Universidade do Minho
Escola de Engenharia

Laboratórios de Informática III

– Relatório fase II –

Grupo: 7

Projeto desenvolvido por:

Filipa Cangueiro Gonçalves(A107329)

Carolina Silva Martins (A107285)

Diogo José Ribeiro e Ribeiro (A106906)

1. Introdução

Na unidade curricular de Laboratórios de Informática III, foi proposta a realização de um projeto em C, com vista a consolidar os princípios fundamentais de Engenharia de Software – modularidade, reutilização, encapsulamento, estruturas dinâmicas de dados, validação funcional e medição de desempenho.

Depois de uma primeira fase que consideramos executada com sucesso concentramos os nossos esforços em fazer algumas melhorias que achamos relevantes ou que foram sugeridas pela equipa docente.

Com a introdução desta nova fase surgiram novos desafios. Deixou de haver apenas a necessidade de realizar as *queries*. Sendo também necessário realizar um programa interativo, algo completamente diferente do que já tínhamos feito. Tudo isto aliado á procura por realizar um programa eficiente tanto em termos de memória como de tempo.

O presente relatório irá abordar as estratégias e decisões que o grupo tomou para alcançar os pontos referidos com sucesso.

2. Desenvolvimento

Depois da entrega da primeira fase do projeto, estávamos satisfeitos com as bases que estabelecemos para esta segunda fase. Mantivemos assim o *parser* geral já implementado, fizemos as alterações necessárias ao programa testes e ao programa-principal para suportarem as novas *queries*. E, assim como foi pedido realizamos o programa interativo que será depois abordado ao pormenor. A equipa docente tinha sugerido fazer um *output* geral para resposta às *queries*, algo que não conseguimos por falta de tempo.

3. Estruturas de dados

Na primeira fase, realizamos o *parsing* dos dados e organizamos a informação em estruturas que consideramos apropriadas, maioritariamente *Hash Tables* compostas por *structs* em que os dados trabalhados eram maioritariamente *strings*.

Apesar de na altura termos estados satisfeitos com essa organização, decidimos mudar alguns dados de *char** para *int.* Esta mudança foi feita com a intenção de melhorar o nosso uso de memória uma vez que sendo os nossos dados do tipo *char** acabávamos por criar bastantes cópias dos mesmos quando precisávamos de os utilizar.

Começamos então por mudar todos os identificadores únicos, *IDs*, de *char** para *int*. Algo que nos fez alterar o tipo dos nossos arrays de *IDs* de *char*** para *int**. ara além disso decidimos apenas trabalhar os dados que seriam necessários para as queries retiramos assim o parâmetro *description* proveniente do *artists.csv*. Isto foi algo que adotamos ao trabalhar os novos dados.

Contrariamente à primeira fase onde respeitamos a maioria dos tipos atribuídos pelos *CSVs*, nesta segunda fase fizemos bastantes mudanças e trabalhamos os dados de forma a facilitar a respostas às *queries*.

Depois de fazermos as alterações que consideramos necessárias aos dados já trabalhados começamos por trabalhar o *albums.csv.*

Seguidamente, trabalhamos o *history.csv*. Decidimos dividir o trabalho deste ficheiro em 2: os dados necessários para a resposta da *Query4* e os dados necessários para a resposta da *Query6*.

Assim, com a Query4 em vista, dividimos os dados fornecidos numa *Hash Table* cuja *key* é a data de início da semana, ou seja, a data de domingo. Essa Hash Table (Domingo) contem uma *Hash Table (artistahistory)* com os dados que consideramos relevantes de um artista, o id e o total de segundos de música desse artista ouvida. Esses dados que se encontram inicialmente na *Hash Table* interna *(artistahistory)*, no fim de todos os dados serem tratados são passados para um *GArray* que ordena os dados e limita a quantidade dos menos. Assim, a *Hash Table* deixa de existir e passamos a ter um *GArray* ordenado por ordem decrescente do total de segundos e que contém no máximo o top 10 de artistas mais ouvidos nessa semana. Tomamos a decisão de transferir os dados de *Hash Table* para um *Garray* pois apesar de a primeira ser bastante útil para procuras uma vez que o tempo de procura é constante, está não é a melhor estrutura quando precisamos de algo que esteja ordenado.

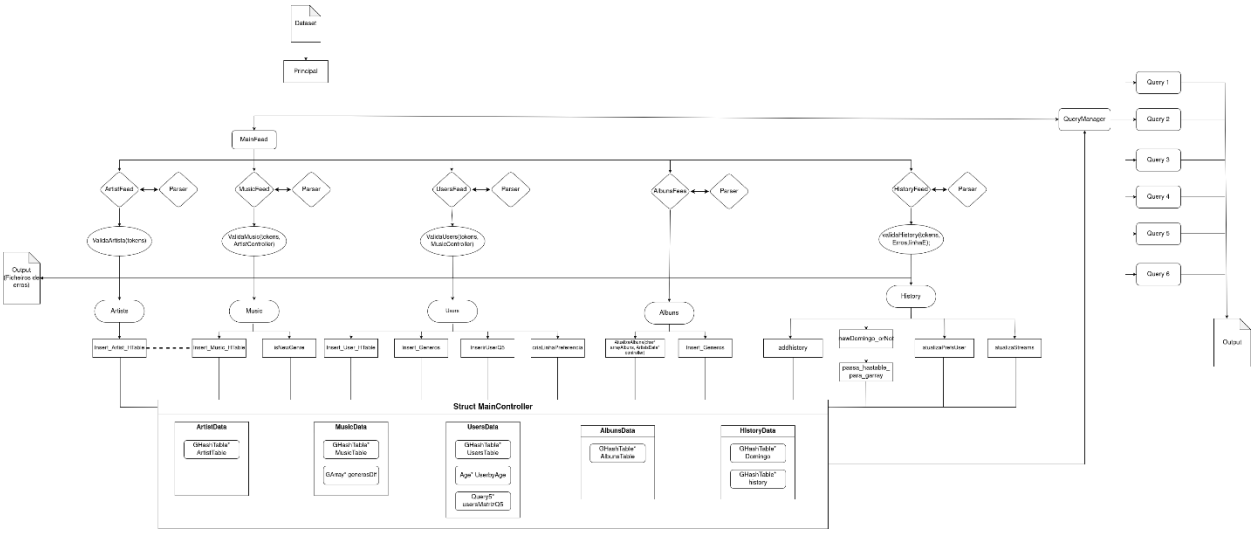
Na *Query6* seguimos a mesma estratégia e tentamos organizar os dados de forma que estes estivessem prontos a ser trabalhados quando chegarem à *query*. Assim, ao analisar os dados fornecidos decidimos dividi-los numa *Hash Table* externa *(History)* cuja *key* é o *user_id*. Essa *Hash Table* contém um *GArray (Anos)*, este é composto por três *GArrays (musicas, horas e dias)* internos que passaremos a explicar.

O *GArray musicas*, contém o *music_id*, o identificador único de uma música e o *tempoAudicao*, o tempo total que um determinado *user* ouviu essa música num determinado ano. O *GArray horas*, contém *int*(0 a 23) que representa a hora e outro *int* com o *tempoAudicao* de música na respetiva hora. Por fim, o *GArray dia*, contém um *int(*0 a 31*)* que representa o dia, um *int(*0 a 12*)* que representa o mês e um *int* que representa o número total de músicas que foram ouvidas nessa data.

4. Estrutura do trabalho

Após a entrega da primeira fase estávamos satisfeitos com a arquitetura que tínhamos estruturado. O nosso trabalho estava dividido em: Main, Controllers, Entities, IOManager e Utils.

Nesta segunda fase, mantivemos a estrutura apenas acrescentando as definições necessárias para suportar os novos dados.



5. Estratégia seguida nas queries

5.1 Query 1

A *Query1*, embora tenha mantido o mesmo objetivo, tornou-se mais abrangente ao passar a fornecer também informações sobre os artistas. A primeira parte da *query*, isto é, responder com informações sobre os *users*, já se encontrava funcional da primeira fase do projeto. Assim, concentramos os nossos esforços no desenvolvimento e tratamento das informações relativas aos artistas.

Inicialmente começamos por definir como é que utilizaríamos as estruturas já implementadas para responder a esta *query*. Posto isto, acrescentamos alguns novos parâmetros à definição de artista: *ganho_por_stream*, *profit*, *albunsIndividuais*.

À medida que trabalhámos os dados dos álbuns, verificávamos o/os artista/artistas e adicionávamos uma unidade ao campo álbuns dos respetivos artistas, de forma a calcular o número de álbuns individuais de cada um.

Posteriormente, pensamos em como obter o número de *streams* e o *profit* de cada artista. Com esse objetivo acedemos ao *historyFeed* e usamos o *musicId* que faz parte de cada entrada do histórico para obter todos os artistas que fizeram parte da música em questão, *music_artists_id.*

Ao iterar pelo *array* de artistas obtido, começámos por verificar o tipo de cada artista. Se este fosse “Individual” apenas multiplicamos o ganho por *stream* do artista respetivo por uma unidade. Já se fosse “coletivo” multiplicávamos o ganho por *stream* por uma unidade e depois dividíamos pelo número de constituintes da “banda”. Ambos os valores eram adicionados a um novo campo que foi acrescentado ao artista, *profit.*

Apesar de acharmos que a ideia seria funcional, acabamos por ter alguns problemas a trabalhar o artista quando este era “coletivo”. Apercebemo-nos que não estávamos a considerar o ganho por

stream da banda como um todo e sim apenas o de cada artista. Fizemos uma melhor análise e acabamos por perceber o erro que estávamos a cometer.

Adicionalmente, alteramos o tipo do ganho por *stream* e do *profit* do artista para *double* em vez do seu tipo original, *float* uma vez que este tem uma maior precisão necessária que *float* não tem.

Concluindo, a *Query1* pretende de forma eficaz fornecer informações referentes a um determinado *user* ou um determinado artista. A resolução apresentada pretende solucionar este problema de maneira equilibrada, isto é, no melhor rácio entre memoria usada e velocidade de resposta e aproveitando as estruturas já definidas.

5.2 Query 2 e 3

A *Query 2* manteve a ideia central, sendo apenas modificada para respeitar os princípios do encapsulamento e modularidade. Já a *query 3* não sofreu alterações.

5.3 Query4

A *Query 4* tem como objetivo identificar o artista mais frequente no Top 10 de um determinado intervalo de datas, ou em todo o histórico, caso o intervalo não seja especificado.

Para abordar este problema, começamos por tentar delinear uma estratégia que fosse o mais eficiente possível tanto em termos de memória como de tempo. Consideramos que o "segredo" para conseguir alcançar isso estaria na forma como trabalharíamos o histórico antes de ele chegar propriamente à *query.*

Como mencionado anteriormente, o nosso trabalho do histórico para esta *query* consistiu em organizar os dados numa *Hash Table* externa (Domingo) cuja *key* é a data do início da semana e os dados que consideramos necessários, isto é, o *artist_id* e o *totalsemanalsegundos* armazenados num *GArray.* Estes já se encontram ordenados e restringidos no máximo a um top 10 dos artistas com mais segundos nessa semana.

Tendo os dados organizados apenas tivemos de percorrer os dados contar as ocorrências de cada artista. Para isso, percorreremos o *GArray* interno da *Hash Table* (Domingo) e usamos uma *Hash Table* auxiliar que usasse como *key* o *artist_id* e guardasse o número de ocorrências desse artista.

Caso seja fornecido um intervalo de tempo, durante a execução da *query*, antes de aceder ao *GArray*, verificamos se a data, correspondente à chave da *Hash Table* externa (Domingo), pertence ao intervalo especificado. Caso contrário, percorremos todos os dados da *Hash Table*.

Em resumo, apesar do desafio apresentado por esta *query*, consideramos que implementámos uma solução eficiente, tendo aproveitado ao máximo a organização prévia dos dados. Estamos satisfeitos com o desempenho e resultados alcançados.

5.4 Query5

A *Query5* tem como principal objetivo recomendar utilizadores com gostos parecidos. Para isso foi fornecido pela equipa docente uma biblioteca com esse objetivo.

Começamos então por criar uma estrutura que contem os argumentos da função *recomendaUtilizadores*, ou seja, as três matrizes (*UsersIds, nomesGeneros, preferenciasDosUsers*) e o número de *users* com o número de géneros.

Inicialmente, os nossos *array* eram do tipo *int*** ou *char***, no entanto sentimos bastantes dificuldades em fazer uma função de resize que funcionasse e fosse eficiente. Acabamos então por utilizar o *GArray* da *GLib* uma vez que este realiza a alocação dinâmica de memória de forma automática, o que permitiu reduzir significativamente a complexidade do código. Com esses fatores em vista, o *array* externo da matriz *UsersIds* e da matriz *preferenciasDosUsers* passaram a ser representados por um *GArray*.

Para o *array* externo de *nomesGeneros*, identificámos que, ao processar as músicas, já era possível determinar o número de géneros diferentes. Assim, conseguimos alocar a memória necessária diretamente, evitando a necessidade de realocação futura. Este método também nos permitiu obter o tamanho preciso para o *array* interno de *preferenciasDosUsers*.

Após tratamos dos *array* externos, concentrámo-nos no preenchimento das matrizes e dos restantes dados necessários.

Assim, ao processar os dados vindos de *users.csv* no *usersFeed*, sempre que um utilizador era considerado válido, alocávamos memória para o ID correspondente, preenchendo assim a matriz *UsersIds* e incrementando o número total de utilizadores.

Simultaneamente alocávamos memoria para o id de um *user* e também alocamos memória para o *array* de preferências, no caso, o *array* interno de *preferenciasDosUsers* com todos os elementos iniciados a zero.

Posteriormente, ao processar os dados de *history.csv* no *historyFeed*, verificávamos o género da música que era mandado para a função *atualizaPrefsUser* que verifica se o género já foi inserido e, caso não tenha sido, insere. Em seguida, era acedido o índice do *array* interno de *preferenciasDosUsers* correspondente ao índice do género da música, incrementando o valor associado

Desta forma, preenchemos todas as matrizes sendo apenas preciso mandar uma copia das mesmas para o recomendador fornecido pela equipa docente. Como ter tantos *getters* para copias é bastante pesado optamos por apenas dar *get* das matrizes no *querieManager* passar as matrizes como argumentos para poupar tempo de execução.

Consideramos assim que a executamos a *query* da melhor forma que nos foi possível tentando ao máximo respeitar os princípios de encapsulamento e modularidade.

5.5 Query6

A Query6 tem como objetivo fazer um resumo das estatísticas anuais de um *user*, sendo ainda opcional pedir os N artistas mais ouvidos por esse *user*.

Pensamos bastante antes de iniciar a execução desta *query*, continuamos a achar que o "segredo" estaria na forma como trabalhamos os dados. Assim, como explicado anteriormente, temos uma *Hash Table* externa (History), cuja *key* é o *user_id* e contém um *GArray* (Anos) composto por três *GArrays* internos *(musicas, horas e dias)*. Esta estrutura representa grande parte dos dados necessários para a execução desta *query*.

Assim, ao receber o input da *query* acedemos aos dados do *user* que nos foi dado como input e preparamo-nos para os trabalhar de forma a obter os resultados necessários.

Começamos então por percorrer o *GArray* interno musicas, a partir do music_id conseguimos ter acesso ao *album_id* e ao *genero*. Após percorrer todas as músicas acabamos com um *GArray album* que contém o *album_id*, o *albumName* e o *tempoAudicao* e um *GArray genero* que contem o *genero* e o *tempoAudicao* correspondente ao *genero*. Utilizamos o algoritmo *Quick Sort* para ordenar estes dois *GArrays* por ordem decrescente. Por fim, utilizavos a função *snprintf* para colocar na mesma *string* o *genero* e o *album_id* mais popular.

Agora com todos os dados necessários já podemos responder à *query*. Então, caso não nos sejam pedido os artistas mais ouvidos, apenas calculamos os dados necessários para a resposta.

O género e o álbum favorito foram calculados anteriormente, o tempo total de audição é obtido pela função *getNArtistas*, que ao procurar o artista mais ouvido vai, simultaneamente, adicionando os tempos de audição de cada música. O dia em que o *user* mais ouviu música é calculado pela função *getDia* onde é percorrido o *GArray Dia* vendo qual é o dia com maior número de músicas. Por fim, para calcular a hora mais ouvida, utilizamos a função *getHora* onde o *GArray* hora é percorrido e são feitas as comparações necessárias para obter a hora em que mais música foi ouvida. Obtemos assim todos os dados necessários para responder à *query*

Já se nos forem pedidos os N artistas mais ouvidos, repetimos o processo anterior, e utilizamos a função *getNArtistas* para obter os N artistas e os dados referentes aos mesmos que nos foram pedidos.

Concluímos assim que apesar de esta *query* ter sido bastante desafiadora, consideramos a sua execução bastante satisfatória.

6. Recomendador

Apesar de usarmos a biblioteca fornecida pela equipa docente, decidimos também fazer o nosso próprio recomendador, assim como foi sugerido.

Este recomendador foi projetado para receber os mesmos argumentos que o disponibilizado pela equipa docente.

Começamos então por percorrer a lista de *users* até encontrar o *InputUser,* ou seja, o *user* usado como input na *query*. Caso este não seja encontrado, o resultado da *query* é automaticamente definido como *NULL*. Caso contrário, acedemos ao *array* de preferências do *InputUser* para obter as suas preferências.

De seguida, percorremos a matriz de preferências fazendo a soma das diferenças dos valores do *array* de preferencias do *InputUser* e das frequências de cada género musical de todos os *users* um por vez. Esta operação é representada por: *somaDiferencas += (preferenciasAlvo[j] - matrizClassificacaoMusicas[i][j]).*

O resultado destas operações, caso seja diferente de zero, é inserido num *array* de somatório/userId. Simultaneamente, se o resultado de *somaDiferencas* for igual a zero, este é inserido num *array* de resultados. Este procedimento visa evitar percorrer a matriz por completo, dado que o *array* de resultados tem um número limitado de posições. Assim, quando o *array* atinge a sua capacidade máxima, este é devolvido como *output*.

Se não forem encontrados valores suficientes com diferença zero, então procuramos os valores mais próximos de zero de forma a obter todas as recomendações necessárias para contemplar o *array* de resultados. Para tal, usamos o algoritmo *Quick Sort* para ordenar os somatórios por ordem crescente e adicionamos ao *array* resultados os mais pequenos até este estar completo.

Desta forma, consideramos que desenvolvemos um recomendador eficiente e adaptado às limitações impostas, garantindo o cumprimento dos requisitos especificados.

7. Programa interativo

O programa interativo foi um desafio e algo completamente diferente do que já fizemos. Sem grandes conhecimentos começamos por estudar e procurar informações sobre *ncurses* e como realizar o *TUI(Terminal user interface)*.

Dedicamos uma parte do tempo a explorar estes conceitos novos antes de começarmos a idealizar e realizar o programa.

Quando começamos a executá-lo apercebemo-nos da necessidade de implementar um *loop* de modo que o utilizador possa pedir mais que uma *query* sem a necessidade de reiniciar o programa.

Após a implementação do *loop*, apercebemo-nos que o input do utilizador tinha de seguir umas certas regras para as *queries* funcionarem então começamos o processo de validar esse input de acordo com a *query*. Algo que foi, sem dúvida o maior desafio deste programa.

Depois de validar o input e tudo o que achamos necessário, começamos a tratar do *output*. No momento o output estava a ser *printado* num ficheiro e não no terminal como deveria ser.

Este problema poderia ser resolvido de várias formas. Pensamos, brevemente em refazer as *queries*, mas em vez de mandar o *output* para um ficheiro o imprimir no terminal. Rapidamente descartamos esta ideia uma vez que teríamos bastante código duplicado algo que contribuiria em nada para o nosso projeto como um todo.

Assim, decidimos usar as nossas *queries* já existentes e simplesmente ler os ficheiros criados para responder ao utilizador e imprimir o conteúdo dos mesmos.

De seguida, implementamos uma *diretoria default* como sugerido no enunciado de modo a facilitar a execução do programa.

Ao longo da execução desta parte do trabalho apercebemo-nos que o nosso programa não era totalmente resistente aos *inputs* dados, isto é, dependendo do *input* dado pelo *user*, o nosso programa poderia *crashar* então aplicamos as modificações anteriores para tentar resolver isto.

Consideramos, assim que realizamos um programa iterativo eficiente e que respeita aquilo que nos foi pedido.

8. Discussão de resultados

Para testar o desempenho do nosso programa, utilizamos a versão mais recente nos seguintes sistemas. O programa foi executado 10 vezes, com os exemplos de input dados no dataset com erros. Consideremos o Computador 1 como *Acer Predator Helios 300 PH315-55- 79YR Core™ i7-12700H 3,5GHz 16 GB,* o Computador 2 como *Asus Vivobook 15 Core™ i7-1165G7 3,5 GHz 16GB* e o Computador 3 como *Asus Vivobook 14 Pro Ryzen™ 5 5600H 8GHz*.

Seguidamente foi calculada a media dos resultados e obtivemos os seguintes resultados:

Small:

Computador	Q1(ms)	Q2(ms)	Q3(ms)	Q4(ms)	Q5(ms)	Q6(ms)	Total
1	2,73	8,16	0,09	4,76	897,51	0,11	7,36s
2	0,77	10,84	0,22	7,64	771,89	0,24	9,45s
3	1,07	8.31	0,27	5,39	927,09	0,43	7,79s

Big:

Computador	Q1(ms)	Q2(ms)	Q3 (ms)	Q4 (ms)	Q5 (ms)	Q6 (ms)	Total
1	6.52	130,48	5,64	58,93	5181,5	0,80	1m04s
2	7,15	201,90	1,44	89,70	8168,37	1,63	1m22s
3	14,94	155,09	2,80	72,09	6052,94	2,28	1m16s

Como podemos observar os tempos apresentados diferem bastante. Em toda a execução não podemos dizer que houve uma máquina que fosse consistente, isto é que obtivesse sempre valores menores ou maiores que as restantes. Atribuímos isso à diferente combinação de processadores e RAMs.

Apesar do que foi observado, independente do *dataset*, o Computador1 com o processador mais recente (Core™ i7-12700H) é mais rápido do que os restantes.

9. Dificuldades sentidas

Durante o desenvolvimento desta segunda fase do projeto deparamo-nos com várias dificuldades. Desde conseguir encontrar a melhor forma para responder às *queries*, a implementar o encapsulamento

Sentimos uma grande dificuldade nas *queries*, principalmente na *query4* e na *query6*. Em relação à *query4* grande parte da sua dificuldade consistiu em escolher as estruturas certas para tratar os dados e em todas os erros causados pela mal implementação das funções que nos permitiriam calcular a data do domingo anterior.

Já no que toca à *query6*, a estrutura escolhida também foi um dos grandes desafios e apesar de consideramos a nossa estrutura satisfatória reconhecemos também que esta gasta bastante memoria. Juntamente a este desafio esteve unido a gestão e obtenção dos diversos dados para responder a esta query

Semelhante à fase anterior consideramos o encapsulamento sempre um desafio, mesmo já estando mais familiarizados com o conceito. Aliado a este, a performance e a utilização de memoria foram também algo que nos preocupou ao longo de todo este projeto.

Dessa forma e com as ferramentas que nos foram disponibilizadas tentamos analisar minuciosamente o código e implementar alterações que não só nos permitissem aprimorar o nosso uso de memoria como também a performance e respeitassem os princípios do encapsulamento

Consideramos assim que todas as dificuldades foram encaradas com uma postura de aprendizagem. Algo que proporcionou uma maior oportunidade para aprimorar as nossas habilidades técnicas e enfatizou a importância ao detalhe e à procura continua por melhorias de forma a desenvolver um programa cada vez melhor.

10. Conclusão

Em conclusão, o projeto apresentou um desafio complexo, abordando temas desde a correta validação de dados, até à pesquisa pelos algoritmos e estruturas mais eficientes para uma determinada tarefa.

Consideramos assim alcançados todos os objetivos propostos pela realização deste projeto