Relatório do Guião III

Laboratórios de Informática III

Grupo 73

Guilherme Sampaio, a96766

Tiago Oliveira, a97254

Miguel Neiva, a92945

Licenciatura em Engenharia Informática

5 de fevereiro de 2022

Resumo

Este relatório diz respeito ao último guião (III) do projeto de Laboratórios de Informática III e tem como objetivo mostrar as metas e formas de resolução dos exercícios propostos e também as dificuldades encontradas nestes.

Em comparação com o guião anterior podemos dizer que este foi bastante mais simples de se realizar e também mais divertido, porém foram encontrados alguns problemas no mesmo. Explorou a nossa criatividade o que nos levou a descobrir aspetos da linguagem C e de programação no geral que nunca imaginávamos, tais como as belas artes que podemos fazer com um bom uso do terminal graças a carateres *unicode* e a cores. E também desenvolveu os nossos conhecimentos em relação a termos como **serialização** de dados e **testes unitários**.

De um modo geral, acreditamos que este guião teve um propósito mais de *end game*, ou seja, não só foi mais simples como também nos deu mais liberdades de escolha sobre o que fazer de modo a finalizar o projeto em grande.

Conteúdo

1 Introdução

No âmbito da unidade curricular Laboratórios de Informática III, este relatório tem como finalidade mostrar os métodos usados na resolução do novo, e último, guião 3 e os maiores obstáculos que surgiram. Este guião tem quatro focos maioritários: a consolidação dos conceitos de modularidade e encapsulamento já postos em prática no guião anterior; o suporte a diferentes mecanismos de interação entre programa e utilizador, ou seja, a criação de uma interface para o utilizador; a manipulação e otimização de consulta de um grande volume de dados e também testes funcionais e avaliação de desempenho.

1.1 Alterações Realizadas

Antes de começar este novo guião, decidimos resolver um assunto pendente, o primeiro exercício do guião I. Este exercício consiste em analisar os dados presentes nos ficheiros que nos foram apresentados gerando, assim, um ficheiro de saída contendo apenas os dados válidos, dados estes, que são considerados válidos de acordo com algumas condições enunciadas.

O grande problema é que havia muita memória que era alocada, mas nunca liberta durante a execução. Isto, dependendo da dimensão, o *kernel* matava o processo devido ao consumo extremo de recursos como memória interna (RAM ou SWAP). Após a análise da execução do programa com a ferramenta `htop` conseguimos ver que a memória que o processo precisava era superior à memória disponível no computador.

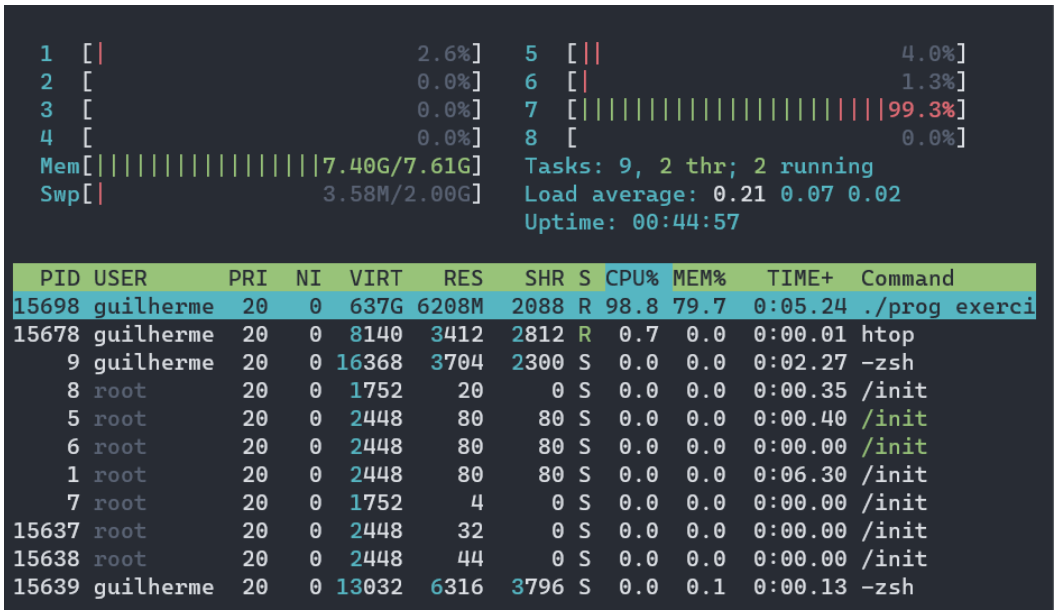

Figura 1: Exaustão da memória durante a execução do guião 1

Após a testagem de cada função, uma a uma, descobrimos que a culpada era a função responsável por transformar a lista dos *followers/following* num *array* de inteiros. Porém, em vez de resolvermos apenas essa função, optamos por reescrever o exercício inteiro de maneira mais eficiente e muito menos custosa em termos de memória. Ou seja, não só ocupa 90% menos memória, como também é muito mais rápida (o tempo de execução passou de cerca de 6 segundos para cerca de 3).

Para além do guião I também foi necessária a reescrita inteira do guião II, porém os porquês e a resolução é apenas discutida no capítulo 3.3 Otimizações.

2 Exercícios

Nesta fase final do projeto foram-nos propostos três exercícios todos independentes entre si.

O primeiro exercício é relativo à implementação de um mecanismo de interação com o utilizador, isto é, uma interface Homem-Máquina através do terminal. Para tal temos duas escolhas: ou tiramos partido de uma biblioteca que permita a construção de uma interface textual (*TUI, text user interface*) ou construímos a nossa. Acabamos por optar pela segunda opção, já que aprender uma biblioteca para construção de *TUI's* (por exemplo, a biblioteca *NCurses*) demora tempo, tempo este que não temos de sobra.

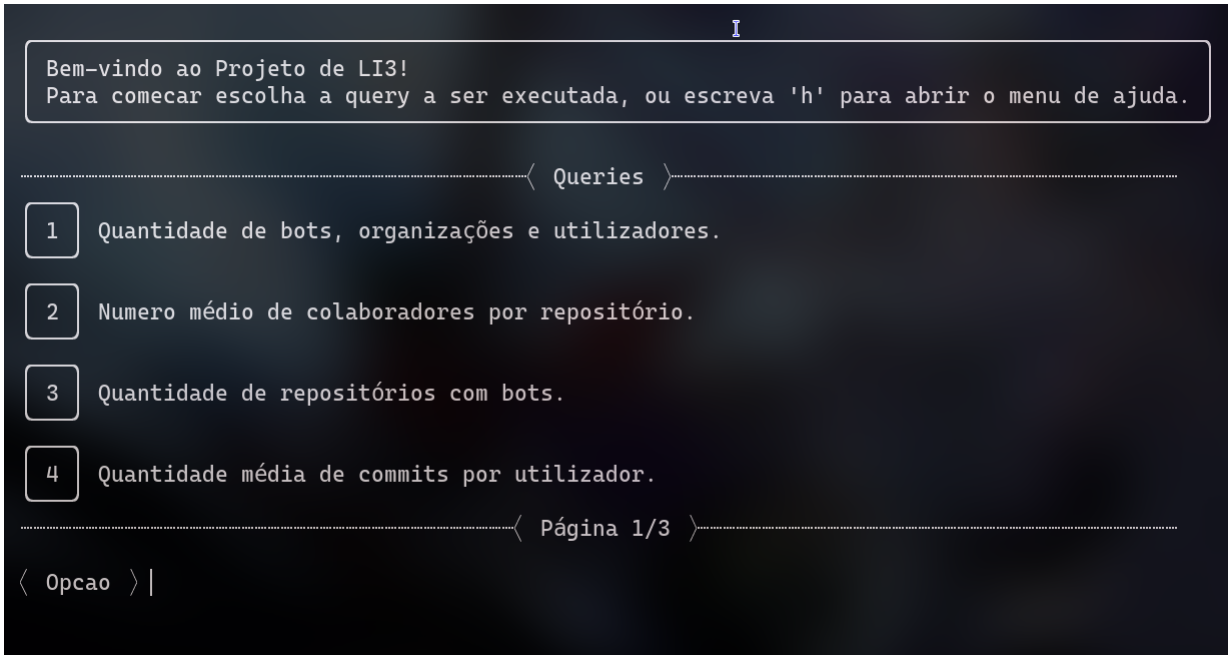

Figura 2: Exemplo de Interface

O segundo exercício dá-nos a conhecer um pouco sobre testes funcionais e como avaliar o desempenho tendo em conta os mesmos. Para isso é preciso implementar uma plataforma de testes que verifica os resultados produzidos pelas *queries* através da comparação entre ficheiros gerados. Para além disso, este exercício estende-se para lá do programável

e, por isso, vamos fazer uma análise aos resultados obtidos assim como aos seus tempos de execução antes e depois das otimizações feitas ao código do guião anterior.

Finalmente, o terceiro e último exercício visa a um melhoramento em relação à gestão de dados, pois com o aumento dos *data-sets*, os mesmos podem já não caber em memória devido a más implementações dos catálogos ou a computadores antigos com pouca memória. Para tal, é nos dada, por exemplo, a solução de serialização de dados ou a reestruturação dos catálogos.

3 Resolução

Como esperado, neste relatório vamos abordar os três exercícios referidos acima abrangendo os assuntos mais importantes do problema, as maiores dificuldades encontradas, como se resolveram e o que poderíamos ter feito mais adequadamente de modo a melhorar tanto a *performance* do programa como a experiência de uso do utilizador.

3.1 Interface Textual

Começamos pelo desenvolvimento de uma interface textual para o programa, pois não só é o exercício mais divertido como também não depende tanto da *performance*. Como dito acima, optamos por não utilizar uma biblioteca para a construção de *TUI's* como a biblioteca *NCurses* mas por, de um certo modo, criar a nossa *lib*. Esta, não interfere com a execução do programa realizado no guião II, pois a interface apenas é iniciada caso não seja passado nenhum argumento ao inicializar o programa.

A nossa interface é construída sob três funções principais: A primeira, `print_boxed`, escreve no terminal texto rodeado por bordas, ou seja, uma caixa de texto com cantos arredondados como podemos verificar no exemplo da imagem abaixo. A segunda utiliza a função `print_boxed` para produzir um item pertencente a uma lista, sendo este constituído por uma caixa de texto que indica a posição na lista e uma linha de texto introduzida na frente do indicador da posição.

Porém, é nesta função onde encontramos o nosso primeiro problema: como é que se escreve o texto na frente da caixa de texto sendo que esta ocupa três linhas e nós queremos a descrição na segunda?

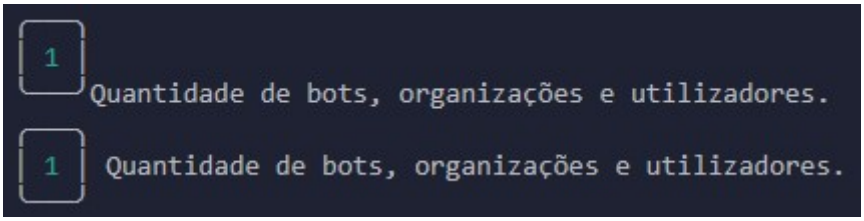

Figura 3: O Problema Encontrado

Este problema é facilmente resolvido com a introdução de *escape sequences*. *Escape sequence* é uma sequência de caracteres que não são representados quando usados dentro de uma *string* mas são traduzidos noutro caracter ou sequência de carateres que possam ser difíceis ou impossíveis de representar diretamente. Normalmente, estes são representados por mais de dois caracteres onde o primeiro é sempre "\". A seguinte função permite a resolução do problema exposto graças ao uso das sequências `\033[%dA\033[%dL` e `\033[%dB` que, dependendo dos parâmetros, sobe ou desce o cursor no terminal e produz como resultado a primeira imagem da Figura 3.

```
1 void print_list_item(char *list_number, char *content)
2 {
3     print_boxed(list_number);
4     printf("\033[%dA\033[%dL", 0, -1);
5     printf(" %s\n", content);
6     printf("\033[%dB", 1);
7 }
```

Passando agora para a última função, `pag_output`, que permite a representação de qualquer ficheiro de texto no terminal de forma paginada e foi sem dúvida a função mais desafiante de todo este exercício. Esta função baseia-se num *loop* sob um *array* constituído por todas as linhas do ficheiro, *loop* este cujas iterações são controladas pelo *input* do utilizador, isto é, começamos no índice 0 do *array* e escrevemos `N` linhas (o nosso *threshold*), caso o utilizador queira avançar para a página seguinte damos início a uma nova iteração, mas no índice $0 + \texttt{N}$ do *array*. Voltar para a página anterior é relativamente o mesmo processo, mas em vez de adicionarmos `N` ao índice, subtraímos.

A maior dificuldade relativa a esta função foi o facto de se ter de escrever o texto sempre a partir de uma determinada linha, ou seja, a cada iteração temos de apagar as linhas escritas anteriormente pela função. Para isso usamos mais *escape sequences* para subir/descer com o cursor e apagar a linha onde o cursor está.

Outro problema encontrado foi o facto de ser **obrigatório** o uso de uma fonte com suporte a caracteres `UNICODE`, o mais provável é que o utilizador esteja a utilizar o *Windows* como sistema operativo cuja fonte só tem suporte parcial o que leva à desfiguração da interface. Daí para um melhor proveito da interface é necessário o uso de uma fonte de texto com total suporte `UNICODE`.

A pior adversidade encontrada, e não resolvida, foi o facto que não conseguir corrigir os ficheiros de entrada e depois lê-los, criando assim os catálogos. O problema, pelo que nós achamos, tem a ver com o tempo de escrita em ficheiros. O que estava a acontecer é que o programa tenta criar os catálogos sem que o guião I tenha acabado de escrever as linhas todas, o que gerou linhas incompletas e *segmentation faults* devido à função `strtok_r()` receber uma linha não completa e atribuir valores errados a vários parâmetros.

Assim, realizamos o exercício sem complicar muito, de maneira eficaz e limpa de modo a oferecer a melhor experiência de uso a qualquer utilizador.

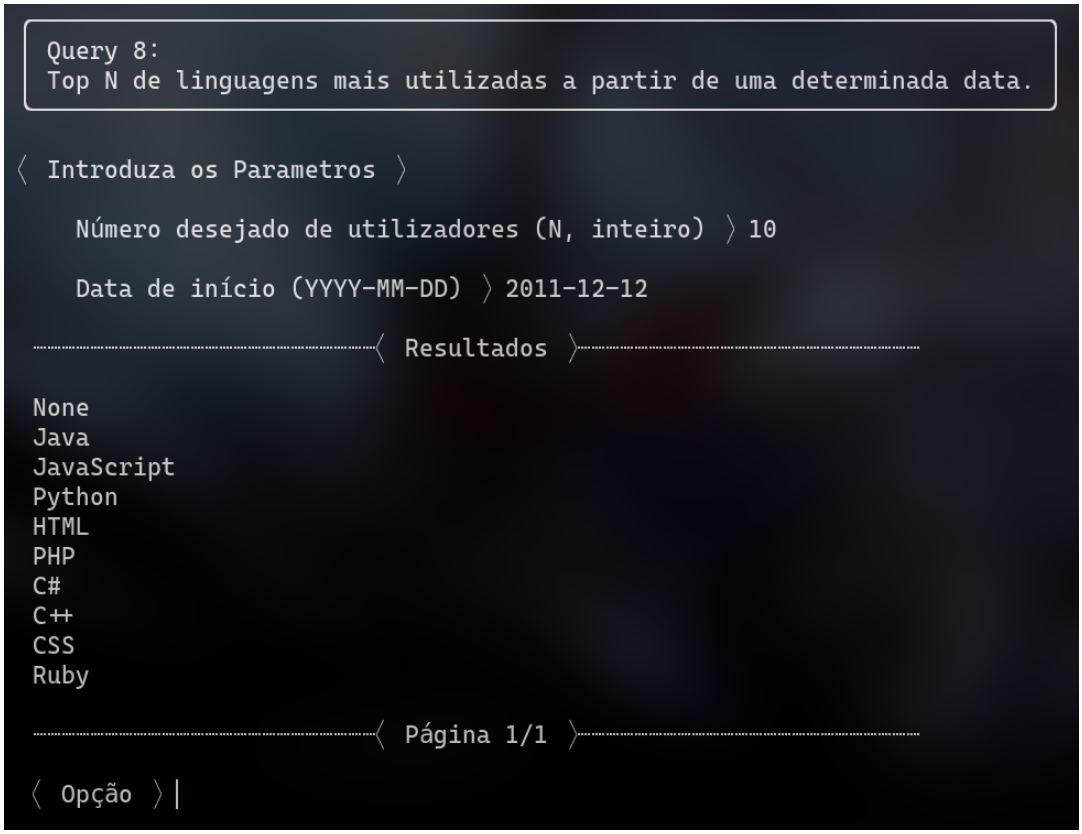

Figura 4: Resultado da *Querie* 8

3.2 Otimizações

Após a realização da interface, decidimos tratar dos testes funcionais e da medição do desempenho do programa, porém não faria sentido primeiro abordar os resultados antes das otimizações realizadas. Todas as otimizações realizadas relativas à gestão de dados e velocidade de execução das queries dão-se no guião II (excluindo o que já foi falado no capítulo 1. Alterações Feitas). Para além das otimizações, também melhoramos o trabalho ao separar os catálogos dos objetos (por exemplo, separar o módulo do catálogo do utilizador do utilizador em si) o que aumentou o nível de modularidade, o único *downside* disto foi a necessidade de criar tantos *getters*/*setters* como quantos parâmetros a estrutura contém.

Relativamente às otimizações realizadas, começamos por repensar o método de armazenamento a utilizar, pois, o uso de *arrays* com procura binária simplesmente não satisfazia a necessidade das *queries* mais complexas (como a 9 e 10). Assim, com exceção ao catálogo dos *commits*, cada catálogo passou a possuir uma tabela de *hash* contendo os pares `<id, estrutura>`, mas porquê o uso de tais tabelas?

Tabelas de *hash* são essenciais à realização deste trabalho, pois possuem um tempo de procura constante, $O(1)$, e tempo de inserção, no pior caso de $O(n)$ e no melhor caso/caso médio de $O(1)$. Ambas operações são usadas frequentemente em todas as *queries* parametrizadas. Para além disso, também introduzimos tabelas de *hash* e *sets*

auxiliares e temporários para cada *query* o que em muitos casos teve um impacto drástico, pois eram posteriormente utilizados *arrays* juntamente com procura linear, uma diferença entre $O(1)$ e $O(n)$, respetivamente.

Em relação à gestão de memória, foi-nos sugerida a implementação de uma base de dados guardada em memória externa devido à dimensão dos ficheiros, com a preocupação da exaustão da `RAM`. Após alguma análise da memória consumida pelo programa, usando o *data set* disponibilizado para o guião III, optamos por não implementar este mecanismo. Em primeiro lugar os três catálogos juntos não chegam a ocupar sequer 1 GB de memória o que é fantástico, pois computadores mais recentes possuem no mínimo 8 GB, e finalmente, implementar uma base de dados em memória externa (que implica a serialização dos dados) resolveria o problema de excesso de memória, porém diminuiria o tempo de execução tanto da geração de catálogos como das *queries*, sendo a `RAM` muito mais rápida do que um disco, qualquer que seja o seu tipo (*solid state/hard disk*).

Assim, com a nossa implementação obtemos "o melhor dos dois mundos", bom desempenho tanto termos de memória e velocidade.

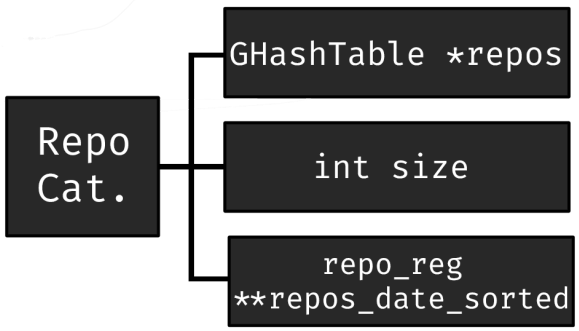

Figura 5: Estrutura do Catálogo dos Repositórios

3.3 Testes Funcionais e Desempenho

Chegamos, finalmente, ao último capítulo onde vamos abordar a implementação de testes funcionais e o desempenho obtido após as otimizações feitas e relatadas no capítulo anterior.

Sendo que todo este trabalho tem como objetivo gerar os resultados das *queries* num ficheiro de texto, para podermos testar a validade dos resultados temos de, inequivocamente, compará-los com outros válidos.

Para tal, acabamos por criar três estados para os testes: `PASS`, ambos ficheiros, produzido e esperado, são iguais; `TIME`, ambos ficheiros são iguais, porém a *query* demorou mais de 5 segundos a executar; `FAIL`, os ficheiros têm pelo menos uma diferença.

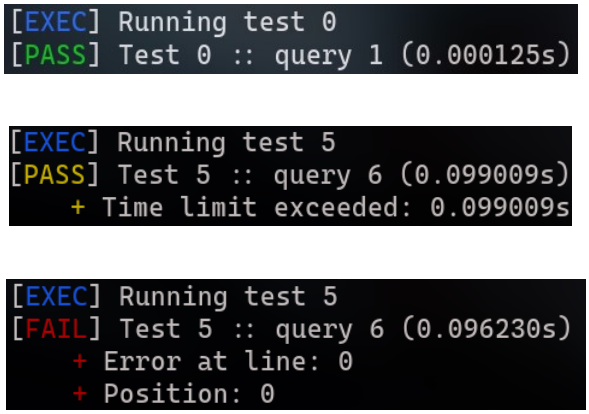

Figura 6: Estados Diferentes dos Testes

Para além dos estados, para uma melhor comparação de resultados ajuda ter as especificações do sistema em conta, pois sistemas mais recentes vão ter um melhor desempenho contrariamente a sistemas mais antigos. Assim, decidimos antes da execução dos testes mostrar ao utilizador com o que trabalha com a ajuda das bibliotecas `<sys/stat.h>` e `<libcpuid.h>`. Porém, a segunda biblioteca não é *default* no sistema, daí ser necessário instalá-la e, também, é uma biblioteca que apenas funciona em sistemas GNU/Linux e FreeBSD o nos leva a apenas mostrar as especificações se o sistema operativo não for Windows.

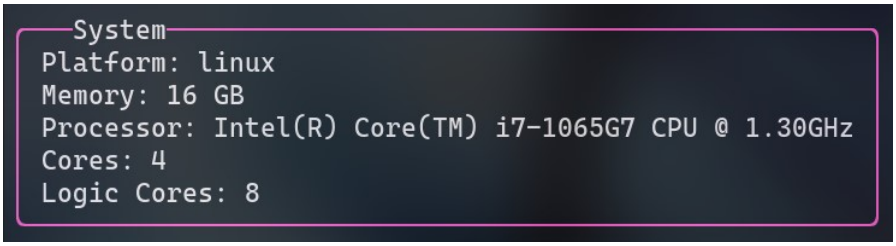

Figura 7: Menu das Especificações

Mas como se adicionam mais testes? Nós desenvolvemos esta plataforma de modo a que tanto a visualização como a inserção/remoção de testes seja o mais simples e intuitivo possível. Para isso, para adicionar cada teste é apenas necessária uma linha de código:

```
1 /* Exemplo do teste para a query 5. */
2 query_5(users, commits,
3         "./entrada/expected/expected_5.txt",
4         test_report);
```

Vemos que no total são passados quatro argumentos à função `query_5()`. O número quatro não é fixo e cada função depende do número de catálogos necessários para a execução da sua *query*. Neste caso, vemos que para executar a *query* 5 são precisos os catálogos dos utilizadores e dos *commits* e, para além disso, são passados mais dois argumentos estáticos: o ficheiro cujo conteúdo é o resultado esperado e uma estrutura de dados que contém a quantidade de testes já feitos, quantos são válidos e quantos falharam.

Uma alteração que poderia ser feita de modo a tornar os testes mais modulares seria introduzir um elemento contendo os parâmetros da *query*, já que, por enquanto, estes são fixos e *hard coded*.

3.4 Desempenho

Para a medida do desempenho do programa, de modo a concluir se as otimizações deram ou não frutos, começamos por fazer cinco sessões de medições para cada *query* antes e depois das mudanças e obtemos as seguintes tabelas:

Queries/Sessão	1	2	3	4	5
Query 1	0,000146	0,000013	0,000013	0,000133	0,000119
Query 2	0,000146	0,000009	0,000005	0,000048	0,000045
Query 3	0,140614	0,146785	0,145207	0,146906	0,147875
Query 4	0,00003	0,000006	0,000006	0,00002	0,000019
Query 5	0,003109	0,002773	0,002776	0,002658	0,002669
Query 6	0,077163	0,074286	0,072949	0,07232	0,073106
Query 7	0,038106	0,03753	0,037551	0,0373	0,037878
Query 8	0,022548	0,2245	0,022339	0,022499	0,022492
Query 9	4,497908	4,647614	4,477638	4,47746	5,343168
Query 10	126,4521	124,1902	126,4821	126,1928	125,9912

Tabela 1: Antes da Otimização (resultados em segundos)

Queries/Sessão	1	2	3	4	5
Query 1	0,000124	0,000119	0,000013	0,000114	0,000012
Query 2	0,15085	0,162484	0,159471	0,160089	0,160174
Query 3	0,000016	0,000018	0,000002	0,000017	0,000002
Query 4	0,000017	0,000056	0,000003	0,000037	0,000003
Query 5	0,000965	0,000957	0,001004	0,000951	0,001047
Query 6	0,097198	0,097106	0,095689	0,096599	0,096054
Query 7	0,010499	0,010666	0,01049	0,010402	0,010556
Query 8	0,032879	0,032424	0,03043	0,030559	0,030941
Query 9	0,109941	0,111896	0,1083	0,109366	0,109527
Query 10	0,328668	0,330227	0,33144	0,309959	0,313104

Tabela 2: Depois da Otimização (resultados em segundos)

Com base na Tabela 1, (3.4) podemos observar que as *queries* mais problemáticas eram a 9 e 10 com tempos de execução médios de $4,7s$ e $125,9s$ respetivamente. A *query* 9 estava mesmo no limite do *threshold* permitido enquanto que a 10 ultrapassava por 120,9 segundos. Porém, na Tabela 2 vemos que ambos os tempos foram reduzidos imensamente! Segue-se abaixo uma tabela que relaciona os tempos médios das *queries* antes e depois das otimizações, onde os valores máximos e mínimos foram descartados:

Queries	Antes	Depois	Speedup
1	0,0000848	0,0000764	0.9x mais rápido
2	0,0000506	0,1586136	3134.6x mais lento
3	0,1454774	0,000011	13252,2x mais rápido
4	0,0000162	0,0000232	0.7x mais rápido
5	0,002797	0,0009848	2.8x mais rápido
6	0,0739648	0,0965292	1.3x mais lento
7	0,037673	0,0105226	3,5x mais rápido
8	0,0628756	0,0314466	1,99x mais rápido
9	4,6887576	0,109806	42,7x mais rápido
10	125,86168	0,3226796	390x mais rápido

Tabela 3: Relação entre as queries

Analisando os resultados, é claro que as alterações feitas no guião II foram muito eficazes, porém foram obtidos alguns valores anormais, é o caso do *speedup* das *queries* 2 e 3, isto deve-se à troca do esforço computacional entre elas, anteriormente a 3 calculava apenas os seus dados, mas agora o esforço passou todo para a 2 que tanto calcula os resultados da 2 como da 3. Em relação às *queries* 9 e 10, vemos uma grande diminuição de tempos de execução como esperado e à exceção da *query* 6 todas tiveram um aumento positivo. Isto é tudo graças ao uso de tabelas de hash e o seu maravilhoso tempo médio de procura constante.

Finalmente, também convém falar na memória utilizada (em *idle*) antes e depois das otimizações que foi o único *downside* das alterações feitas, pois:

	Antes	Depois	Diferença
Catálogos	0.66GB	0.89GB	+0.23GB

Tabela 4: Uso de memória antes e depois

Apesar desta nova versão ocupar mais memória, julgamos que a diferença não seja significativa e que não nos devemos preocupar com a mesma.

Nota: Todos os valores foram obtidos numa máquina cujo sistema operativo é `EndeavourOS` (distribuição de Linux), processador é `CPU: Intel i7-1065G7 (8) @ 3.900GHZ` e possui `16GB` de memória RAM; foi utilizado o *data-set* do guião anterior por ser menor e parâmetros constantes em todas as *queries*.

4 Conclusão

Finalizando, apesar das adversidades encontradas aquando a realização do projeto, conseguimos obter os resultados esperados. Foi refeito o exercício 1 do guião I, e o guião II inteiro e feito do zero o guião III.

A parte onde foram encontradas mais dificuldades foi, provavelmente, durante a realização da reescrita do guião II, pois aprender a utilizar a biblioteca `GLib` é um processo um pouco demorado, assim como saber trabalhar com as tabelas de hash da mesma. Mas, ainda assim, conseguimos chegar onde queríamos, que era fazer com que todas as *queries* executassem em menos dos 5 segundos propostos.

Em relação aos exercícios deste guião, não foram encontradas grandes dificuldades na sua resolução, talvez porque estes não exigiam tanta computação de dados, mas mais criatividade e paciência. Porém, poderíamos ter feito uma análise mais detalhada do desempenho ao testar em diferentes máquinas.

Deste modo, apesar de algumas imperfeições do trabalho, todos estes recursos usados e soluções encontradas nas adversidades, permitiram-nos diminuir o impacto em termos de complexidade no programa e aprimorar a eficiência deste.