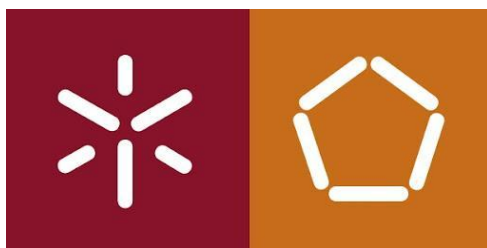


Universidade do Minho
Licenciatura em Engenharia Informática



LABORATÓRIOS DE INFORMÁTICA III

Guião 3

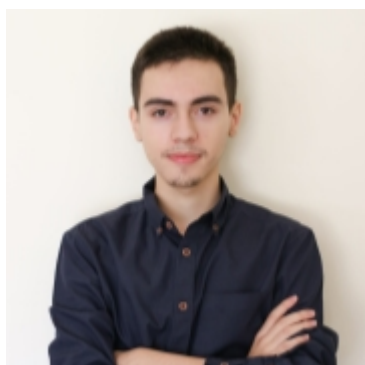
Trabalho realizado por :

Grupo 10

Nuno Guilherme Cruz Varela - a96455

Gabriela Santos Ferreira da Cunha - a97393

Miguel de Sousa Braga - a97698



a96455
Guilherme Varela



a97393
Gabriela Cunha



a97698
Miguel Braga

1. Introdução

Na terceira e última fase do projeto, foi-nos solicitada a resolução de três exercícios: a implementação de um mecanismo de interação pelo terminal, a realização de testes funcionais e de desempenho às *queries* e o desenvolvimento de um mecanismo eficiente de gestão de dados que permitisse algumas otimizações posteriores.

Estes novos módulos do nosso projeto teriam de manter as boas práticas evidenciadas pela fase anterior do projeto, tais como a modularidade e o encapsulamento do código.

Com os ficheiros disponibilizados para este novo guião a crescerem bastante, ficou claro que o nosso foco para esta fase seria a eficiência na resposta às diversas *queries*. Estipulamos, por isso, como objetivo um tempo máximo de 5 segundos para cada uma das *queries*, algo referido também no enunciado. No entanto, não deixamos de otimizar cada uma das *queries* o máximo que nos fosse possível. Mais à frente discutiremos os resultados obtidos.

2. Tratamento de dados

Como nesta fase do trabalho os ficheiros de entrada iniciais podem conter registos inválidos, voltamos a adicionar as funcionalidades do primeiro guião ao nosso *parser*. Com isto, adaptamos a nossa função de *parsing* do segundo guião de forma a validar as linhas e, ao mesmo tempo, fazer o cruzamento de dados entre ficheiros. Relativamente ao *data crossing*, decidimos adicionar uma passagem adicional pelos dados relativamente às que foram referidas no enunciado do primeiro guião, de modo a tornar este cruzamento totalmente válido (sem os repositórios que foram considerados válidos por conter *commits* posteriormente considerados inválidos). Com o objetivo de evitar futuras passagens pelos catálogos, mantivemos o cálculo das *queries* estatísticas em tempo de *parsing*.

3. Gestão de dados

Depois de algumas falhas no guião 2 relativas ao encapsulamento dos dados, começamos neste guião por implementar o mecanismo de gestão de dados ao nível dos catálogos, resolvendo simultaneamente os erros que tinham ficado do segundo guião.

Esta fase revelou-se, mais tarde, como a mais trabalhosa e crucial no nosso trabalho, visto que era a primeira vez que geríamos dados de enorme

dimensão, tendo de guardar estes parcial ou totalmente em disco. Após alguns testes e tentativas de implementação, ficou claro que a organização dos diferentes catálogos como simples *hashtables* não seria prática nem eficiente. Para além disso, uma das falhas da nossa implementação dos catálogos para o guião 2 foi o facto de os dados não estarem ordenados da forma mais conveniente à execução das *queries*. Relembramos que no guião 2 organizamos o catálogo dos *commits* por *repo_id*, de modo a que para cada *repo_id* existisse uma lista ordenada por comprimento das mensagens do *commit*, o que favorecia apenas a execução da *query* 10. Uma ordenação temporal dos *commits* poderia tornar-se muito mais favorável à realização das *queries*, uma vez que a maior parte delas envolvia datas. Como tal, pensamos numa solução que garantisse tempos de acesso bastantes reduzidos e que, simultaneamente, evidenciasse a ordenação dos dados. Com isto em mente, elaboramos os novos catálogos da seguinte forma:

- **Catálogo dos Users:**

Uma vez que os ficheiros que estamos a tratar têm uma dimensão bastante grande, decidimos guardar a informação útil dos utilizadores num ficheiro, de modo a não correremos o risco de a memória corromper devido à sua elevada utilização. Para além disso, ponderando sobre a maneira mais eficiente de resolver as *queries*, chegamos à conclusão que a ordenação destes dados seria irrelevante, pois este catálogo serviria essencialmente para consulta de dados. Assim, procuramos a melhor forma de aceder aos dados em disco, optando por resolver este problema com a criação de uma *hashtable* (em memória) que guardasse os índices de cada utilizador no ficheiro. Com esta adaptação conseguimos reduzir a quantidade de memória utilizada substancialmente. Se, por um lado, antes guardávamos em memória o *id* (4 *bytes*), o *login* (1 *byte* * tamanho da *string*) e o tipo (1 *byte*), agora apenas precisamos de guardar o *id* e o índice (4+4 = 8 *bytes*). Num ficheiro com milhões de entradas, esta diferença torna-se de facto muito significativa. Contudo, um problema surgiu: onde guardar as listas de *followers* e *following*? Não encontrando nenhuma forma eficiente de guardar esta informação em disco (devido à existência de listas bastante consideráveis), decidimos guardar em memória, recorrendo a uma *hashtable*, tal como fizemos para o guião 2. De forma a termos acesso ao tamanho do ficheiro em número de *bytes*, decidimos também incluir um inteiro na estrutura do catálogo para esse efeito, algo que nos ajuda a saber os índices em *bytes* dos utilizadores no ficheiro.

```

struct cat_users{ /** Users catalogue **/
    FILE *hash; /** Where the hashtable will be stored **/
    GHashTable *index; /** Index from the users' positions on file **/
    int file_size; /** Users' file size **/
};

```

Figura 1 - Estrutura que representa o catálogo dos utilizadores

- **Catálogo dos Commits:**

Já tínhamos chegado à conclusão, no guião 2, que este era o catálogo mais importante para a realização das *queries*, pois quase todas requeriam uma travessia por este conjunto de dados. Contudo, a estratégia utilizada no guião 2 não se revelou a mais acertada, devido à falta de ordenação por datas, essencial para a resolução eficiente de metade das *queries*.

Tendo isso em mente, elaboramos este catálogo ordenando os *commits* dentro do mesmo repositório por datas por ordem decrescente, ou seja, o *commit* mais recente em primeiro. Trata-se de um compromisso entre a ordenação total de todos os *commits* por datas e o facto de todos os *commits* do mesmo repositório estarem em posições contíguas, algo que também favorece a realização de algumas *queries*.

De modo a ordenar os dados, recorremos a uma lista ligada que em tempo de parsing vai armazenando os *commits* adicionados. Finda a fase de *parsing*, os dados são enviados para ficheiro, guardando-se em memória, também numa *hashtable*, as posições correspondentes a cada repositório - seguido dos respetivos *commits* - no ficheiro. De forma a permitir “saltar” para o próximo repositório a partir do repositório anterior, guardamos ainda em ficheiro o número de commits associados a um dado repositório. Em algumas das *queries* esta informação ser-nos-á útil quando deixarmos de processar os *commits* de um dado repositório.

Apesar do potencial *overhead* em termos de gastos de memória com a lista temporária, esta solução permitiu-nos poupar sensivelmente 22 *bytes* por cada *commit*. Excluímos desta contagem o total de *bytes* gastos com o armazenamento em memória das posições de cada repositório no ficheiro. Apesar disso, o número de repositórios é bastante inferior ao número de *commits*, logo esta aproximação não é de todo desadequada.

```

struct cat_commits{ /** Commits catalogue **/
    FILE *hash; /** Where the hashtable will be stored **/
    GHashTable *index; /** Index from the commits' positions on file **/
    GList *aux_list; /** Array that will be sorted after all insertions and sent to the file **/
    int file_size; /** Commits' file size **/
};

```

Figura 2 - Estrutura que representa o catálogo dos commits

- **Catálogo dos Repositórios:**

Assim como no catálogo dos utilizadores, a informação útil não precisaria de estar ordenada no ficheiro, pois este catálogo serviria também como meio de consulta de alguns dados para as *queries*. De entre todos os catálogos este foi o que permitiu poupar mais memória, devido à existência de dois campos com o formato *string* (descrição e linguagem), *strings* essas que poderiam ser bastante extensas, como vemos na descrição de alguns repositórios. Assim, conseguimos poupar $(4 + \text{sizeof}(\text{descrição}) + \text{sizeof}(\text{linguagem}) \text{ bytes})$ com esta atualização.

Desta forma, tal como fizemos para o catálogo dos utilizadores, inserimos os dados dos repositórios em ficheiro com auxílio de uma *hashtable* que nos indica os índices de cada repositório.

```
struct cat_repos{ /** Repos catalogue **/  
    FILE *hash; /** Where the hashtable will be stored **/  
    GHashTable *index; /** Index from the repos' positions on file **/  
    int file_size; /** Repos' file size **/  
};
```

Figura 3 - Estrutura que representa o catálogo dos repositórios

4. Adaptação de módulos e *queries*

Tal como no guião 2, optamos por continuar a usar uma estratégia semelhante para a resolução das *queries*, onde criamos uma *hashtable* temporária e, no final, inserimos ordenadamente os dados da *hashtable* numa lista para devolver o *output*.

- **Query 5:**

Com a nova implementação do catálogo dos *commits*, a resolução desta *query* torna-se muito mais fácil e eficiente a nível computacional. Como agora os *commits* estão organizados por repositórios e, posteriormente, por datas, basta verificar se a data do *commit* é anterior ao intervalo de datas passado como argumento. Em caso afirmativo, não necessitamos de percorrer os seguintes *commits* do repositório atual e podemos passar diretamente para o próximo, o que nos poupa imenso trabalho. Caso contrário, fazemos o incremento na respetiva *hashtable*.

- **Query 6:**

Também na implementação da *query* 6, usufruímos da nova atualização do catálogo dos *commits*. Percorremos, então, o ficheiro dos *commits* e para cada repositório verificamos se é da linguagem passada como parâmetro. Se não

for, passamos diretamente para o próximo repositório e escusamos de percorrer os restantes *commits* associados ao repositório atual.

- **Query 7:**

No guião 2, para cada repositório, verificávamos se existia algum *commit* após a data indicada no *input*. Para este guião, ao fazermos a pesquisa do *commit*, trabalhamos automaticamente com o mais recente associado ao *repo_id*. Se o *commit* for após a data limite (data do *input*) então existe pelo menos um *commit* depois da data, o que torna o repositório ativo. Por outro lado, se o *commit* for antes da data limite, sabemos que todos os *commits* efetuados anteriormente são também antes dessa data limite. Desta forma, podemos determinar se o repositório é inativo ou ativo através de uma avaliação a um único *commit*, o *commit* mais recente.

- **Query 8:**

Esta *query* beneficiou da nova implementação pois, mais uma vez, apenas precisamos de verificar se o *commit* mais recente é posterior à data do *input*, visto que, em caso negativo, descartamos imediatamente o repositório associado a esse *commit* e procedemos à análise do próximo.

- **Query 9:**

Relativamente ao guião 2 pouco foi alterado relativamente à resolução desta *query*. Esta envolve uma travessia completa pelo catálogo dos *commits*, recorrendo ao módulo “counter_hash” de forma a contar o número de *commits* em repositórios cujo *owner* é um amigo do utilizador que faz o *commit*. Uma otimização adicional que fizemos nesta *query* foi guardar os resultados da ordenação completa destes utilizadores num ficheiro para nas seguintes invocações não termos de realizar todo o trabalho repetido de percorrer os *commits*.

- **Query 10:**

Com a atualização do catálogo dos *commits*, este deixou de estar organizado pelo tamanho da mensagem de *commit* de cada utilizador. Isto motivou a necessidade de ordenar (em tempo de execução) os *commits* de cada repositório, sendo estes resultados enviados para um ficheiro auxiliar. Nas próximas invocações desta *query*, apenas teríamos que atravessar o dito ficheiro, selecionando as N primeiras entradas de cada repositório.

- **counter_hash.c :**

Uma das alterações feitas neste módulo, em relação ao guião 2, foi o facto de podermos agora trabalhar com *inteiros* nestas estruturas, uma vez que apenas mantivemos uma *string* como *key* na resolução da *query* 8. A inserção é feita na *hashtable* e a ordenação e consequente impressão é feita em listas ligadas. A transformação para listas ($\Theta(N)$) representa um custo computacional adicional neste problema. Contudo, este *overhead* é compensado pela rápida inserção na *hashtable*.

5. Mecanismos de interação

Para além do mecanismo de interação desenvolvido no guião anterior, foi-nos pedido agora o desenvolvimento de um mecanismo alternativo, através de um menu interativo. Para tal, quando o programa é invocado sem argumentos, é perguntado ao utilizador qual a *query* que pretende executar e os determinados dados de *input* através do terminal. Em função da opção escolhida, a *query* é executada e a lista resultado produzida será passada à função *display_page*, que trata de mostrar ao utilizador um número de linhas fixo (5) provenientes da lista. Esta função permite ainda ao utilizador avançar, recuar e saltar de página. Se o programa é invocado com argumentos, recorreremos ao módulo “command_interpreter” desenvolvido no guião 2, apenas responsável pela chamada e impressão das *queries* para ficheiros *output* do tipo “.txt”.

6. Testes funcionais e de desempenho

De forma a testar e a avaliar os resultados e o desempenho do programa desenvolvido foi realizado um módulo de testes capaz de executar cada uma das *queries* apurando a correção do resultado obtido e o tempo de execução, desejavelmente inferior a 5 segundos.

Para isso, criamos um executável independente que, com recurso à biblioteca “time.h”, calcula o tempo que cada *query* demora a ser executada. Para além disso, criamos uma pasta com os resultados esperados para a execução de cada um dos testes, obtidos numa fase anterior à entrega do trabalho, tendo por base resultados calculados pelo executável deste guião. Para comparar resultados obtidos e esperados, criamos uma função que compara linha a linha o conteúdo de dois ficheiros.

7. Resultados e Discussão

Em relação à comparação dos *outputs* obtidos com os *outputs* esperados, devido à diferente organização dos dados nos catálogos, muitos dos resultados apresentavam diferenças em pontos onde vários resultados seriam possíveis. Por exemplo, a listar as N maiores mensagens de *commit* por repositório, poderia dar-se o caso de haver vários utilizadores com mensagens de *commit* do mesmo tamanho, aparecendo com ordens diferentes no guião 2 e no guião 3. Todavia, escrutinadas essas diferenças, acreditamos que os resultados são convergentes, o que traz mais garantias acerca da correção do nosso trabalho.

Apresentamos, em seguida, a comparação entre os tempos obtidos para a execução das *queries*. É importante referir que executamos as *queries* 10 vezes e os resultados exibidos provêm da média dessas 10 execuções.

Primeiramente, analisamos a comparação entre os resultados do guião 2 e o guião 3, fornecendo os mesmos *inputs*, de modo a testar a eficiência das otimizações implementadas.

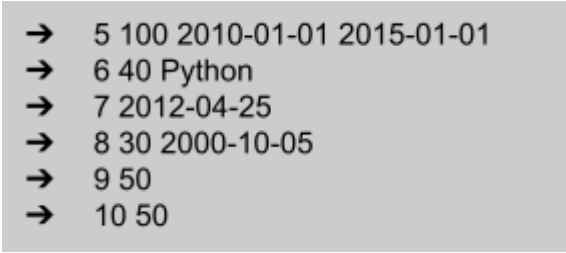
Em seguida, expomos os resultados obtidos no guião 3 para cada membro do grupo, de forma a analisar a divergência dos resultados quando obtidos através de componentes computacionais diferentes.

● Performance antes *versus* depois da otimização

De modo a avaliar o ganho de performance com as otimizações, comparamos os tempos de execução das *queries* do guião 2 com o guião 3. Para isto, utilizamos um computador com as seguintes especificações:

- CPU : AMD Ryzen 3600 6-Core 3.6 GHz
- DISCO : SSD Western Digital Blue 1TB 3D M.2
- RAM : 16GB DDR4 3200MHz
- SO : Ubuntu 20.04.3 LTS

Os comandos executados para todas as seguintes *queries* foram:



```
→ 5 100 2010-01-01 2015-01-01
→ 6 40 Python
→ 7 2012-04-25
→ 8 30 2000-10-05
→ 9 50
→ 10 50
```

Figura 4 - Inputs das *queries* para os testes

Recolhemos os seguintes dados:

	Tempos de execução (s)	
	Antes	Depois
Query 5	21,4637	1,2990
Query 6	4,6962	0,8029
Query 7	0,3717	0,6313
Query 8	0,3645	2,2499
Query 9	1,2099	0,2911
Query 10	1,2213	0,2874

Figura 5 - Resultados obtidos na execução do guião 2 e do guião 3

Através destes resultados, podemos avaliar como positivos os melhoramentos efetuados nesta fase do projeto. Houve ganhos bastante consideráveis nas *queries* 5, 6, 9 e 10, fruto das otimizações adicionadas. Já nas *queries* 7 e 8 houve um aumento dos tempos de execução, em parte porque estas já apresentavam tempos bastante razoáveis numa medição do guião 2. De notar, também, que a gestão e tratamento dos dados no guião 2 foram realizados em memória, enquanto que, nesta última fase, acedemos aos dados em disco, um método mais lento, dado que os acessos à memória são muito mais rápidos. De maneira geral, os ganhos são bastante positivos, facto que evidencia e enaltece as otimizações feitas nos catálogos nesta parte final do projeto.

Em relação aos tempos de execução obtidos, estes apresentam-se na tabela abaixo, juntamente com os ambientes de execução de cada elemento. Relembramos que estes tempos (em segundos) são a média de uma sequência de 10 execuções, excluindo o melhor e o pior resultado, por cada membro do grupo:

Ambientes de execução		
Guilherme	Gabriela	Miguel
<u>CPU</u> : AMD Ryzen 3600 6-Core 3.6 GHz <u>DISCO</u> : SSD Western Digital Blue 1TB 3D M.2 <u>RAM</u> : 16GB DDR4 3200MHz <u>SO</u> : Ubuntu 20.04.3 LTS	<u>CPU</u> : Intel Core i7-5500U Dual Core 2,4 GHz <u>DISCO</u> : SSD PNY CS900 240GB <u>RAM</u> : 8GB <u>SO</u> : Ubuntu 20.04.3 LTS	<u>CPU</u> : Intel Core i5-8265U 8-Core, 1.6 GHz <u>DISCO</u> : SSD TOSHIBA-RC500 250 GB <u>RAM</u> : 32GB <u>SO</u> : Ubuntu 20.04.3 LTS

Figura 6 - Tabela com os ambientes de execução de cada elemento

	Tempos de execução (s)		
	Guilherme	Gabriela	Miguel
Query 5	1,2990	2,4251	1,7212
Query 6	0,8029	1,9559	1,4186
Query 7	0,2874	1,5395	1,0019
Query 8	2,2499	5,5571	3,7586
Query 9	0,2911	0,8406	0,6236
Query 10	0,2874	0,5158	0,7295

Figura 7 - Resultados obtidos na execução do guião 3 por membro do grupo

Vale a pena mencionar, também, que as primeiras execuções para as *queries* 9 e 10 são mais lentas, por terem de carregar os dados para ficheiro e as seguintes mais rápidas, pois parte do processamento já se encontra feito.

Estes resultados vão de encontro à meta estipulada no início deste guião de menos de 5 segundos para cada *query*.

8. Testes à memória

De modo a manter a boa prática adotada nos guiões anteriores ao testar a libertação da memória alocada, utilizamos, mais uma vez, a ferramenta *valgrind*, de forma a identificar e corrigir eventuais *memory leaks* do código.

```
==18041== HEAP SUMMARY:
==18041==    in use at exit: 69,542 bytes in 138 blocks
==18041== total heap usage: 188,232,514 allocs, 188,232,376 frees, 5,234,216,593 bytes allocated
==18041==
```

```
==18041== LEAK SUMMARY:
==18041==    definitely lost: 512 bytes in 128 blocks
==18041==    indirectly lost: 0 bytes in 0 blocks
==18041==    possibly lost: 0 bytes in 0 blocks
==18041==    still reachable: 69,030 bytes in 10 blocks
==18041==    suppressed: 0 bytes in 0 blocks
```

Figura 8 - Relatório da memória usada para a execução de todas as queries

9. Conclusão

Com este trabalho fomos introduzidos a uma série de conceitos fundamentais para o desenvolvimento correto e estruturado de aplicações em C (e em outras linguagens). Não estávamos de todo familiarizados com as boas práticas que são o encapsulamento e a modularidade do código. Contudo, o balanço que fazemos depois destes 3 guiões é muito positivo, pois acreditamos que ficamos muito mais sensibilizados em relação a estes conceitos.

Um outro aspeto em que ficamos a ganhar com este trabalho foi na gestão de grandes quantidades de dados, quer em memória, quer em ficheiro. Pudemos experimentar as vantagens e as desvantagens da utilização de diversas estruturas de dados em memória para o armazenamento de informação estruturada e como guardar de forma eficiente dados em ficheiro para serem utilizados em tempo de execução.

Entendemos a necessidade de realizar testes ao código, num processo modular em que, após a elaboração de um dado módulo, este era testado de forma a inseri-lo no projeto. Foram realizados diversos testes de desempenho que nos levaram a procurar maneiras de aumentar a eficiência do código sem colocar em causa o resultado final.

Tudo isto não seria possível sem o excelente trabalho de equipa que o nosso grupo demonstrou, tendo sido esse um dos grandes ingredientes para o sucesso do nosso trabalho.

Apesar de tudo isto, reconhecemos que o nosso trabalho não foi perfeito, havendo alguns aspetos que podíamos vir a melhorar com tempo adicional, como aumentar a eficiência na resolução das *queries* e realizar um maior número de testes funcionais às mesmas. Também a elaboração de um menu mais gráfico e a introdução de paralelismo ao nível das *threads* foram objetivos para este guião que, infelizmente, acabamos por não cumprir com a sua implementação.