LABORATÓRIOS DE INFORMÁTICA III

RELATÓRIO DO TRABALHO PRÁTICO FASE 2

Realizado por:
Francisco Ribeiro Martins - A106902 - franciscormartins
Hugo Ferreira Soares - A107293 - yhugosoares
Marco Rafael Vieira Sèvegrand - A106807 - marcosevegrand

ÍNDICE

1. INTRODUÇÃO

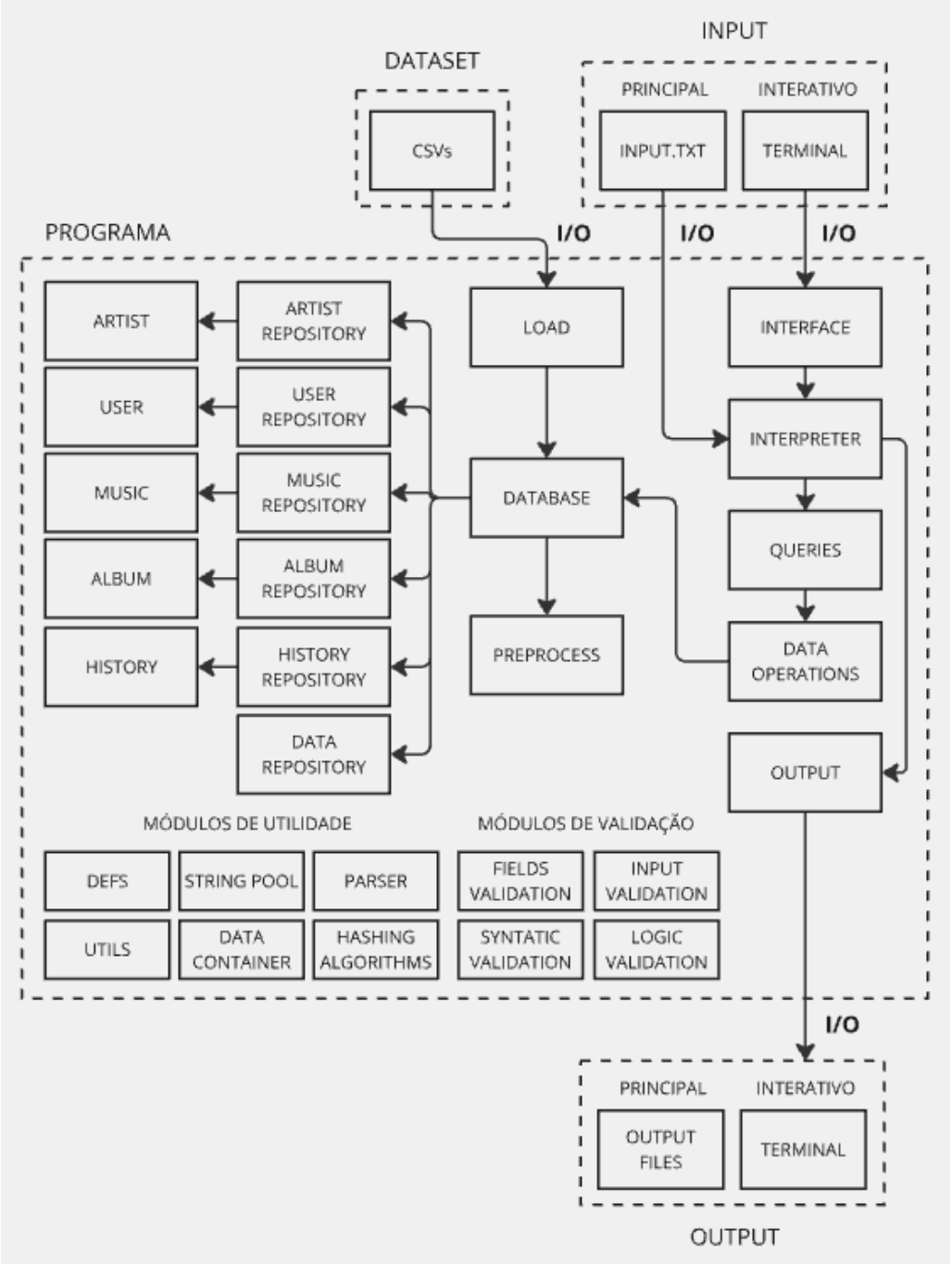
Este relatório visa apresentar o desenvolvimento, a estrutura e a implementação de um sistema de gestão e processamento de dados de artistas, músicas, utilizadores, álbuns e históricos de utilização.
Serão detalhados os objetivos do trabalho, o funcionamento do programa e a organização do sistema. Serão ainda discutidos a modularidade e encapsulamento presentes no projeto e apresentadas justificações para as estruturas de dados utilizadas e uma análise de desempenho a partir dos testes realizados. Por fim, terminaremos com uma breve conclusão onde serão sugeridas possíveis melhorias onde refletimos sobre o aprendizado.

1.1 Objetivos do Trabalho Prático

O trabalho prático tem como objetivo o desenvolvimento de um sistema de streaming de música, utilizando a linguagem de programação C, com particular foco na construção de um código modular, eficiente e encapsulado. O sistema deve ser capaz de carregar, processar e manipular dados relativos a músicas, artistas, utilizadores, álbuns e históricos de utilização, permitindo responder a consultas (queries) específicas sobre os dados. Com a realização deste projeto, pretende-se consolidar competências fundamentais em C, incluindo o uso de conceitos de modularização e encapsulamento, e a aplicação prática de ferramentas de análise de memória e debugging, como o valgrind e o gdb.

2. SISTEMA

2.1 Arquitetura do Sistema (Esquema)



2.2. Descrição e Justificação dos Módulos

O projeto está organizado em módulos lógicos de forma a separar as funcionalidades do programa. Abaixo é apresentada uma descrição dos mesmos tendo sido assinalados os módulos que foram introduzidos na fase 2 do projeto ou que substituíram módulos já existentes:

Entidades
artist, music, user, album e *history*: Contêm as definições da estrutura de dados Artist, Music, User, Album e History, respetivamente, e funções relacionadas à criação, libertação e acesso aos seus campos.

Repositórios
repository_artist, repository_music, repository_user, repository_album e *repository_history*: Módulos de repositórios que fornecem estruturas de dados baseadas em GHashTable para armazenar e gerenciar as estruturas artist, music, user, album e history.
repository_data: Módulo de repositório responsável por armazenar os dados pré-processados que serão utilizados para dar uma resposta mais eficiente às queries existentes.

I/O
load: Responsável pelo carregamento dos dados dos ficheiros CSV para os repositórios correspondentes.
output: Módulo para manipulação de saídas, incluindo a criação e escrita de outputs em ficheiros.
interface: Módulo responsável por gerir a interação com o utilizador no modo interativo do programa recorrendo à biblioteca NCurses para manipular o terminar.

Base de Dados
database: Módulo que encapsula as estruturas de repositórios e centraliza a inicialização e libertação de recursos.
data_operations: Contém as funções usadas pelos módulos das queries para aceder e realizar operações/cálculos com os dados presentes na database

Validação de Dados e Input
input_validation, fields_validation, logic_validation e *syntatic_validation*: Módulos de validação que verificam a coerência e a sintaxe dos dados e inputs fornecidos.

Pré Processamento
preprocess, histories_preprocess, musics_preprocess, users_preprocess: Módulos responsáveis pelo pré processamento dos dados de forma a garantir uma resposta mais eficiente às queries.

Utilidades
defs: Inclui definições de constantes, funções genéricas e macros de verificação de alocação de memória.
parser: Contém funções para manipulação de strings e que fazem parsing de dados para listas.
utils: Inclui funções de tratamento de datas e tempo, libertação de memória e outras utilidades.
string_pool: Módulo responsável pela criação, gerenciamento e libertação de memória da String Pool usada para armazenar os dados da database de forma eficiente.
data_container: Módulo que implementa um Data Container utilizado para facilitar a passagem de argumentos nas funções.

hashing_algorithms: Módulo que implementa funções de hashing com objetivo de aumentar a eficiência das hashtables usadas no programa.

Queries
interpreter: Módulo responsável por processar e executar as consultas com base nos dados carregados.
query1 query2, query3, query4, query5 e query6: Módulos que implementam diferentes tipos de consultas: a query1 procura um utilizador a partir de um ID e lista alguns dos seus atributos; a query2 determina e lista os artistas com a maior discografia em duração podendo ser especificado o país; a query3 determina e lista os gêneros musicais por popularidade dentro de uma faixa etária; a query4 determina o artista que esteve mais vezes no top 10 semanal, com opção de filtro por intervalo de datas; a query5 gera recomendações de utilizadores com gostos parecidos; a query6 fornece um resumo anual para um utilizador específico, incluindo estatísticas como tempo total de audição, artista mais ouvido, gênero favorito, entre outros.

Modes
batch: Módulo responsável pela coordenação das etapas do programa-principal.
interactive: Módulo responsável pela coordenação das etapas do programa-interativo.

2.3 Funcionamento do Programa

O programa foi projetado para gerir e consultar dados sobre artistas, músicas, utilizadores, álbuns e históricos de forma eficiente e organizada. Começa com o carregamento de dados a partir de ficheiros CSV que contêm informações detalhadas sobre cada entidade. Estes arquivos são lidos e os dados são validados para garantir que estejam em conformidade com o formato esperado.
O programa verifica se as informações estão no formato correto e se cumprem os critérios lógicos, como formatos apropriados de datas e durações e a existência de IDs válidos. Esta etapa assegura que apenas dados consistentes sejam processados nas consultas. Entradas validadas são armazenadas em repositórios compostos por HashTables, que facilitam um acesso rápido e estruturado aos dados. Entradas com erros são ignoradas e escritas num ficheiro. De seguida, o programa procede ao pré-processamento dos dados armazenando os resultados num repositório próprio. O programa depois lê um ficheiro com consultas (queries) e, após executadas, os resultados das consultas são organizados e escritos em ficheiros de saída.
A execução do programa é começada por um módulo que inicializa e carrega a base de dados e chama a função responsável pelo processamento das consultas. No final da execução, o programa assegura a libertação adequada dos recursos, garantindo que a memória utilizada seja devolvida ao sistema.

3. DISCUSSÃO E JUSTIFICAÇÕES

3.1. Modularidade

Modularidade: Cada módulo tem uma responsabilidade bem definida, o que facilita a compreensão e a manutenção. Por exemplo:
O módulo *artist* gerencia apenas a lógica relacionada aos artistas, permitindo alterações na estrutura Artist sem impactar o *music* ou *user*. Isto reduz a complexidade do código e torna mais intuitivo fazer manutenções ou expansões do código.
O módulo *interpreter* coordena a execução das consultas, enquanto que a *query1*, *query2*, *query3, query4, query5 e query6* implementam lógicas específicas para cada tipo de consulta, permitindo que novas funcionalidades de consulta sejam adicionadas sem a necessidade de alterar a lógica de processamento.
Reutilização de Código: Implementações genéricas e funções utilitárias estão localizadas na subpasta *utils*, permitindo o seu reaproveitamento em vários módulos. Por exemplo, macros como CHECK_FILE, as funções de parsing ou a implementação de a String Pool são amplamente usadas nos diversos módulos do programa e podem ainda ser reutilizadas para outros projetos.
Testabilidade: Cada módulo pode ser testado isoladamente para garantir que suas funcionalidades estejam corretas. Por exemplo, o *syntatic_validation* pode ser testado independentemente de outros módulos.
Manutenção: Mudanças num módulo (como melhorias na estrutura de Music ao adicionar o campo album_id) não impactam de forma significativa outros módulos. Isto significa que aprimoramentos ou correções num módulo específico não requerem revisões em cascata em outros módulos, minimizando a propagação de erros.
Escalabilidade: A estrutura modular permite adicionar novos módulos, como novas funções de consulta ou repositórios (como se verificou na fase 2 do projeto), com impacto mínimo no código existente. Por exemplo, a adição de uma nova query pode ser feita sem modificar os módulos de repositório ou o módulo *database*. Isto possibilita a expansão do projeto para atender a novos requisitos sem comprometer o código existente.

3.2. Encapsulamento

Encapsulamento: foi aplicado de forma rigorosa em todos os módulos. Por exemplo, as estruturas de dados relacionadas com os artistas, músicas, utilizadores, álbuns e históricos foram implementadas como estruturas opacas nos seus respectivos módulos (*artist*, *music*, *user*, etc.). Apenas as funções expostas nestes módulos permitem armazenar e aceder aos dados de forma controlada.
Flexibilidade: A omissão dos detalhes de implementação na interface (não confundir com o módulo *interface* usado no programa) permitiu que cada módulo utilizasse diferentes representações internas. Por exemplo, o módulo *repository_artist* utiliza uma GHashTable para armazenar os artistas, mas esta decisão pode ser alterada no futuro sem impactar os módulos que utilizam a interface do repositório.
Robustez: O encapsulamento assegurou que qualquer erro associado a dados é responsabilidade do módulo que os controla. Isto reduz significativamente a propagação de erros e facilita a sua identificação e correção. Por exemplo, erros relacionados à

manipulação de datas estão confinados ao módulo *utils*, que é responsável por tratar esse tipo de dado.
Manutenção: Ao encapsular a lógica interna, conseguimos introduzir alterações ou melhorias sem necessidade de modificar módulos externos. Por exemplo, poderíamos otimizar a implementação da String Pool no módulo *string_pool* sem que isso afetasse os módulos que dependem dela.
Outros exemplos práticos no projeto:
O módulo *database* encapsula todas as operações sobre repositórios e abstrai os detalhes de inicialização e liberação de memória. Dessa forma, os módulos que interagem com a base de dados não precisam se preocupar com a estrutura ou alocação de memória dos repositórios.
As consultas (query1, query2, etc.) não acedem diretamente aos dados armazenados nas HashTables. Em vez disso, utilizam funções específicas da interface do módulo data operations que por sua vez recorre a funções específicas a cada repositório, promovendo um alto nível de encapsulamento.

3.3. Justificações das Escolhas de Estruturas de Dados

As estruturas de dados foram escolhidas com base em critérios de eficiência, flexibilidade e adequação ao problema. A seguir, destacamos as principais escolhas e as devidas justificações:

GHashTable (Tabela Hash):
Utilizadas nos repositórios para armazenar entidades como artist, music e user ou os dados pré-processados. Esta estrutura permite procuras rápidas em tempo médio constante O(1), essencial para lidar com grandes volumes de dados. A decisão de usar GHashTable foi baseada na sua eficiência, além das funcionalidades oferecidas pela biblioteca GLib, como hashing automático, mitigação de colisões e ainda a possibilidade de obter as chaves e valores em forma de GLists.
String Pool:
Implementada para armazenar e reutilizar strings de forma eficiente, reduzindo a duplicação de dados e o uso de memória. Apesar de promissora, a análise de desempenho indicou a possível necessidade de otimização para alcançar os resultados esperados.
GPtrArray (Array de Apontadores):
GPtrArray foi usado no módulo *output*, onde a manipulação sequencial e indexada dos dados é predominante. Esta estrutura de dados disponibilizada pela GLib permitiu uma manipulação simples e eficiente dos outputs.
Estruturas Personalizadas:
Módulos como o *data_container* foram desenvolvidos para atender às necessidades específicas do projeto, neste caso facilitar a passagem de argumentos às funções.

Estas decisões tiveram em consideração o desempenho e simplicidade do código, tendo sido ajustadas ao longo do projeto para atender aos requisitos do sistema.

4. ANÁLISE DE DESEMPENHO

4.1 Ambiente de Testes

Os testes foram realizados em 3 ambientes:

Ambiente 1
Processador: AMD Ryzen 7 4800H
Memória: 16GB (DDR4)
Disco: 512GB SDD
Placa Gráfica: NVIDIA GeForce GTX 1650
Sistema Operativo: Kubuntu 24.04 LTS

Ambiente 2
Processador: Intel i5-6200U
Memória: 16GB (DDR4)
Disco: 512GB SDD
Placa Gráfica: Intel Skylake GT2 [HD Graphics 520]
Sistema Operativo: Fedora Linux 41

Ambiente 3
Processador: Intel Core i7-12700H
Memória: 16 GB (DDR4)
Disco: 1 TB SSD
Placa Gráfica: NVIDIA GeForce RTX 3070 Ti
Sistema Operativo: Ubuntu 22.04.3 LTS

3.2 Casos de Testes e Cobertura

Para analisar a desempenho do programa foram elaborados alguns casos-teste tendo em conta diversas métricas. Foram testados 2 datasets (com erros), um de aproximadamente 400MB e outro de 3GB, e dois ficheiros com queries, um com 75 queries e outro com 500 (com a atenção de adequar as queries aos datasets), tendo em consideração que deveriam ser testados para cada query o caso de não serem encontrados quaisquer resultados para a procura solicitada e o caso de ser fornecida uma query de tipo não implementado.

Casos:

Caso 1.1: Dataset 400MB / 75 queries
Caso 1.2: Dataset 400MB / 500 queries

Caso 2.1: Dataset 3GB / 75 queries
Caso 2.2: Dataset 3GB / 500 queries

4.3 Resultados dos Testes

Em cada ambiente, foram executados 3 testes por caso tendo sido a média dos resultados apresentada (consultar tabelas de resultados em 6. Anexos).

4.4 Discussão dos Resultados e Identificação de Limitações

Os testes revelaram o bom desempenho do sistema nas consultas após a etapa inicial de carregamento e pré-processamento dos dados. No entanto, identificaram-se algumas limitações e pontos de melhoria:

Tempo de Carregamento e Pré Processamento:
Esta fase consome um tempo significativo, especialmente para datasets maiores. Uma otimização adicional é necessária para reduzir o tempo de execução, nomeadamente no *parsing* e pré-processamento.
Desempenho das Queries:
A maioria das queries apresentou tempos de resposta satisfatórios, exceto a query5, que por recorrer a uma biblioteca externa incorre em tempos de execução superiores às restantes, uma vez que nos são limitadas as possibilidades de otimização do tempo através do pré-processamento. A implementação de um módulo próprio mais otimizado para o contexto deve ser considerado.
Uso de Memória:
Apesar da implementação de uma String Pool, o uso de memória não atingiu a diminuição esperada. Uma revisão da implementação no sentido de identificar e corrigir possíveis ineficiências seria necessária.
Repositórios com Nested HashTables:
Algumas estruturas de dados baseadas em tabelas hash aninhadas podem ter levado ao aumento da complexidade e do consumo de memória. Substituí-las por matrizes (particularmente na query 3) ou listas para casos específicos pode melhorar a eficiência.

Em resumo, o sistema apresentou um bom desempenho no geral, mas há espaço para melhorias na otimização de tempo e uso de memória, especialmente em cenários de maior carga de dados.

5. CONCLUSÃO

5.1 Propostas de Melhoria

Na fase 2 do projeto, fomos capazes de implementar várias das melhorias sugeridas no relatório da primeira fase, destacando-se a correta modularização do loading dos ficheiros de dados, a implementação de um pré-processamento de dados e a introdução de uma String Pool.
Após a análise de desempenho do programa, identificamos diversas melhorias que poderiam contribuir para otimização do uso de memória e tempo de execução. Nesse sentido, consideramos que a simplificação dos repositórios (removendo uma parte significativa das hashtables adicionais) e o uso de matrizes no lugar de *nested hashtables* trariam uma melhoria significativa nos dois quesitos. Uma revisão da implementação da String Pool também seria algo a considerar.
Estas melhorias propostas visam tornar o sistema mais eficiente, garantindo um desempenho otimizado e uma experiência de uso mais fluida.

5.2 Reflexão sobre o Aprendizado e Aplicação dos Conhecimentos

Este projeto foi uma ótima oportunidade para juntar o que aprendemos sobre programação em C e desenvolvimento de sistemas modulares e encapsulados. Durante a implementação, enfrentamos desafios que nos ajudaram a aplicar conceitos de organização de código, alocação de memória e otimização de performance.
Aprendemos bastante sobre como usar bibliotecas como a GLib para trabalhar com tabelas de hash e outras estruturas de dados, o que reforçou o nosso conhecimento em como criar soluções eficientes. O uso da biblioteca NCurses também nos permitiu ganhar conhecimentos na área de implementação de UI e *user experience*.
Dividir o projeto em módulos mostrou a importância de ter uma arquitetura bem pensada, facilitando a manutenção e expansão do sistema. A aplicação de técnicas de encapsulamento ajudou-nos a criar interfaces limpas entre os diferentes componentes do sistema, escondendo detalhes de implementação e promovendo um código mais robusto e fácil de manter. Isto não só melhorou a organização do nosso código, mas também nos permitiu fazer alterações em módulos específicos sem afetar o funcionamento de outros, demonstrando na prática os benefícios da modularidade e encapsulamento.
Outro ponto importante foi a experiência com ferramentas de debugging e análise de performance, como o Valgrind, que nos ajudou a encontrar leaks de memória e outros problemas, consolidando boas práticas de programação.
Em suma, o projeto foi uma oportunidade valiosa que nos preparou para desafios maiores e mostrou a aplicação prática do que aprendemos durante o curso.

6. ANEXOS

6.1 Tabelas de Resultados dos Testes

6.1.1 Tabela de Resultados no Ambiente 1

	Casos			
	1.1	1.2	2.1	2.2
Tempo de carregamento e pré processamento	21.6 s	20.81 s	186.45 s	183.56 s
Tempo de Execução Médio Query 1	0.006 ms	0.004 ms	0.005 ms	0.005 ms
Tempo de Execução Médio Query 2	0.053 ms	0.217 ms	0.093 ms	0.038 ms
Tempo de Execução Médio Query 3	0.018 ms	0.017 ms	0.008 ms	0.009 ms
Tempo de Execução Médio Query 4	0.299 ms	0.195 ms	0.282 ms	0.442 ms
Tempo de Execução Médio Query 5	52.097 ms	23.588 ms	68.812 ms	68.016 ms
Tempo de Execução Médio Query 6	0.007 ms	0.006 ms	0.045 ms	0.043 ms
Tempo de Execução Total	22.13 s	22.40 s	187.07 ms	187.01 s
Uso de memória	390.38 MB	390.49 MB	2968.16 MB	2968.16 MB

6.1.2 Tabela de Resultados no Ambiente 2

	Casos			
	1.1	1.2	2.1	2.2
Tempo de carregamento e pré processamento	42.72 s	32.17 s	307.33 s	288.66 s
Tempo de Execução Médio Query 1	0.005 ms	0.006 ms	0.009 ms	0.005 ms
Tempo de Execução Médio Query 2	0.068 ms	0.371 ms	0.230 ms	0.069 ms
Tempo de Execução Médio Query 3	0.025 ms	0.024 ms	0.015 ms	0.024 ms
Tempo de Execução Médio Query 4	0.547 ms	0.330 ms	0.758 ms	0.843 ms
Tempo de Execução Médio Query 5	104.746 ms	36.438 ms	100.914 ms	108.618 ms
Tempo de Execução Médio Query 6	0.011 ms	0.007 ms	0.075 ms	0.053 ms
Tempo de Execução Total	43.77 s	34.63 s	308.26 s	294.19 s
Uso de memória	359.08 MB	358.87 MB	2751.17 MB	2751.11 MB

6.1.3 Tabela de Resultados no Ambiente 3

	Casos			
	1.1	1.2	2.1	2.2
Tempo de carregamento e pré processamento	11.96 s	11.94 s	105.54 s	105.27 s
Tempo de Execução Médio Query 1	0.002 ms	0.002 ms	0.004 ms	0.003 ms
Tempo de Execução Médio Query 2	0.039 ms	0.159 ms	0.045 ms	0.025 ms
Tempo de Execução Médio Query 3	0.010 ms	0.010 ms	0.005 ms	0.006 ms
Tempo de Execução Médio Query 4	0.235 ms	0.141 ms	0.219 ms	0.334 ms
Tempo de Execução Médio Query 5	30.621 ms	13.949 ms	34.009 ms	39.190 ms
Tempo de Execução Médio Query 6	0.005 ms	0.004 ms	0.039 ms	0.034 ms
Tempo de Execução Total	12.27 s	12.88 s	105.85 s	107.27
Uso de memória	400.69 MB	400.72 MB	3106.82 MB	3107.25 MB