



Projeto LI3 24/25

Relatório 2ª Fase

Gabriel Dantas, a107291

José Fernandes, a106937

José Martins, a104443

Simão Oliveira, a107322

Grupo 15

Engenharia Informática
Universidade do Minho
2024/2025

Índice

Índice.....	1
Introdução.....	2
Objetivo.....	3
Sistema.....	4
Arquitetura da Aplicação.....	4
Funcionamento da Aplicação.....	5
Discussão.....	6
Encapsulamento.....	6
Estruturas de dados.....	6
Recomendador.....	9
Programa interativo.....	9
Eficiência.....	10
Conclusão.....	13

Introdução

O presente relatório complementa o previamente elaborada, sendo um acréscimo explicativo do projeto que se realiza no âmbito da unidade curricular de Laboratórios de Informática III, do 2º ano de Engenharia Informática, na Universidade do Minho.

Incidir-se-á com maior notoriedade, neste relatório, sobre os seguintes assuntos descritos abaixo em tópico, visto terem sido parte notória do desenvolvimento da atual fase do projeto.

- Encapsulamento
- Estruturas dinâmicas de dados
- Medição de desempenho

A nova estrutura básica do repositório do projeto mostra-se abaixo:

```
grupo 15/  
  |README.md  
  |trabalho-prático/  
    |Makefile  
    |include/  
      |...  
    |src/  
      |...  
    |recomendador/  
      |...  
    |resultados/  
      |...  
    |resultados-esperados/  
    |programa-principal (após make)  
    |programa-testes    (após make)  
    |programa-interativo (após make)  
    |relatorio-fase1.pdf  
    |relatorio-fase2.pdf
```

Objetivo

Tem-se por objetivo o desenvolvimento de um projeto final, escrito em C, que consista na implementação de uma base de dados em memória a partir de ficheiros .csv, fornecidos pelos docentes, que sumarizam informações relativas a um sistema de streaming de música. Estes dados serão armazenados e implementar-se-ão métodos de pesquisa, o mais eficientes possíveis, para atender às *queries* impostas pela chamada do programa.

O projeto final instrui aos discentes conceitos de modularidade, encapsulamento, abstração e reutilização de código, usabilidade e eficácia de estruturas dinâmicas de dados, validação do código funcional e capacidade de medir o desempenho do software. Com esta proposta criar-se-á uma base sólida de conhecimentos e habilidades no que diz respeito à engenharia de software.

Na 1ª fase desenvolveu-se a validação lógica e sintática do *dataset*, a etapa de *parsing* dos dados fornecidos, os modos executáveis principal e de testes e as primeiras 3 *queries* propostas.

Nesta 2ª etapa de realização do projeto, baseado em 2 novos ficheiros de álbuns e histórico, desenvolveram-se os novos catálogos, juntando-os aos previamente existentes, realizaram-se 3 novas queries e fizeram-se alguns ajustes às já existentes. Fez-se ainda o executável do modo interativo.

Sistema

Arquitetura da Aplicação

A arquitetura do projeto atualizada, representada no Diagrama 1, foi pensada tendo em conta uma organização modular, visando sempre a abstração e opacificação dos módulos criados, bem como o fácil gerenciamento de informação e transmissão da mesma entre eles.

Utilizou-se uma estrutura denominada *Database* como um gestor de nível superior para interagir com os gestores das entidades (considerando-se as entidades os tipos de dados criados diretamente da informação fornecida nos .csv), estes chamados de *Entity Catalog*.

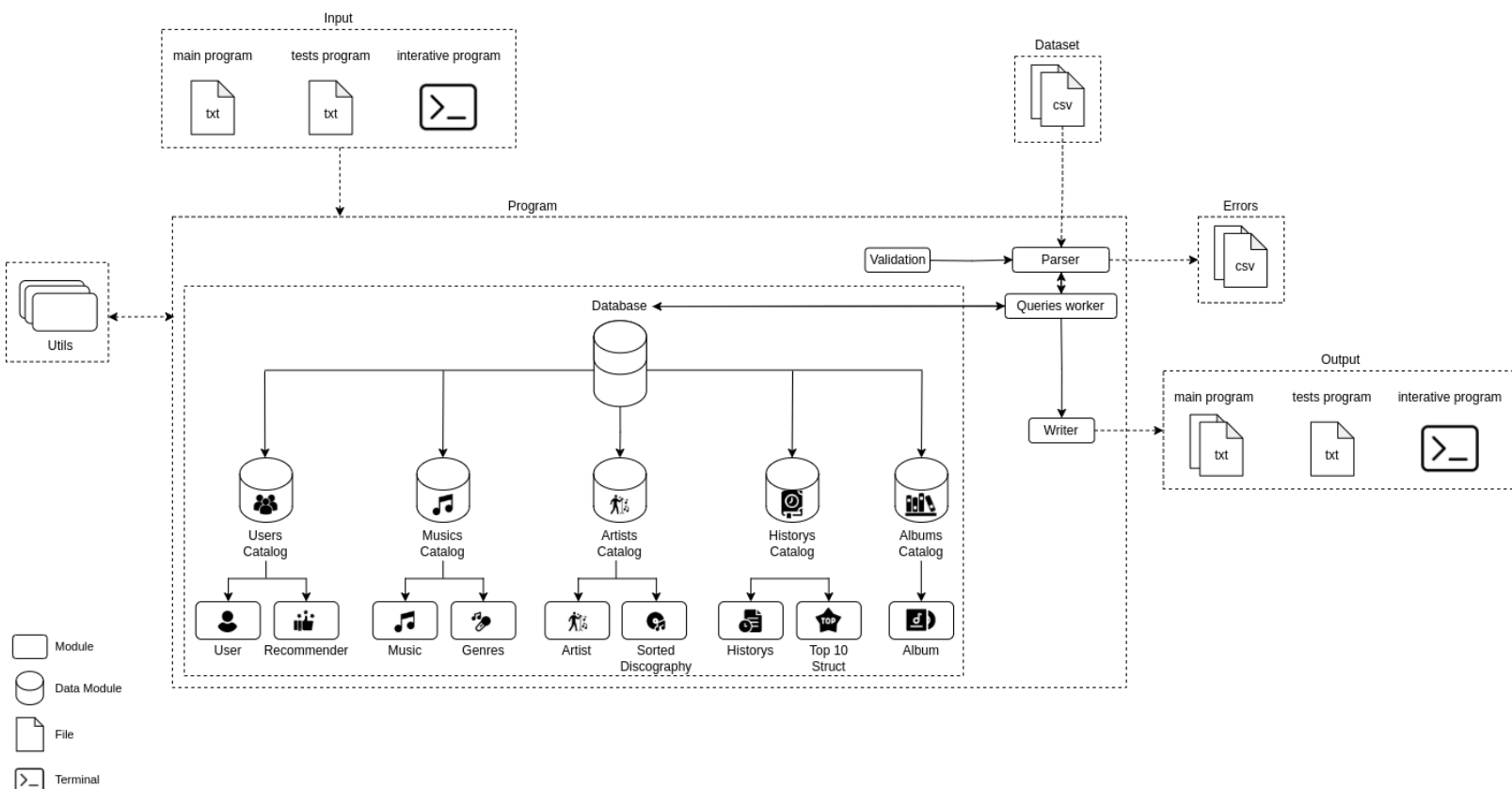


Diagrama 1. Estrutura da Arquitetura

Funcionamento da Aplicação

Todo o funcionamento da aplicação permanece análogo à primeira fase do projeto, não tendo havido alterações fundamentais no fluxo do programa.

Laconicamente, a aplicação é chamada via terminal e, inicialmente, são criadas a *DataBase* e os catálogos, sendo os catálogos gestores das entidades do sistema e a *DataBase* um gestor de nível superior a estes.

De seguida os arquivos .csv são lidos por um *parser* genérico e é feito o seu tratamento e adição aos respectivos catálogos ou adição a um arquivo de erros. São ainda criadas estruturas de dados auxiliares, visando a eficiência temporal na resposta imediata à chamada das *queries*.

A resposta a estas é gerida pelo *queries worker*, módulo que solicita informação à *DataBase* para fornecer o devido retorno à *query*. Finalmente, o *writer*, módulo de escrita genérico responsável por criar e escrever nos ficheiros de *output*, recebe o *buffer* de retorno pela *query*.

No momento terminal da execução, é feita uma limpeza da memória do programa, onde todas as estruturas criadas durante o mesmo são eliminadas.

Algumas das estruturas adicionais específicas para aprimorar a eficiência à resposta das *queries* foram já explicadas no relatório que a este se precedeu, enquanto as restantes serão elucidadas adiante, na seção de discussão do projeto, onde se aprofundarão os aspetos mais proeminentes do desenvolvimento nesta fase.

Discussão

Esta seção tem por objetivo debater os métodos de modularização e encapsulamento utilizados, bem como a escolha das estruturas de dados para realizar eficazmente as *queries*. Dado já terem sido abordadas as técnicas de modularização e as estruturas de dados utilizadas nas *queries* anteriormente atribuídas, esta seção responsabilizar-se-á por explicar as técnicas de encapsulamento e as estruturas de dados para as novas *queries* e alterações às anteriores.

Encapsulamento


Os módulos estão encapsulados, tendo sido feita utilização de todos os métodos lecionados em aula, a opacificação de estruturas de modo a restringir o acesso direto a módulos de informação e a utilização de *getters* e *setters* com partilha de informação imutável, fazendo cópias ou manuseios pertinentes das estruturas antes de as retornar ao utilizador.

Esta característica do programa permite que haja maior flexibilidade, robustez e manutenibilidade do código, não havendo forma de alterar dados internos das estruturas de dados e permitindo uma fácil alteração de código nos módulos a estes relativos, sem que prejudique o ambiente externo a eles.

Estruturas de dados

Para a realização da atualizada *query* 1, retornar o resumo de um artista, foi utilizado um array atribuído a cada artista, guardando os *id* 's das bandas a que pertencem, permitindo facilmente obter o número de artistas coletivos a que cada artista individual pertence, sendo bastante rápido o cálculo da receita dos artistas. Têm ainda um campo que armazena o número de álbuns em que o artista é o único autor. Estas informações são calculadas durante a fase de parsing dos artistas.

A *query* 4 pretende conhecer o artista que esteve mais vezes no *top* 10 semanalmente entre 2 datas, ou, na omissão destas, em todo o histórico existente. Esta *query* é realizada utilizando uma *hashtable*. As chaves desta são *week offsets* da semana a que a data do histórico pertence, em que esse valor é calculado com base




numa data referencial do início do calendário gregoriano. Os valores que esta armazena são estruturas que contêm uma árvore binária, *ArtistDurationTree*, e um array com os *id* 's dos artistas no top 10 dessa semana. A *ArtistDurationTree* é uma árvore binária que tem como chave os *id* 's de todos os artistas ouvidos nessa semana e guarda, como valor, a duração de audição destes. Este processo de criação da árvore é feito durante o *parsing* do histórico. No final deste processo, é chamada uma função responsável por preencher os top 10 para todas as semanas da *hashtable*, que percorre a árvore, gerenciando o top 10 dos artistas.

Tendo esta estrutura em memória, chamando a *query* 4, o resultado torna-se trivial de ser retornado, sendo calculado apenas o *week offset* das datas inicial e final (ou pedindo os valores inicial e final guardados na *hashtable* ao módulo, valores previamente computados) e atualizando um campo pertencente aos artistas, que guarda o número de vezes que estiveram no top 10 para todas as datas entre as obtidas. Finalizando este processo, alcança-se a inevitabilidade de percorrer o catálogo dos artistas e guardar o id do artista que esteve mais vezes no top 10 conjuntamente com essa quantidade. Para cada um, verifica-se se a quantidade de vezes que este esteve num top 10 semanal é maior do que as anteriores guardadas e, em caso positivo, atualizar estes campos. Ao final deste processo teremos o id do artista pretendido e as vezes que entrou no top 10 semanal entre as datas pedidas.

Para a realização da *query* 5, utiliza-se uma *hashtable* com as chaves sendo utilizadores e, contendo como valor, uma lista ligada de histórico associados a esse utilizador específico, para eficiência de procura. Usa-se ainda uma estrutura chamada *recommender* que guarda a *matrizClassificacaoMusicas*, matriz argumento da função de recomendação, uma lista com *id* 's dos utilizadores, outra com os nomes dos gêneros, o número de utilizadores e o número de gêneros. A *matrizClassificacaoMusicas* é gerada iterando sobre a *hashtable* e, para cada *user*, percorre-se a lista ligada do histórico e incrementa-se o valor no índice representativo do género pretendido na matriz.

Para a realização da *query* 6 utiliza-se uma abordagem diferente de todas as outras *queries*. Ao invés de criar a estrutura previamente, apenas o fazemos na chamada da *query* para o *user* respectivo. Utilizando a mesma *hashtable* mencionada para a *query* 5, ao chamar a *query* apenas temos que procurar o *id* recebido, manipulando, para a nova estrutura, apenas os históricos do ano que nos são pertinentes.



A estrutura utilizada é uma coleção de 4 árvores binárias (albumTree, genreTree, artistTree e musicTree), em que todas são utilizadas para guardar o tempo de audição por cada um destes parâmetros (álbuns, géneros, artistas e músicas) e o seu respetivo identificador único.

Durante a criação destas árvores, vamos atualizando outros campos de informação, informações essas que realmente respondem ao exigido pela *query*. Existem, para além das árvores, 4 inteiros e 2 listas de inteiros. O mostHeardAlbum e o mostHeardAlbumTime são inteiros que mantêm a lógica de armazenar sempre o maior tempo de audição de um álbum registado. O inteiro musicCount guarda o número total de músicas diferentes ouvidas pelo utilizador (faz uma busca na musicTree por cada histórico a tratar e, caso não exista nesta, adiciona-a e incrementa este valor). O totalListeningTime é um inteiro que armazena o tempo total de audição do utilizador. As 2 listas, hourList (tamanho 24) e dayList (tamanho 365) são listas que terão os seus índices incrementados na hora e dia que um histórico tem, para ser possível, no final, iterando sobre estas, descobrir a hora e o dia com maior tempo de audição pelo utilizador.

Existe, para finalizar, um outro parâmetro, designado mostHeardArtists que consiste numa estrutura do tipo Top_artist_info. Esta apresenta uma lista com n artistas (número dado no input) que contém o tempo de audição do utilizador de músicas de cada um dos artistas e o número total de músicas ouvidas pelo utilizador de cada artista. Esta lista é alterada quando acedemos à artistTree, verificando se o artista que estamos a processar atualmente tem um tempo de audição maior que o último elemento da lista de mostHeardArtists (esta lista está sempre ordenada), caso tenha, inserimos e reordenamos.

Depois desta grande estrutura ser processada, temos toda a informação pedida, apenas temos que a formatar corretamente para o output.

Esta estrutura é bastante complexa pelo baixo número de históricos existentes por cada utilizador neste projeto académico, porém, pensando num caso de um sistema integrado real, tentámos implementar uma maneira mais elegante e eficiente de realizar a *query*.

Recomendador

Decidimos elaborar o nosso próprio recomendador. Baseamo-nos na conceção de que o produto escalar entre vetores é uma medida de similaridade entre objetos matemáticos. Concretamente, partindo de uma lista com tempos de audição de um utilizador, chamar-lhe-emos o utilizador a recomendar, relativamente a algum parâmetro especificado, podemos calcular o produto escalar do utilizador a recomendar com outro utilizador, denominá-lo-emos o utilizador recomendado. Fazendo este processo, obteremos um valor que mede a similaridade entre os 2 utilizadores em termos do parâmetro pretendido. Portanto, repetindo este método para todos os utilizadores, será possível obter, entre todos os utilizadores, o que tenha maior similaridade em termos do parâmetro escolhido com o utilizador a recomendar (por comparação dos produtos escalares).

Pensando nesta ideia chave, elaborámos algo um pouco mais situacional. Cada *user* contém uma estrutura que guarda 3 listas físicas (implementadas através de árvores binárias) contendo o tempo de duração de audição do utilizador por géneros, países e músicas. Serão calculados 3 produtos escalares por cada emparelhamento de utilizadores, um por cada árvore e, a estes, foram atribuídas ponderações distintas de importância para o que achamos que sejam as características mais importantes para medir a similaridade entre gostos. Ao produto escalar resultante das listas por países, atribui-se uma percentagem menor, 20%, visto soar-nos menos relevante, ao resultante das listas por géneros atribuímos uma percentagem de 50%, dado ser um fator altamente significativo para a comparação de gostos pessoais e os restantes 30% foram atribuídas às músicas ouvidas em si.

Após uma computação exaustiva entre o utilizador a recomendar e todos os outros utilizadores, computando este valor final acima explicado, selecionamos os *n* (inteiro parâmetro recebido pela nossa função própria de recomendação) *users* com maior similaridade (valores desta ponderação de produtos escalares mais elevados) e são colocados numa lista de tamanho *n* e devolvida para formatar.

Programa interativo

O modo interativo é uma interface gráfica de utilizador (UI). Consegue executar com um *dataset* personalizado, sendo o caminho para este fornecido em tempo de

execução e realiza *queries* pedidas pelo utilizador na interface. A implementação do modo gráfico foi feita utilizando a biblioteca *ncurses*. Seguem-se algumas figuras ilustrativas (Figura 1, 2, 3 e 4).

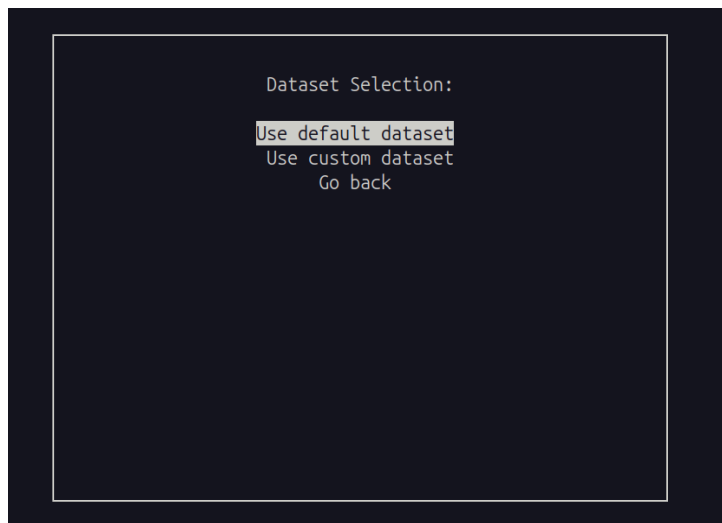


Figura 1. Menu principal

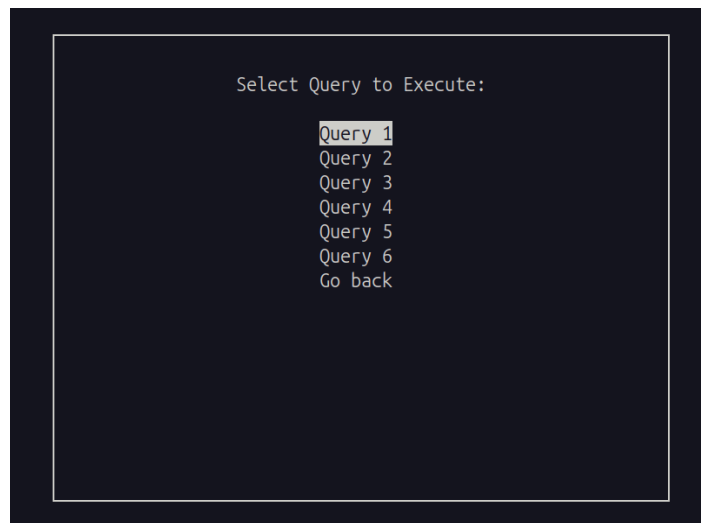


Figura 2. Menu seleção de queries



Figura 3. Menu de dados da query 2

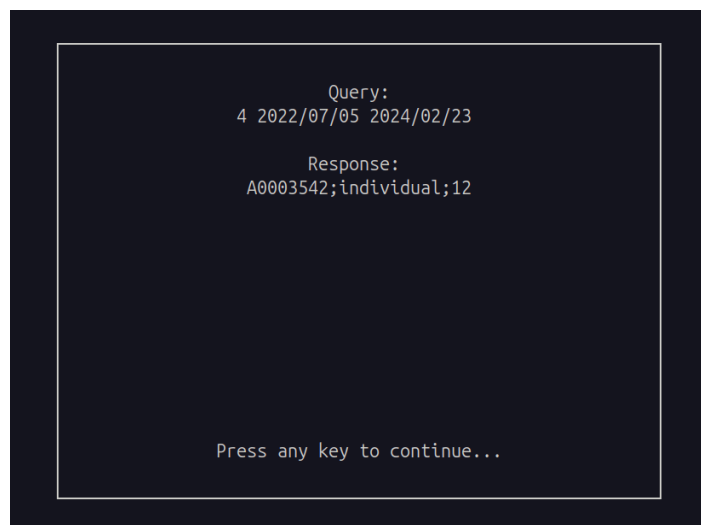


Figura 4. Resultado da query 4

Eficiência

Em termos de eficiência dos processos de criação e busca das estruturas referidas na seção de “Estruturas de dados” com objetivo de responder às *queries*,

pensamos que tenhamos encontrado soluções interessantes, com procuras rápidas o suficiente para satisfazer as necessidades e dificuldades a que o projeto nos expõe.

Apresenta-se na Tabela 1 o resumo da performance do programa, calculado através dos resultados dados pelo programa de testes (utilizando os *datasets*, pequeno e grande, com erros), nos dispositivos de cada um dos integrantes do grupo, com as seguintes especificações:

- Dispositivo 1 (ASUS Vivobook S 15 S5506MA_S5506MA)
 - Especificações: Intel® Core™ Ultra 7 155H × 22, 16.0 GB RAM LPDDR5X, Ubuntu 24.04.
- Dispositivo 2 (Yoga 7i 2-in-1 Gen 9)
 - Especificações: Intel® Core™ Ultra 7 155U 16.0 GB RAM LPDDR5X-7.467MHz Ubuntu 24.04
- Dispositivo 3 (ASUS Zenbook 14X UM3402YA_UM3402YA)
 - Especificações: AMD® Ryzen 7 5825u with radeon graphics × 16, 16.0 GB RAM, Pop!_OS 22.04 LTS
- Dispositivo 4 (HP Victus by HP Laptop 16-d0xxx)
 - Especificações: 11th Gen Intel® Core™ i5-11400H @ 2.70GHz × 12, 20,0 GB RAM, Ubuntu 22.04.4 LTS

Achámos ainda relevante falar, nesta seção de eficiência, sobre a escalabilidade do nosso programa. Dado que no menor *dataset* estão a ser recebidos 1,578,961 linhas de texto (10,000 de álbuns, 11.501 de artistas, 287,500 de utilizadores, 119,960 de músicas e 1,150,000 de históricos) e no maior *dataset* processam-se 16,725,433 linhas de texto (100,000 de álbuns, 11.500 de artistas, 460,000 de utilizadores, 1,203,933 de músicas e 14,950,000 de históricos), isto quer dizer que a razão entre as linhas que sofrem *parsing* e são processadas pelo nosso programa nos *datasets* grandes comparados aos pequenos é de 10.593 mais. Já a razão do tempo de execução destes é de 7.722 (razão das médias dos tempos totais de execução dos *datasets* maiores sobre os menores nos 4 dispositivos). Ou seja, o programa apresenta uma escalabilidade que se pode considerar positiva, visto que um aumento de valores recebidos no programa de 10.6x não resultará em aumentos catastróficos no tempo de execução do programa, que demora apenas 7.7x a mais.

A proposta feita no anterior relatório, de analisar uma nova abordagem à *query* 3, acabou por não se realizar. Isto deveu-se à perceção de uma boa escalabilidade deste

processo, sendo um processo eficiente relativamente aos tempo na execução do código para os novos *datasets* maiores.

Dispositivo	1		2		3		4	
Dataset	Small	Big	Small	Big	Small	Big	Small	Big
Inicialização DB (s)	10.705	115.926	9.676	62.103	14.926	69.467	21.378	195.822
Inicialização Q2 (s)	0.004	0.004	0.002	0.002	0.003	0.004	0.009	0.009
Inicialização Q3 (s)	3.241	9.867	2.312	6.024	2.674	6.432	8.526	17.150
Inicialização Q4 (s)	0.172	0.980	0.095	0.483	0.107	0.526	0.203	1.097
Inicialização Q5 (s)	0.523	11.713	0.3865	6.589	0.578	9.148	1.242	18.013
Query 1 (ms)	0.014	0.023	0.0047	0.030	0.084	0.060	0.062	0.078
Query 2 (ms)	0.061	0.069	0.041	0.075	0.113	0.118	0.206	0.184
Query 3 (ms)	0.024	0.024	0.03	0.078	0.060	0.078	0.082	0.081
Query 4 (ms)	0.985	0.801	0.839	0.768	0.756	0.768	2.885	1.684
Query 5 (ms)	49.040	66.578	43.234	64.356	42.322	73.620	112.282	155.268
Query 6 (ms)	0.014	0.042	0.035	0.63	0.065	0.082	0.088	0.078
Free (s)	2.512	4.527	1.540	2.998	1.760	7.464	2.612	9.764
Total Execução (s)	13.914	133.148	12.021	78.207	17.148	93.045	25.438	224.700
Pico Memória (KB)	0501600	2645280	0512640	2686884	0512128	2686884	0503424	2646368

Tabela 1. Tempo de Inicialização da DataBase, bem como das estruturas criadas especificamente para executar as *queries* (2, 3, 4, 5), tempo de execução médio das 6 *queries*, tempo de libertar a DataBase, tempo total de execução e máximo de memória utilizado. Valores obtidos da média de 10 execuções para *datasets* pequenos e 3 para *datasets* grandes.

Conclusão

Toda a boa base proveniente da fase anterior possibilitou que o projeto se elaborasse da melhor maneira possível, sem grandes entraves nem dificuldades.

Ao longo do projeto, dedicamo-nos a entender de forma curiosa e empenhada os conceitos lecionados durante as aulas, destacando-se a modularidade e o encapsulamento, e aprendemos a utilizar as ferramentas necessárias e atenuantes para o desenvolvimento do mesmo, tais como o *Valgrind* e o *GDB*.

O trabalho em grupo foi maioritariamente positivo, havendo uma base de entreajuda e discussão de ideias com finalidade de aprimorar a organização e “beleza” do nosso código, a sua eficiência e manter a coerência e organização do projeto. Sentimos que fomos capazes de superar as dificuldades a que a realização do projeto nos sujeitou e terminamos este projeto com mais um nível de conhecimento adquirido, melhores habilidades de organização e gerenciamento de projetos e noções de código essenciais para um engenheiro informático que se prestigie.

Assim, concluímos com a convicção de que foram alcançados todos os objetivos apresentados na iniciação de ambas as fases, dando como concluído o projeto final da unidade curricular de Laboratórios de Informática III.