence a duas categorias:X e Y, e é reposto em dez unidades então o numero de unidades repostas da categoria X aumenta em 10 unidades assim como o da categoria Y. Poder-se-ia ter assumido apenas a categoria definida na relação produto.

# Consultas OLAP

Consultas SQL que permitam analisar o número total de artigos vendidos:

1. num dado período (i.e. entre duas datas), ***por*** dia da semana, ***por*** concelho e no total
   Dado a forma como está enunciado o problema **por*,por*** assumimos que o que é pedido é partições em conjuntos separados, isto é, por dia da semana e por concelho e no total.

```
SELECT v.dia_semana, v.concelho, SUM(v.unidades) FROM Vendas AS v
WHERE  make_date(CAST(v.ano AS integer), CAST(v.mes AS integer),CAST(v.dia_mes AS integer)) BETWEEN '2015-01-01' AND '2022-07-01'
GROUP BY GROUPING SETS ((v.dia_semana),(v.concelho),());
```

2. num dado distrito (i.e. “Lisboa”), **por** concelho, categoria, dia da semana e no total

   2.1) Tendo em conta o que assumimos na alínea anterior, neste caso não seria em conjuntos separados.
   Logo assumimos que (concelho,categoria,dia da semana) seria um conjunto, dessa forma:

```
SELECT v.concelho, v.cat, v.dia_semana, SUM(v.unidades) as soma_unidades FROM Vendas as v
WHERE v.distrito = 'Lisboa'
GROUP BY GROUPING SETS ((v.concelho,v.cat,v.dia_semana),());
```

   2.2) Por outro lado, se o pedido foi o número total de artigos vendidos por todos os subconjuntos possíveis entre concelho, categoria e dia da semana, a query seria:

```
SELECT v.concelho, v.cat, v.dia_semana, SUM(v.unidades) as soma_unidades FROM Vendas as v
WHERE v.distrito = 'Lisboa'
GROUP BY CUBE(v.concelho,v.cat,v.dia_semana);
```

Nota: Dado que a 2.1 nos parece a mais plausível pela forma como o problema está enunciado, **por concelho, categoria, dia da semana** e no total, assumimos essa opção como resposta em caso de dúvida.

# Índices

```
SELECT DISTINCT R.nome
FROM retalhista R, responsavel_por P
WHERE R.tin = P.tin and P.nome_cat = 'Frutos'
```

- ***retalhista:***
  - tin é PK de retalhista, logo por omissão retalhista tem um indice em Btree(tin).
  - Como o SELECT DISTINCT não tem a sua performance melhorada com um índice *****, e por omissão já temos o index para tin então não existe nenhum índice novo para Retalhista que ajude a otimizar esta query.

- ***responsavel_por:***
  - tin não é PK de responsavel_por, nem nome_cat é PK logo não têm índices associados.
  - Nesta query e com base nos dados, o nome_cat = 'Frutos' é mais seletivo, isto é, existe uma menor percentagem de dados que fazem parte desta restrição. Ora vejamos, a condição r.tin = p.tin teria que ser verificada para cada row de retalhista e responsavel_por, porém nome_cat = 'Frutos' filtra a tabela responsavel_por e consequentemente torna a verificação da condição do JOIN mais eficiente, dado que existem após a filtragem menos registos para comparar.
  - Por outro lado, recorrendo ao EXPLAIN ANALYZE, verificámos que na execução da query é filtrado primeiro pelo nome_cat e só depois é que verifica a condição r.tin = p.tin. O que nos pareceu comprovar o ponto anterior.

Assim, com base na informação que apresentamos acima, decidimos criar um índice em Btree sobre a combinação das chaves nome_cat e tin.Dado que nome_cat é a chave mais seletiva, o index é organizado inicialmente por nome_cat e só depois por tin.

***create index nome_cat_tin_index on responsavel_por(nome_cat,tin);***

Informação Adicional (NÃO AVALIAR)
***** informação confirmada num horário de atendimento por zoom com outro professor.A minha ideia inicial seria até que o select distinct fazia um sort e comparava chave a chave, e que dessa forma um indice Create index new_index on retalhista using hash(nome) ajudaria. Mas dado que a performance não é melhorada, excluímos essa hipótese.

---

```
SELECT T.nome, count(T.ean)
FROM produto P, tem_categoria T
WHERE P.cat = T.nome and P.descr like 'A%'
GROUP BY T.nome
```

- ***produto:***
  - Nem cat, nem descr são PK de produto, logo não têm índices associados.
  - Pegando no raciocínio anterior, a filtragem de produtos cujas descrições começam por A (começadas por, logo posso usar índice) é feita antes da verificação da condição p.cat = t.nome. Por outro lado, ter uma descrição começada por A é um fator comum a uma percentagem baixa de registos quando comparada com a verificação da condição que se aplica a todos nomes e cats ( se fosse verificada a condição primeiro, antes da filtragem). Assim, a condição da descrição parece-nos mais seletiva.
- ***tem_categoria:***
  - Nem nome, nem cat são PK de tem_categoria. (nome,ean) é PK.
  - É feita um agrupamento por T.nome, logo esta seria a chave principal para um possível indice para tem_categoria. Por outro lado, no comando select é usada uma função de agregação que conta os ean por nome, por isso ean é chave secundaria do índice.
  - No entanto, se (nome,ean) é PK já está criado automaticamente um índice para a combinação destas chaves.

Recorrendo a EXPLAIN ANALYZE, verificámos que na execução da query a ordem de procedimentos é a seguinte:

1) Sort e Agregação por t.nome
2) Filtragem pela descrição começada por A
3) Verificação da condição P.cat = T.nome
4) Contagem de T.ean

Assim, com base na informação que apresentamos acima, decidimos criar um indice em Btree sobre a combinação das chaves descr e cat na tabela produto:

***create index cat_do_produto_index on produto(descr,cat);***

Como ter (nome,ean) ou (ean,nome) é igual para PK de tem_categoria, assumimos no nosso caso que a chave primária é (nome,ean). E dessa forma, já é criado um indice em Btree com (nome,ean) o que já está de acordo com o nosso raciocínio. Temos primeiro o indice organizado por nome, que é a chave principal que usamos no GROUP BY, e depois por ean, que é a chave que vamos contar *para cada um dos nomes*. Logo o índice gerado para a PK de tem_categoria já é eficiente para esta query.