Sistema de Gestão de Dados

Engenharia e Ciência de Dados
2º Ano, 1º Sem

Alunos:
Alice Mangara
N.º 2020242411

Afonso Rodrigues
N.º2020242400

Núria Silva
N.º 2020242444

Departamento de Engenharia Informática

Faculdade de Ciências e Tecnologia da Universidade de Coimbra
FCTUC

Índice

1. Introdução

Este trabalho foi realizado no âmbito da Unidade Curricular de Sistema de Gestão de Dados (SGD) e teve como objetivo o desenvolvimento de uma API RESTful e um banco de dados relacional para servir como *backend* de uma loja virtual especializada em alimentos e equipamentos para animais de estimação. Foram implementados nove *endpoints* essenciais, mais 3 opcionais. Estes *endpoints* envolvem funcionalidades como gestão e análise estatística de dados de vendas.

O utilizador interage com o servidor web por meio de uma troca de solicitação/resposta REST, e, por sua vez, o servidor web interage com o servidor de banco de dados por meio de uma interface SQL.

A API foi testada usando o Postman.

2. Desenvolvimento

Para começar o projeto, desenvolveu-se um **modelo de entidade-relacionamento (ER)** que vai servir como base para nosso banco de dados. Neste foram identificadas todas as entidades relevantes, os seus atributos e as relações entre elas. Posteriormente, realizou-se a implementação da base de dados, a infraestrutura que vai armazenar todas as informações sobre produtos, clientes e transações. Criaram-se as tabelas com base nas entidades identificadas no modelo ER, especificando os campos, tipos de dados, chaves primárias e estrangeiras.

Com a base de dados estabelecida, desenvolveu-se a RESTful API, que serve como interface de comunicação entre os utilizadores e o banco de dados, e criaram-se *endpoints* que vão permitir aos utilizadores realizar todas as operações necessárias.
A partir do ficheiro csv fornecido pelo professor foram criados mais 6 ficheiros csv: cliente.csv, item.csv, category.csv, item_purchase.csv, purchase_csv, cart_csv.

Por último, usou-se o Postman que é o local de teste para a nossa API, ou seja, o local onde são feitas os *requests* associadas à API.

2.1. Modelo Entidade Relacionamento-ER

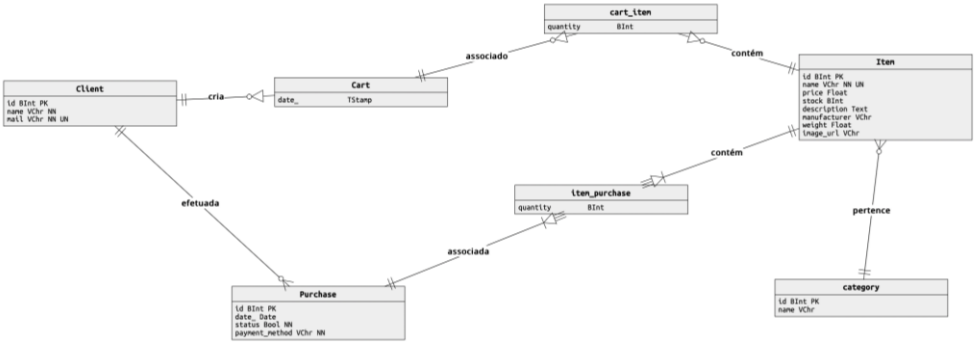

Figura 1-Conceptual Model

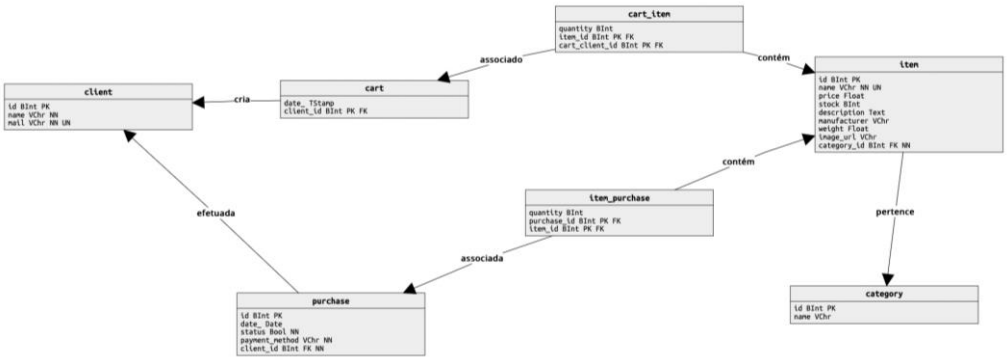

Figura 2-Physical Model

Na Figura 1, é apresentado o modelo de dados implementado. Foram criadas 6 tabelas *Client*, *Cart*, *Item*, *cart_item*, *Purchase*, *item_purchase*, *category*.
Foram feitas algumas alterações desde a apresentação intermédia, nomeadamente e mais evidente, a passagem de todos os nomes de tabelas e atributos para inglês de forma ao código se tornar mais legível, à exceção da data que passou a ser date_ uma vez que “date” é uma palavra reservada.
Foi também removida a ligação compra cesto e criada a ligação (dada pela entidade fraca item_purchase) bem como a criação da nova intermédia fraca cart_item, alterando as ligações respetivas para o produto de 0-n para 0-1, formando uma chave primária composta pelo id do cliente (cart_client_id) e pelo id do item (item_id).

A maioria das tabelas no esquema possuem um atributo id, que atua como chave primária, representada pela notação PK. Além disso, algumas tabelas incluem chaves estrangeiras, indicadas por FK, para estabelecer relações significativas entre elas. Este modelo foi desenvolvido com base nas necessidades específicas do sistema.

A entidade Client armazena informações sobre os clientes do sistema. Cada cliente é identificado exclusivamente pelo atributo id, chave primária. O campo name representa o nome do cliente, enquanto o campo mail armazena o endereço de e-mail único do cliente. Foi aplicada uma restrição de unicidade ao campo mail para garantir que cada cliente tenha um e apenas um endereço de e-mail associado. Um cliente pode realizar várias compras, mas cada compra é associada a um único cliente. Essa relação é estabelecida pela chave estrangeira client_id na tabela purchase, que referencia o campo id na tabela client.

A tabela purchase registra informações sobre as compras realizadas no sistema. Cada compra é identificada exclusivamente pelo atributo id. O campo date_ armazena a data da compra, enquanto status indica o estado da compra. A chave estrangeira client_id está vinculada à tabela client, assegurando que cada compra esteja associada a um cliente existente.

A tabela item contém detalhes sobre os itens disponíveis para compra. Cada item é identificado pelo atributo id, que serve como chave primária. O campo name é único o que garante a singularidade de cada item, enquanto category_id é uma chave estrangeira vinculada à tabela category, estabelecendo a categoria à qual o item pertence.

A tabela cart representa os carrinhos de compras dos clientes. Cada carrinho é identificado pelo atributo client_id, que serve como chave primária. O campo date_ armazena a data do carrinho. A chave estrangeira client_id está vinculada à tabela client, garantindo que cada carrinho esteja associado a um cliente existente. Um cliente pode ter apenas um carrinho de compras ativo por vez. Um carrinho de compras pode conter vários itens, e cada item pode estar presente em vários carrinhos

A tabela category define as categorias às quais os itens podem pertencer. Cada categoria é identificada exclusivamente pelo atributo id, que serve como chave primária. O campo name armazena o nome da categoria. Vários itens podem pertencer à mesma categoria, mas cada item está associado a apenas uma categoria.

A tabela item_purchase associa itens a compras, indicando a quantidade comprada. A chave primária é composta pelos atributos purchase_id e item_id. As chaves estrangeiras purchase_id e item_id estão vinculadas às tabelas purchase e item, respetivamente.

A tabela cart_item associa itens a carrinhos, indicando a quantidade. A chave primária é composta pelos atributos item_id e cart_client_id. As chaves estrangeiras item_id e cart_client_id estão vinculadas às tabelas item e cart, respetivamente.

2.1.1. Triggers

Inicialmente haviam sido dois triggers criados no pgadmin, porém, apercebemo-nos que para os triggers seriam específicos ao pgadmin e que teriam que ser importados de computador para computador, tendo portanto sido importados diretamente no código python.
Os triggers implementados foram então:

- descréscimo do stock na quantidade de compra realizada para o determinado produto;
- eliminação do cart_item para o client_id no qual a compra foi realizada

2.2. API Rest

A API que faz a ligação ao DB server (pgAdmin) foi editada para conter todos os *requests* necessários.
A função *db_connection* apresenta uma conexão direta com o PostgreSQL, fornecendo detalhes de acesso diretamente no código. No entanto, é crucial notar que essa prática, especialmente expor a senha diretamente no código, representa uma vulnerabilidade de segurança séria. Para uma implementação mais segura e de acordo

com as melhores práticas, deveriam usar-se variáveis de ambiente para armazenar essas informações.

Segue-se a explicação da implementação dos 12 *endpoints*.

2.2.1. Create Item (POST)

A primeira função desenvolvida foca-se em criar um produto para acrescentar aos já existentes na tabela *item*. Na nossa função, começámos por validar os payloads que serão introduzidos. No entanto, neste caso, uma vez que necessitamos de uma coluna ID, ao invés de nós o introduzirmos manualmente, a função foi desenvolvida de forma a ir buscar o id anterior, que no caso dos dados pré-definidos disponibilizados é o 20, e acrescenta 1 a este.

2.2.2. Update Item (PUT)

A função “update_item” foca-se em atualizar os atributos de um item dado um certo id. No nosso caso, não precisamos de fornecer todos os atributos para que a função seja atualizada, necessitando apenas no payload os atributos que queremos alterar. Por exemplo, no caso de alterarmos o id 19, podemos simplesmente colocar no Postman, no método PUT {"name": "Pedro"} e apenas o name será alterado.

2.2.3. Delete Item from Shopping Cart (DELETE)

A função “delete_item” recebe um determinado id e apaga-o da table cart_item, caso este item exista

2.2.4. Add Item to Shopping Cart (POST)

A função “add_item” ao contrário do método PUT anteriormente definido e explicado, necessita que o payload contenha todos os atributos do produto que serão inseridos na tabela Cart. Para que tal se verifique,

necessitamos de validar cada atributo, sendo estes 'item_id', 'quantity', 'cart_client_id'

2.2.5. Get Items List (GET)

A função “get_items_list” foca-se em devolver produtos existentes na tabela item. Por pré-definição, a função não devolve nem todos os produtos (estando pré-definida para mostrar 10) nem os devolve ordenados, porém a query foi para que com uma alteração consigamos fazer com que ambos os aconteçam, bem como separar unicamente para uma categoria.

Caso seja desejável organizar a lista de produtos por id ou nome, podemos alterar a requisição na web, do previamente pedido para *http://localhost:8080/proj/api/item?sort=id*

2.2.6. Get Items Details (GET)

A função “get_item_details” é parecida à anterior, focando-se não em mostrar a lista de todos os produtos, mas sim um produto em específico em detalhe. Para alterar a query do anterior basta adicionar o id produto que queremos analisar em detalhe no fim da requisição web, ficando, por ex.: *http://localhost:8080/proj/api/item/12*

2.2.7. Search Item (GET)

A função “search_item” pode ser compreendida como uma junção das funções 2.2.5 e 2.2.6, na medida que caso nada seja dado então devolve todos os produtos sem ordem e com o detalhe de todos os items.

Porém, se alterarmos o requisito, e definirmos ou o nome que queremos, ou a descrição de um certo item, então obtemos a descrição para tal item. O requisito passando este a ser *http://localhost:8080/proj/api/item/search?query=Item%20A*

2.2.8. Get top 3 sales per category (GET)

A função *get_top_sales_per_category* é responsável por calcular e retornar as 3 principais vendas por categoria.

A consulta SQL é executada para cada categoria (Food, toys ou accessories) onde é recuperado os nomes dos produtos e a soma das quantidades vendidas de uma categoria específica, isto é garantido através da linha de código *WHERE c.id=%s*. A consulta é agrupada por nome de produto e ordena em ordem decrescente o total de vendas*(total_sales)*. Para garantir que apenas 3 produtos sejam selecionados é usado o *LIMIT 3*.

O primeiro JOIN é realizado entre as tabelas *item_purchase e item* através dos IDs dos produtos (ou seja, apenas os produtos com ID presente na tabela produto compra são escolhidos) e o segundo é entre as tabelas item e category que conecta as informações do produto à categoria à qual pertence.

Por último, as tabelas item_purchase e purchase, associa a tabela de compras à tabela de produtos comprados.

2.2.9. Purchase Items (POST)

Nesta secção do código, o objetivo pretendido é a passagem de um ou mais items do carrinho e realizá-los numa compra para o cliente que as tinha colocado no carrinho.

O código divide-se essencialmente em três partes. Na validação de argumentos, verificamos se o item_id existe na table item e/ou se o stock desse item é nulo.

Temos também dois try_except importantes, nos quais o primeiro verifica e seliciona o/os produtos existentes no carrinho e remove na quantidade específica o stock do determinado produto na tabela *item*, e o segundo insere os items especificados no json body na tabela *purchase*. De forma a não haver problemas transacionais, o commit está incluído no segundo try de forma a que não haja diminuição de stock sem a compra ser efetivamente realizada.

Um exemplo de um request que poderá ser feito para devolver a passagem de um ou alguns produtos do cart é (cliente_id 790 foi criado por nós):

{"cart":[{"item_id":12,"quantity":2}],"client_id":790}

2.2.10. Get Clients With Filters

A função get_all_clients() retorna informações sobre os clientes, incluindo o ID, nome, e-mail, data da última compra e o nome do último produto comprado.

Em primeiro lugar, são combinadas as tabelas client com as linhas da tabela purchase onde o id do cliente na tabela cliente é igual ao client_id na tabela compra, para que seja possível relacionar um cliente com as informações da compra relacionada.

Depois, combina-se as linhas anteriores (que já contêm informações sobre os clientes e as suas compras) com as linhas da tabela item_purchase onde o purchase_id na tabela item_purchase é igual ao id da compra na tabela purchase. De seguida, são comparadas as linhas da tabela item onde o item_id na tabela item_purchase é igual ao id do produto na tabela item. Cada linha agora representa um cliente com informações da compra e detalhes do produto comprado, os detalhes do produto comprado são necessários uma vez que é preciso imprimir o nome do produto.

O resultado final é agrupado pelo id do cliente e são aplicadas funções de agregação MAX às colunas compra.data e produto.nome dentro de cada grupo formado pelo GROUP BY. Isso retorna a data da última compra e o nome do último produto comprado para cada cliente.

2.2.11. Add Client (POST)

Este código adiciona clientes à base de dados, validando os campos obrigatórios, verificando se os campos obrigatórios (id, nome, mail) estão presentes no payload. Se algum campo estiver ausente, retorna uma resposta de erro 400 (Bad Request). Este request será testado no Postman mais à frente.

2.2.12. Get Client Orders (GET)

Em resumo, essa consulta SQL retorna informações sobre os produtos associados a um pedido específico. O resultado da consulta é uma lista de tuplos, onde cada tuplo contém o produto_id, a quantidade comprada e o preço do produto.

É feita uma junção (JOIN) com a tabela item (abreviada como p) de forma a obter os detalhes sobre cada produto. O JOIN combina as informações do item_purchase com item. A condição de junção é pc.produto_id = p.id, garantindo que as linhas correspondam aos produtos certos. Os resultados são filtrados para garantir que se incluem apenas as linhas onde pc.compra_id é igual ao ID do pedido específico (order['order_id']).

3. Transactions, Concurrency Faults and Error

No que toca a transações, os problemas apareceriam unicamente quando se estaria a publicar dados a tempo real. No nosso caso, problemas de transações seriam unicamente evidenciados quando se adiciona um produto ao carrinho, ou quando o produto, ou produtos, são comprados. No caso da compra, um possível problema que teríamos de transação era se duas pessoas tentassem comprar o último elemento disponível em stock, contudo, no nosso projeto tentamos minimizar tal problema ao máximo.

4. Testes

Como referido anteriormente, os *requests* foram testadas no Postman.

GET REQUEST:

- Exemplo de resposta obtida pelo request mencionado no 2.2.7


```
{
    "results": [
        {
            "category_id": 2,
            "description": "Premium pet food",
            "id": 1,
            "image_url": "https://example.com/item-a.jpg",
            "manufacturer": "ABC Pet Foods",
            "name": "Item A",
            "price": 9.99,
            "stock": 1000,
            "weight": 2
        }
    ],
    "status": 200
}
```


POST REQUEST:


```
{
    "cart": [
        {
            "item_id": 12,
            "quantity": 2
        }
    ],
    "client_id": 790
}
```

5. Conclusão

Este trabalho permitiu adquirir conhecimentos teóricos e práticos sobre sistema de gestão de dados. A implementação das funcionalidades solicitadas permitiu consolidar conceitos fundamentais relacionados com a conceção de modelos de dados, utilização de sistemas de gestão de bases de dados e o design de uma arquitetura por meio de uma API REST.

Para um futuro trabalho, poder-se-ia incluir uma nova entidade com o objetivo de ser "Store Credit", no qual, ao ser realizada uma compra, verifica se o cliente em específico tem crédito em loja no qual este possa ser descontado, baixando o preço da compra.

Uma possível melhoria no trabalho que poderia ter sido feita era a criação de um trigger que apagasse o item_purchase dinamicamente para que a entidade não cresça infinitamente. Outra melhoria, seria a verificação do stock ao adicionar item ao carrinho e não só quando a compra é realizada.