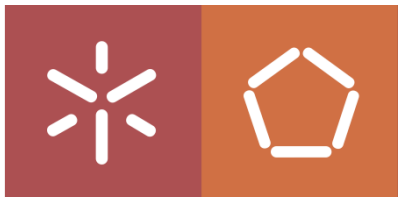
Universidade do Minho

Licenciatura em Engenharia Informática



Laboratórios de Informática III

2024/2025

Relatório 2ª Fase Projeto

Grupo 73

a106804, Alice Soares

a106853, Ana Beatriz Freitas

a106894, Francisco Barros

Janeiro 2025

Índice

Capítulo 1

Introdução

Este relatório aborda a continuação do desenvolvimento do trabalho prático da disciplina de Laboratórios de Informática III, implementado em linguagem *C*, inserido no 2º ano da Licenciatura em Engenharia Informática.

Sendo o tema centrado num sistema de *streaming* de música, a partir do qual exploramos um conjunto de dados relativos a músicas, artistas, utilizadores, álbuns, histórico e estatísticas de uso do sistema, carregamos os dados para estruturas de dados e utilizamos essa informação para responder a um conjunto de *queries*. Pretendemos fornecer uma visão geral do contexto, abordagem e métodos adotados, recorrendo à aplicação prática de conceitos como modularização e encapsulamento.

Nesta segunda fase, a estrutura do projeto continua a enfatizar a organização modular e ainda o encapsulamento. Para garantir a modularidade, o projeto é dividido em diversos módulos, cada um com diferentes responsabilidades, como *parse* de dados, input/output, execução de *queries* e gestão de estruturas de dados. Esta abordagem facilitou a manutenção e a evolução do sistema. Também o encapsulamento, através da criação de métodos, facilitou o desenvolvimento do código de forma organizada.

Aliado a estes, a implementação de estratégias eficientes e o uso de bibliotecas, como a *GLib* e o *nCurses*, e ferramentas auxiliares, como o GDB para *debugging*, o *Valgrind* para análise de uso de memória, o *Doxygen* para documentação, o *GProf* para análise do tempo de execução e o *Docker* para simularmos um ambiente idêntico à plataforma de testes e aí desenvolver todo o projeto.

O desenvolvimento deste trabalho prático inclui a criação de um programa principal que processa dados a partir de arquivos CSV, lidando com validações de dados e gerando respostas para *queries*. Inclui também a criação de um programa de testes e, a par disso, criamos também um programa de testes unitários. Nesta segunda fase disponibilizamos também um programa interativo que inclui um menu interativo via terminal que permite executar *queries* sobre os dados carregados.

Capítulo 2

Metodologia utilizada e Descrição do sistema

2.1 Arquitetura Geral

Utilizando como base a arquitetura da primeira fase realizamos algumas mudanças e melhorias para garantir maior modularidade, como é o caso do novo módulo de input, generalizamos o módulo de resultado *query* para evitar a repetição de código e criamos também um novo módulo de estatísticas a fim de retirar o cálculo dos tempos de execução do programa-principal.

Como é visível pela Figura 1 nos anexos, procuramos manter o fluxo do programa, com ligeiras nuances: ambos os programas (principal e teste) começam por executar o módulo de gestor de programa, onde são carregados os dados, é lido o *input* e é executado o gestor de *queries*. No entanto, apenas o programa de testes inicializa as *estatísticas* para o cálculo dos tempos. Por sua vez, cada gestor das *queries* utiliza o módulo de resultados para os construir e passar ao módulo de *output*, sendo ambos os módulos genéricos. Continuamos ainda a usar um *repositório-dados* que engloba os repositórios de todas as entidades e eventuais dados auxiliares para evitar dependências pois, desta forma, cada um dos repositórios das entidades não dependem entre si.

Temos ainda um módulo de *utils* constituído por funções de utilidades que podem ser usadas em qualquer um dos módulos, um módulo de recomendador próprio e módulos de implementações de estruturas, como é o caso dos comandos, do gestor de *quarks* e das *Min-heaps*.

2.2 Estrutura de dados

Nesta fase foram acrescentadas duas novas estruturas de dados às entidades e os respetivos repositórios: álbuns e histórico. Nos álbuns procedemos da mesma forma que as restantes entidades, já no repositório do histórico, decidimos guardá-lo num índice. Isto é, o histórico referente a um utilizador num ano especifico fica guardado no mesmo sítio. Assim, foi nos permitido manter os dados organizados e a procura destes tornou-se mais fácil e eficiente.

Assim como na primeira fase, criamos ainda novas estruturas auxiliares úteis para a implementação das *queries*.

2.3 Modularidade e encapsulamento

O objetivo do encapsulamento é restringir o acesso direto aos dados apenas ao módulo que os controla, para uma maior facilidade de manutenção, robustez e flexibilidade no código. Esta abordagem promove uma separação clara entre a interface pública e a implementação interna, resultando em um código mais seguro, modular e fácil de manter.

Para manter o encapsulamento ao longo do projeto, fizemos uso de estruturas opacas, criando métodos para as aceder e atualizar. Desta forma, quando queremos fazer algo referente a uma entidade, chamamos uma função do seu módulo que implementa o que for necessário, sem o módulo que chamou essa função ter de saber como é a sua implementação. Isto permitiu que, nos casos em que tínhamos que adaptar certo método, apenas tínhamos que mexer numa parte do código, em vez de estar espalhado pelos diferentes módulos. Assim, cada módulo do projeto mantém o controlo sobre os seus próprios dados e funcionalidades, expondo apenas as operações necessárias para o seu uso externo.

Nos *getters* optámos por retornar sempre que possível um *const* em vez de fazer uma cópia, por causa da eficiência, mesmo sabendo que uma cópia seria mais seguro. As estruturas são definidas e manipuladas apenas no ficheiro de implementação, enquanto o ficheiro de cabeçalho expõe apenas as funções públicas necessárias, mantendo a flexibilidade do sistema.

Capítulo 3

Implementação das funcionalidades

Começamos por melhorar aspetos que tínhamos como objetivo de melhoria no relatório da primeira fase bem como alguns apontados na defesa da mesma, como é o caso da criação de alguns novos módulos, a generalização de outros, o *parsing* dos comandos e melhorias no código em geral.

Das primeiras adaptações que fizemos para ir de encontro aos objetivos desta fase foi atualizar as validações dos dados, o carregamento das novas entidades e a adaptação a eventuais mudanças no enunciado (por exemplo, da query1). Com isto, passamos para a implementação das novas *queries* e funcionalidades, e ainda algumas otimizações.

3.1 Queries

As *queries* são executadas a partir do *gestor-queries*, estando a implementação destas no seu próprio gestor.

3.1.1 Query 1 - Atualização

Listar o resumo de um utilizador ou artista, consoante o identificador recebido por argumento

Como já tínhamos a lógica para os utilizadores implementada da primeira fase, apenas tivemos que acrescentar para os artistas também. Para ter todas as informações necessárias, adicionamos os parâmetros necessários na entidade artista e fizemos os cálculos aquando ao *parse*.

3.1.2 Query 4

Qual é o artista que esteve no top 10 mais vezes?

Para esta *query*, decidimos implementar um módulo de *Min-heaps* com o objetivo de as utilizarmos de modo a guardar e calcular os top 10 de forma mais eficiente. Consideramos as *Min-heaps* uma estrutura adequada tendo em conta que mmantêm na primeira posição o artista menos ouvido, sendo uma maneira mais fácil e eficiente de manter esse top 10.

Assim, guardamos o tempo total de reprodução por artista por semana numa *HashTable* que é posteriormente destruída quando criamos a estrutura auxiliar que guarda os top 10 semanais. Esta estrutura criada posteriormente é também uma *HashTable* em que a *key* é a semana (facilitando a procura das semanas) e o valor é uma *Min-heap* de artistas por reproduções(facilitando a manutenção do top10, como explicado anteriormente).

Posteriormente, já na *query*, esta pode receber o intervalo de tempo que queremos analisar e apenas procura pelas semanas referentes, percorrendo as *Min-heaps* de modo a encontrar o artista que aparece

mais vezes, caso contrário percorre todas as semanas existentes. Para calcular as semanas usamos os dias julianos, de forma a que a cada semana fosse associado um inteiro único.

3.1.3 Query 5

Recomendação de utilizadores com gostos parecidos

Ao longo do carregamento dos dados fomos criando as estruturas auxiliares necessárias, pelo que na *query* apenas tínhamos que ir buscar esses dados e passá-los. Com esta implementação ocupamos muita memória sendo esta também a *query* que mais tempo demora a executar. Consideramos assim que esta é a *query* menos eficiente e pensamos que uma das maneiras para se tornar mais eficiente seria se os ID's dos utilizadores fossem passados como inteiros (ignorando o primeiro caracter 'U') em vez de usar *strings*.

3.1.4 Query 6

Resumo anual para um utilizador

Inicialmente, tentamos usar a mesma estratégia usada nas demais *queries*: guardar apenas os dados necessários à resposta e calculá-los aquando o carregamento dos dados, tornando a execução da *query* mais rápida. Assim não seria necessário guardar o histórico em si, mas sim uma estrutura de resumo anual que ia sendo preenchida durante o carregamento do histórico. No entanto, esta abordagem mostrou-se ineficiente pois a cada estrutura auxiliar usada para fazer os cálculos observamos um aumento de cerca de 200 MB ocupados (com o dataset grande), ou seja, ainda nem tínhamos feito os cálculos para todos os campos necessários e já estávamos a exceder limites de memória. Chegamos à conclusão que teríamos de tentar outra maneira e refletimos em diferentes abordagens.

Depois de umas pesquisas, vimos que cada utilizador não tinha muitas entradas no histórico e até havia utilizadores sem histórico. Optámos então por voltar a guardar o histórico, mas desta vez usando um índice, ou seja, estava guardado por utilizador e por ano. Com isto, tornou-se de rápido acesso os históricos pedidos na *query* e os cálculos necessários para a resposta passaram a ser lá efetuados pois em vez de fazer para todos os utilizadores em todos os anos apenas teriam que ser feitos para os que são pedidos.

3.2 Recomendador

Após alguma pesquisa, descobrimos que normalmente as funções de recomendação utilizam a semelhança entre vetores de características como métrica para escolher as recomendações a dar. Assim, decidimos fazer a nossa própria implementação do recomendador para a *query* 5 e criamos um módulo, estando este a ser utilizado apenas no programa interativo. Este módulo tem como objetivo sugerir utilizadores semelhantes a um determinado utilizador alvo com base no número de audições de géneros musicais.

Para isso, experimentamos duas formas de calcular a similaridade entre os vetores: com a distância euclidiana entre os vetores e com a similaridade dos cossenos. No caso específico deste sistema de recomendação, achamos a distância euclidiana mais adequada para calcular essa semelhança, pois estávamos a lidar com tempos de audições absolutas de géneros musicais, e a magnitude da diferença entre as preferências torna-se relevante, mostrando mais detalhes sobre as divergências de gostos entre utilizadores. A distância do cosseno, por outro lado, é frequentemente utilizada em sistemas de recomendação que trabalham com vetores de características normalizados, onde a direção (não a magnitude) dos vetores é mais importante. Além disso, consideramos ser uma métrica simples e intuitiva que é fácil de implementar e entender, oferecendo uma solução eficiente e eficaz para o tipo de dados tratados.

Assim, o módulo implementa uma abordagem simples baseada na distância entre vetores. Estes representam as preferências de cada utilizador, onde cada elemento do vetor corresponde ao número de audições de um determinado género. A função *recomendaUtilizadores* começa por encontrar o índice do utilizador alvo na lista de utilizadores, de seguida, para cada utilizador existente, é calculada a distância euclidiana entre ele e o utilizador alvo. Uma vez calculadas as distâncias, os utilizadores são ordenados por ordem crescente de distância (onde menor distância implica maior semelhança). Após a ordenação, os N utilizadores mais semelhantes (de acordo com a distância calculada) são selecionados pela ordem da lista, armazenando-os no *array* que é retornado.

3.3 Programa Interativo

O programa implementa um sistema de interface interativa possível de ser manipulado pelo terminal. Este fornece a possibilidade do utilizador navegar entre diferentes janelas e executar operações relacionadas a *queries* definidas anteriormente e estatísticas das mesmas.

Utilizamos as bibliotecas *ncurses* e *panel* para ser possível a manipulação de janelas de texto no terminal bem como outras *features* bastante relevantes das mesmas.

O programa utiliza duas estruturas centrais para gerir o estado e as funcionalidades:

GestorDeProgramaInterativo - Controla o fluxo geral do programa e mantém o estado do programa, estatísticas e os dados carregados.

EstadoProgramaInterativo - Gere especificamente o estado da interface (janela atual, linha atual, se os dados estão carregados...). Possui um *mutex* (*pthread_mutex_t*) para gerir concorrência ao acesso de dados partilhados, garantindo que estes não são corrompidos em operações simultâneas.

O programa começa por criar o estado do programa, configurando variáveis iniciais, como o tema e o estado de carregamento de dados. De seguida, após pedir a introdução do seu nome e a pasta dos dados a ser carregados, entra num *loop* de eventos, aguardando a interação do utilizador. Este pode navegar no menu inicial com o *mousepad* e entre janelas usando as teclas. Assim, as ações deste determinam qual janela será exibida, cada uma tendo uma função específica e o programa muda de estado conforme necessário.

O programa permite ao utilizador personalizar a aparência da interface, através da janela de configurações. Esta funcionalidade sugere a capacidade de alternar entre diferentes esquemas de cores ou estilos visuais, melhorando a experiência de uso e acessibilidade do utilizador e personalizando a mesma.

Para manter o utilizador informado durante processos mais demorados, como o carregamento de dados, o programa implementa uma animação de carregamento. O programa inclui também funcionalidades para processar e exibir estatísticas e resultados de consultas, utilizando as estruturas *gestorQueries* e *estatisticas*. Essas funcionalidades permitem ao utilizador aceder a informações detalhadas e resultados de operações realizadas no sistema, aumentando a utilidade do programa para análise e monitorização de dados.

O utilizador pode ainda escolher entre utilizar um menu interativo mais intuitivo ou uma consola de *queries*, na qual, passando um comando inteiro, consegue vê-lo juntamente com a resposta e as estatísticas em simultâneo.

Capítulo 4

Discussão

4.1 Gestão de memória

Apesar de inicialmente termos dado prioridade a formas de fácil implementação dos aspetos do programa e à praticabilidade das estruturas usadas, fomos percebendo que a atenção à gestão de memória era fundamental para a eficiência de qualquer sistema computacional. Por isso, realizamos diversas otimizações para reduzir o uso de memória e melhorar o desempenho.

Uma das primeiras otimizações implementadas foi a substituição do armazenamento redundante dos dados do histórico. Em vez de manter o histórico em duas estruturas separadas, passamos a armazená-lo apenas pelo índice. Isso não só reduziu significativamente o uso de memória como também simplificou a lógica de busca e acesso aos dados.

Outra otimização que achamos importante foi a substituição de alguns *GArray*. Inicialmente usávamos estas estruturas por serem práticas e de fácil uso. Embora o *GArray* ofereça flexibilidade e funcionalidades adicionais, este consome mais memória. A troca por *arrays* convencionais permitiu a diminuição de memória não sacrificando a funcionalidade necessária.

Também implementamos o uso do *g_slice* em vês do *g_new*, para melhorar a alocação de memória e a sua libertação. Essa funcionalidade da *GLib* é projetada para gerir pequenos blocos de memória com alto desempenho, reduzindo fragmentações e melhorando o tempo de execução.

Sempre que possível, substituímos variáveis do tipo *int* por *short*, pois apesar de não ser tão significativo, também ocupa menos espaço na memória. Essa modificação foi aplicada apenas quando os valores armazenados eram garantidamente pequenos o suficiente para caber no intervalo suportado pelo *short*.

Tentamos usar o *GQuark*, para reduzir o uso redundante de memória com *strings* repetidas, sendo este uma funcionalidade da *GLib* que mapeia *strings* para identificadores únicos. Contudo, durante o uso do *GQuark*, identificamos problemas de *leaks* na memória. Para resolver essa questão, implementamos o nosso próprio 'Quark', uma versão simples de um mecanismo similar, e usámos nos géneros das músicas onde *strings* repetidas eram muito comuns. Esta implementação resultou numa redução significativa do uso de memória uma vez que evitou múltiplas cópias da mesma *string* na memória, reduzindo o consumo global.

Optamos também por substituir o uso de *strsplit* por *strtok* para manipulação de *strings*. A função *strtok* demonstrou ser mais eficiente em termos de tempo de execução e uso de memória, especialmente em operações de *parsing* repetitivas.

Por fim, aplicamos técnicas de alinhamento de tipos nas estruturas para otimizar a disposição dos dados na memória. Essa abordagem assegurou que as estruturas ocupassem o menor espaço possível, reduzindo desperdícios causados por alinhamento inadequado.

A adoção de práticas como substituição de estruturas, uso de tipos mais compactos e implementação

de mecanismos personalizados demonstrou a importância de uma gestão cuidadosa da memória em todo o projeto.

Para percebermos onde havia falta de libertação de memória, usamos o *Valgrind* como auxílio. No momento de entrega apresentamos 0,019 MB de *memory leaks* associados à utilização da biblioteca *GLib* e 305 MB de ocupação de memórias com o modo de economia de memória (com o *large dataset*).

4.1.1 Modo Economia de Memória

A dado momento do desenvolvimento desta fase do projeto percebemos a necessidade de criar um modo de economia de memória uma vez que, depois de todas as otimizações que fizemos, apesar de já não ocuparmos demasiada memória (cerca de 1750 MB com o *large dataset*), tentamos pensar em maneiras de baixar o seu consumo ainda mais.

Assim, percebemos que o repositório do histórico era a estrutura mais dispendiosa em termos de memória e que, muitas vezes, guardava demasiadas entradas que não eram utilizadas nas *queries*. Então, fizemos este "modo"que está ativo por *default*, a não ser que seja passado ao programa principal a informação de correr o programa no modo "normal". Com isto, o programa faz um pré-processamento do *input* para ver quais utilizadores serão necessários guardar o histórico para responder às *queries*. Esta informação é passada ao carregamento dos dados, que apenas vai guardar as estruturas de histórico necessárias e consequentemente, conseguimos baixar significativamente o consumo de memória.

De seguida decidimos também repetir este mecanismo com os utilizadores necessários às *queries*, para apenas carregar os que fossem necessários à sua resposta. Além da memória, também tornou mais eficiente o programa diminuindo o tempo de execução.

Apenas fizemos com estas duas entidades pois foram as mais evidentes em termos de necessidade de diminuição de espaço ocupado e as que eram relativamente mais fáceis de implementar. Admitimos que talvez fosse possível ter feito com mais algumas entidades mas, tendo em conta que teríamos que criar mais estruturas auxiliares para guardar partes da sua informação, não achamos que teria um enorme impacto reconhecendo a eficiência das mesmas se implementadas.

Reconhecemos que este modo é bastante eficiente com ficheiros de *inputs* que não sejam demasiado grandes, ou que obriguem a guardar praticamente todas as entidades de determinado repositório e consideramos uma boa estratégia para programas que tenham que funcionar com um limite de memória.

4.2 Programa de testes

Tal como na primeira fase, o programa de testes foi indispensável ao bom desenvolvimento do código, tanto para verificar os resultados das *queries* como para nos mantermos a par da memória utilizada e dos tempos de carregamento e execução. Continuamos também a usar o programa de testes unitários para validar pequenas funções como, por exemplo, o cálculo das semanas para a *query* 4, tendo sido fundamental para percebermos as discrepâncias nos resultados desta.

4.3 Análise de desempenho

Como é possível ver na Tabela 1 dos anexos incluímos uma tabela que compara o desempenho do programa nos três sistemas operacionais com diferentes configurações de *hardware* utilizados ao longo da execução do projeto.

Capítulo 5

Conclusões

5.1 Análise Critica e Resultados alcançados

Percebemos que um dos grandes desafios deste projeto estava na análise da grande quantidade de dados. Acreditamos que com a modularidade apresentada e o encapsulamento conseguimos estruturar o código de forma eficiente, tornando-o fácil de alterar e analisar. Consideramos que conseguimos cumprir todos os objetivos propostos para esta segunda fase.

Durante esta fase, tivemos a oportunidade de explorar mais a fundo a biblioteca *GLib*, nomeadamente o *g_slice* e o *GQuark*, que tentámos utilizar, mas sem sucesso e, devido a isto, procedemos a uma implementação adaptada do mesmo. Além disso, sentimos que conseguimos aprofundar o uso de ferramentas auxiliares e consolidar ainda mais os conceitos de modularização e encapsulamento.

5.2 Limitações

Durante o desenvolvimento, enfrentámos várias limitações que impactaram o desempenho e a eficiência do programa bem como o desenvolvimento do mesmo. Um dos principais problemas foi identificado durante os testes de carregamento do histórico, que apresentava um tempo de execução excessivamente elevado em comparação com o restante do sistema. Após uma análise detalhada, verificámos que a construção e destruição das entidades, após a transferência dos dados para as estruturas auxiliares, consumia cerca de dois terços do tempo total do nosso carregamento.

Adicionalmente, com a ajuda da ferramenta *gprof*, descobrimos que a função *musica_get_generoMusica* era responsável por cerca de 40 por cento do tempo total de execução. Esta função realizava múltiplas chamadas a *strdup*, duplicando as *strings* associadas aos géneros musicais em cada uma das aproximadamente 7.500.000 execuções, o que causava um impacto significativo no desempenho global, tendo sido um dos fatores decisivos ao uso de *const* nas funções deste tipo.

Uma tentativa inicial para resolver este problema foi substituir os géneros musicais em formato de texto por identificadores numéricos *(generoid)*, criando um repositório de géneros baseado em tabelas de *hash* para facilitar o mapeamento bidirecional. Contudo, esta solução resultou num código mais complexo e difícil de manter, além de não reduzir significativamente o tempo de execução.

Já no programa interativo, como não estávamos familiarizados com a biblioteca *ncurses* fomos obrigados a reescrever o código várias vezes com vista a melhorar e aprimorar o mesmo. Contudo, conseguimos implementar várias *features* extras que permitiram refinar o programa final. Todas estas alterações e reestruturações resultaram numa dedicação extra de tempo que condicionaram a aprimoração e dedicação a outras partes do projeto.

Por fim, considerámos a possibilidade de adicionar um parâmetro ao *parser* para permitir o pré-

processamento das linhas antes da criação dos objetos, de modo a evitar o armazenamento direto das *strings* nas entidades. No entanto, esta solução acabou por não ser utilizada, pois outras otimizações implementadas revelaram-se mais eficazes.

Em relação ao encapsulamento, temos também a noção que poderia estar melhor, principalmente no caso específico da *query* 5 em que passamos diretamente a lista da ordem dos utilizadores. Sabemos que estando a passar uma lista de *strings* a uma função desconhecida devíamos ter passado apenas uma cópia, pois não sabemos se a função a alteraria, no entanto, quando reparámos tarde que estávamos com esta quebra de encapsulamento ainda tentámos resolver mas não tivemos muito tempo para ver uma solução viável com atenção.

5.3 Otimizações e melhorias

Apesar das limitações encontradas, realizámos várias otimizações que melhoraram significativamente o desempenho do programa.

Uma das principais melhorias foi implementada no processo de *parsing*. As validações sintáticas passaram a ser realizadas pelas funções responsáveis pela construção das entidades a partir das linhas de dados antes da criação dos objetos. Caso a validação falhasse, o objeto não era construído, reduzindo, assim, o consumo de tempo e recursos. Adicionalmente, os erros detetados eram imediatamente registados num ficheiro de erros, evitando etapas desnecessárias no fluxo de execução.

Outra otimização importante foi a modificação dos *getters* para retornarem *const char** em vez de duplicarem as strings através de *strdup*. Esta alteração reduziu significativamente o tempo de execução, não só da função *musica_get_generoMusica*, mas também de todas as outras funções que manipulavam *strings* de forma intensiva, tendo sido uma das decisões tomadas em relação ao encapsulamento.

Embora a abordagem inicial de tentar substituir as *strings* por identificadores numéricos tenha introduzido complexidade adicional, os ganhos de desempenho com a utilização de *const char** mostraram-se suficientes para responder às nossas necessidades. Esta solução simplificou o código e melhorou o desempenho, tornando o programa mais eficiente.

Além disso, além das otimizações de memória especificadas no capítulo acima, adotámos outras práticas de otimização, como evitar o carregamento de dados desnecessários e reorganizar o código, o que contribuiu para uma redução significativa do consumo de memória e um aumento da eficiência global do programa.

Anexos

5.4 Primeiro Anexo

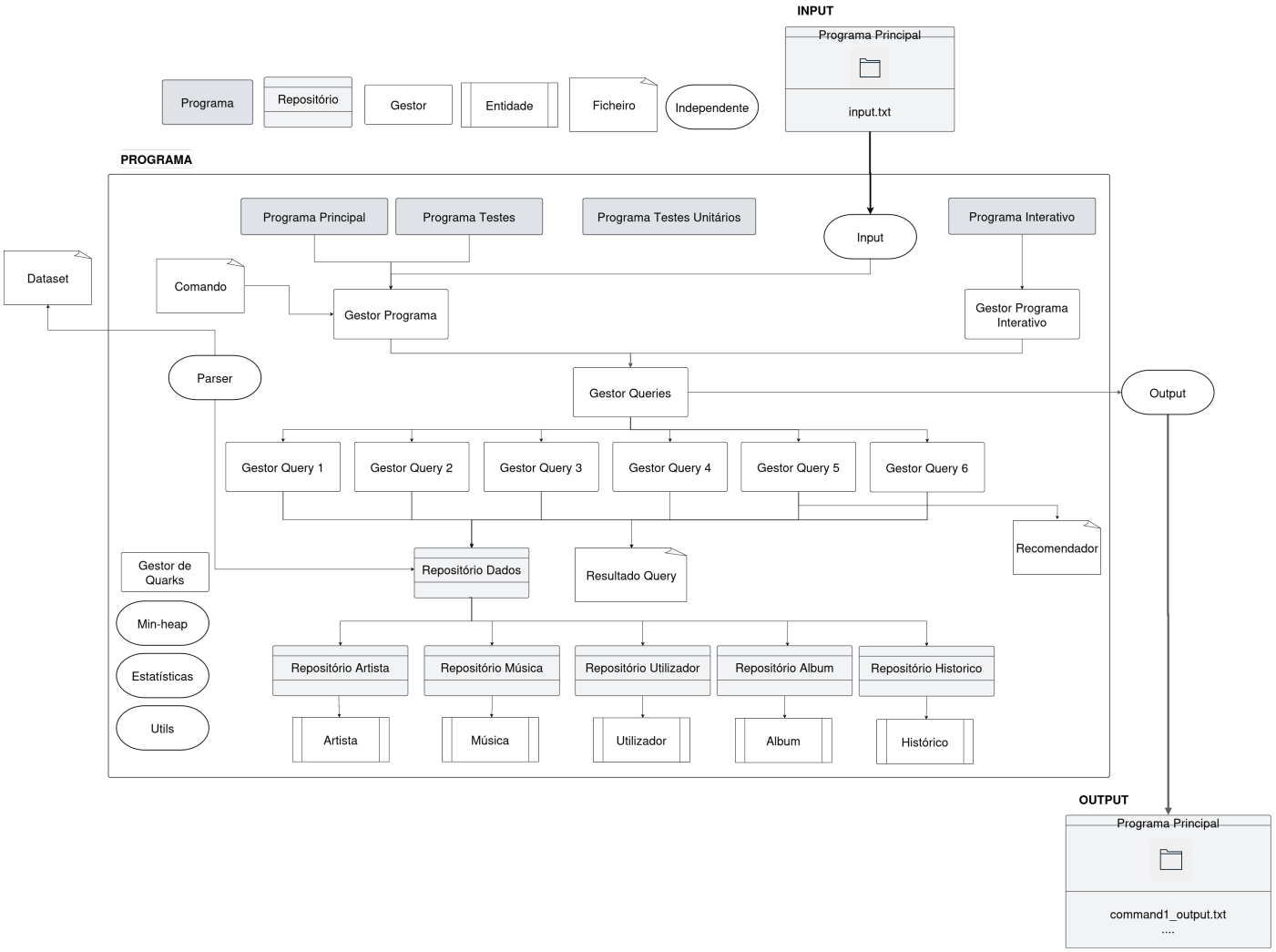


Figura 5.1: Arquitetura de referência para a aplicação a desenvolver

5.5 Segundo Anexo

Dataset	Sistema Operativo	Processador	RAM	Memória	Query	Tempo de Exec (s)
Regular	Linux Ubuntu 24.04.1	i7-1165G	16GB	89.12 MB	1	0.000071
					2	0.002600
					3	0.000033
					4	0.001753
					5	0.001753
					6	0.000034
	Linux Mint 21.7	i7-10875H	32GB	77.29 MB	1	0.000122
					2	0.004360
					3	0.000070
					4	0.002880
					5	0.380264
					6	0.000072
	Windows 11	i7-11800H	64GB	77.27 MB	1	0.000206
					2	0.003483
					3	0.000075
					4	0.003203
					5	0.619959
					6	0.000081
Large	Linux Ubuntu 24.04.1	i7-1165G	16GB	348.46 MB	1	0.000263
					2	0.003700
					3	0.000372
					4	0.014447
					5	7.429860
					6	0.000503
	Linux Mint 21.7	i7-10875H	32GB	297.72 MB	1	0.000253
					2	0.005987
					3	0.000578
					4	0.020029
					5	4.548978
					6	0.000655
	Windows 11	i7-11800H	64GB	297.72 MB	1	0.000849
					2	0.009376
					3	0.000635
					4	0.032657
					5	4.015722
					6	0.001080

Tabela 5.1: Desempenho do Programa com o Dataset Regular e Large