Escola de Engenharia
Universidade do Minho

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
Mestrado Integrado em Engenharia Informática
Laboratórios de Informática III

Gestão de Vendas de uma cadeia de Distribuição com 3 filiais

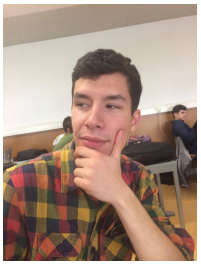
GEREVENDAS

Parte JAVA

Grupo 84


Célia Figueiredo
a67637


Gil Gonçalves
a67738


Humberto Vaz
a73236


Ricardo Lopes
a72062

Braga, 16 de Junho de 2016

1. Introdução

No âmbito da unidade curricular de Laboratórios de Informática III do 2ºano da licenciatura de Engenharia Informática, foi proposto o desenvolvimento de um projeto em linguagem Java, que tem por objectivo ajudar à consolidação dos conteúdos teóricos e práticos e enriquecer os conhecimentos adquiridos na UC de Programação Orientada aos Objectos.

Este projeto considera-se um grande desafio para nós pelo facto de passarmos a realizar programação em grande escala, uma vez que se trata de grandes volumes de dados e por isso uma maior complexidade. Como primeira etapa, o mais importante é definir as estruturas utilizadas no trabalho. Assim sendo iremos explicar ao longo do relatório detalhadamente cada uma e os aspetos mais importantes que definem a base do nosso projeto.

Por fim, explicaremos todas as nossas decisões quanto as resolução das diversas consultas ("queries"), os resultados obtidos para cada uma destas e apresentaremos também uma detalhada explicação das mesmas. Nisto englobaremos algumas estatísticas e alguns testes de "performance" do nosso programa em relação aos tipos de estruturas de dados oferecidas pela linguagem. Estes testes serão baseados na alteração das nossas estruturas iniciais e serão comparados através dos tempos de execução da leitura e de algumas queries.

Conteúdo

2. Arquitetura da Aplicação

A arquitetura da aplicação a desenvolver é definida por quatro módulos principais: Catálogo de clientes, Catálogo de produtos, Faturação Global e Vendas por Filial, cujas fontes de dados são três ficheiros de texto detalhados abaixo.

No ficheiro **Produtos.txt** cada linha representa o código de um produto vendável no hipermercado, sendo cada código formado por duas letras maiúsculas e 4 dígitos (que representam um inteiro entre 1000 e 1999), como no exemplo:

```
AB9012
XY1185
BC9190
```

O ficheiro de produtos contém cerca de 200.000 códigos de produto.

No ficheiro **Clientes.txt** cada linha representa o código de um cliente identificado no hipermercado, sendo cada código de cliente formado por uma letra maiúscula e 4 dígitos que representam um inteiro entre 1000 e 5000, segue um exemplo:

```
F2916
W1219
F2915
```

O ficheiro de clientes contém cerca de 20.000 códigos de cliente.

O ficheiro **Vendas_1M.txt**, no qual cada linha representa o registo de uma venda efectuada numa qualquer das 3 filiais da Cadeia de Distribuição. Cada linha (a que chamaremos compra ou venda, o que apenas depende do ponto de vista) será formada por um código de produto, um preço unitário decimal (entre 0.0 e 999.99), o número inteiro de unidades compradas (entre 1 e 200), a letra **N** ou **P** conforme tenha sido uma compra **Normal** ou uma compra em **Promoção**, o código do cliente, o mês da compra (1 ... 12) e a filial (de 1 a 3) onde a venda foi realizada, como se pode verificar nos exemplos seguintes:

```
KR1583 77.72 128 P L4891 2 1
QQ1041 536.53 194 P X4054 12 3
OP1244 481.43 67 P Q3869 9 1
JP1982 343.2 168 N T1805 10 2
IZ1636 923.72 193 P T2220 4 2
```

O ficheiro de vendas inicial, **Vendas_1M.txt** , conterá 1.000.000 (1 milhão) de registos de vendas realizadas nas 3 filiais da cadeia de distribuição. Existirão também os ficheiros **Vendas_3M.txt** e **Vendas_5M.txt** utilizados para as questões de performance da aplicação.

A aplicação possuiu uma arquitectura tal como apresentado na figura seguinte, em que se identificam as fontes de dados, a sua leitura e os módulos de dados a construir:

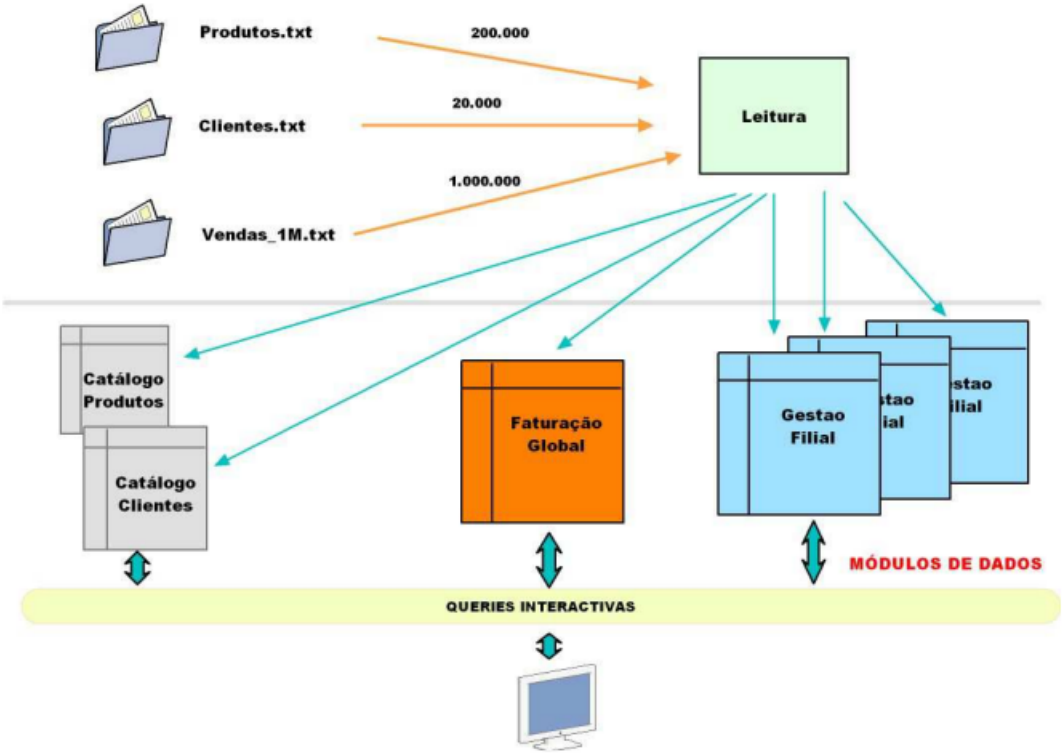


Figura 2.1: Arquitetura da aplicação

3. Diagrama de classes

De seguida é apresentado o Diagrama de classes gerado no BlueJ com os ficheiros .java usados.

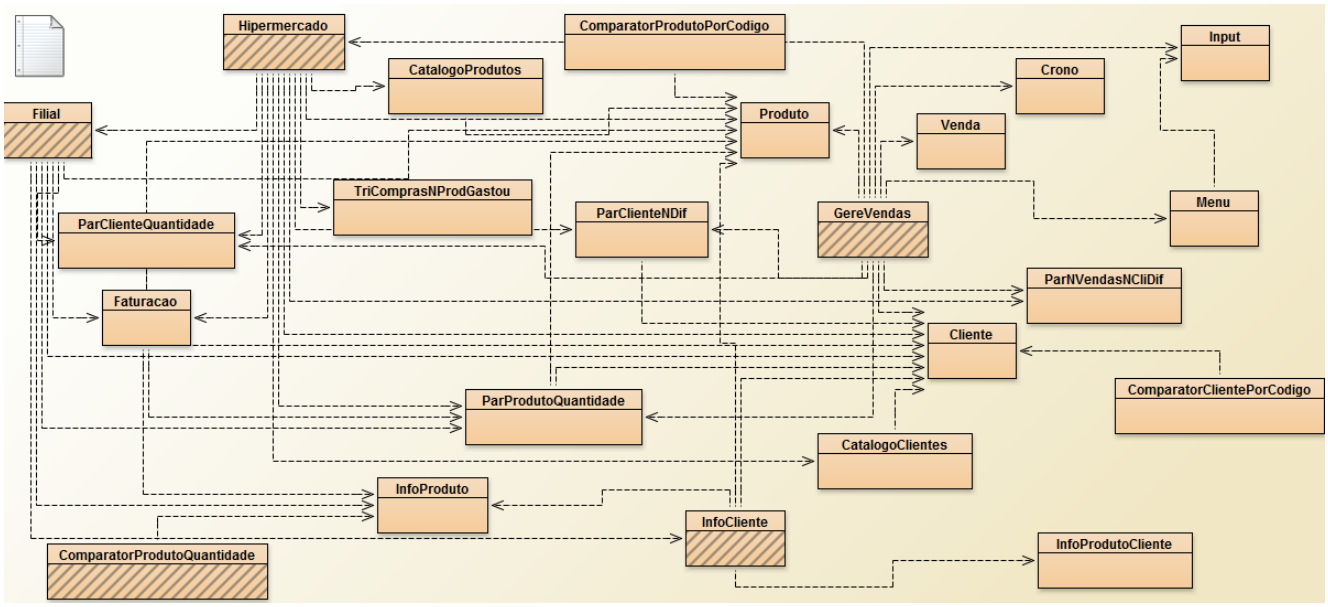


Figura 3.1: Diagrama de classe do Blue J

4. Módulos de dados

4.1 Catálogo de Clientes

É o módulo de dados onde são guardados os códigos de todos os clientes do ficheiro **Clientes.txt**, organizados por índice alfabético;

4.1.1 Classe CatalogoClientes

Para guardar os clientes presentes no ficheiro de clientes.txt, criámos a classe **CatalogoClientes** que irá guardar a lista de todos os clientes presentes no ficheiro. Utilizando HashSet que guarda todos os códigos de cliente do tipo **String** e retira os repetidos, proporcionando assim uma pesquisa rápida.

A classe **CatalogoClientes** servirá também para validar os clientes.

```
private Set<Cliente> catalogoClientes;
```

4.1.2 Classe Cliente

A classe **Cliente** foi implementada para guardar o código de cada cliente. Foi utilizada a classe especial em Java que é responsável por fazer comparações de objetos, que é a classe Comparable . Usada como padrão de comparação, serve para comparar o tipo de objectos para os quais o objeto Cliente pode ser comparado.

```
public class Cliente implements Comparable<Cliente> {

        private String codigo_cliente;
```

4.1.3 Classe InfoCliente

Esta classe foi criada para guardar informações relativas ao cliente, tais como o dinheiro que um cliente gastou num mês, a quantidade de produtos comprados num mês, as compras efetuadas, a quantidade de compras efetuadas, total de dinheiro gasto, assim como a lista de todos os produtos comprados em modo de Promoção(P), ou em modo Normal (N).

Fica aqui o código que define as variáveis de instância da classe InfoCliente:

```
class InfoCliente {
private double[] dinheiroGastoMes;
private int[] totalCompradoMes;
private int [] vendas;
private int totalComprado;
```

```
private double totalGasto;
private Map <Produto,InfoProdutoCliente> produtosComprados[]; // modos
```

Esta classe contém os construtores, clone e os respetivos set e get definidos que servirão de auxilio noutras classes.

4.2 Catálogo de Produtos

Módulo de dados onde são guardados os códigos de todos os produtos do ficheiro **Produtos.txt**, organizados por índice alfabético, o que irá permitir, de forma eficaz, saber quais são os produtos cujos códigos começam por uma dada letra do alfabeto e saber quantos produtos são contabilizados.

4.2.1 Classe CatalogoProdutos

Para guardar os produtos presentes no ficheiro de produtos.txt, criámos a classe **CatalogoProdutos** que irá guardar a lista de todos os produtos presentes no ficheiro. Utilizando HashSet que guarda todos os códigos de produto do tipo **String** e retira os repetidos, proporcionando assim uma pesquisa rápida.

A classe **CatalogoProdutos** servirá também para validar os produtos.

```
private Set<Produto> catalogoProdutos;
```

4.2.2 Classe Produto

A classe **Produto** foi implementada para guardar o código de cada produto. Esta classe é semelhante à classe Cliente.

```
public class Produto implements Comparable<Produto> {

private String codigo_produto;
```

4.2.3 Classe InfoProduto

A classe InfoProduto serve para guardar as informações relativas ao produto, tais como a quantidade de um produto em determinado mes numa filial, a quantidade total, o total faturado, o faturado num mes em determinada filial, e o total de vendas efetuadas.

```
public class InfoProduto {
private int [][] quantidade; // [12][3] mes, filial
private int quantidadeTotal;
private double [][] faturado; // [12][3] mes, filial
private double totalFaturado;
private int[][] vendas;
```

4.3 Faturação Global

Módulo de dados que contém as estruturas de dados responsáveis pela resposta a questões quantitativas que relacionam os produtos às suas vendas mensais, em modo Normal (N) ou em Promoção (P), para cada um dos casos guardando o número de vendas e o valor total de faturação de cada um destes tipos. Este módulo refecencia todos os produtos, mesmo os que nunca foram vendidos, não contém qualquer referência a clientes, mas é capaz de distinguir os valores obtidos em cada filial.

4.3.1 Classe Faturacao

A classe **Faturacao** serve para guardar o número de vendas, total faturado, e os produtos vendidos, assim como os não vendidos

```
public class Faturacao {
private int totalVendas;
private double totalFaturado;
private HashMap<Produto, InfoProduto> produtosVendidos;
```

4.4 Gestão da Filial

Módulo de dados que, a partir dos ficheiros lidos, contém as estruturas de dados adequadas à representação dos relacionamentos, fundamentais para a aplicação, entre produtos e clientes, ou seja, para cada produto, saber quais os clientes que o compraram, quantas unidades cada um comprou, o mês e a filial.

Para a estruturação optimizada dos dados deste módulo de dados tivemos em atenção que pretendemos ter o histórico de vendas organizado por filial para uma melhor análise, nunca esquecendo que existem 3 filiais nesta cadeia.

4.4.1 Classe Filial

Guarda a informação de todos os cliente quer tenham comprado ou não.

```
public class Filial {
private Map<Cliente,InfoCliente> informacaoClientes;
```

4.5 Classe Hipermercado

Esta é a classe principal onde se guardam todas as informações relativas ao funcionamento do hipermercado.

```
public class Hipermercado implements Serializable {

private CatalogoClientes clientes;
private CatalogoProdutos produtos;
private Filial filiais[];
```

```
private Faturacao faturacao;
private int vendaserradas;
private int comprasnulas;
private double faturacaoglobal;
```

4.6 GereVendas

Todo o programa é inicializado nesta função, o hipermercado é inicializado nesta classe que posteriormente iniciará as restantes classes, o menu principal do programa começa a executar permitindo assim ao utilizador navegar pelo programa e executar as queries e obter os resultados que quiser consultar. Esta classe contém quais os ficheiros para leitura, assim como o caminho para eles assim como o último estado guradado.

4.7 Venda

A classe Venda simplifica os dados introduzidos, sendo que assim os dados deixam de ser apenas uma string e passam a ser uma venda concreta para o programa poder consultar livremente e fazer as devidas inserções nas estruturas de forma mais simples. Esta classe é tambem responsável pela validação de todas as vendas introduzidas.

```
public class Venda implements Serializable{

private String codigoProduto;
private String codigoCliente;
private char modoDeCompra;
private int mes;
private int filial;
private double preco;
private int quantidade;
```

5. Interface com utilizador

Nesta secção apresenta-se a interface com o utilizador, fazendo algumas considerações sobre as decisões tomadas. Ao iniciar o programa GereVendas, o utilizador tem um menu que lhe permite carregar ou não o último estado do programa, isto é o ficheiro hipermercado.dat.

```
Para poder executar o programa GereVendas tem de efetuar um primeiro carregamento dos dados.

Deseja carregar o ultimo estado do programa? Caso contario, os ficheiros sao carregados (S/N)
S

Deseja introduzir o nome do ficheiro a carregar? (S/N)  S
Introduza o nome do ficheiro a carregar sem a extensao .dat:
```

Figura 5.1: Inicio da execução do programa

Se não quiser carregar o último estado do programa carrega os ficheiros Clientes.txt Produtos.txt automáticamente e pede para escolher qual o ficheiro de Vendas que pretende escolher, se escolher a opção N (não) carrega o ficheiro de Vendas_1M:

```
Para poder executar o programa GereVendas tem de efetuar um primeiro carregamento dos dados.

Deseja carregar o ultimo estado do programa? Caso contario, os ficheiros sao carregados (S/N)
N

Carregamento default dos ficheiros...
A proceder a leitura do ficheiro C:\Users\gil\Documents\GitHub\LI3-Parte-java\data\Produtos.txt...
A proceder a leitura do ficheiro...C:\Users\gil\Documents\GitHub\LI3-Parte-java\data\Clientes.txt

Deseja introduzir o nome do ficheiro Compras a carregar? (S/N)  S

Introduza o nome do ficheiro Compras a carregar:
Exemplo : Vendas_1M
```

Figura 5.2: Escolha da opção N (não)

Depois do carregamento dos ficheiros de dados é apresentado o seguinte menu com as 5 opções:

- Carregar Programa -
- Gravar estado do programa -
- Consultas Estatisticas -
- Consultas Interativas -

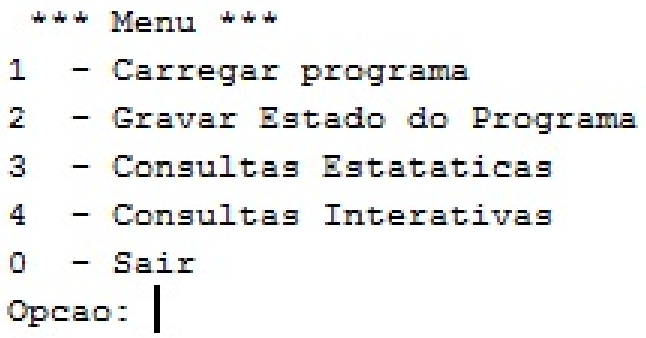


Figura 5.3: Diagrama de classe do Blue J

6. Resultados e comentários sobre os testes de performance

Depois de desenvolver e codificar todo o projeto foi-nos proposto realizar alguns testes de performance que consistem em comparar os tempos de carregamneto de ficheiros e de execução das queries 5, 6, 7, 8 e 9.

O programa foi corrido num computador ASUS k55v com um processador intel(R) core(TM) i5-3210M CPU@2.50GHz, memória RAM 6 GB onde 5.89 são utilizáveis, sistema operativos de 64 bits, processador baseado em x64.A máquima possui uma placa gráfica GEFORCE* GT 630M* 2GB e um disco rigido de HDD 500GB, o windows utilizado pela máquina é o windows 10.

6.1 Performance Leitura dos Ficheiros

Para verificar qual a melhor classe a usar para fazer a leitura dos ficheiros corremos testes para as classes **BufferedReader** e para a classe **Scanner** realizando testes fazendo parsing da string e criando instância da class **Venda** e testes sem parsing.

De seguida ficam tabelas com os resultados obtidos e também dois gráfico para comparação de resultados.

Para cada uma das situações corremos 5 vezes, para os ficheiros Vendas_1M, Vendas_3M, Vendas_5M , no final calculamos a média destas 5 execuções e foi o tempo que se registou.

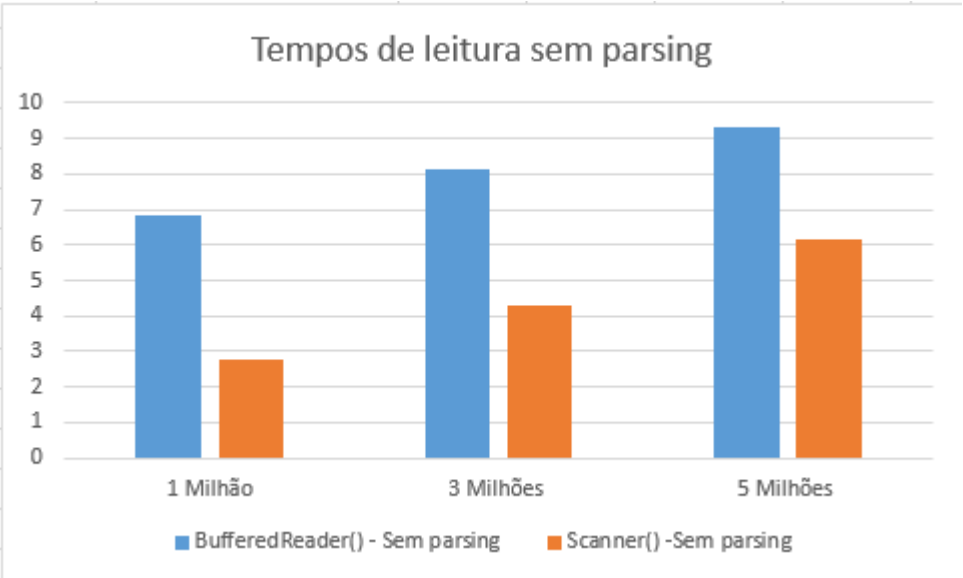


Figura 6.1: Inicio da execução do programa

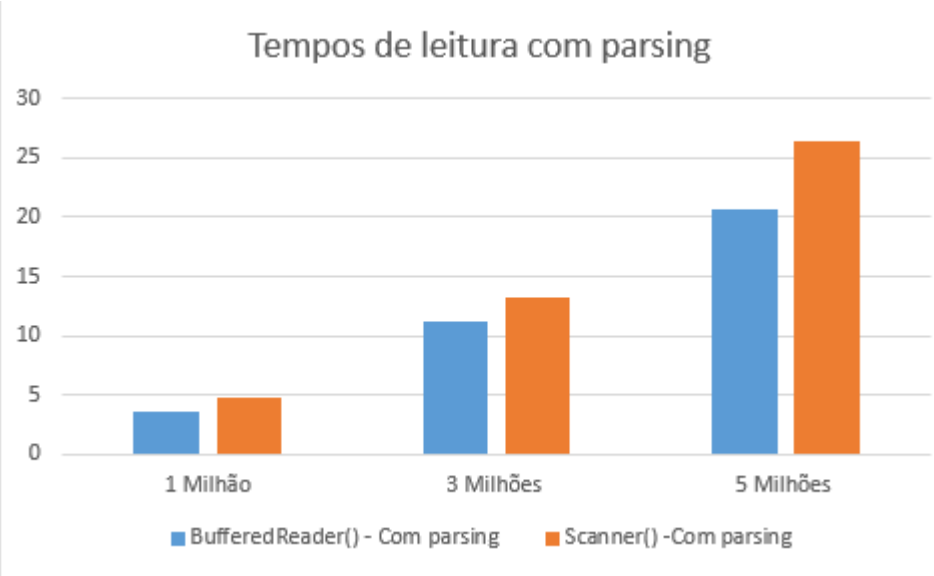


Figura 6.2: Inicio da execução do programa

Assim pode-se concluir que apesar de **BufferedReader** ser mais rápido em todas as situações, ao contrário do que era esperado por nos a class **Scanner** tem um comportamento mais eficiente durante parsing, pois é esta a sua principal utilização.

6.2 Performance Queries

Para testar a performance das queries foi proposto que fossem substituidas as estruturas inciais por outras.

No inicio a estrutura utilizada no catalogo de clientes e de produtos foi um HashSet e mudou-se para um TreeSet, registando-se 5 vezes o valor da execução e fazendo a média.

Nas classes Faturação, Filial e InfoCliente inicialmente utilizou-se uma HashMap depois mudou-se para uma TreeMap.

Foram consideradas várias situações diferentes, a primeira situação é onde a estrutura do módulo de catálogos de clientes , a class **catalogo Clientes** tem a sua estrutura definida da seguinte forma:

Na segunda situação a estrutura foi modifica para em vez de utilizar **TreeMap** utilizar um **HashMap** . Ficando então a estrutura definida da seguinte forma:

Para medir os tempos executamos as queries 6,7,8 e 9 num total de 5 medições fazendo de seguida a média.

Na query 8 utilizamos o códiGo de cliente Z5000 .

Na querie 9 utilizamos o códiGo AF1184.

De seguida apresentamos a tabela dos tempos registados seguido de um gráfico que irá ajudar na conclusão destes testes de performance das diferentes estruturas.

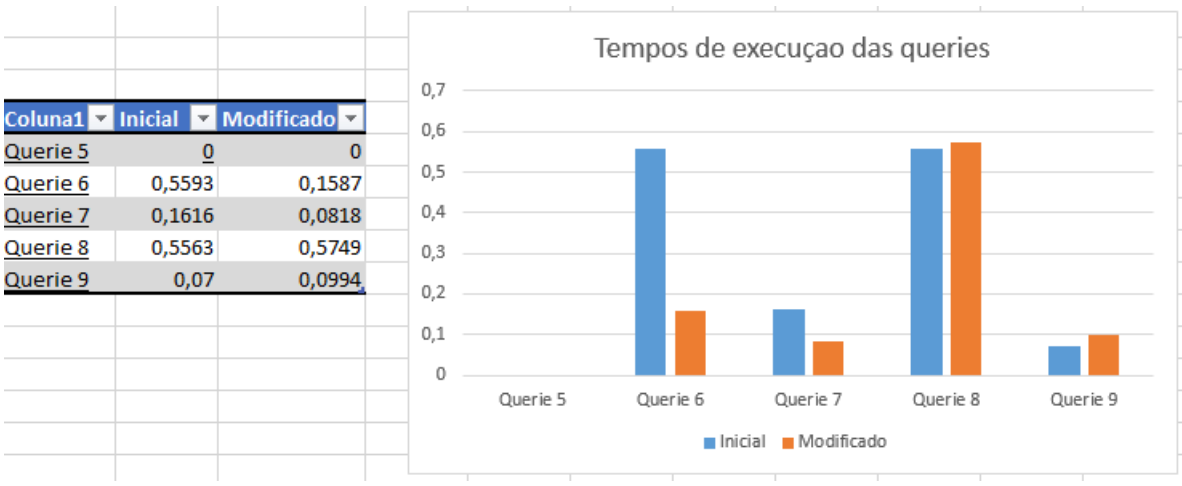


Figura 6.3: Tabela e gráfico com a execução dos tempos das diferentes estruturas utilizadas

7. Conclusão

Uma vez que se tratou de um trabalho de uma dimensão já considerável comparando com o que estávamos habituados envolveu utilização de técnicas particulares e tivemos sempre como objetivo que este trabalho fosse concebido de modo a que seja facilmente modificável, e seja, apesar da complexidade, o mais optimizado possível a todos os níveis.

A modularidade foi garantida através da criação de classes com API completa, com constructores apropriados e métodos que permitem um bom número de operações com as classes e módulos do programa. Nesse espírito, foram ainda incluidos não só nas classes dos módulos como em todas as que tal se justificava, métodos essenciais a qualquer classe "bem comportada" nomeadamente hashCode(), toString(), equals() e clone(). O encapsulamento dos módulos foi garantido através do uso de clone's ao inserir nas estruturas e ao serem procurados elementos nas mesmas. Além da modularidade e encapsulamento, uma boa estruturação do programa e legibilidade do mesmo também foram aspectos tidos em conta, o que levou à criação de tipos enumerados e métodos e variáveis com nomes sugestivos, ainda que isso tenha levado a que esses nomes fossem por vezes longos. Os tempos de resposta às queries e de leitura dos ficheiros são também bastante satisfatórios na nossa opinião. O tempo de leitura dos ficheiros é melhor que linear no tamanho dos ficheiros de compras e nenhuma querie demora mais que um minuto a ser executada, sendo que a maioria demora menos de 1 milisegundo. Estes resultados reflectem um bom planeamento da arquitectura do programa e boa escolha das estruturas usadas.

A implementação de um programa de gestão de hipermercados em Java provou ser mais fácil quando comparado com a implementação em C. Tal deve-se, em grande maioria, à necessidade de definir de raiz um grande número de funções como por exemplo estruturas. Esta vantagem de implementaçãoo permite desenvolver soluções adicionais que teriam um custo considerável em C. Assim sendo foi possível concretizar as várias queries com maior facilidade, sem prejudicar a sua velocidade, graças à facilidade em trocar e testar diferentes estruturas, desde que estas partilhem da mesma API.