Universidade do Minho
Escola de Engenharia

Laboratórios de Informática III

Trabalho prático - Fase 1

novembro,2025

Grupo 80

Pedro Manuel Macedo Rebelo, A104091

Eduarda Mafalda Martins Vieira, A104098

Margarida Cunha da Silva, A104357


A104091


A104098


A104357

Conteúdo

1. Introdução 3
2. Sistema 4
3. Discussão 5
   3.1. Análise de desempenho 5
   3.2. Modularização e Encapsulmaneto 5
   3.3. Estruturas de dados 5
   3.4. Programa de testes 6
4. Queries 6
   4.1. Query 1: Listar o resumo de um aeroporto 6
   4.2. Query 2: Top N aeronaves com mais voos realizados 6
   4.3. Query 3: Listar o aeroporto com mais partidas entre 2 datas 7
5. Resultados obtidos 7
6. Conclusão 7

1. Introdução

O presente relatório apresentará informações relativas à 1ª Fase do Trabalho Prático da Unidade Curricular de Laboratórios de Informática III, pertencente ao 2º Ano da Licenciatura em Engenharia Informática da Universidade do Minho, realizada no ano letivo 2025/2026.

O projeto final referido consiste na implementação de uma base de dados em memória, utilizando-se ficheiros .csv fornecidos pela equipa docente. Estes ficheiros contêm dados relativos a um **sistema de gestão de voos** (Aeroportos, Aeronaves, Voos, Passageiros e Reservas). O objetivo principal é armazenar estes dados de forma eficiente e implementar métodos de pesquisa e associações entre eles.

São aplicados neste trabalho conceitos fundamentais como modularidade, encapsulamento, reutilização e abstração de dados.

Na 1ª fase, agora concluída, era necessário implementar o **parsing** e a **validação** dos dados de entrada (dos ficheiros airports.csv, aircrafts.csv, flights.csv, etc.) e o modo *batch* (programa-principal), bem como a realização das **3 queries** propostas pela equipa docente.

2. Sistema

A aplicação, através do número de argumentos recebidos, seleciona o seu modo de funcionamento. Nesta fase foi apenas implementado o modo *batch* (programa-principal), sendo apenas este analisado nesta secção.

No início da aplicação, são imediatamente criadas todas as estruturas de catálogo necessárias para armazenar os dados (Catálogos de Aeroportos, Aeronaves, Voos, etc.).

Através do primeiro argumento (o caminho para a pasta dos ficheiros .csv do dataset), é populada a base de dados que contém os catálogos de cada entidade. A leitura de dados do ficheiro é realizada através de um *parser* genérico que recebe apontadores para funções que fazem o tratamento, validação e inserção dos dados nos catálogos (ex: create_and_store_aeroporto). Caso a validação dos dados falhe, o *parser* regista essa linha inválida no respetivo ficheiro de erros (na pasta resultados/).

Uma otimização de performance foi implementada logo após o carregamento dos dados: o programa pré-processa os dados (por exemplo na query2). Isto é, em vez de executar a operação pesada (iterar todos os voos, contar e ordenar) a cada chamada da query, esta lógica é executada uma única vez no arranque do programa, dentro da main.c. Nesse momento, uma lista de aeronaves já ordenada por número de voos é gerada e guardada. Desta forma, quando a query é chamada mais tarde, o trabalho intensivo já está concluído.

Seguidamente, é lido o ficheiro de comandos. Para cada linha, é identificada a query a ser chamada e os seus argumentos. As queries utilizam as funções dos catálogos (e os dados já calculados) para encontrar a informação.

Finalmente, cada query escreve a sua "resposta" no ficheiro de output apropriado e, no final, é feita uma limpeza da memória, evitando *memory leaks*.

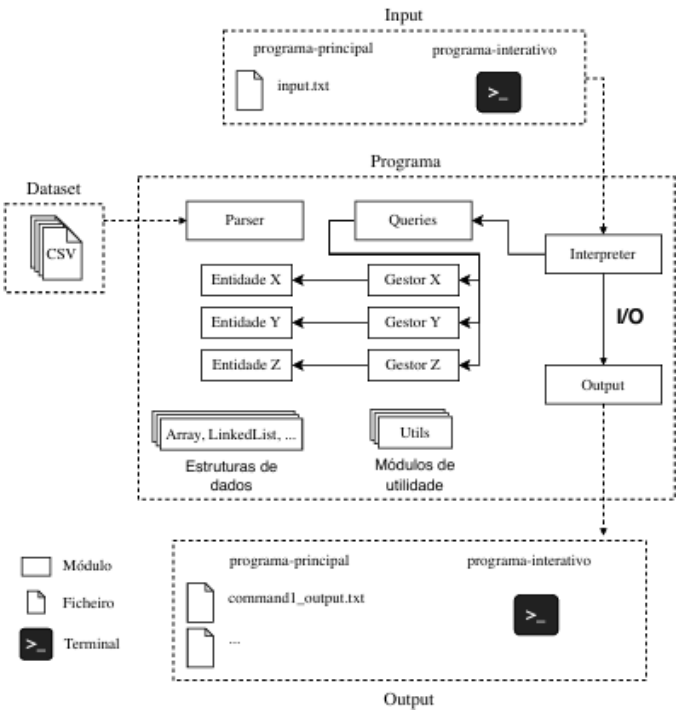

Figura 1- Arquitetura de referência para a aplicação a desenvolver.

3. Discussão

3.1. Análise de desempenho

O desempenho do projeto foi avaliado com base nas queries realizadas, testando-se o tempo de execução e o consumo de memória nas diferentes máquinas dos membros da equipa. Embora as diferenças de desempenho possam ser observadas devido ao hardware variado, os resultados são consistentes com o esperado, dado o uso de algoritmos eficientes, como as HashTables da biblioteca GLib.

3.2. Modularização e Encapsulmaneto

A modularização foi aplicada ao longo do projeto, dividindo-o em várias unidades de código bem definidas e independentes. Criámos módulos para tipo de dados principal (aeroporto, aeronave, voo, passageiro, reserva), módulos para os seus catálogos de gestão (aeroportos, aeronaves, etc.), e módulos de responsabilidade única como parser, validacao, queries e utils.

Cada módulo tem uma responsabilidade clara, facilitando a manutenção e a expansão do código. Por exemplo, a lógica de validação de um aeroporto está contida em validacao.c e é completamente separada da lógica de armazenamento (aeroportos.c) ou da sua definição de dados (aeroporto.c).

A interface de cada módulo é exposta através de um ficheiro .h, escondendo os detalhes de implementação (como a definição da struct aeroporto em aeroporto.c), o que minimiza os efeitos colaterais e a possibilidade de erros. A criação de novas entidades é feita através de funções como create_and_store_aeroporto, que asseguram a integridade dos dados.

3.3. Estruturas de dados

As escolhas de estruturas de dados foram fundamentais para o desempenho e eficiência do sistema, alinhadas com os requisitos para notas superiores.

- **GHashTable (GLib):** Escolhemos GHashTable para os catálogos (por ex., CatalogAeroporto, CatalogAeronave, CatalogVoo) porque precisamos de procurar entidades pelo seu identificador único — por exemplo, código IATA para aeroportos ou ID para voos — de forma muito rápida. Em vez de percorrer listas, um hashtable dá-nos esse acesso quase, o que torna queries como a Query 1 (validações e pesquisas rápidas) muito mais rápidas e simples.
- **GPtrArray (GLib):** Usámos arrays de ponteiros dinâmicos da GLib para armazenar as coleções de dados de forma sequencial (ex: voos_array em voos.c). Isto facilita a iteração sobre todos os elementos, o que foi essencial para implementar as queries 2 e 3, que necessitam de processar *todos* os voos num determinado período ou de um determinado tipo.
- **Structs Auxiliares:** Para a Query 2, criámos uma struct aeronaveAux (utils.c) temporária para facilitar a contagem e ordenação dos voos por aeronave, demonstrando uma solução otimizada para um problema específico. Essas structs são usadas apenas enquanto precisamos de agregar ou ordenar dados e depois descartadas

3.4. Programa de testes

Para garantir a correção do nosso projeto, foi implementado o *programa-testes*, conforme solicitado no enunciado. Este programa automatiza a verificação dos outputs. O seu funcionamento é o seguinte:

- Recebe três argumentos: o caminho para o dataset, o ficheiro de comandos (input.txt) e a pasta com os outputs esperados.
- Executa o programa-principal num diretório temporário para gerar os ficheiros de resultado.
- Compara, linha a linha, cada ficheiro de *output* gerado (ex: command1_output.txt) com o respetivo ficheiro de *output* esperado.
- No final, apresenta um relatório no terminal que indica, para cada query, quantos testes passaram (Q1: 100 de 100 testes ok) e, em caso de falha, aponta a primeira linha onde a discrepância foi encontrada.

Este programa foi fundamental para validar a lógica das queries e para detetar regressões à medida que otimizávamos o código.

4. Queries

4.1. Query 1: Listar o resumo de um aeroporto

A Query 1 (query_aeroport_by_code) recebe um código IATA. A implementação é muito eficiente: utiliza-se o código recebido para fazer uma única pesquisa (catalog_get_aeroporto_by_code) na GHashTable de aeroportos. Se encontrado, os seus dados (nome, cidade, país, tipo) são formatados e guardados. Caso contrário, é retornada uma linha vazia.

4.2. Query 2: Top N aeronaves com mais voos realizados

A implementação da Query 2 (query_list_top_n_aeronaves_by_flightcount) foi desenhada para uma performance elevada através de pré-processamento.

Para evitar calcular a contagem de voos e ordenar as aeronaves em cada chamada, esta operação pesada é feita uma única vez no arranque do programa (em main.c). Após carregar todos os voos, o main.c cria uma lista filtrada (excluindo voos cancelados) e usa a função fill_aeronaveAux para preencher uma tabela de contagem. Esta tabela utiliza uma struct auxiliar (aeronaveAux) que nós definimos para armazenar os resultados, guardando a contagem de voos (fligthCount) associada a cada identificador de aeronave. No final, esta tabela é convertida num GPtrArray e ordenada, ficando guardada

Assim, quando a query é efetivamente executada:

- Obtém a lista de aeronaves já ordenada que foi calculada no main.
- Verifica se existe o filtro opcional de manufacturer.
- Aplica o filtro (se necessário) e devolve eficientemente apenas os primeiros N resultados dessa lista.

Esta abordagem de pré-processamento muda a complexidade da query de uma operação pesada (iterar todos os voos e ordenar a cada chamada) para uma operação muito leve (apenas ler o topo de uma lista já pronta).

4.3. Query 3: Listar o aeroporto com mais partidas entre 2 datas

A Query 3 (query_aeroporto_mais_voos_by_data) primeiro converte os argumentos de data (que vêm como strings) para o formato Data.

De seguida, a função get_aeroporto_mais_voos itera sobre o GPtrArray de todos os voos. Dentro do *loop*, aplica um filtro para considerar apenas os voos cuja data real de partida (actual_departure) está dentro do intervalo especificado.

Para armazenar os resultados da contagem, é usada uma GHashTable temporária (auxHashAeroportos no utils.c). Esta tabela mapeia o código do aeroporto de origem (uma char*) a um ponteiro para um inteiro (int*) que é incrementado. Finalmente, o código itera esta tabela de contagens para encontrar o aeroporto com o valor mais alto, aplicando a regra de desempate lexicográfica

5. Resultados obtidos

Os resultados obtidos nos testes são coerentes com as expectativas, dado que os algoritmos e estruturas de dados utilizadas (HashTables, arrays dinâmicos) são conhecidos pela sua eficiência em situações de grande volume de dados. A modularização e a separação de responsabilidades também desempenham um papel importante na redução da complexidade do código e na facilidade de manutenção, sem afetar o desempenho geral do sistema.

6. Conclusão

O trabalho desenvolvido nesta primeira fase achamos que satisfez, de forma geral, os requisitos pedidos. Este projeto foi desenvolvido com o objetivo de ser o mais eficiente e rápido possível, tentando sempre cumprir as regras de modularidade.

O grande desafio deste projeto foi a programação à larga escala, com base numa grande quantidade de dados disponíveis, sendo imperativo trabalhar com estruturas de dados e algoritmos que se desenvolvessem tanto em complexidade como em eficiência para que pudéssemos concretizar os objetivos.

Isto obrigou-nos a explorar e a conhecer a **GLib** e, consequentemente, ter em consideração os métodos mais eficientes para cada caso. A escolha de estruturas de dados como **HashTables** garantiu-nos que as consultas possam ser feitas de forma rápida e eficaz.

Algumas dificuldades que fomos enfrentando ao longo da realização do projeto foram a resolução de *memory leaks* (através da implementação cuidadosa de funções free para cada entidade e catálogo) e a contínua adaptação das queries para otimização. Uma dificuldade particular foi a complexidade da sintaxe encontrada em certas partes do projeto, como no parser.c. Esta sintaxe, por vezes menos intuitiva, exigiu um esforço adicional de compreensão e depuração.

Outro desafio foi a depuração de erros subtis, isto porque, a nossa Query 3 apresenta uma pequena falha nos testes automáticos que, apesar dos nossos esforços, se revelou muito difícil de encontrar e corrigir.

O sistema cumpriu os objetivos propostos, mas poderá ser aprimorado para a fase futura.