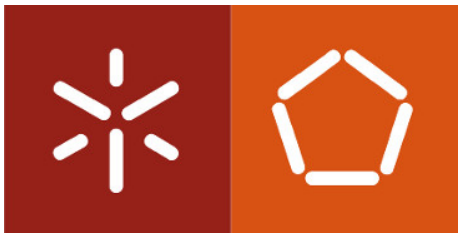
Laboratórios de Informática III

Trabalho Prático

Linguagem C

Grupo nr. 4

a84485	Tiago Magalhães
a85517	Duarte Oliveira
a89512	Manuel Novais



Mestrado Integrado em Engenharia Informática
Universidade do Minho

Conteúdo

1 Introdução

O projeto da disciplina de Laboratórios de Informática III visava a consolidação a nível teórico-prático de conhecimentos previamente lecionados em unidades curriculares deste curso. Para além do desenvolvimento de capacidades a nível individual, também se procurava introduzir novos princípios e bons hábitos na área de Engenharia de Software. Por conseguinte, esperava-se um aprofundamento dos conhecimentos associados à manipulação de grandes volumes de dados ao mesmo tempo que, e intrinsecamente, se inseria uma maior complexidade algorítmica para o tratamento destes.
Do início ao fim a evolução deste projeto acompanhou as ideias e os conhecimentos passados nas aulas práticas pelos docentes, tendo em conta que os membros deste grupo possuíam muito pouco conhecimento em relação a esta área da engenharia informática e foi graças à abordagem pedagógica que conseguiu-se desenvolver certas partes do mesmo.
Inicialmente, quando foi dado o acesso aos ficheiros de dados (Produtos, Clientes e Vendas) conseguimos criar uma imagem mais concreta da imensidão e da complexidade característica dos códigos de produtos, de clientes e de vendas. Tornou-se assim claro o porquê da necessidade primordial de uma codificação que favorecesse a performance mas também a sua confiabilidade. Ou seja, com um grande volume de dados há a clara necessidade de proceder à sua verificação de uma forma, no mínimo, razoável, evitando assim que o seu tempo de leitura seja pouco aprazível para o ser humano e que não consuma recursos desnecessários da máquina que os trata.

Em retrospetiva concluímos assim o quão necessária foi a adoção de diferentes abordagens que permitissem tirar o maior partido possível das capacidades de manipulação da memória que a linguagem C nos proporciona.

1.1 Problema e Objetivo

Como referido anteriormente, o problema levantado para este projeto era a gestão de um sistema de vendas, possivelmente uma cadeia empresarial com um vasto leque de recursos a serem vendidos em três filiais e com um extenso número de clientes, todos devidamente identificados. Nesta ótica, havia também um conjunto de vendas que representava a compra de um produto por um determinado cliente. Essa compra tinha em si identificada a quantidade adquirida, se a compra tinha sido feita em promoção ou não e em qual filial é que tinha ficado registada a compra, para além do cliente e do produto devidamente assinalados. É de notar que havia a perceção de que dentro dos dados haveria a existência de entradas inválidas, ou seja, códigos de produtos ou de clientes inválidos. Para além disso, tanto os clientes poderiam estar registados sem terem feito qualquer tipo de compra como também os produtos podiam estar identificados sem fazerem parte de qualquer tipo de vendas. Era essencial saber percorrer estes dados e selecionar os que realmente convinham para .
Após esta verificação, era necessário elaborar um conjunto de características que este sistema devesse conter, de modo a que a manipulação e o tratamento de dados pudesse invocar valores em concreto, em correlacionados com os dados disponibilizados. Com isto remete-se para um dos principais e mais trabalhosos objetivos do trabalho: a construção e desenvolvimento de *queries* apresentadas pela equipa docente, tendo sempre presente que havia a necessidade de manter o histórico de vendas organizado por filiais.

Finalmente esperava-se uma arquitetura conveniente, que de modo evolutivo e individual fosse continuamente testada, para se proceder a uma mais completa implementação e finalmente uma total acoplação, dando origem a um programa devidamente estruturado - arquitetura *Model View Controller*.
Estes e outros aspetos irão ser tratados neste manuscrito de forma mais detalhada, para uma melhor perceção do leitor.

1.2 Estrutura do Relatório

O relatório apresenta-se de forma detalhada. Cada tópico remete para uma parte em específico que foi abordada e tratada no nosso projeto. Primamos pelo seguimento delineado do que foi sugerido pelos docentes desta unidade curricular de modo a varrer o maior número de tópicos possível. Esses tópicos são devidamente descritos e identificados ao longo do comento. Optamos pelo uso de imagens tanto da esquematização de estruturas de dados como de blocos de código para corroborar as nossas afirmações e para mais facilmente fazermos compreender a nossa perspetiva e o nosso planeamento para este projeto. Na conclusão fazemos uma retrospetiva deste projeto, apresentando problemas, dificuldades, ambições e também possíveis melhorias.

2 Concepção/Desenho da resolução

Para a implementação do projeto optamos pelo uso de uma das bibliotecas usadas no C, a *Glib*, aproveitando assim a vasta série de estruturas de dados já implementadas nesta biblioteca e também pelo facto de ser multi-plataforma, o que permite que não seja preciso modificar funções noutros sistemas operativos ou compiladores, não pondo em causa a estrutura do projeto.

2.1 Estruturas de Dados

Aqui será abordado o esqueleto por nós usado para a implementação de informação neste projeto.

Estruturas onde são carregados os dados:

- Catálogo de Produtos
- Catálogo de Clientes
- Faturação
- Filial

2.1.1 Catálogos

Nos catálogos de Produtos e Clientes, optámos por guardar tanto os códigos dos produtos como o dos clientes num *array* de 26 posições. Assim sendo, cada posição do *array* corresponde a uma letra do abecedário, o que, havendo 27 letras, faz com que a correspondência seja:

$$\text{A} \rightarrow 0,\ \text{B} \rightarrow 1,\ldots,\text{Z} \rightarrow 26$$

Por sua vez cada índice deste *array* possui uma árvore binária (*AVL*). Deste modo temos acesso direto às árvores bastando usar a letra inicial de um código produto/cliente. O motivo pelo qual escolhermos o uso de *AVLs* foi devido à listagem de códigos em algumas das *queries* terem de ser por ordem alfabética. Como se sabe, as *AVLs* permitem uma inserção ordenada alfabeticamente e mesmo havendo a necessidade de verificação nas vendas para se ver se um produto ou cliente pertence ao seu respetivo catálogo e mesmo tendo em conta que a *AVL* não tem um tempo de acesso direto, a sua complexidade continua a ser razoável $O(\log n)$ tendo em conta os propósitos deste projeto.

Além disto tudo, os catálogos, nas suas estruturas, contêm ainda informação relativa aos respetivos ficheiros que carregaram as suas estruturas.

Estruturas dos Catálogos:

```
/* Estrutura de dados Catalogo de produtos */
struct cat_produtos{
GTree* produtos[Produtos]; /*Array de 26 Avl's a que cada índice deste
                           array corresponde a uma letra do abecedário.*/
int num_linhas_lidas;      /* Número de linhas lidas pelo ficheiro que
                           carregou a estrutura */
char* filename;            /* Nome do ficheiro que carregou o Catálogo */
float tempo_leitura;      /* Tempo de leitura do ficheiro */
};

/* Estrutura de dados do catalogo de clientes */
struct cat_clientes{
GTree* clientes[Clientes]; /* Array de 26 Avl's a que cada índice deste
                           array corresponde a uma letra do abecedário. */
int num_linhas_lidas;
char* filename;
float tempo_leitura;
};
```

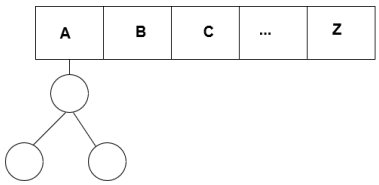


Figura 1: Representação gráfica dos catálogos

2.1.2 Faturação

Dentro da estrutura faturação optámos pelo uso de uma *HashTable*, uma vez que esta permite guardar informação dentro de cada *Value* associando a sí uma *Key*. Isto deu-nos a possibilidade de termos uma leitura do ficheiro de vendas sempre que precisamos de atualizar o valor do número de vendas realizadas ou o total faturado de um produto, basicamente possibilitando que o seu acesso seja direto. Neste caso a *Key* é o código de produto e o *Value* uma estrutura com 2 matrizes que representam os modos de compra: normal e de promoção, sendo que estas guardam o valor total faturado e número de vendas por filial e meses. Assim para a resolução da query 3 o uso de matrizes possibilita acedermos diretamente aos seus valores por filial e mês. Como auxílio, para a query 8, procedemos à criação de um array que guarda o número de vendas e o total faturado por mês.

Estrutura da Faturação:

```
/* Estrutura da faturação */
struct faturacao {
  GHashTable* produtos;   /* HashTable com código de produtos */
  Data TotaisMes[MES];    /* Informações globais acerca dos meses */
  int num_linhas_lidas;   /* numero de linhas lidas do ficheiro que contém
                             a informação a armazenar na faturação e filial */
  int num_linhas_validadas; /* numero de linhas validadas do ficheiro que
                               contém a informação a armazenar na faturação e filial */
  char* filename; /* nome do ficheiro que contém a informação a armazenar  */
  float tempo_leitura; /* tempo de leitura */
};

/* Value da HashTable produtos */
struct fat{
  char* code; /* Código do produto */
  Data infoN[MES][FILIAL]; /* Informação mês a mês e filial a filial
                              acerca do produto no modo Normal */
  Data infoP[MES][FILIAL]; /* Informação mês a mês e filial a filial
                              acerca do produto no modo Promoção */
  int vendas; /* Número de unidades vendidas global do produto */
};

/* Estrutura Data */
struct data{
  int quant; /* Número de vendas do produto */
  float precofat;  /* Total faturado pelo produto */
};
```

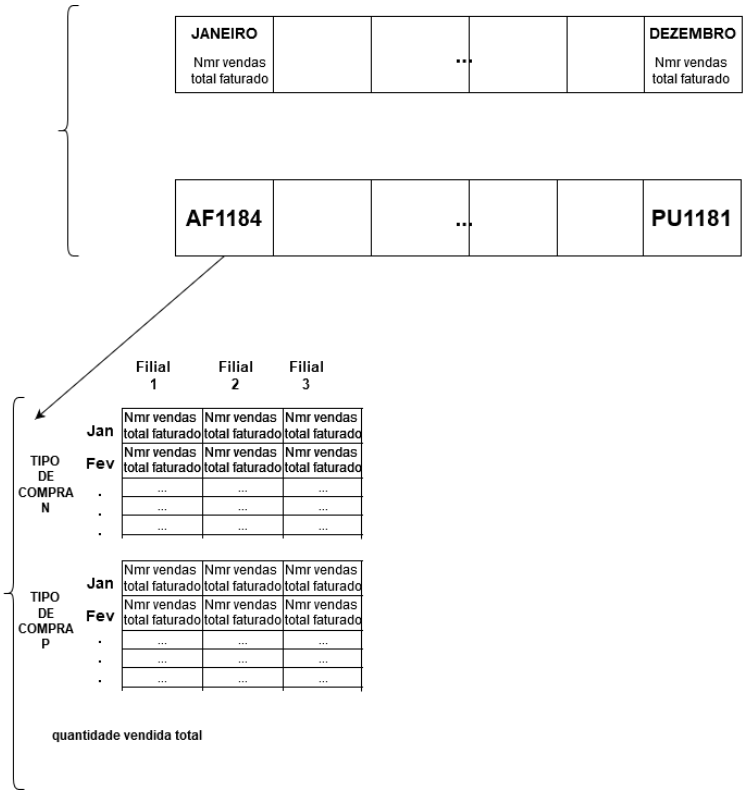


Figura 2: Representação gráfica faturação

2.1.3 Filial

Na filial, tal como na faturação, devido à necessidade de atualização de informação, optámos pelo uso de *HashTables*, uma de produtos que nos indica os produtos comprado na filial em questão - permitindo resolver a query para saber quais o produtos nunca comprados, por exemplo - sendo o *Value* o número de unidades vendidas do produto para sabermos quais os produtos mais vendidos. Também temos uma *Hashtable* de clientes que tem como *value* um *array* de meses contendo uma *hashtable* dos produtos comprados em cada mês pelo cliente e cada produto com o número de unidades compradas pelo cliente e o se já o comprou em Promoção ou normalmente - servindo de resolução de query 10. Este *Value* também tem uma *hashtable* de produtos, permitindo melhorar o tempo de resolução da query 12, uma vez que não precisa percorrer mês a mês nem encontrar produtos repetidos.

Estrutura da Filial:

```
/* Estrutura de uma Filial */
struct filial{
```

```
  GHashTable* produtos; /* HashTable de produtos comprados na filial */
  GHashTable* clientes; /* HashTable de clientes  que realizaram compras na filial*/
};

/* Estrutura que é o value da HashTable Clientes */
struct infoCli{
  GHashTable* prods; /* HashTable de produtos  que um cliente comprou */
  GHashTable* produtos[MES]; /* Array de 12 meses com uma hashtable
                              de produtos que o cliente comprou */
};

/* Estrutura InfoProd value das HashTable's do array produtos */
struct infoProd{
  gboolean modoCompra[2];//0->N 1->P /* Indica em que modos o cliente
                                       comprou o produto */
  int qnt; /* Indica o número de unidades comprados pelo cliente de um produto */
};
```

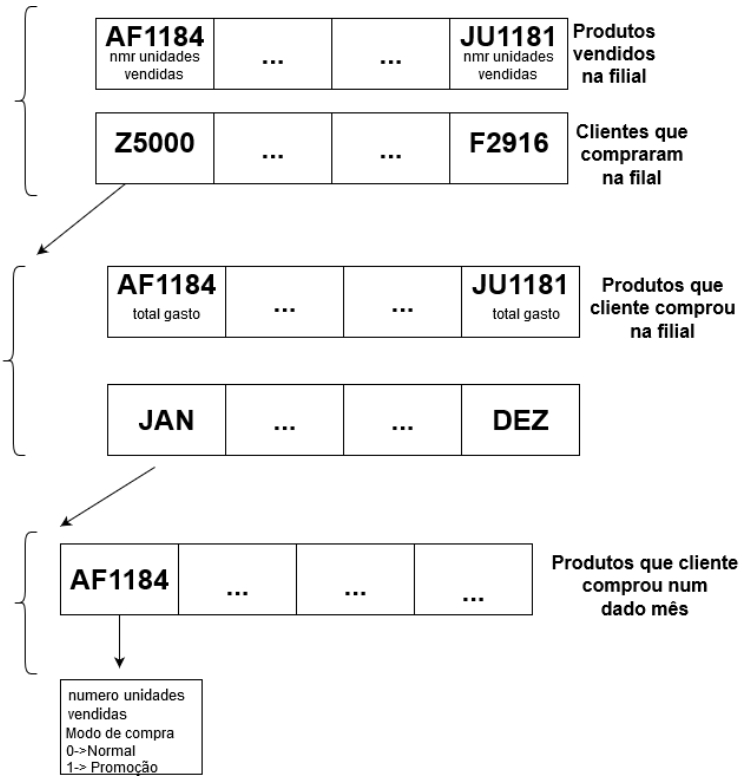


Figura 3: Representação gráfica filial

2.2 Estruturas Auxiliares

Como estruturas auxiliares, de maneira a existir um navegador que permitisse apresentar por páginas códigos, usamos uma estrutura denominada Lista. Esta tem na sua representação um *GArray* de *char** como se fosse uma Lista de *strings*. Também utilizamos uma*GList* que nos permitiu resolver algumas *queries* devido à possibilidade de ordenar informação de diferentes maneiras. Para além disso foram usadas estruturas com base em *arrays* de *structs* com inteiros e *floats* para representar as respostas a determinadas *queries*.

2.3 Modularidade

O projeto encontra-se distribuído por módulos, visto que uma vez que trabalha com grande volume de dados, torna-se fundamental organizar o programa desta forma. A utilização de módulos também facilita com que várias pessoas possam implementar diferentes partes do projeto, agilizando o nosso processo de manutenção, reconstrução e deteção de erros em código. Para além disso, a modularidade é crucial para uma fácil otimização das funções desenvolvidas no decorrer do mesmo, também contribuindo para a boa gestão de um grande volume de dados.O nosso projeto segue a arquitetura *Model-View-Controller* (MVC), contendo os seguintes módulos presentes no projeto:

1. model
   (a) CatClientes - contém funções de inicialização,remoção e inserção no Catálogo;
   (b) CatProds - contém funções de inicialização,remoção e inserção no Catálogo;
   (c) Navegador - contém funções para fornecer páginas,e inserir em listas;
   (d) Filial contém funções de inicialização,remoção da Filial;
   (e) Faturacao contém funções de inicialização,remoção e inserção na Faturação;
   (f) Queries - contém funções e estruturas de resposta às queries;
   (g) SGV - contém funções de inicialização,remoção do SGV e queries;
   (h) Venda - contém funções de inicialização e criação de vendas;
   (i) Files - contém funções que carregam as estruturas.
2. controller
   (a) Controller - contém funções de interação utilizador sistema;
   (b) Input - permite validar e receber Input's do utilizador;
3. view
   (a) View - funções que apresentam menus e respostas às queries.

Mais documentação sobre as API's encontra-se no repositório.

2.4 Encapsulamento

A garantia de proteção e acesso controlado a dados foi obtida através da: Declaração de tipos opacos que escondem a informação.

```
/**
 * @brief Declaração do tipo opaco Cat_Clientes.
 */
typedef struct cat_clientes *Cat_Clientes;
```

Figura 4: Tipo opaco no projeto

O uso da keyword *static* que torna a funções acessíveis apenas no módulo onde estão declaradas.

```
static int compare(gpointer a,gpointer b){
```

Figura 5: Funções privadas ao módulo

O uso de *"clones"* para evitar o retorno de apontadores para estruturas internas.

```
GList* clone=g_list_copy_deep(l,cloneTopProds,NULL);
```

Figura 6: Clone

```
gpointer cloneTopProds(gconstpointer src,gpointer data){
  TopProds tp=(TopProds) src;
  TopProds clone=initTopProds();
  float gasto=getGastoTop(tp);
  char *code=getCodeTop(tp);
  clone=setTopProds(clone,code,gasto);
  free(code);
  return clone;
}
```

Figura 7: Clone

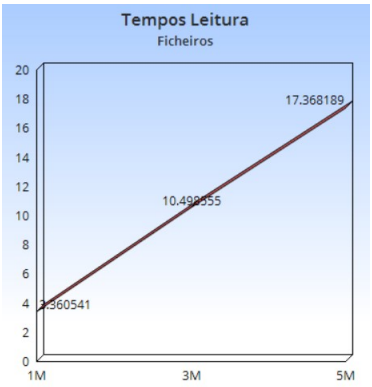
O encapsulamento também foi garantido,através de *"getters"*, assim os atributos só são acedidos através destas funções e não diretamente fora dos seus módulos.

```
char* getFileNameClientes(Cat_Clientes cc);
```

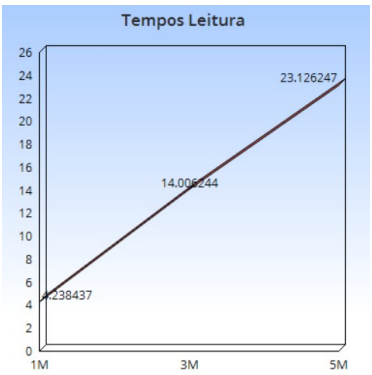
Figura 8: Getter

3 Testes de desempenho

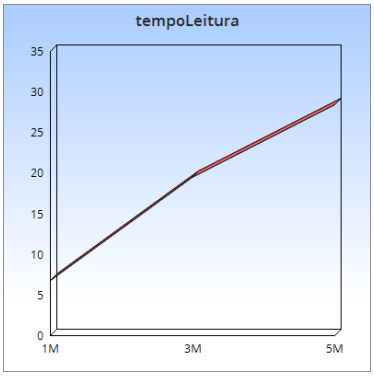
3.1 Tempos de Leitura e Queries



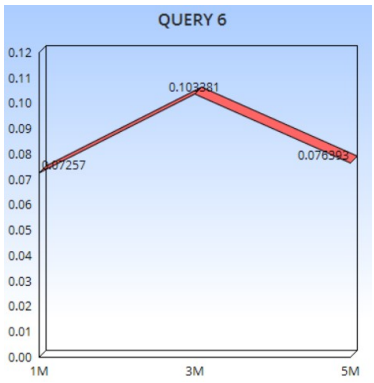
(a) Tempos Leitura PC1



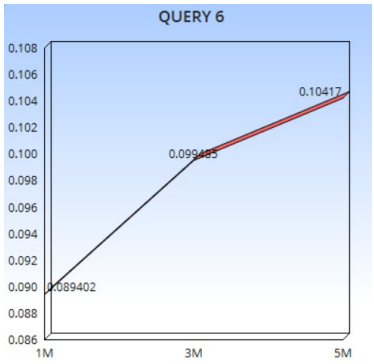
(b) Tempos Leitura PC2



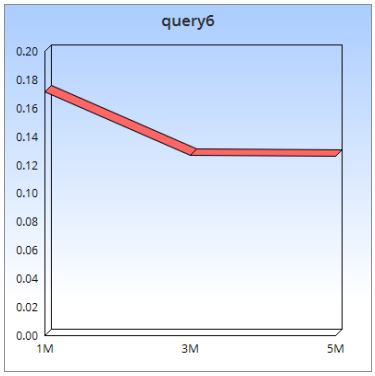
(c) Tempos Leitura PC3



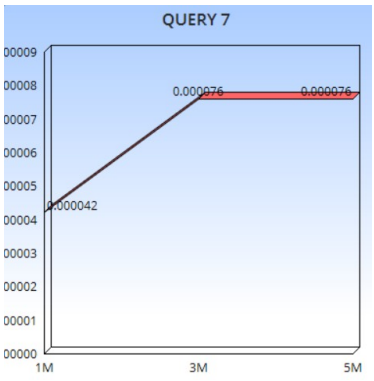
(a) Query 6 PC1



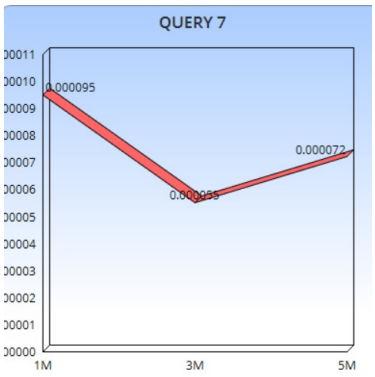
(b) Query 6 PC2



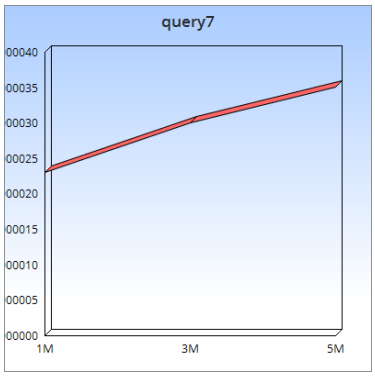
(c) Query 6 PC3



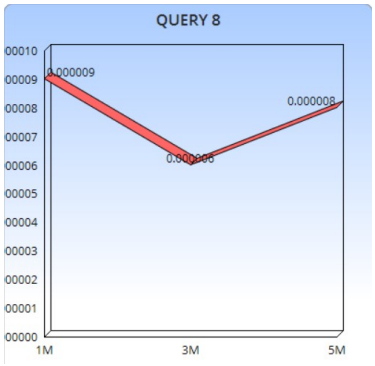
(a) Query 7 PC1



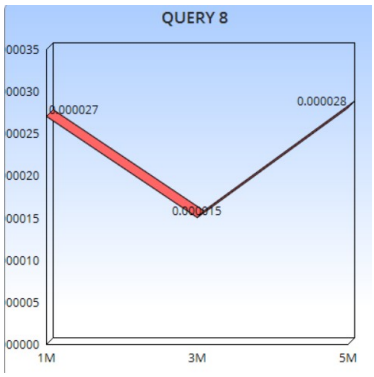
(b) Query 7 PC2



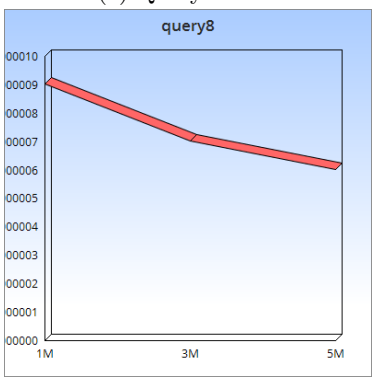
(c) Query 7 PC3



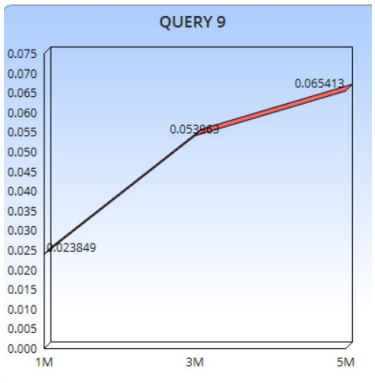
(a) Query 8 PC1



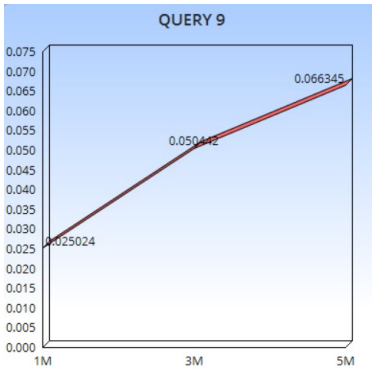
(b) Query 8 PC2



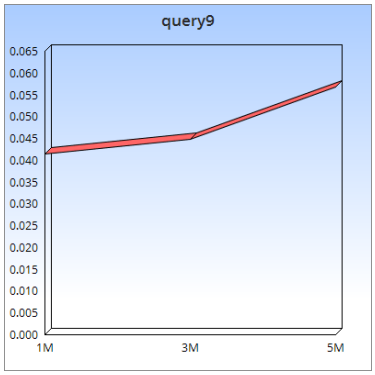
(c) Query 8 PC3



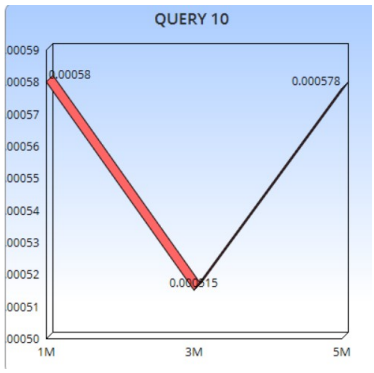
(a) Query 9 PC1



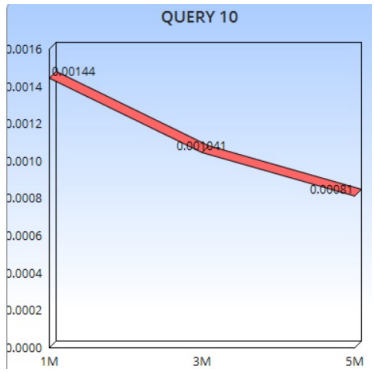
(b) Query 9 PC2



(c) Query 9 PC3

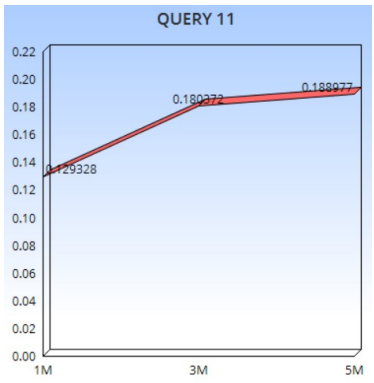


(d) Query 10 PC1

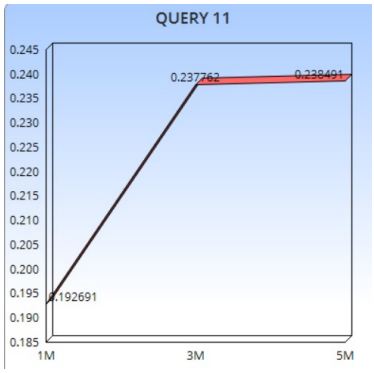


(e) Query 10 PC2

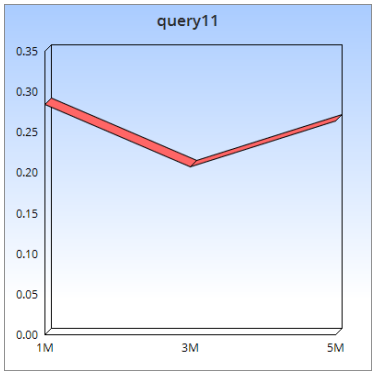
(f) Query 10 PC3



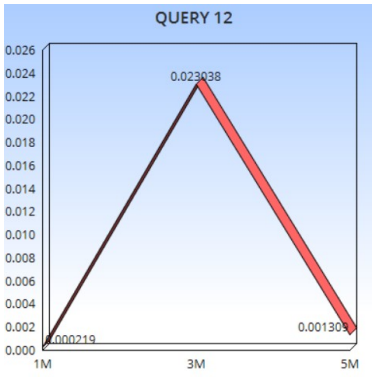
(a) Query 11 PC1



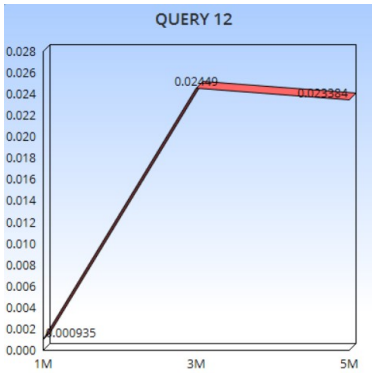
(b) Query 11 PC2



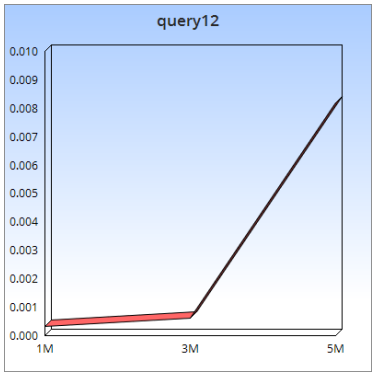
(c) Query 11 PC3



(d) Query 12 PC1



(e) Query 12 PC2



(f) Query 12 PC3

3.2 Gestão de memória

Para evitar memory leaks e acessos inválidos a memória foram realizados testes com a ferramenta Valgrind.

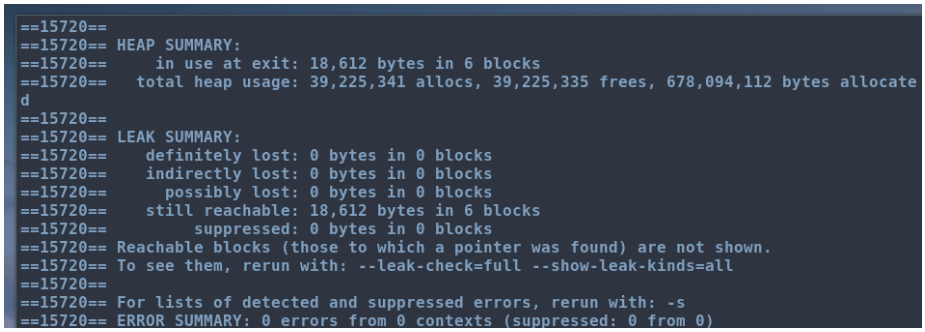


Figura 15: Utilização do Valgrind

4 Dificuldades

Houve uma grande evolução deste projeto ao longo desta primeira parte do semestre, muito graças às orientações dadas nas aulas e aos vídeos partilhados pelos docentes. Independentemente disso é inevitável referir algumas dificuldades sentidas neste projeto. Em primeiro lugar, a biblioteca *Glib* trouxe alguns problemas na compilação do programa, por erros de compatibilidade com as *flags* presentes na *Makefile* do projeto ou mesmo por causa da partição *linux* que os elementos do grupo possuíam. O *github* também foi motivo para alguns retrocessos no nosso projeto, e aproveitamos este espaço para referir que o repositório oficial deveria ter sido disponibilizado muito mais cedo, sentimos que algumas das dificuldades que sentimos - essencialmente problemas de *merge* e conflitos - poderiam ter sido colmatadas muito mais cedo. Para além disso tudo, ao longo do projeto em si, houve problemas com o uso de memória que o programa utilizava levando a que problemas de execução começassem a ocorrer na máquina que o executava. Finalmente tratou-se de *leaks* memória usando a plataforma *valgrind*, sendo para nós revelador de alguns erros que tínhamos tido feito até então. Obviamente que todos estes problemas conseguiram ser resolvidos, custando um pouco de tempo e alguma frustração ao grupo, mas trazendo sempre algo de positivo.

5 Conclusão

É para nós importante referir que se dedicou muito tempo não só à qualidade das *queries*, mas também à organização do projeto em si. De modo a tornar as queries instantâneas existiu um trade off com a alocação de memória que tornava os tempos de leitura maiores. Tendo isto em conta, foi feito um programa o mais fluído possível, a nível de performance, sendo que testes para verificação de tempos foram feitos vezes sem conta procurando obter os melhores tempos possíveis de leitura dos ficheiros e das *queries*. Por conseguinte, e noutra variante, este é um programa que na sua plenitude é bastante sólido, evitando ter que carregar o programa vezes sem conta aquando erros de *input* do *user*.Em qualquer outro caso foi-se recorrentemente testando as funções e o *"menu"* no qual o utilizador iria interagir, ao mesmo tempo procurando *bugs* e outros pequenos problemas que pudessem afetar esta fluidez.
Para além disto tudo procuramos manter um código legível, indentado e acima de tudo documentado. A documentação deste projeto foi gerada pelo programa *doxygen* e os seus comentários foram detalhadamente inseridos. Esta documentação está visível para todos os que tenham acesso ao projeto, na pasta *DOCS*.

Findamos referindo a nossa vontade de criar uma *config* que aceitasse um número N de filiais por exemplo. Sabíamos que conseguíamos fazê-lo mas preferimos tornar prioritário e efetivo aquilo que os docentes nos propunham.