Universidade do Minho
Escola de Engenharia

Laboratórios de Informática III

Relatório Fase 2

2024 - 2025

Grupo 10
A106837 - Filipa Cosquete Santos
A106915 - Andreia Alves Cardoso
A107382 - Cátia Alexandra Ribeiro da Eira

Conteúdo

1 Introdução **2**
2 Sistema **2**
2.1 Arquitetura do projeto 2
2.2 Processos de resolução das queries 3
2.2.1 Query 1 3
2.2.2 Query 2 4
2.2.3 Query 3 4
2.2.4 Query 4 4
2.2.5 Query 5 4
2.2.6 Query 6 5
2.3 Programa Interativo 5
3 Discussão **5**
3.1 Encapsulamento 5
3.2 Tempos de execução 5
3.3 Estruturas escolhidas 7
4 Conclusão **7**

1 Introdução

Este relatório aborda o processo de realização da segunda fase do projeto de grupo no âmbito da UC de Laboratórios de Informática III, no ano letivo de 2024/2025.

Nesta fase, o foco esteve na realização de três novas *queries*, na criação do programa interativo, e na otimização de *queries* anteriores para um *dataset* maior. O *parsing* e *output* das *queries* implementados na primeira fase mostrou-se capaz de ser facilmente usado nas novas *queries*, pelo que não sofreram alterações, para além do seu reuso nos novos *datatypes*.

2 Sistema

2.1 Arquitetura do projeto

A arquitetura do programa manteve-se muito semelhante à da primeira fase, tendo sido utilizado o mesmo método de armazenamento para os novos *datatypes*. Assim, os dados começam por ser lidos dos ficheiros CSV que são fornecidos ao programa, cada linha enviada então para uma série de funções que extraem da linha a informação útil sobre a respetiva entidade, e a analisam de forma a determinar se se trata de uma entidade válida, de acordo com os requisitos do projeto. Caso tal se verifique, é introduzido um novo elemento na estrutura de armazenamento da entidade correspondente. Caso contrário, a linha original é impressa no respetivo ficheiro de erros.

Uma alteração implementada foi a exclusão de informações que não seriam usadas em nenhuma parte do programa, procurando minimizar o uso de memória.

Depois de todos os dados serem avaliados, o programa efetua uma análise semelhante no ficheiro que contém as instruções a correr, discriminando em cada uma o tipo de *query*, e as informações essenciais à sua resposta. Após cada instrução corrida, é criado o respetivo ficheiro de *output*, que apresenta a resposta à *query* em questão. Mais à frente falaremos detalhadamente sobre o processo de resolução de cada tipo de *query*.

Para manter o código organizado, dividimo-lo em diferentes contextos:

- *datatypes*, que consiste nas *structs* das entidades e comandos, os respectivos *getters*, *setters*, e funções de libertação de memória;
- *managers*, que inclui as definições das estruturas de dados escolhidas e funções que permitem gerir e controlar os dados de forma a respeitar o encapsulamento dos mesmos;
- *input handling*, que consiste nas funções de *parsing*;
- *validation*, que consiste nas funções de validação dos dados;
- *queries*, que consiste nas funções de resolução dos diferentes tipos de *queries* propostas;
- *output handling*, que consiste nas funções de escrita dos *outputs* nos ficheiros devidos;
- *interactive mode*, que consiste nas funções relativas ao funcionamento do programa interativo.

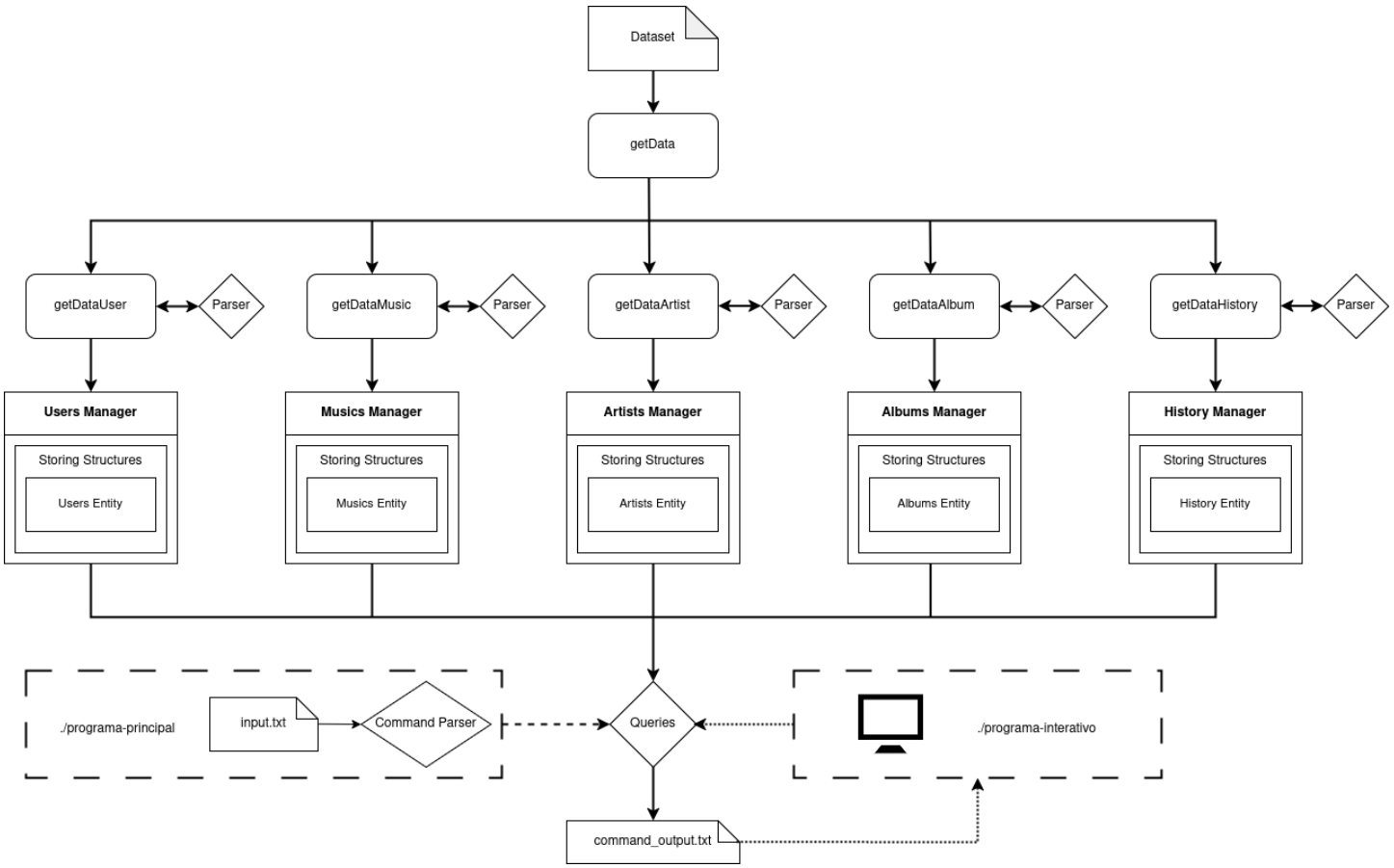


Diagrama simplificado da arquitetura do programa

2.2 Processos de resolução das *queries*

A base do programa da segunda fase manteve-se geralmente igual à da primeira, uma vez que a modularidade torna pequenas alterações fáceis de realizar e facilita o *debugging*. Uma destas modificações foi o acréscimo da formatação do *output*, onde agora é possível escolher qual o separador a usar no comando dado.

2.2.1 *Query* 1

A *query* 1 foi alterada de modo a poder apresentar também o resumo de um artista, além do resumo de um utilizador implementado na primeira fase. Agora, quando recebe o ID, o programa verifica se o mesmo se refere a um utilizador ou artista, recolhendo as informações necessárias consoante.

Assim, se o ID pertencer a um utilizador, o processo seguido é o mesmo implementado na primeira fase. Se pertencer a um artista, começa por conferir se o mesmo é de tipo individual ou grupo. Percorre então a tabela *hash* dos álbuns, contando aqueles que pertencem ao artista pretendido. Por fim, itera sobre as listas ligadas das músicas que pertencem ao artista, de forma a determinar o lucro do mesmo.

2.2.2 *Query* 2

A *query* 2 foi modificada de modo a manter um *array* ordenado com os artistas com maior discografia, juntamente com a tabela *hash* que previamente existia. Para cada artista será verificado se pertence ao país pedido (caso o filtro do país esteja ativo), e será adicionado à tabela e ao *array*. A tabela é usada para manter os artistas e as durações totais, informação necessária para o programa funcionar corretamente, e manter o *array* ordenado evita a iteração e posterior ordenação da tabela *hash*, o que poupa tempo significativo.

2.2.3 *Query* 3

A *query* 3 foi alterada de modo a ter melhor encapsulamento, substituindo o uso direto da tabela *hash* por funções específicas do gestor.

2.2.4 *Query* 4

A *query* 4 pede pelo artista cujas músicas foram ouvidas durante mais tempo, com a opção de filtrar a data de audição. Podemos dividir este processo em 2 fases: guardar e organizar os dados de todas as audições por semanas, e procurar o artista mais ouvido no tempo pretendido.

Para guardar os dados começamos por percorrer a tabela *hash* com os dados do histórico, e, para cada um, é guardada a informação do artista da música ouvida, juntamente com a duração e o dia da audição. Organizamos estes dados em tabelas *hash*, uma por semana, usando a data do primeiro dia da semana correspondente como chave. Quando este processo termina, criamos uma árvore binária de pesquisa apenas com os 10 artistas mais ouvidos em cada semana, organizada também pelo primeiro dia da semana. Todo este processo ocorre apenas uma vez e a árvore fica guardada durante o resto do tempo do programa.

O motivo de usarmos uma árvore é para facilitar a segunda fase da *query*: A pesquisa. Ao ter os dados organizados de tal forma, apenas precisamos de percorrer a árvore de pesquisa e guardar as durações dos artistas dentro do limite das datas que queremos para uma tabela *hash* temporária. No fim, percorremos esta tabela, obtemos o artista que apareceu mais vezes, e finalmente temos o nosso *output* desejado.

2.2.5 *Query* 5

A *query* 5 pretende recomendar utilizadores com gostos musicais semelhantes ao utilizador dado. Optámos por utilizar a biblioteca de recomendador disponibilizada pelos professores.

Começamos por verificar se o utilizador alvo existe e se o número de recomendações pretendidas não é zero. Se uma destas condições não for verdadeira, é criado o ficheiro vazio e a função termina.

Com tudo válido, a função averigua se a matriz e o *array* de IDs necessários já existem, criando-os caso tal não se confirme. Por fim, é então chamada a função do recomendador disponibilizado, e impresso no ficheiro de *output* os utilizadores recomendados.

2.2.6 *Query* 6

A *query* 6 tem como objetivo apresentar o resumo anual de um utilizador. Para isso, recebe o ID do utilizador pretendido, o ano a que as estatísticas se devem referir, e, opcionalmente, um filtro numérico de forma a apresentar os N artistas mais ouvidos pelo utilizador no ano dado, em função do tempo de reprodução.

Começamos então por aceder ao histórico de um determinado utilizador. Para cada um, existe uma lista ligada onde cada nodo diz respeito a uma linha de histórico do mesmo. Daí, o programa verifica se o ano de cada nodo é o pretendido, e, se for, faz o tratamento dos dados do mesmo, caso contrário passa imediatamente para o próximo nodo.

A informação pretendida é armazenada em *arrays*, respeitando todas as condições necessárias de contagem, como por exemplo, se uma determinada música já foi escutada pelo utilizador. Para verificarmos tais condições, armazenamos as informações que precisam ser analisadas também em *arrays* ou *linked lists*, como por exemplo, a lista de músicas já escutadas pelo utilizador.

2.3 Programa Interativo

Para iniciar o modo interativo, usa-se o comando "./programa-interativo", que utiliza a biblioteca *ncurses*. O programa solicita ao utilizador o caminho para o dataset e, por meio de menus e *inputs*, o programa utiliza as funções previamente definidas das queries para calcular o resultado final. O programa então analisa o *output* criado e mostra-o na tela. É importante notar que embora a janela possa ser redimensionada antes do início do programa, ela possui um tamanho mínimo necessário para seu funcionamento adequado.

3 Discussão

3.1 Encapsulamento

Um dos requisitos fundamentais da realização do projeto era seguir o princípio do encapsulamento. Para tal, isolámos todos os dados, criámos funções de *getters,* para poder ter acesso aos conteúdos sem os poder modificar, recorrendo à duplicação de memória, e *setters,* para conseguirmos manipular os dados respeitando sempre o encapsulamento do código, ou seja, inibindo alterações a dados externos ao próprio contexto.

3.2 Tempos de execução

Nesta secção apresentamos uma análise do desempenho do programa nas máquinas dos três elementos do grupo, durante a execução do programa de testes. Esta análise é acompanhada por uma tabela que contém os tempos médios de execução, e que indica ainda especificações do *hardware* das três máquinas, de modo a auxiliar na compreensão das diferenças no desempenho. Apresentamos ainda dois gráficos para auxílio visual na diferença de tempos médios de execução por *query* entre máquinas, no *dataset* pequeno com erros e no *dataset* grande com erros.

Para assegurarmos uma maior confiabilidade dos resultados, foram realizadas 5 execuções de aquecimento, seguidas de 10 medições para cada conjunto de dados. Os valores apresentados na tabela são as médias diretas dos valores obtidos.

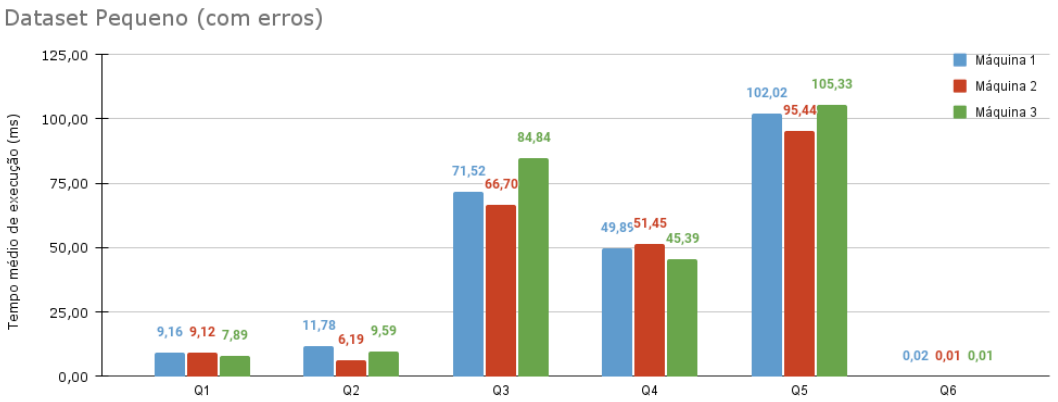
MÁQUINA	1	2	3
OS	Ubuntu Linux	Ubuntu Linux	Ubuntu Linux
CPU	Intel® Core™ i7-1065G7 CPU @ 1.30GHz × 8	13th Gen Intel® Core™ i7-1355U × 12	Intel® Core™ i7-9850H CPU @ 2.60GHz × 12
RAM (GiB)	12	16	16
Cores/Threads	4/8	10/12	6/12
Disco (GB)	512.1	512.1	512.1
		Dataset Pequeno Com Erros	
Tempo Médio Execução (s)	6.41	6.02	6.51
Tempo Médio Q1 (ms)	9.16	9.12	7.89
Tempo Médio Q2 (ms)	11.78	6.19	9.59
Tempo Médio Q3 (ms)	71.52	66.70	84.84
Tempo Médio Q4 (ms)	49.89	51.45	45.39
Tempo Médio Q5 (ms)	102.02	95.44	105.33
Tempo Médio Q6 (ms)	0.02	0.01	0.01
		Dataset Grande Com Erros	
Tempo Médio Execução (s)	116.83	130.07	130.66
Tempo Médio Q1 (ms)	75.25	81.06	73.08
Tempo Médio Q2 (ms)	168.84	99.01	161.62
Tempo Médio Q3 (ms)	158.52	159.20	439.43
Tempo Médio Q4 (ms)	77.45	88.43	71.32
Tempo Médio Q5 (ms)	170.59	159.06	182.29
Tempo Médio Q6 (ms)	0.03	0.02	0.02

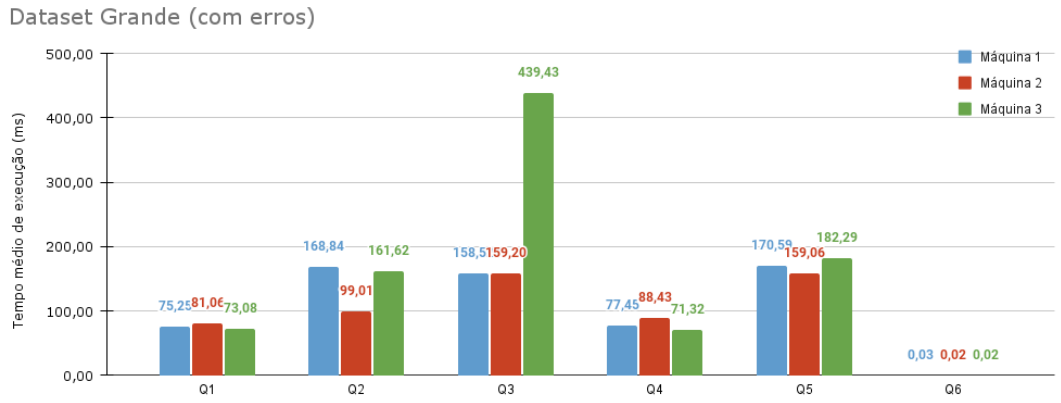
Pela observação dos valores da tabela, vemos que não existem diferenças significativas quanto ao *dataset* pequeno. A máquina 2 apresenta uma *performance* minimamente superior às restantes, devendo-se possivelmente ao facto de possuir um processador mais recente.

Já no *dataset* grande, as máquinas 2 e 3 comportam-se muito similarmente, enquanto que a máquina 1 é cerca de 13 segundos mais rápida. Tendo em conta as especificações de hardware dos três aparelhos, seria de esperar o oposto, dada a menor quantidade de *cores*/*threads* e RAM da máquina 1. Assim, estes resultados podem indicar que o programa não está explicitamente otimizado para paralelismo, dependendo fortemente numa execução de *thread* única, pelo que as arquiteturas mais complexas das máquinas 2 e 3 não são particularmente vantajosas.

Nos gráficos a seguir apresentados, podemos notar que, para ambos os *datasets*, a *query* 2 apresenta a maior variabilidade de tempo médio de execução entre máquinas, sugerindo talvez que a sua implementação possa ser mais sensível ao desempenho do CPU e da memória.

A *query* 3 tem tempos médios de execução especialmente elevados para o *dataset* grande na máquina 3 (439,43 ms), o que indica que pode estar a ser prejudicado pela arquitetura da mesma ou por ineficiências na implementação.





3.3 Estruturas escolhidas

Dada a necessidade de pesquisas frequentes, a principal estrutura de armazenamento de dados que escolhemos para a realização deste projeto foi a tabela de hash. As pesquisas em tabelas de hash são muito eficientes, tornando-as ideais para operações que exigem acesso frequente a elementos individuais, como é o nosso caso.

Além de serem excelentes em pesquisas, as tabelas de hash também suportam iterações eficientes sobre todas as suas chaves e valores, o que foi útil, por exemplo, para o cálculo da duração total da discografia de determinado artista, que precisou de verificar informações em todas as músicas do conjunto de dados.

Recorremos também a outras estruturas auxiliares, como arrays, árvores binárias de procura, e matrizes, dependendo do que cada query necessitava. Por exemplo, se estamos a guardar números que terão de ser todos impressos, é mais vantajoso utilizar um array, já que a sua travessia é eficiente, e a estrutura em si ocupa menos memória do que uma tabela hash.

4 Conclusão

Em conclusão, este projeto permitiu-nos evoluir em vários aspetos, desde a aquisição dos conceitos de modulação e encapsulamento, ao processamento e gestão de grandes quantidades de dados.

Particularmente, pensamos que foi feito um bom trabalho na criação de funções de *parsing* bem encapsuladas. Por outro lado, não chegámos a modificar a forma como os géneros de música são tratados, tendo sido mantido um método de *hardcoding* que apenas considera os 10 géneros existentes, não possibilitando uma eventual adição de mais géneros sem modificação de código.

Assim, quanto a melhorias futuras, poderíamos aplicar uma codificação dinâmica para os géneros de música existentes, e também guardar informação da *query* 3 para reuso nas várias *queries*. Poderíamos também aprimorar a aparência do modo interativo, através, por exemplo, do uso de mais cores. Por último, seria ainda interessante aprofundar o aspecto do paralelismo, fazendo uso do processamento *multithread* para obter tempos de execução mais reduzidos.