LI3 - Relatório da Fase II - Grupo 32

João Delgado - A106836 Simão Mendes - A106928
Pedro Pereira - A107327

3 de janeiro de 2025

Conteúdo

Capítulo 1

Introdução

O presente documento tem como finalidade relatar o trabalho realizado na **Fase II** do projeto de **LI3**, que envolve a criação de uma aplicação destinada ao **processamento de ficheiros CSV** relacionados com componentes de uma **plataforma musical**. Este projeto não se limita à extração e manipulação de dados, abrangendo também a exploração de funcionalidades que proporcionem uma melhor interação com a informação musical.

Os principais objetivos deste trabalho prático incluem o desenvolvimento de um sistema que permita a **leitura otimizada** de dados provenientes de ficheiros CSV, a execução de ***queries*** para a obtenção de informações relevantes e a apresentação de **resultados** de forma clara e sucinta. A aplicação foi concebida de forma **modular**, de modo a facilitar a **manutenção e a expansão** futura do sistema, valorizando igualmente a usabilidade e a eficácia no processamento dos dados.

Entre as funcionalidades mais marcantes do programa, salientam-se a aptidão para efetuar ***queries*** complexas que interligam diversos conjuntos de dados e a aplicação de técnicas de **encapsulamento** que asseguram uma **estrutura limpa e bem organizada**. Este documento descreve detalhadamente as decisões de design tomadas pelo grupo, os resultados alcançados durante o desenvolvimento e as considerações sobre o processo de implementação.

Capítulo 2

Sistema

2.1 Arquitetura

O diagrama abaixo apresenta, de forma simplificada, **os principais módulos funcionais do programa**. Este diagrama ilustra as **interações entre os módulos** e descreve o fluxo de dados desde a entrada até à saída, abrangendo ambos os componentes desenvolvidos nesta fase do projeto.

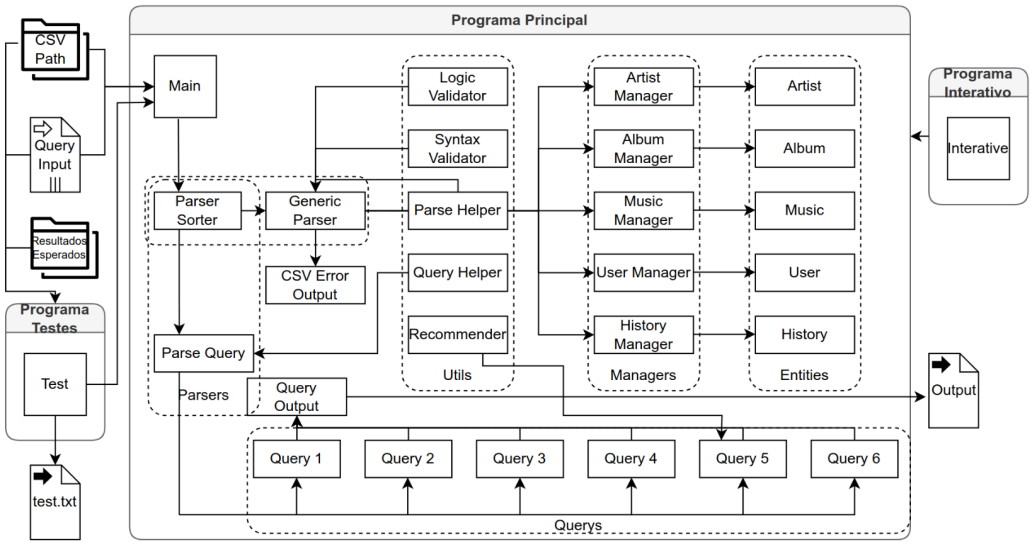

Figura 2.1: Arquitetura geral do projeto

2.1.1 Programa-Principal

O programa principal está estruturado de modo a que os parâmetros de entrada sejam os seguintes:

```
./programa-principal  <dataset-path> <queries-input-path>
```

Após a introdução dos parâmetros, o programa inicia a execução de acordo com o diagrama apresentado anteriormente. Com a análise do diagrama podemos concluir que o sistema está estruturado em várias camadas:

1. **Gestores:** Módulos responsáveis pela gestão das entidades. São responsáveis pela criação, libertação e validação das estruturas de dados que armazenam os conjuntos das mesmas. Estes módulos contêm tabelas *hash* referentes aos principais dados: **artistas**, **músicas**, **utilizadores**, **álbuns** e **histórico**. Além disso, têm como função fornecer às *queries* os dados necessários para a sua execução, preservando o funcionamento interno de cada estrutura de dados e garantindo o **encapsulamento**.

2. **Parsers:** Incluem componentes dedicados ao *parsing* de dados, organizados em três módulos principais:

   - **Parser Sorter:** Responsável por inicializar as estruturas de dados nos gestores, solicitar o *parsing* dos dados em cada categoria e preparar os dados necessários para o funcionamento das *queries*.
   - **Generic Parser:** Implementa toda a lógica de análise dos ficheiros *CSV*, devolvendo um *array* de campos com o conteúdo necessário para que o gestor crie as respetivas estruturas.
   - **Parse Query:** Executa a validação e a divisão do conteúdo de cada *query*, preparando-as para a execução.

3. **Entidades:** São as estruturas de dados que formam o núcleo do projeto. Têm a responsabilidade de armazenar os campos extraídos pelos módulos de *parsing* dos ficheiros CSV de entrada, de acordo com um formato previamente definido. As principais estruturas incluem **artista** (*artist*), **música** (*music*), **utilizador** (*user*), **álbum** (*album*) e **histórico** (*history*), que contêm os dados fundamentais para a execução do projeto. Importa referir que, por **questões de otimização** de **memória**, optámos por não armazenar certos dados, como as letras das músicas e os dispositivos do histórico, pois estes não são necessários para a execução das *queries*. As cinco entidades mencionadas estão detalhadamente descritas no final desta subsecção.

4. **Queries:** Após o *parsing*, cada *query* é executada de forma semi-isolada em módulos independentes. Em termos gerais, as *queries* aproveitam as estruturas já definidas na biblioteca *glib*, o que facilita a sua execução de maneira eficiente. Na secção seguinte, faremos uma análise detalhada de cada uma delas.

5. **Utilidades:** Foram definidos diversos módulos utilitários, responsáveis por tarefas como a validação **sintática** e **lógica** dos dados de entrada. Incluímos também funções destinadas à manipulação de *strings*, com o objetivo de facilitar o *parsing* dos dados e a execução das *queries*. O módulo de **recomendação** (efetuado por nós) para a *query* cinco também faz parte desta categoria e será abordado de forma mais detalhada no capítulo dedicado ao desempenho.

6. **Output:** Módulos responsáveis pela geração de *output* para as linhas que não cumpram os requisitos predefinidos de validação **sintática** e **lógica**, bem como para a saída individual de cada *query*. Existe um módulo genérico para o *output* das *queries*. Além disso, os *parsers* partilham um módulo de saída comum para registar os ficheiros de erro, que reconstrói a linha inválida, caso a validação falhe.

Albúm	Artista	Músicas	Músicas	Histórico
Identificador Artista(s)	Identificador Nome Receita por reprodução Membros (grupo) País Tipo	Identificador Artista(s) Albúm Duração Género	Identificador Email Nome Sobrenome Data de Nascimento País Músicas Favoritas	Utilizador Música Horário de audição Duração de audição

Figura 2.2: Estruturas de dados

Os módulos foram cuidadosamente organizados para garantir o isolamento dos dados específicos de cada um. Esta abordagem facilita a **manutenção** e a **escalabilidade** do sistema, permitindo que cada módulo seja desenvolvido, testado e modificado de forma **independente**. Existem vários módulos implementados com base nas respetivas estruturas de dados, que apresentam semelhanças entre si. Um exemplo disso é o ecossistema *gestor-entidade*, que pode ser identificado em diversos pontos ao longo do projeto.

2.1.2 Programa-Testes

O programa de testes, funciona de forma muito similar ao principal, utilizando da maioria dos módulos do mesmo, no entanto os seus parâmetros de entrada, são ligeiramente diferentes, sendo estes os seguintes:

```
./programa-testes <dataset-path> <queries-input-path> <output-folder-path>
```

Além disso, redefinimos algumas das funções originais com o propósito de registar o tempo de execução de cada *query*. O *output* gerado pelo programa de testes apresenta os seguintes campos: a **execução** de cada *query*, incluindo discrepâncias ou a execução correta; o **tempo médio de execução** para cada tipo de *query*; a **quantidade de *queries* corretas** em cada categoria; o **tempo total de execução** do programa; e **a memória total utilizada**.

2.1.3 Programa-Interativo

O programa interativo não recebe nenhum argumento de entrada, como pode ser observado na sua invocação:

```
./programa-interactivo
```

Dentro do programa, é possível, de forma interativa, carregar os *datasets* e executar as *queries* individualmente. O programa é responsável pela validação dos inputs do utilizador, solicitando especificamente o que deve ser inserido em cada *query*. Além disso, suporta comandos predefinidos, como o **comando "*default*"** para carregar o *dataset* na pasta atual e o **comando "*quit*"** para sair do programa, libertando toda a memória alocada durante a sua execução. Visualmente, o programa é simples, contudo, inclui algumas cores no texto gerado durante a execução.

2.2 Queries

Nesta secção, iremos detalhar a implementação de cada uma das seis *queries* propostas nesta fase do trabalho. O nosso objetivo foi **maximizar** a **eficiência** e **minimizar** o custo de **memória**, o que acreditamos ter sido parcialmente alcançado. As seis *queries* em questão são as seguintes:

1. **Listar o resumo de um utilizador ou artista, consoante o identificador recebido como argumento:** A *query* inicialmente era simples, bastando procurar o utilizador na sua tabela de *hash* e obter

os seus parâmetros. Com a introdução do artista na execução, a situação tornou-se mais complexa, pois foi necessário iterar pela tabela de histórico e realizar diversos cálculos, utilizando estruturas auxiliares, para obter o **dinheiro gerado pelo artist**a e os **álbuns nos quais participou**.

2. **Listar os Top N artistas com maior discografia:** Esta *query* utilizou uma estrutura de dados auxiliar para percorrer a tabela de músicas apenas uma vez, permitindo que as execuções subsequentes reutilizem os resultados obtidos. Foi também considerado um parâmetro adicional, **país**, que restringe a seleção dos artistas. Por fim, os dados foram organizados numa lista ligada, utilizando uma função da *GLib*, e apenas os **N** primeiros artistas foram apresentados.

3. **Listar géneros de música mais populares numa faixa etária:** Esta *query* revelou-se relativamente simples após a implementação da segunda, uma vez que os princípios aplicados foram semelhantes. Utilizámos, desta vez, um *array*, no qual o índice representava a **idade do utilizador** e o valor correspondia à **popularidade por cada música**. Mais uma vez, conseguimos percorrer a tabela *hash* de utilizadores apenas uma vez. Por fim, juntámos todos os elementos do *array* correspondentes às idades solicitadas e, utilizando uma função da *GLib*, ordenámos os dados conforme os parâmetros necessários.

4. **Qual o artista que esteve no top 10 mais vezes:** Sem dúvida, esta foi a *query* mais complicada do projeto, não tanto pelo seu funcionamento, mas pelo **cálculo das semanas** de cada entrada no histórico, que se revelou um desafio, especialmente devido aos casos de exceção que falhavam múltiplas vezes. De forma geral, o procedimento consiste em **iterar** pela tabela *hash* do **histórico** e adicionar a **duração da música ao respetivo artista** na semana correspondente, guardada em um *array* de listas ligadas. No final percorre-se as semanas solicitadas e obtém-se **o primeiro artista da lista**.

5. **Recomendação de utilizadores com gostos semelhantes:** Relativamente simples, não há muito a acrescentar. Seguiu-se o mesmo procedimento da *query* quatro, com a diferença de que, neste caso, se guarda um tipo de dado diferente, nomeadamente um inteiro que representa **o número de vezes que o utilizador ouviu uma música de determinado género**. Tivemos alguns problemas com o recomendador fornecido pela equipa docente, mas acabaram por ser resolvidos

de forma simples. Também implementámos **o nosso próprio recomendador**, sobre o qual discutiremos mais detalhadamente na secção de discussão.

6. **Resumo anual para um utilizador:** Embora tenha sido a *query* que exigiu mais código, não foi a mais complexa. Foi necessário realizar um **extenso processamento de dados** e utilizar algumas estruturas auxiliares para armazenar e ordenar os dados processados. Para **otimizar** o **desempenho**, guardámos uma tabela de *hash* com os identificadores das entradas dos utilizadores no histórico, permitindo iterar pela tabela **apenas uma vez**.

2.3 Makefile e Documentação

O **Makefile** desenvolvido para este projeto é eficiente e de fácil utilização, incluindo algumas *flags* importantes para otimização, como a **-O3**, que permite uma compilação mais rápida e eficiente, e a **-march=native**, que ajusta o código gerado especificamente para a arquitetura da máquina onde está a ser compilado. Adicionalmente, foram integradas as *flags* necessárias para a biblioteca **GLib** e para **math.h**, garantindo o uso adequado das suas funcionalidades no programa.

Para manter a estrutura organizada, os **arquivos objeto** (.o) para os três modos do programa são armazenados em pastas dedicadas dentro da pasta *resultados*. Assim, podemos compilar o programa facilmente com o comando **make**, e remover todos os arquivos de compilação e execução com **make clean**.

Além disso, para a documentação do projeto, utilizamos o gerador **Doxygen**, que facilita a criação de uma documentação detalhada e completa. Com o comando **make doc**, o Doxygen gera uma documentação que descreve todo o código-fonte, incluindo detalhes sobre **parâmetros de funções**, **valores de retorno** e **descrições básicas** de cada função individualmente. Caso necessário, os ficheiros resultantes da documentação podem ser removidos com o comando **make docclean**. Comparativamente com a primeira fase do projeto, optámos por remover a documentação dos arquivos `.c`, à exceção das estruturas de dados, a fim de tornar o **código** mais **limpo** e evitar documentação repetida. Atualmente, a documentação encontra-se apenas nos arquivos `.h`.

Capítulo 3

Discussão

3.1 Estruturas de dados

De forma simples, usamos duas grandes estruturas de dados no decorrer do projeto, **tabelas de *hash*** (*GHashTables*) e **listas ligadas simples** (*GSLists*).

As tabelas de ***hash*** foram fundamentais para armazenar as principais estruturas do projeto, proporcionando acesso rápido em tempo **O(1)** e facilitando operações como inserções e buscas. Embora possam ocorrer colisões, não explorámos em detalhe como a *GLib* as resolve.

Também utilizámos **listas ligadas simples** para processar grandes volumes de dados em *queries* intensivas, como as *queries* dois e quatro, e para armazenar listas dentro de outras estruturas, como músicas favoritas ou artistas associados. Apesar de eficazes, estas listas podem reduzir ligeiramente o desempenho devido à criação individual de cada nó.

3.2 Modularidade e Encapsulamento

Recorremos à separação dos módulos, como descrito no guião três da UC, para tornar o código mais **legível**, **autónomo** e **modificável**. Para isso, além de separar o código em diversos ficheiros independentes, também os organizámos por **categorias** (pastas), como *queries* e utilidades. Esta abordagem permitiu **minimizar a repetição de código**.

No que diz respeito ao **encapsulamento**, aplicámos técnicas como o **uso de estruturas opacas** para omitir a definição das estruturas nos ficheiros *header* e a implementação de ***getters*** e ***setters*** que garantem a **imutabilidade dos dados**, evitando modificações acidentais. A principal técnica foi

definir as tabelas de *hash* como **globais e estáticas** no módulo correspondente, o que impede alterações externas e garante a **segurança** dos dados.

3.3 Recomendador

Criámos **o nosso próprio recomendador** para encontrar utilizadores semelhantes ao especificado na *query* cinco, com uma função semelhante à fornecida pela equipa de *LI3*, mas não utilizando um *array* com géneros musicais. O recomendador baseia-se na **fórmula da similaridade do cosseno:**

$$\textbf{Similaridade do Cosseno} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Após calcular a similaridade entre o utilizador alvo e os outros, subtraímos 1 ao valor obtido. **Valores próximos de zero** indicam preferências musicais **semelhantes**, enquanto valores distantes indicam **menores semelhanças**. Ordenamos os utilizadores pelos valores mais próximos de zero para obter as **N** recomendações. A comparação entre o nosso recomendador e o fornecido pela equipa de LI3 encontra-se no fim deste documento, em anexo.

3.4 Análise de Performance

Em termos de **desempenho**, acreditamos ter alcançado uma **otimização adequada**, embora tenha sido observada uma **desaceleração significativa** ao lidar com *datasets* maiores. Evitamos **operações desnecessárias**, como múltiplas iterações sobre as tabelas de *hash*, realizando-as apenas uma vez por *query*, o que contribuiu de forma significativa para a **redução do tempo de execução** em comparação com a **fase I** do projeto. No entanto, um dos principais desafios foi o aumento anormal do consumo de memória com os *datasets* maiores. Após uma análise detalhada, acreditamos que este comportamento se deve a uma certa **ineficiência das estruturas da biblioteca utilizada**, que ocupa cerca de **1.2 GB** de espaço apenas para o *overhead* das tabelas de *hash* do **histórico**. Além disso, o estilo de programação adotado não foi o mais adequado ao tipo de trabalho exigido por este projeto.

Apesar desses desafios, conseguimos manter o **desempenho dentro dos limites estabelecidos** pela equipa docente da Unidade Curricular. Um detalhe particularmente interessante é que, da maneira como o projeto foi implementado, caso se executem mais *queries*, o tempo de execução por *query* será próximo da centésima de segundo. No final deste relatório, encontram-se as várias *benchmarks* realizadas, incluindo a benchmark dessa curiosidade.

Capítulo 4

Conclusão

De forma geral, consideramos que o nosso **código** está **bem estruturado**, **modular** e **encapsulado**, características que certamente facilitariam a **manutenção** e a **expansão** futura.

No âmbito desta unidade curricular, tivemos a oportunidade de aprender diversos conceitos anteriormente desconhecidos, como a aplicação eficiente de técnicas de **modulação** e **encapsulamento**, bem como a manipulação de **estruturas de dados avançadas**, como tabelas de *hash* e listas ligadas de estruturas.

Este projeto não se revelou apenas uma componente de avaliação, mas também uma oportunidade valiosa para aprimorarmos as nossas habilidades de programação e identificar erros a evitar no futuro, como o uso excessivo de memória mencionado anteriormente. Apesar dos desafios enfrentados, acreditamos ter atingido os objetivos definidos para a unidade curricular de Laboratórios de Informática III.

Capítulo 5

Anexos

Dispositivo	CPU	Memória	OS	Compilador
Zenbook 15 OLED (UM3504)	AMD Ryzen™ 7 7735U	16 GB LPDDR5 6400 MT/s	Manjaro 24.2.1	GCC 14.2.1
Vivobook 16 (F1605 12th Gen Intel)	Intel® Core™ i7-1255U	16 GB DDR4 3200 MT/s	Ubuntu 24.04.1 LTS	GCC 13.3.0
Raspeberry PI 5 8GB	ARM Broadcom BCM2712	8 GB LPDDR4X 4267 MT/s	Raspeberry Pi OS 12	GCC 12.2.0

Figura 5.1: Características dos **dispositivos de teste**

	Recomendador Equipa LI3	O nosso recomendador
Small	Texec: 4.309 s Mem: 502.94 MB	Texec: 4,113 s Mem: 503.38 MB
Big	Texec: 37,700 s Mem: 3 631.42 MB	Texec: 41.705 s Mem: 3 631.40 MB

Figura 5.2: Benchmark de **Recomendadores** (Memória e tempo do *parse* + execução *query* 5)

	Inputs Originais (75/500)	100000 Inputs	Tempo Esperado	Fator de Aceleração	Queries Por Segundo
Small	Texec: 8.602 Mem: 579.252	Texec: 678.108 s Mem: 592.484 MB	11469.305 s	11469.305 / 678.108 = 16.91	147
Big	Texec: 87.054 Mem: 4338.822	Texec: 741.81 s Mem: 4484.332 MB	17410.800 s	17410.800 / 741.81 = 23.47	135

Figura 5.3: Benchmark com **100000** Inputs e comparação com testes normais

Dispositivo	Texec Q1(s)	Texec Q2(s)	Texec Q3(s)	Texec Q4(s)	Texec Q5(s)	Texec Q6(s)	Total Memory (MB)	Texec Total(s)	
Zenbook 15 OLED (UM3504)	0.02016	0.005632	0.319556	0.487258	0.120824	0.030014	578.996	11,16	**Small (sem erros)**
Vivobook 16 (F1605 12th Gen Intel)	0.015464	0.004324	0.28341	0.259626	0.082652	0.0201	579.226	7.262	
Raspeberry PI 5 8GB	0.035876	0.010222	0.527316	0.539956	0.21655	0.057074	568.5	16.67	

Dispositivo	Texec Q1(s)	Texec Q2(s)	Texec Q3(s)	Texec Q4(s)	Texec Q5(s)	Texec Q6(s)	Total Memory (MB)	Texec Total(s)	
Zenbook 15 OLED (UM3504)	0.09625	0.003448	0.068136	0.648786	0.293514	0.139858	4388.522	126.238	**Big (sem erros)**
Vivobook 16 (F1605 12th Gen Intel)	0.07836	0.00268	0.060866	0.362728	0.197418	0.11033	4388.892	81.47	
Raspeberry PI 5 8GB	0.13238	0.005762	0.087082	0.678422	0.412958	0.179832	4238.25	153.342	

Dispositivo	Texec Q1(s)	Texec Q2(s)	Texec Q3(s)	Texec Q4(s)	Texec Q5(s)	Texec Q6(s)	Total Memory (MB)	Texec Total(s)	
Zenbook 15 OLED (UM3504)	0.021408	0.00672	0.341242	0.50374	0.128582	0.038444	578.752	13.554	**Small (com erros)**
Vivobook 16 (F1605 12th Gen Intel)	0.01769	0.005336	0.296546	0.270404	0.089178	0.030872	579.252	8.602	
Raspeberry PI 5 8GB	0.037338	0.010334	0.531074	0.545832	0.219234	0.060122	568.5	18.744	

Dispositivo	Texec Q1(s)	Texec Q2(s)	Texec Q3(s)	Texec Q4(s)	Texec Q5(s)	Texec Q6(s)	Total Memory (MB)	Texec Total(s)	
Zenbook 15 OLED (UM3504)	0.09634	0.003482	0.069072	0.649562	0.291546	0.141538	4388.462	140.156	**Big (com erros)**
Vivobook 16 (F1605 12th Gen Intel)	0.07889	0.002722	0.059596	0.365504	0.19809	0.110046	4388.822	87.054	
Raspeberry PI 5 8GB	0.132692	0.005736	0.086602	0.676436	0.408188	0.177508	4237.776	162.678	

Figura 5.4: **Benchmark Geral**