Universidade do Minho

Departamento de Informática

Projeto de Laboratórios de Informática III
Grupo nº 31

Ricardo Gomes (a93785) Rui Fernandes (a89138)

Maio 2020

Resumo

O presente relatório descreve o projeto realizado no âmbito da disciplina de *Laboratórios de Informática III* (LI3), ao longo do segundo semestre do segundo ano do Mestrado Integrado em Engenharia Informática da Universidade do Minho.

Este projeto teve como principal objetivo o processamento e tratamento de grandes volumes de dados provenientes de três ficheiros de texto, e posterior desenvolvimento de um sistema de gestão de vendas modular capaz de armazenar informação de vendas e relacionar, de forma eficiente, produtos, clientes e vendas.

Conteúdo

1 Problema

1.1 Descrição do problema

São fornecidos três ficheiros de texto que contêm informação sobre as transacções de uma cadeia de distribuição:

- O ficheiro *Produtos.txt* contém cerca de 200.000 códigos de produto, sendo que cada linha deste ficheiro representa o código de um produto da de distribuição. Cada código é formado por duas letras maiúsculas seguidas de quatro dígitos (ex.: *HG1389*).
- O ficheiro *Clientes.txt* contém cerca de 20.000 códigos de clientes, sendo que cada linha deste ficheiro representa o código de um cliente da cadeia de distribuição. Cada código é formado por uma letra maiúscula seguida de quatro dígitos (ex.: *A2657*).
- O ficheiro *Vendas_ 1M.txt* contém 1.000.000 de registos de vendas, sendo que cada linha deste ficheiro representa uma venda efetuada numa das três filiais da cadeia de distribuição. Cada um destes registos é formado por um código de produto, preço unitário do produto comprado, número de unidades compradas, um caractere (*'N'* ou *'P'*) conforme tenha sido uma compra normal ou uma compra em promoção, o código do cliente que efetuou a compra, assim como o mês e a filial em que esta ocorreu (ex.: *VC1429 768.94 153 N G3562 11 2*).

Com este trabalho prático prende-se extrair a informação necessária dos vários ficheiros, de forma a responder, de forma eficiente, a um conjunto de *queries* que relacionam com o conteúdo dos mesmos. tendo especial atenção aos tempos de execução, assim como ao encapsulamento de dados e gestão eficiente de memória.

1.2 Conceção da solução

Uma vez que com este projeto se pretende desenvolver um sistema capaz de simular o funcionamento de uma cadeia de distribuição, este deve ser capaz de:

- Armazenar, após a devida validação, os códigos de produtos e clientes.
- Admitir a existência de várias filiais.
- Armazenar informação relativa à faturação global da cadeia de distribuição.
- Registar a relação entre um cliente e um produto e vice-versa.

Assim, foram criadas três classes principais: *Catálogo, Faturação* e *Filial*. Tendo em conta esta organização dos dados e as características do projeto, optamos por estruturar a aplicação seguindo os princípios da arquitetura *MVC - Model, View, Controller*, dada a inerente modularidade desta arquitetura.

Além disso, optamos por utilizar interfaces uma vez que desta forma é possível que o código possa ser desenvolvido apenas com recurso à interface, sem saber qual a sua implementação. As declarações que constam de uma interface especificam o comportamento que as respetivas implementações oferecem e, assim, o programador apenas necessita de saber qual o comportamento oferecido e pode construir o programa em função disso. A vantagem desta forma de programar é evidente se tivermos em consideração eventuais alterações nalguma parte do código. Desta forma, basta alterar a implementação da classe em questão, garantindo que esta implementa obrigatoriamente os métodos impostos pela interface, mantendo-se o restante código inalterado.

A título de exemplo, podemos implementar um par cujas componentes são a quantidade e a faturação das seguintes formas e, se eventualmente decidirmos mudar a sua implementação, as alterações a efetuar restringir-se-ão apenas a esta classe.

```
public class ParQtFaturacao implements IParQtFaturacao {
    private int quantidade;
    private double faturacao;
    (...)
}

public class ParQtFaturacao implements IParQtFaturacao {
    private Map.Entry<Integer, Double> par;
    (...)
}
```

2 Classes principais

2.1 Catálogo

O *Catálogo* é a estrutura responsável por armazenar, a partir dos ficheiros lidos, todos os códigos de produtos e cliente que forem válidos. Uma vez que tanto produtos como clientes são identificados unicamente pelo seu código alfanumérico, a escolha da estrutura seria entre *Set<String>* ou *List<String>*. Como no contexto do problema a existência de códigos duplicados não é justificável, a escolha recaiu sobre um *Set*.

Tendo optado por um *Set*, teremos de decidir qual a implementação da interface *Set* a utilizar. Uma vez que as operações mais frequentes sobre esta estrutura serão consultas e inserções, optámos por um *HashSet*, tirando partido da complexidade *O(1)* destas operações, que contrastam com a complexidade *O(logN)* das mesmas operações para um *TreeSet*. Relativamente ao *LinkedHashSet*, a vantagem da sua utilização reside no facto de os seus elementos poderem ser iterados por uma ordem específica, algo que não se revela particularmente útil no contexto do problema.

```
public class Catalogo implements ICatalogo, Serializable {
    private Set<String> codigos;
    (...)
}
```

2.2 Faturação

Sendo esta a classe responsável pela resposta eficiente a quesões quantitativas que relacionam os produtos e as suas vendas, é evidente que a estrutura adequada é um *Map* em que a cada código de produto se associa uma estrutura que contém informação sobre o número de vendas, faturação e quantidade vendida desse mesmo produto.

Desta forma, desenvolvemos a classe auxiliar *InfoProduto* que contém informação relevante sobre as vendas de um produto, nomeadamente o número de vendas, faturação e quantidade comprada, valores esses que são separados por mês e filial.

Tal como no *Catálogo*, as operações mais frequentes serão consultas e inserções e, uma vez que não se revela particularmente vantajoso manter os dados ordenados de uma forma específica, a escolha recaiu sobre um *HashMap*.

```
public class Faturacao implements IFaturacao, Serializable {
    private Map<String, IInfoProduto> faturacao;
    (...)
}

public class InfoProduto implements IInfoProduto, Serializable {
    private String produto;
    private boolean comprado;
    private int[][] nVendas;
    private double[][] faturacao;
    private int[] quantidade;
    (...)
}
```

2.3 Filial

É a classe que, a partir dos ficheiros lidos, irá conter as estruturas de dados necessárias para fazer a ligação entre clientes e os respetivos produtos por eles comprados, e vice-versa.

Importa salientar que tivemos em consideração o facto de que o sistema deve suportar não só a existência de 3 filiais da cadeia de distribuição, mas deve também ser escalável de forma a garantir que à medida que a cadeia de distribuição aumenta, o sistema suporta a existência de

mais filiais. Tendo isto em consideração, optamos por, na classe *GestVendasModel* guardar as filiais numa estrutura *List<IFilial>* uma vez que esta estrutura é capaz de crescer e armazenar um maior número de filiais.

Tendo em consideração o conjunto de *queries* apresentado, consideramos que seria pertinente manter os dados organizados por meses, motivo pelo qual optamos pela estrutura *List*, em que cada elemento da lista seria a informação relativa a cada um dos meses considerados, isto é, uma mapeamento em que a cada código de cliente se faria corresponder uma estrutura com a informação relativa às compras que este efetuou nesse mesmo mês. Assim, optamos por armazenar os dados utilizando a estrutura *List<Map<String, IInfoMensalCliente>>*. Além disso, uma vez que as operações mais frequentes sobre esta estrutura serão consultas consideramos que a implementação da interface *Map* mais adequada a este problema seria o *HashMap*.

```
public class Filial implements IFilial, Serializable {
    private List<Map<String, IInfoMensalCliente>> clientes;
    (...)
}

public class InfoMensalCliente implements IInfoMensalCliente, Serializable {
    private Map<String, ITernoQtVendasFaturacao> produtos;
    (...)
}
```

3 Testes de *performance* e análise de resultados

Durante a execução do projeto, foram efetuados diversos testes, quer ao nível da forma de leitura dos ficheiros de texto, em particular utilizando as classes *BufferedReader* e *Files*, assim como ao nível das diversas implementações das interfaces *Map*, *Set* e *List*. Importa salientar que os valores apresentados, resultantes de testes realizados numa máquina com um processador *Intel Core i7-8550U 4.00 GHz* e 8GB de memória RAM DDR4, não são resultado de uma só medição, mas da média de um conjunto de 15 medições.

3.1 Tempos de leitura e escrita

Ao nível de leitura, foram realizados testes com o intuito de comparar a *performance* da leitura de ficheiros de texto utilizando as classes *Files* e *BufferedReader*, sendo os resultados obtidos apresentados nas tabelas e gráficos que se seguem.

	Vendas_1M.txt	Vendas_3M.txt	Vendas_5M.txt
Leitura	0.183	0.406	0.701
Leitura e *parse*	0.754	1.952	3.206
Leitura, *parse* e validação	0.906	2.575	3.983

Tabela 1: Tempos, em segundos, de leitura, *parse* e validação, utilizando *BufferedReader*.

	Vendas_1M.txt	Vendas_3M.txt	Vendas_5M.txt
Leitura	0.179	0.668	0.758
Leitura e *parse*	0.771	2.048	3.220
Leitura, *parse* e validação	1.221	2.628	4.380

Tabela 2: Tempos, em segundos, de leitura, *parse* e validação, utilizando *Files*.

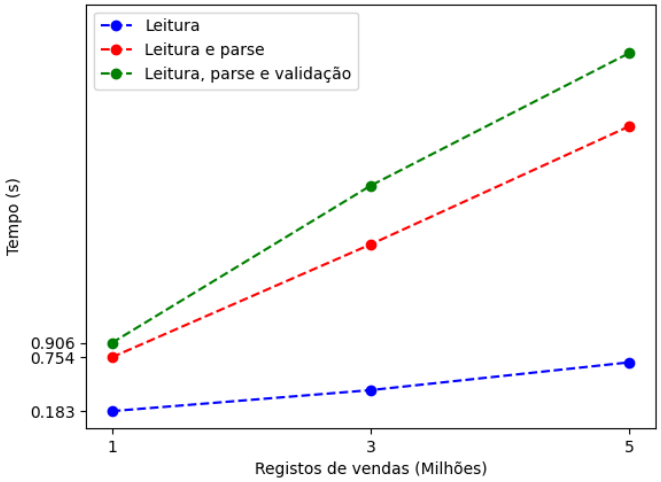

Figura 1: Tempos, em segundos, de leitura, *parse* e validação utilizando a classe *BufferedReader*.

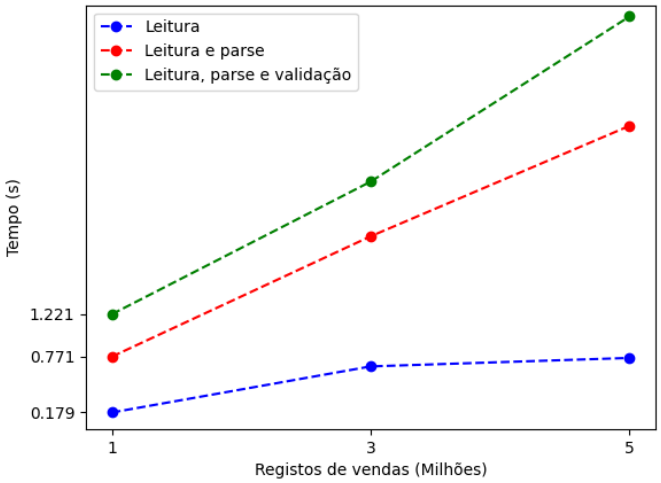

Figura 2: Tempos, em segundos, de leitura, *parse* e validação utilizando a classe *Files*.

Com estes resultados, podemos notar que a leitura com recurso a um *BufferedReader* é relativamente mais eficiente, e, por essa razão, foi esse o método usado para a leitura.

	Vendas_1M.txt	Vendas_3M.txt	Vendas_5M.txt
Leitura dos ficheiros de texto	3.11	8.84	11.63
Gravação do estado do programa	365.63	496.22	754.85
Recuperação do estado do programa	55.86	69.12	79.21

Tabela 3: Tempo, em segundos, de leitura e escrita.

Tendo em consideração os resultados apresentados, facilmente se conclui que não existe qualquer vantagem em guardar o estado do programa, utilizando *Object Streams* e, posteriormente

recuperá-lo, uma vez que a inicialização das estruturas de dados partindo dos ficheiros de texto revela-se consideravelmente mais eficiente. Além disso, comparando estes resultados com os obtidos no trabalho prático desenvolvido em *C*, podemos notar que a inicialização e população das estruturas de dados é relativamente mais eficiente.

3.2 Tempos de execução

Com recurso à classe fornecida *Crono*, efetuamos alguns testes de *performance*, obtendo, para cada um dos ficheiros de vendas fornecidos, a tabela com os tempos médios de execução de cada *query*, que se apresenta de seguida.

Foram também testados tempos de execução das implementações da interface *Map*, nomeadamente *HashMap* e *TreeMap*, tendo obtidos os seguintes resultados.

	Vendas_1M.txt	Vendas_3M.txt	Vendas_5M.txt	Input
Query 1	0.0257	0.0397	0.0269	-
Query 2	0.159	0.419	0.595	2
Query 3	0.769	2.033	3.266	*F2916*
Query 4	0.149	0.136	0.147	*AF1184*
Query 5	0.00884	0.00890	0.146	*Z5000*
Query 6	1.583	5.315	9.368	15
Query 7	0.423	0.807	0.713	-
Query 8	2.012	8.678	12.75	20
Query 9	0.377	0.335	0.393	Produto *AF1184*, 10 clientes
Query 10	0.0290	0.139	0.132	Mês 3, Filial 2

Tabela 4: Tempo de execução, em segundos, de cada *query*. Foram utilizados *ArrayList, HashMap* e *HashSet* como implementação das interfaces *List, Map* e *Set*, respetivamente.

Podemos observar que as únicas *queries* que o tempo de execução da maioria das *queries* não varia de forma significativa. De facto, as únicas que realmente aumentam um consideravelmente o tempo de espera por parte do utilizador à medida que o número de registos de venda aumenta são as *queries* 6 e 8. Este aumento acontece sobretudo porque, uma vez que optamos por representar a estrutura de uma filial utilizando *HashMaps*, e dado que estes não admitem noção de ordem, torna-se necessário, para cada uma das filiais, iterar sobre todos os clientes.

Relativamente às diferentes implementações da interface *List*, nomeadamente *ArrayList* e *Vector*, obtivemos os resultados que se apresentam a seguir. Uma vez que estes implementam uma mesma interface, possuem os mesmos métodos mas não as mesmas implementações dos métodos, e é aqui que podemos notar a principal diferença, que se dá principalmente ao nível da *performance*.

	Tempo de carregamento
Vendas_1M.txt	3.35
Vendas_3M.txt	8.17
Vendas_5M.txt	13.15

Tabela 5: Tempo, em segundos, de carregamento das estruturas de dados, utilizando *Vector* como implementação da interface *List*.

	Vendas_1M.txt	Vendas_3M.txt	Vendas_5M.txt	Input
Query 1	0.030	0.0120	0.0265	-
Query 2	0.230	0.711	0.735	2
Query 3	1.332	3.105	3.262	*F2916*
Query 4	0.0899	0.150	0.144	*AF1184*
Query 5	0.00323	0.00371	0.00855	*Z5000*
Query 6	1.412	8.025	9.103	15
Query 7	0.317	0.777	0.766	-
Query 8	2.078	13.203	13.62	20
Query 9	0.254	0.365	0.365	Produto *AF1184*, 10 clientes
Query 10	0.0290	0.117	0.144	Mês 3, Filial 2

Tabela 6: Tempo de execução, em segundos, de cada *query*, utilizando *Vector* como implementação da interface *List*.

Não havendo diferenças significativas, acabamos assim por optar pela implementação mais comum *ArrayList*.

Por outro lado, testamos diferentes implementações das interfaces *Map*, nomeada- mente, *HashMap* e *TreeMap*, tendo obtido os resultados que se apresentam a seguir.

	Tempo de carregamento
Vendas_1M.txt	3.25
Vendas_3M.txt	9.60
Vendas_5M.txt	16.88

Tabela 7: Tempo, em segundos, de carregamento das estruturas de dados, utilizando *TreeMap* como implementação da interface *Map*.

	Vendas_1M.txt	Vendas_3M.txt	Vendas_5M.txt	Input
Query 1	0.0246	0.0314	0.0493	-
Query 2	0.110	0.441	0.724	2
Query 3	0.798	2.073	3.329	*F2916*
Query 4	0.146	0.138	0.148	*AF1184*
Query 5	0.00753	0.00914	0.0102	*Z5000*
Query 6	1.221	7.018	12.501	15
Query 7	0.383	0.650	0.990	-
Query 8	1.946	6.676	13.688	20
Query 9	0.251	0.391	0.532	Produto *AF1184*, 10 clientes
Query 10	0.0275	0.0683	0.179	Mês 3, Filial 2

Tabela 8: Tempo de execução, em segundos, de cada *query*, utilizando *TreeMap* como implementação da interface *Map*.

Concluímos, assim, que , utilizando *HashMaps*, o tempo de carregamento de informação para memória é menor, mantendo os tempos das *queries* praticamente inalterados. Relativamente a *TreeMaps*, notamos que para as *queries* 6 e 8, os tempos de execução aumentaram consideravelmente.

Por fim, testamos tempos de execução das implementações da interface *Set*, nomeadamente *HashSet* e *TreeSet*. Apresentamos, então, os resultados nas tabelas que se seguem.

	Tempo de carregamento
Vendas_1M.txt	4.08
Vendas_3M.txt	12.78
Vendas_5M.txt	21.39

Tabela 9: Tempo, em segundos, de carregamento das estruturas de dados, utilizando *TreeSet* como implementação da interface *Set*.

	Vendas_1M.txt	Vendas_3M.txt	Vendas_5M.txt	Input
Query 1	0.0266	0.0306	0.0294	-
Query 2	0.215	0.455	0.664	2
Query 3	1.276	1.996	3.186	*F2916*
Query 4	0.0983	0.159	0.152	*AF1184*
Query 5	0.00931	0.0120	0.00940	*Z5000*
Query 6	1.590	4.232	8.546	15
Query 7	0.390	0.626	0.756	-
Query 8	2.319	9.974	17.691	20
Query 9	0.245	0.332	0.386	Produto *AF1184*, 10 clientes
Query 10	0.0394	0.0874	0.121	Mês 3, Filial 2

Tabela 10: Tempo de execução, em segundos, de cada *query*, utilizando *TreeSet* como implementação da interface *Set*.

Desta forma, podemos concluir que a diferença mais significativa ocorre ao nível do carregamento de dados para memória. Neste caso, um *HashSet* revela-se mais eficiente dada a complexidade *O(1)* das inserções, que contrasta com a complexidade *O(logN)* da mesma operação para um *TreeSet*.

4 Conclusão

Através do *parsing* dos ficheiros de texto, foi possível extrair toda a informação considerada relevante para a resolução das *queries* propostas, guardando a mesma em estruturas de dados que consideramos apropriadas. Conseguimos cumprir todos os requisitos propostos, implementando todos os módulos e estruturando-os como pedido, sendo assim capaz de responder a todas as *queries* da forma que nos pareceu mais eficiente, motivo pelo qual consideramos que atingimos o objetivo proposto, ainda que haja espaço para posteriores melhorias. Além disso, o nosso projeto foi elaborado com a ideia de criar algo que permita uma fácil expansão tanto de funcionalidades, como de adição de novos dados às estruturas, como por exemplo a existência de mais filiais.

Como trabalho futuro, gostaríamos de melhorar a forma como organizamos os dados relativos às filiais, de forma a diminuir o tempo de resposta a *queries* que consultam estes dados.

5 Anexos

5.1 Diagrama de Classes

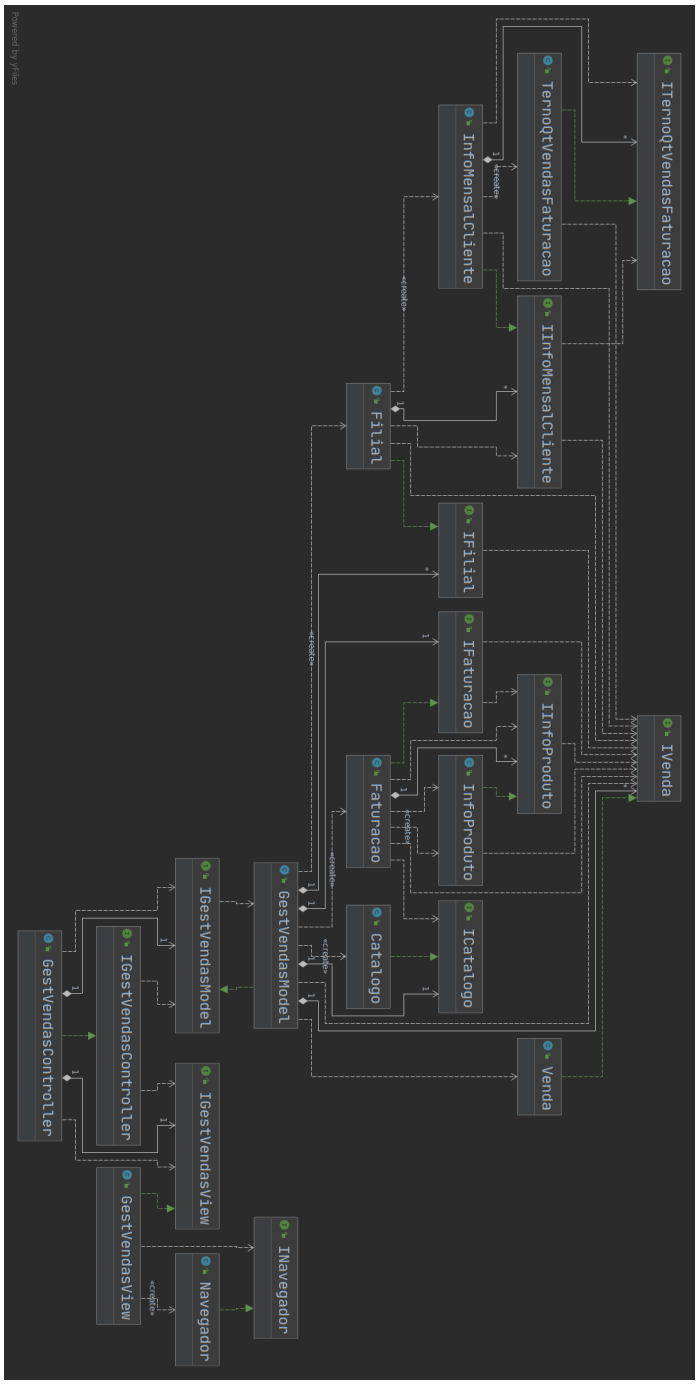

Figura 3: Diagrama de Classes