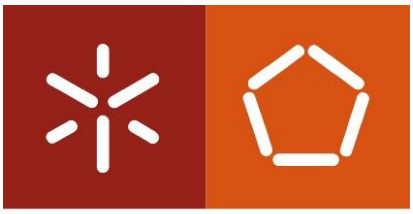

Universidade do Minho

Mestrado Integrado em Engenharia Informática

LI3 - Sistema de Gestão de Vendas

Grupo 12

Sofia Santos (A89615) Ana Filipa Pereira (A89589)
Carolina Santejo (A89500)

Ano Letivo 2019/2020



Conteúdo

Capítulo 1

Introdução e principais desafios

Este projeto consistiu no desenvolvimento de um Sistema de Gestão de Vendas (SGV) na linguagem de programação C. Sendo um projeto de programação em larga escala, foi necessário recorrer a princípios mais avançados de programação, como o uso de estruturas de dados eficientes para armazenar e consultar grandes quantidades de informação, garantindo sempre o encapsulamento dos dados. Todas as estruturas de dados que usámos neste trabalho ou foram criadas por nós ou pertencem à biblioteca GLib da GNOME, nomeadamente Hash Tables e Balanced Binary Trees, às quais nos iremos referir daqui em diante como tabelas de hash e árvores binárias, para efeitos de simplificação da linguagem.

O maior desafio, numa fase inicial, foi encontrar uma forma eficiente de armazenar os dados dos ficheiros, de forma a poder responder às queries de forma eficiente. Depois de termos definido todas as estruturas necessárias que constituem o SGV, o resto do trabalho foi relativamente fácil. Outro desafio que tivémos foi o de ter que apresentar listas com centenas ou milhares de elementos ao utilizador, e a solução que encontrámos para este problema foi definir uma estrutura que contém uma lista de strings e que é navegável por páginas, sendo que o número de elementos por página depende do tamanho de cada elemento, para que as páginas tenham sempre um tamanho semelhante.

Capítulo 2

Módulos e estruturas de dados

2.1 Clientes

Para armazenar todos os códigos de clientes usámos uma árvore binária, que a cada código de cliente (uma string) faz corresponder uma lista de 3 valores booleanos, que nos dizem se aquele cliente fez compras numa dada filial. A API associada a este módulo permite realizar operações de inserção, pesquisa, validação e alteração de dados associados a um dado cliente.

2.2 Produtos

Tal como o módulo dos clientes, o módulo dos produtos usa uma árvore binária, que a cada código de produto (uma string) faz corresponder uma lista de 3 valores booleanos, que nos dizem se aquele produto foi comprado numa dada filial. A API associada a este módulo permite realizar operações de inserção, pesquisa, validação e alteração de dados associados a um dado produto.

2.3 Faturas (Bills)

Para armazenar as faturas usámos uma tabela de hash, que associa um código de produto a uma estrutura de dados à qual chamámos Bill (Fatura). Uma fatura permite armazenar, para um dado produto, o lucro total obtido com a venda desse produto e o número total de vezes que esse produto foi vendido (total de registos de venda). Tanto o lucro total como a quantidade total de vendas de um produto estão separadas por filial, por mês e por modo (isto é, se o produto foi comprado em promoção ou não). Temos assim a nossa definição de uma fatura:

```
struct bill {
        double totalProfit[3][12][2];
        int totalSales[3][12][2];
};
```

A API associada a este módulo permite realizar operações de inserção, pesquisa e atualização de dados.

2.4 Filial (Branch)

Para armazenar as relações entre clientes e produtos por filial usámos uma estrutura de dados constituída por duas tabelas de hash, uma que faz corresponder cada cliente aos produtos que este comprou, e outra que faz corresponder cada produto aos clientes que o compraram. No SGV é usada uma lista de três elementos, onde cada elemento é uma cópia desta estrutura para armazenar estas relações para cada uma das três filiais.

A primeira tabela (clientsWhoBought) associa cada produto (chave) aos clientes que o compraram (valor). Esta lista de clientes é outra tabela de hash, que a cada cliente (chave) faz corresponder a informação relativa à compra do produto inicial por esse cliente (valor), nomeadamente a quantidade desse produto comprada por mês, separada por modo (normal ou em promoção).

A segunda tabela (productsBoughtBy) associa cada cliente (chave) aos produtos que este comprou (valor). Tal como na outra tabela, nesta a lista de produtos é também uma tabela de hash, que a cada produto (chave) faz corresponder a informação relativa à compra desse produto pelo cliente inicial (valor). Esta informação inclui a quantidade desse produto comprada por mês e o total de dinheiro gasto nesse produto pelo cliente dado.

Estas duas tabelas acabam por conter na sua maioria a mesma informação, mas esta duplicação de informação resulta numa enorme eficiência na execução das queries. Se apenas tivéssemos a tabela que associa cada produto aos clientes que o compraram, por exemplo, e quiséssemos uma lista de todos os produtos comprados por um dado cliente, teríamos que percorrer a tabela toda, visto que o cliente pode ter comprado qualquer produto, enquanto que usando duas tabelas podemos simplesmente consultar uma vez a tabela que associa cada cliente aos produtos que comprou e teríamos imediatamente a nossa resposta.

A API associada a este módulo permite realizar operações de inserção, pesquisa e atualização de dados, para além de operações específicas para ajudar a responder a certas queries, como preencher uma lista com os códigos de cliente que compraram um dado produto.

2.5 Informação dos ficheiros (FileInfo)

Este módulo armazena informação relativa aos ficheiros lidos aquando do carregamento do SGV, nomeadamente os caminhos dos ficheiros, o número total de linhas e o número de linhas válidas. A API deste módulo permite realizar operações de consulta e alteração de dados.

2.6 SGV

O nosso Sistema de Gestão de Vendas é composto pelas cinco estruturas mencionadas acima. A API associada ao SGV, definida em *interface.h*, permite preencher o SGV a partir de ficheiros e permite responder a todas as queries. Podemos então definir o nosso SGV da seguinte forma:

```
struct sgv {
        Products products;
        Clients clients;
        Bills bills;
        Branch branches[3];
        FileInfo fileInfo;
};
```

2.7 Venda (Sale)

Definimos um módulo auxiliar que nos permite armazenar uma venda, tal como consta no ficheiro das vendas. Esta estrutura permite-nos aceder à informação de cada venda de forma mais eficiente, aquando do preenchimento dos módulos das faturas e das filiais. A API associada a este módulo apenas inclui operações de criação, consulta e validação de uma venda, visto que este é um módulo temporário, que apenas é usado para auxiliar o carregamento de dados dos ficheiros no SGV.

2.8 Estruturas auxiliares (AuxStructs)

Para poder responder às queries de forma eficiente definimos três estruturas de dados para armazenar os resultados das queries, quando estes são demasiado complexos para armazenar num tipo de dados simples. Temos assim as estruturas StringArray, SalesProfit e ProductTable, defjnidas da seguinte forma:

```
struct stringArray {
        char **array;
        size_t N;
        int data;
};

struct salesProfit {
        int sales[3][2];
        double profit[3][2];
};

struct productTable {
        int totalQuant[3][12];
};
```

A API associada a este módulo permite realizar operações de inserção e consulta de dados nas três estruturas.

2.9 UI

O módulo UI, tal como o nome indica, trata da parte visual do programa, como mostrar ao utilizador os resultados das queries, os menus do programa, e os *input prompts*. A sua API contém *macros* para poder imprimir texto colorido no terminal.

2.10 Navegador (Navigator)

Este módulo permite navegar pela estrutura StringArray, mostrando a sua informação ao utilizador pelo módulo UI de forma organizada, por páginas. A navegação pelas outras estruturas devolvidas pelas queries fica ao encargo do próximo módulo, pois são simples o suficiente para não ser necessário recorrer a outro módulo para as navegar.

2.11 Queries

O módulo Queries é o responsável pelo fluxo das queries, ou seja, é o que trata de receber *input* do utilizador, chamar a função da query correspondente, fornecendo-lhe o *input* necessário, e enviar o *output* para o módulo UI, para poder ser apresentado ao utilizador. Para além das queries, este módulo tem ainda uma função que realiza um *benchmark*, medindo o tempo de execução de todas as queries, uma por uma.

2.12 Controlador (Controller)

Por último, o módulo Controlador é o módulo principal, é este que controla o fluxo do programa. É este módulo que diz ao módulo Queries que query o utilizador escolheu, e que manda libertar a memória alocada ao SGV ao sair do programa. Os módulos Queries e Navegador são "sub-módulos"deste módulo.

Capítulo 3

Estrutura do projeto

O nosso projeto segue a estrutura *Model View Controller* (MVC), estando por isso organizado em três camadas:

- A camada de dados (o modelo) é composta pelo SGV/Interface, que por sua vez é constituído pelos módulos referidos no capítulo anterior, Clientes, Produtos, Faturas, Filial e FileInfo. Para além destes, temos ainda as estruturas auxiliares definidas nos módulos AuxStructs e Venda.
- A camada de interação com o utilizador (a vista, ou apresentação) é composta unicamente pelo módulo UI.
- A camada de controlo do fluxo do programa (o controlador) é composta pelos módulos Controlador, Queries e Navegador. É esta camada que gere o funcionamento do programa, interagindo com as outras duas camadas para que este possa funcionar da forma pretendida. As outras duas camadas nunca interagem diretamente uma com a outra.

Nesta estrutura, o controlador é responsável por enviar pedidos ao modelo, como por exemplo responder a uma query. O modelo calcula essa resposta e envia-a ao controlador, que irá pedir à vista para a apresentar ao utilizador. Temos assim três camadas distintas que funcionam juntas para formar o nosso programa.

Capítulo 4

Complexidade das estruturas

Para as estruturas mais complexas, responsáveis por armazenar grandes quantidades de dados, decidimos usar tabelas de hash, com duas exceções. Usámos árvores binárias em vez de tabelas de hash para armazenar os catálogos de clientes e de produtos pois com árvores binárias conseguimos percorrer os seus elementos por ordem (neste caso alfabética), permitindo-nos assim devolver facilmente listas de clientes ou de produtos ordenadas alfabeticamente como resposta a algumas queries, que devem ter como resultado uma lista ordenada, sem ser preciso ordenar a lista.

Também tentámos reduzir ao máximo o número de estruturas diferentes usadas para guardar os resultados das queries. Acabámos por ter que definir apenas três.

A primeira (StringArray) contém, como o nome indica, um array de strings, com um comprimento N, e um valor auxiliar **data** que armazena informação pertinente para responder a algumas das queries. Por exemplo, na query 2, este valor contém a letra pela qual os produtos da lista devem começar, e na query 11 contém a quantidade de produtos que pretendemos encontrar, ou seja, o comprimento final que a lista deve ter. Através do módulo Navegador, somos capazes de percorrer esta estrutura por páginas.

A segunda (SalesProfit) contém duas listas, uma para armazenar o número total de vendas, e outra para armazenar o lucro total. Estes valores podem ser divididos por filial ou por modo de compra (em promoção ou não). Dividimos a estrutura em duas listas pois o total de vendas é um valor inteiro e o lucro total é um valor de vírgula flutuante, logo não podem estar armazenados na mesma lista.

A terceira e última estrutura (ProductTable) é responsável por armazenar a quantidade total de um produto que foi vendida, estando dividida por mês e por filial. Esta estrutura apenas consiste numa lista bidimensional (uma matriz, por outras palavras), e não era necessário ter criado esta estrutura, podíamos simplesmente ter trabalhado com a lista que esta contém, mas como em C uma lista multidimensional devolvida por uma função "decai"para um apontador para uma lista unidimensional, seria muito mais complicado obter os valores da nossa lista depois da query a devolver. Desta forma não temos esse problema.

Capítulo 5

Testes de desempenho

Para medir o tempo de execução do nosso programa usámos a biblioteca *time.h*. Fizemos várias medições, e podemos ver o resultado de uma dessas medições na tabela da figura a seguir.

BENCHMARK

query	1M vendas	3M vendas	5M vendas
1	3.924651s	13.686257s	21.535181s
2	0.001039s	0.001155s	0.001220s
3	0.000001s	0.000001s	0.000001s
4	0.011286s	0.011930s	0.012443s
5	0.001628s	0.001852s	0.002120s
6	0.009967s	0.011791s	0.012451s
7	0.000017s	0.000026s	0.000036s
8	0.042992s	0.044215s	0.044354s
9	0.000009s	0.000009s	0.000010s
10	0.000044s	0.000092s	0.000150s
11	0.428006s	0.904372s	1.313287s
12	0.001665s	0.000340s	0.000591s

Figura 5.1: Tempos de execução de cada query

Poderíamos ter calculado o tempo médio de várias medições, mas não sentimos a necessidade de o fazer, visto que os resultados que obtivemos diferiram muito pouco uns dos outros, a única diferença "considerável"registou-se na primeira query, mas nunca foi superior a 0,2s.

Decidimos não incluir a query 13 neste *benchmark* porque esta apenas vai buscar dados ao SGV, e o seu tempo de execução é sempre praticamente nulo.

Pela análise da tabela, podemos concluir o seguinte:

- todas as queries, exceto a query 1 e a query 11, têm um tempo de execução bastante baixo, nunca ultrapassando os 0,1 segundos. Sem contar com a query 1, a query 11 é a mais lenta, visto que tem que percorrer todas as filiais e ordenar todos os produtos vendidos em função da quantidade vendida, mas mesmo assim consegue fazê-lo num tempo aceitável.
- o aumento do número de vendas lidas leva a um aumento do tempo de execução de algumas queries, mais especificamente aquelas que interagem com os módulos de faturação e das filiais.
- este aumento faz-se sentir com mais força na query 1, onde a complexidade de tempo aparenta ser linear, ou próxima de linear. As queries 10 e 11 também apresentam algum aumento no tempo de execução, mas aqui a complexidade já é inferior a linear.
- em queries que apenas vão buscar informação aos catálogos de produtos e clientes, como as queries 2, 3 e 8, o número de vendas não parece afetar o seu tempo de execução, como seria de esperar. Pode-se atribuir a pequena variação registada a erros estatísticos. A query 5 apresenta um ligeiro aumento no tempo de execução, mas este aumento é causado pelo facto de que a lista que a query devolve ser maior, dado que com mais vendas há mais clientes a comprar em todas as filiais, logo haverá mais alocações de memória, que acabam por também ter um peso no tempo de execução.

Para além de testes de tempo de execução, também realizámos testes de memória, usando a ferramenta *Valgrind*. Concluímos que o único tipo de *memory leaks* presentes no nosso programa se devem às estruturas de dados do GLib utilizadas, e que são do tipo "still reachable", e portanto não são "graves"[1]. Para além disso, usando a ferramenta *memcheck*, verificámos que o nosso SGV ocupa um tamanho máximo na heap de cerca de 398,74 MB.

Capítulo 6

Conclusão

A nível geral, e tendo em conta o que foi explicado nos capítulos anteriores, podemos afirmar que temos um projeto bem conseguido. As estruturas de dados implementadas possuem um acesso eficiente à informação guardada, o que permite que os tempos de resposta das queries sejam bastante curtos. Mesmo a query 11, a que demora mais tempo a gerar uma resposta, consegue fazê-lo em menos de um segundo para um milhão de vendas, e para três milhões de vendas num computador moderno. Para além disso, é possível verificar que não existe uma grande perda de performance com o aumento do tamanho do ficheiro de vendas. Conseguimos com que o nosso programa não só fosse eficiente, como também que o seu desempenho se mantivesse aceitável em diferentes cenários, que era um dos principais desafios deste projeto.

Sentimos que as nossas estruturas e funções, apesar de funcionarem bem e serem eficiente, poderiam ser um pouco mais simples, da forma como estão atualmente podem parecer um bocado confusas para alguém que não tenha lido este relatório, o que não é ideal, especialmente no mundo profissional.

Apêndice A

Grafo de dependências

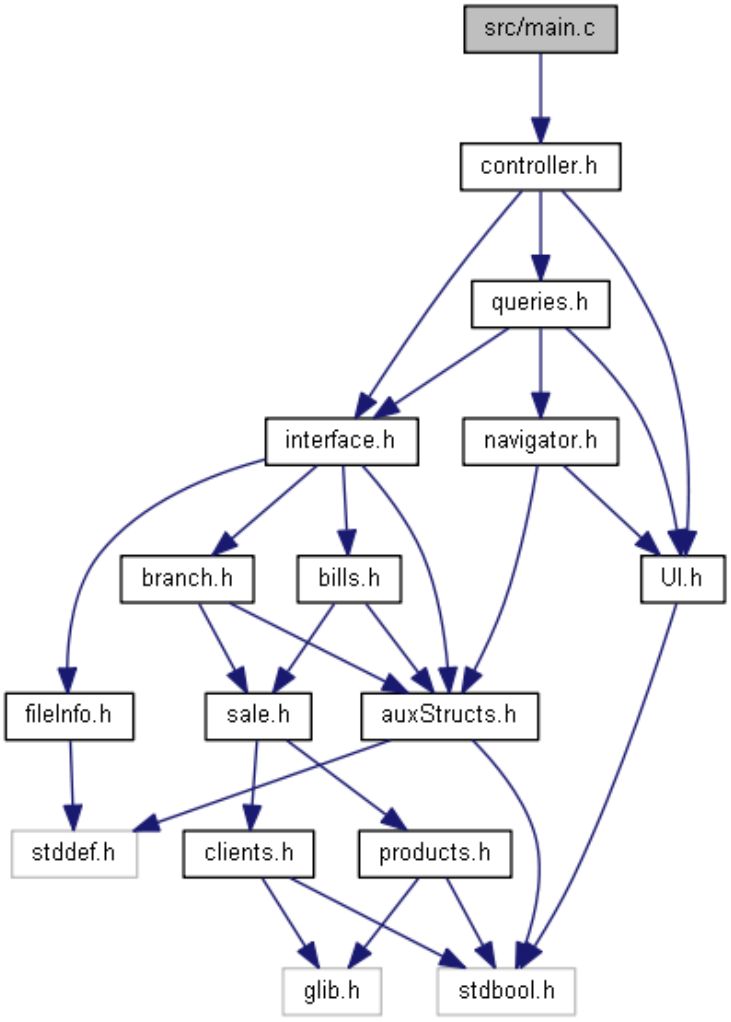

Figura A.1: Grafo de dependências do programa, gerado pelo *Doxygen*