Universidade do Minho
Escola de Engenharia

Sistema de Gestão de Vendas de uma Distribuidora

Laboratórios de Informática 3 - Grupo 2

João Pedro da Santa Guedes A89588
Luís Pedro Oliveira de Castro Vieira A89601
Pedro Miguel de Soveral Pacheco Barbosa A89529



19 de abril de 2020

Índice

1 Introdução

No âmbito da unidade curricular de Laboratórios de Informática 3 foi-nos proposta a implementação de um sistema de resposta a queries sobre a gestão de vendas de uma distribuidora com 3 filiais.

A segunda fase do projeto consistiu em implementar este sistema na linguagem de programação Java. Tendo como objetivo tirar o máximo partido da eficiência e rápida execução das funcionalidades do nosso programa, tivemos também como foco alguns princípios da programação como modularidade e encapsulamento de dados, criação de código reutilizável, uma escolha otimizada das estruturas de dados bem como a sua reutilização e testes de performance e de profiling. Não podemos deixar de mencionar a grande diferença em termos de ferramentas disponíveis comparativamente com o projeto em C.

Estes objetivos vão de encontro aos conhecimentos adquiridos neste semestre na unidade curricular de Programação Orientada aos Objetos, sendo que a segunda fase deste projeto compreende a implementação do mesmo sistema na linguagem de programação Java.

2 Ferramentas Auxiliares

Ao contrário do que aconteceu no projeto em C, nesta segunda fase nem sequer consideramos implementar estrutruras de dados feitas por nós, sendo que a linguagem Java já apresenta uma API enriquecida com ferramentas que servem este propósito.

3 Estruturas de Dados

A abordagem feita às estruturas de dados é um dos passos mais importantes de todo o projeto pois é essencial tanto para organizar a quantidade de informação presente nos ficheiros de texto como para responder às queries do trabalho.

Assim, começamos por analisar como eram constituídos os produtos, os clientes e as vendas bem como o que nos era pedido em cada querie de forma a percebermos quais eram as informações às quais iríamos recorrer constantemente. Desta forma, e muito similarmente áquilo que fizemos no projeto em C criamos quatro classes: uma para o catálogo de clientes, uma

para o catálogo de produtos, uma para a faturação e uma para as filiais. Mais tarde fomos criando outras classes que foram sendo necessárias com o aumento de complexidade do projeto.

3.1 Tipos de Dados

A base de todo o trabalho são os dados dos clientes, dos produtos e das vendas.

3.1.1 Clientes e Produtos

Foi-nos pedido no enunciado para criarmos um catálogo tanto de clientes como de produtos. Assim sendo, criamos duas classes, uma para cada catálogo.

```
private Map<Integer, Set<String>> clients;
```

```
private Map<Integer, Set<String>> products;
```

Para organizarmos a informação ao longo do projeto acabamos quase sempre por utilizar a interface Map de Java.

Assim sendo, para ambos os catálogos de clientes e de produtos utilizamos um Map em que a chave é um Integer correspondente a uma das 26 posições, que são as letras do alfabeto. O valor do nosso Map é um Set com os códigos dos clientes e dos produtos, dependendo do catálogo.

3.1.2 Vendas

Arranjar uma forma de tratar das vendas de maneira a facilitar a resposta das queries foi um pouco mais complexo. Desta vez criamos duas classes diferentes no nosso trabalho, uma para a faturação e outra para as filiais.

Em relação à primeira classe, em que tivemos de relacionar os produtos com as suas vendas, decidimos que o melhor seria criar um Map em que a chave é um Integer correspondente a uma das 26 letras do alfabeto. Por sua vez o valor do Map é outro Map em que a chave é uma String com o código do produto e o valor é a informação da faturação desse mesmo produto. Essa informação está presente noutra classe que é a InformacaoFat na qual organizamos a informação de qualquer produto em arrays tridimension-

ais organizados pelo modo de compra, pela filial onde foi realizada e pelo respetivo mês.

```
private Map<Integer, Map<String, InformacaoFat>> faturacao;
```

```
private double [][][][]fat;
private int [][][][]qtd;
private int [][][][]n_vendas;
private int []qtd_total;
```

A outra classe foi a das filiais. Nesta, utilizamos mais uma vez Maps. Criamos um em que relacionamos as filiais com os clientes e outro em que as relacionamos com os produtos.

A chave dos nossos Maps é um Integer referente a uma das 3 filiais existentes e o valor do mesmo é novamente um Map em que a chave é uma String com o código do cliente ou do produto, dependendo se são as filiais de clientes ou as filiais de produtos, e o valor é a informação do respetivo cliente ou produto. Estas informações estão noutras duas classes, na classe InfoCliente e na classe InfoProduto, respetivamente.

Em relação à classe InfoCliente, esta contém o código do cliente e três arrays: um com a quantidade total que esse cliente comprou num determinado mês, um com a faturação total do cliente num determinado mês e outro com o número de vendas desse cliente num determinado mês. Também criamos um Map, que contém informações dos produtos que esse cliente comprou, no qual a chave é uma String com o código do produto e o valor é a informação de venda do mesmo. Esta informação, por sua vez, está presente noutra classe, a classe InformacaoVendaProduto.

Em relação à classe InfoProduto, esta contém o código do produto e três arrays: um com a quantidade total desse produto comprada num determinado mês, um com o número de vezes que esse produto foi comprado num determinado mês e um com a faturação total referente a esse produto num determinado mês. Também cirmaos um Map, que contém as informações dos clientes que compraram esse produto, no qual a chave é uma String com o código do cliente e o valor é a informaçao de venda do mesmo. Esta informação, por sua vez, está presente noutra classe, a classe InformacaoVendaCliente.

```
private Map<Integer, Map<String, InfoCliente>> filialClientes;
private Map<Integer, Map<String, InfoProduto>> filialProdutos;
```

```
private String code; // client code
private Map<String, InformacaoVendaProduto> products_bought;
private int[] qtd_total_mes;
private double[] fat_total_mes;
private int[] nvendas_mes;
```

```
private String code;
private Map<String, InformacaoVendaCliente> clients_who_bought;
private int[] qtd_total_comprada_mes;
private int[] numero_vezes_comprada_mes;
private double[] faturacao_total_mes;
```

4 Modularização e Abstração dos Dados

Como aprendemos durante este semestre, é uma boa prática manter o nosso código modular e abstraído do tipo de dados com que trabalhamos.

Assim sendo, utilizamos encapsulamente ao longo do nosso projeto. Em Java, este processo torna-se relativamente fácil uma vez que esta linguagem já tem mecanismos que facilitam esta prática. São exemplos disso, a declaração de variáveis de instância como private e a utilização do método clone sempre que necessário de forma a clonar-se tanto a informação que se recebe como aquela que se retorna.

A utilização das diferentes APIs de Java tornou o código bastante abstrato sendo que a qualquer altura conseguimos, por exemplo, alternar entre a utilização de Hashmaps e Treemaps sem termos de modificar muito o nosso código. Este comportamente verifica-se na maior parte do nosso código.

5 Queries

5.1 Query 1 Estatística

Para a Query 1 Estatística começamos por passá-la no parser para contabilizar o número total de vendas, o número total de vendas corretas e com a diferença das duas as erradas, o número total de produtos e o número total

de clientes. Depois, acedemos à faturação para calcular o total de produtos comprados e não comprados, o número total de compras de valor igual a 0 e o total da faturação. Acedemos também à filialClientes para determinar quantos clientes compraram e quantos não realizaram nenhuma compra.

5.2 Query 2 Estatística

Em relação à Query 2 Estatística acedemos às filiais, primeiro à filial de clientes para calcular o número de clientes distintos que compraram em cada mês e depois à filial de produtos para calcular o total faturado em cada mês de cada filial. Depois acedemos à faturação para calcular o total de vendas em cada mês.

5.3 Query 1

Esta Query pretende obter o código dos produtos não comprados bem como o seu número total. Desta forma, criamos uma função que percorre a faturação e verifica se um dado produto existe. Se este não existir, adiciona-o o seu código a um TreeSet. Por fim, retornamos o size do TreeSet para assim obtermos o seu número total.

5.4 Query 2

A Query 2 pretende obter o número total de vendas realizadas bem como o número total de clientes distintos que as realizaram num determinado mês, tanto de forma global como para um filial específica. O método que adotamos para a resolver foi o de percorrermos a filial dos clientes e sempre que, nesse mês, o número de vendas desse cliente fosse diferente de 0 iriamos adicionar o seu código a um TreeSet e aumentavamos um contador do número de vendas com o número de compras realizadas por esse cliente. No final retornamos esse contador que continha o número total de vendas realizadas bem como o size do nosso TreeSet que iria dar o número total de clientes distintos que as realizaram.

Criamos também outra função igual mas que iria percorrer uma filial específica para deste modo obtermos estes valores mas para essa determinada filial e não de forma global.

5.5 Query 3

O objetivo da Query 3 era dado um cliente determinar, para cada mês, quantos compras fez, quantos produtos diferentes comprou e o total gasto. Desta forma, percorremos a filialClientes e, para cada filial, utilizamos funções auxiliares que guardavam em arrays, cujas posições eram referentes a cada mês, o número de compras, o número total de produtos e o total faturado desse mesmo cliente.

5.6 Query 4

Para respondermos à Query 4 o método que utilizamos foi exatamente o mesmo da Query 3. A única diferença é que como os números que pretendemos obter são referentes aos produtos e não aos clientes, desta vez, percorremos a filialProdutos.

5.7 Query 5

Esta Query pretende obter, dado um código de cliente, uma lista com o código e a quantidade dos produtos que mais comprou. Deste modo, percorremos a filialClientes, e para cada filial fomos ao cliente dado e guardamos num TreeSet o código dos produtos que este mais comprou bem como a quantidade dos mesmos.

5.8 Query 6

De modo a obtermos o número de produtos mais comprados bem como o código dos diferentes clientes que os compraram tivemos de dividir a Query6 em dois. Consequentemente, percorremos o map da faturação e guardamos num TreeSet ordenado pela quantidade o código dos produtos bem como a quantidade comprada dos mesmos. De seguida, percorremos a filialProdutos e para cada filial encontramos o produto pretendido e calculamos o número de clientes distintos que o comprou.

5.9 Query 7

A query 7 pretende obter os 3 maiores compradores em termos de dinheiro faturado para cada uma das filiais. Para a resolvermos, percorremos a

filialClientes e para cada filial guardamos num TreeMap a string com o código de cada cliente bem como o dinheiro por ele gasto. Guardamos também num Arraylist o código dos diferentes clientes e através de um Comparator que os compara pelo dinheiro que eles gastaram retornamos os 3 clientes mais gastadores de cada filial.

5.10 Query 8

O objetivo da Query 8 é obter os códigos dos clientes que compraram mais produtos diferentes. Assim sendo, percorremos a filialClientes e através da nossa classe InfoCliente obtemos quantos produtos diferentes cada cliente comprou. Finalmente, guardamos os códigos dos clientes num Arraylist utilizando um Comparator que os ordenou pelo número de produtos diferentes que comprou de forma decrescente.

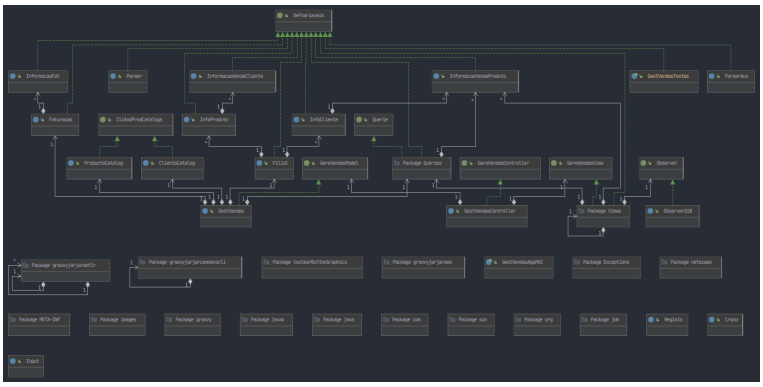
5.11 Query 9

De modo a obtermos os clientes que mais compraram um determinado produto e quanto gastaram no total nesse mesmo produto decidimos percorrer a filialProdutos e para cada filial, com o respetivo código de produto, calculamos os diferentes clientes que o compraram bem como a quantidade do mesmo e o valor que gastaram nele. No fim, guardamos numa estrutura, através do recurso a um Comparator que a ordenou pelo valor gasto, a string com o código dos clientes bem como o total por eles gasto no produto dado.

5.12 Query 10

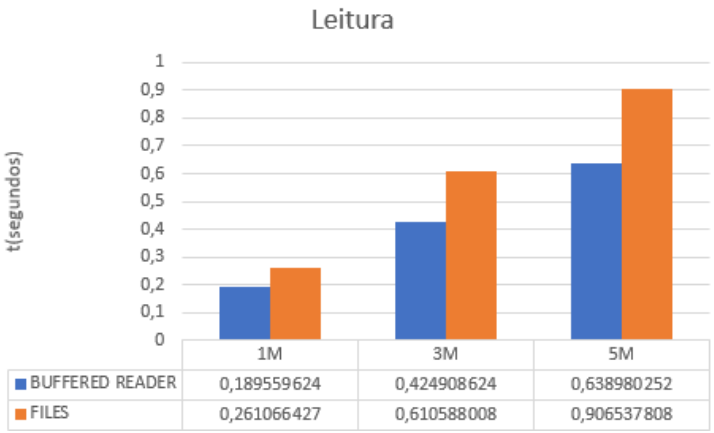
A Query 10 pretende obter para cada filial e para cada mês o total faturado para cada produto. Assim sendo, percorremos a filialProdutos, e depois com recurso à infoProduto guardamos a informação contida no array que guardava a faturação de cada produto em cada mês. Por último, guardamos numa estrutura a filial, o código do produto e a faturação por mês referente ao mesmo.

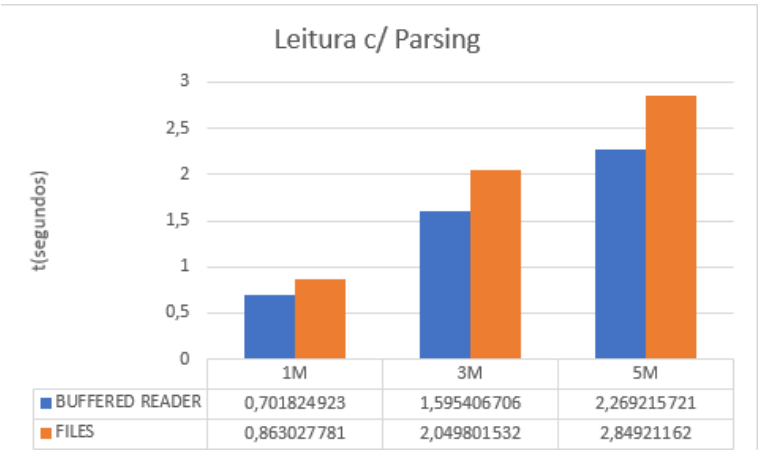
6 Diagrama de Classes

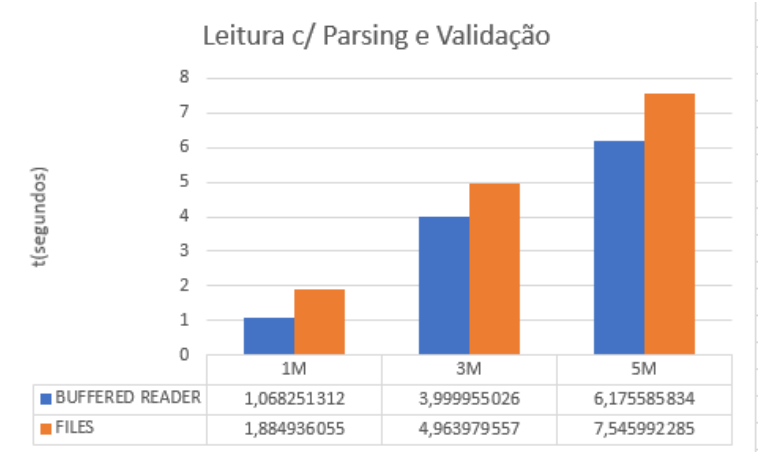


7 Testes e Resultados

7.1 Leitura, Leitura com Parsing e Leitura com Parsing e Validação

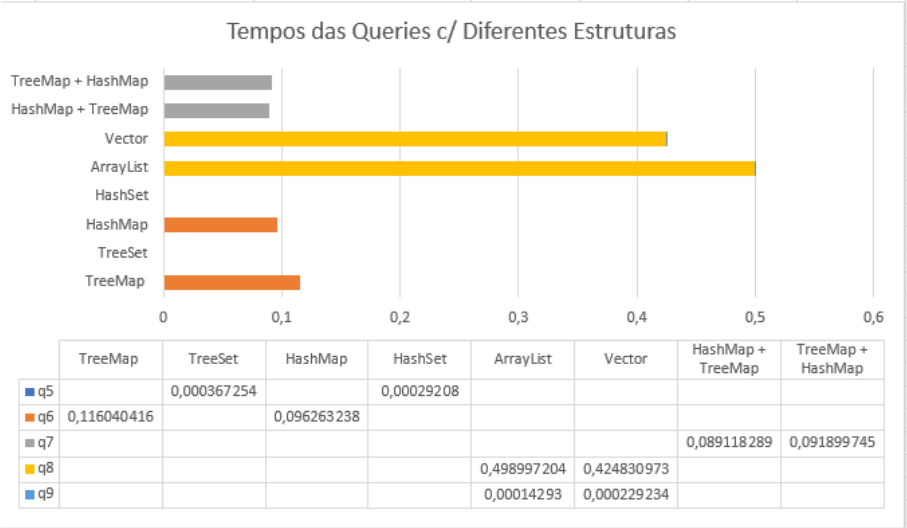






Como podemos ver pelos gráficos acima apresentados, em todos os tópicos pedidos e medidos,a classe BufferedReader apresenta melhores tempos quando comparada à classe Files.

7.2 Tempo das Queries com diferentes Estruturas



	TreeMap	TreeSet	HashMap	HashSet	ArrayList	Vector	HashMap + TreeMap	TreeMap + HashMap
q5		0,000367254		0,00029208				
q6	0,116040416		0,096263238					
q7							0,089118289	0,091899745
q8					0,498997204	0,424830973		
q9					0,00014293	0,000229234		

Como pedido no enunciado, de forma a testar a perfomance das estruturas escolhidas por nós para o trabalho, foi-nos pedido que, onde usámos HashMap fosse usado TreeMap, onde foi usado TreeSet usar HashSet ou LinkedHashSet e onde foi usado ArrayList usar Vector, ou vice-versa em todos os casos. Acima apresentamos o gráfico com as medições de tempo efetuadas e comparando os dois resultados.

No entanto é preciso notar que, face ao pedido para ser realizado em cada query, a troca destas estruturas, apesar de poder apresentar tempos mais favoráveis, torna o objetivo do programa obsoleto, uma vez que, e acontecendo recorrentemente, em queries que pedem ordenação, a troca das estruturas impossibilita essa ordenação.

Assim, e apesar de serem apresentadas melhorias nos tempos para algumas queries como 5, 6 e 8, a troca de estruturas é ineficiente. Concluímos assim que as estruturas a utilizar para que o trabalho corresponda ao pedido, e mantenha uma performance viável são Query5 TreeSet, Query6 TreeMap, Query7 HashMap + TreeMap, Query8 ArrayList e Query9 ArrayList.

7.3 Estruturas que contêm informação

FILIAL	FATURAÇÃO	CATALOGOS	TEMPO
HashMap + TreeMap + HashMap	HashMap + HashMap	HashMap + TreeSet	8,97431209
TreeMap + TreeMap + TreeMap	HashMap + HashMap	HashMap + TreeSet	10,250588
HashMap + HashMap + HashMap	HashMap + HashMap	HashMap + TreeSet	6,176587
HashMap + HashMap + HashMap	HashMap + TreeMap	HashMap + TreeSet	7,39014429
HashMap + HashMap + HashMap	TreeMap + HashMap	HashMap + TreeSet	6,12629619
HashMap + HashMap + HashMap	TreeMap + TreeMap	HashMap + TreeSet	7,52173978
HashMap + HashMap + HashMap	TreeMap + HashMap	TreeMap + Tree Set	6,28269892
HashMap + HashMap + HashMap	TreeMap + HashMap	TreeMap + HashSet	5,40457281
HashMap + HashMap + HashMap	TreeMap + HashMap	HashMap + HashSet	5,35682671

De forma a conseguir testar quais seriam as estruturas que seriam mais eficientes em termos de tempo começámos por alterar a estrutura das filiais. Inicialmente tinhamos HashMap + TreeMap + HashMap e fomos alterando, como é possivel ver na tabela, até obtermos uma estrutura que tivesse um desempenho melhor que as outras. Assim chegámos à conclusão que, somente alterando a estrutura das filiais, no caso de esta ser HashMap + HashMap + HashMap seria a que tem melhor desempenho.

Procedemos da mesma maneira para com a faturação e procedemos à troca de estruturas e comparação de resultados, sendo que inicialmente usávamos HashMap + HashMap, e chegámos à conclusão de que, após alterar a estrutura da filial, a estrutura da faturação que permitiria um melhor desempenho seria TreeMap + HashMap.

Por fim, o mesmo processo foi realizado para os catálogos de clientes e produtos onde inicialmente a estrutura usada era HashMap + TreeSet, que acabaria por ser a estrutura com melhor performance face a outras experimentadas.

	ESTRUTURA FINAL
FILIAL	HashMap + HashMap + HashMap
FATURAÇÃO	TreeMap + HashMap
CATÁLOGOS	HashMap + TreeSet
TEMPO(s)	5,341525601

Como podemos ver imagem acima apresentada, conseguimos obter uma estrutura mais eficiente em termos de load ao realizar os testes de performance, demorando cerca de menos 3 segundos comparado ao valor inicial. Desta forma conseguimos manter um tempo de load eficiente, não comprometendo a abstração e encaplusamento do nosso código.

QUERIES	TEMPOS(s) DAS QUERIES COM ESTRUTURA FINAL
Q5	0,000190324
Q6	0,109605053
Q7	0,084807002
Q8	0,38787732
Q9	0,000136467

Para testar a Query 5 utilizamos o cliente Z5000, para a Query 6 utilizamos 20 produtos, para a Query 8 utilizamos 20 clientes e para a Query 9 utilizamos o produto AF1184 e 10 clientes.

8 Conclusão

Tal como na primeira fase do projeto podemos concluir que há sempre um compromisso entre performance e segurança e que a organização das nossas estruturas tem muita importância no que toca à eficiência das queries.

Concluimos também aquilo que já tinhamos referido no relatório da primeira fase do projeto, ou seja, que foi bastante mais fácil implementar esta solução na linguagem Java do que em C dado a panóplia de ferramentas que a primeira linguagem disponibiliza.

Finalmente fazendo uma retrospetiva global sobre as duas fases do projeto, chegamos à conclusão de que, apesar de existirem alguns aspetos do projeto que apenas a linguagem em C pode oferecer, acabariamos sempre por perferir executar o mesmo na linguagem Java. Vivemos num mundo cada vez mais orientado aos objetos e exigente do ponto de vista produtivo e, assim sendo, é normal que vigore a utilização de linguaguens com um maior nível de abstração.