Universidade do Minho

Relatório - Fase 1

Grupo 16

Laboratórios de Informática III

Developers:
Eduardo Luís Dias Pinto
Diogo Silva Fernandes
João Salvador de Magalhães Silva Coelho

Universidade do Minho
2025
Licenciatura em Engenharia Informática

Índice

1 | Introdução

Este relatório descreve o desenvolvimento da primeira fase do projeto prático da cadeira de Laboratórios de Informática III, que teve como objetivo principal o desenvolvimento de competências essenciais à implementação de programas de forma estruturada na linguagem C. O foco pedagógico assenta na aplicação de técnicas de modularização e encapsulamento, garantindo o isolamento entre estas componentes. Adicionalmente, o trabalho visa consolidar a utilização de ferramentas auxiliares de depuração e análise de memória, como o gdb e o valgrind.

É um facto que o tema do trabalho é comum para os 84 grupos, pelo que não deve fazer sentido abordar *o óbvio* que é partilhado entre todos os projetos.

Divisão estrutural e camadas

Organizámos o projeto em camadas bem definidas, de forma a adotar uma separação lógica, garantir a modularidade e **facilidade de manutenção** (que foi essencial para evitar conflitos nos commits entre nós). Fizemos questão de manter cada ficheiro relevante à sua função, no sentido em que houve cuidado na separação do código entre os diversos módulos.

Estruturas de dados

A estrutura de dados que dominou a fase 1 do nosso grupo foi a **HashTable** - já disponibilizada pela GLib - que se deve à própria natureza do projeto: grande parte das operações consiste em procurar rapidamente entidades a partir de um single identifier. Adicionalmente, foi implementada uma **Fenwick Tree** (Binary Indexed Tree) que permite à query 3 ser respondida de forma **quase instantânea** e uma **Min-Heap** para determinar a seleção do Top N de aeronaves sem recorrer a uma ordenação completa.

Nas secções seguintes, detalhamos a arquitetura de sistema que suporta esta divisão, justificamos as escolhas de implementação e apresentamos a análise de desempenho, validada pelo `programa-testes`, que comprova a eficácia da nossa abordagem.

2 | Sistema

A nossa solução foi estruturada de forma modular, seguindo uma arquitetura em camadas que reflete uma clara separação de responsabilidades. Esta abordagem foi adotada não só por ser uma boa prática de engenharia, mas também por uma razão prática de desenvolvimento: com uma equipa de três elementos, esta separação foi essencial para minimizar conflitos de commits e permitir a manutenção e o desenvolvimento paralelo, assim como facilitar o processo de debugging.

```
trabalho-pratico/
├── src/
│   ├── core/
│   │   ├── dataset.c
│   │   ├── fenwick.c
│   │   ├── report.c
│   │   └── utils.c
│   ├── entities/
│   │   ├── aircrafts.c
│   │   ├── airports.c
│   │   ├── flights.c
│   │   ├── passengers.c
│   │   └── reservations.c
│   ├── queries/
│   │   ├── queries.c
│   │   ├── query1.c
│   │   ├── query2.c
│   │   └── query3.c
│   ├── tests/
│   │   ├── comparison.c
│   │   ├── runner.c
│   │   └── stats.c
│   ├── main.c
│   ├── tests_main.c
│   └── validation.c
└── Makefile
```

Listagem 1 : Estrutura de diretórios do código fonte do projeto com omissão dos headers (em `include/`).

2.1 | Arquitetura do Sistema

O design do nosso sistema pode ser visualizado como uma pirâmide de camadas, onde cada camada superior consome os serviços da camada imediatamente inferior, e nunca o contrário. O fluxo de dados é ascendente: o texto (CSV) entra pela Camada 1 e os resultados (queries) saem pela Camada 3.

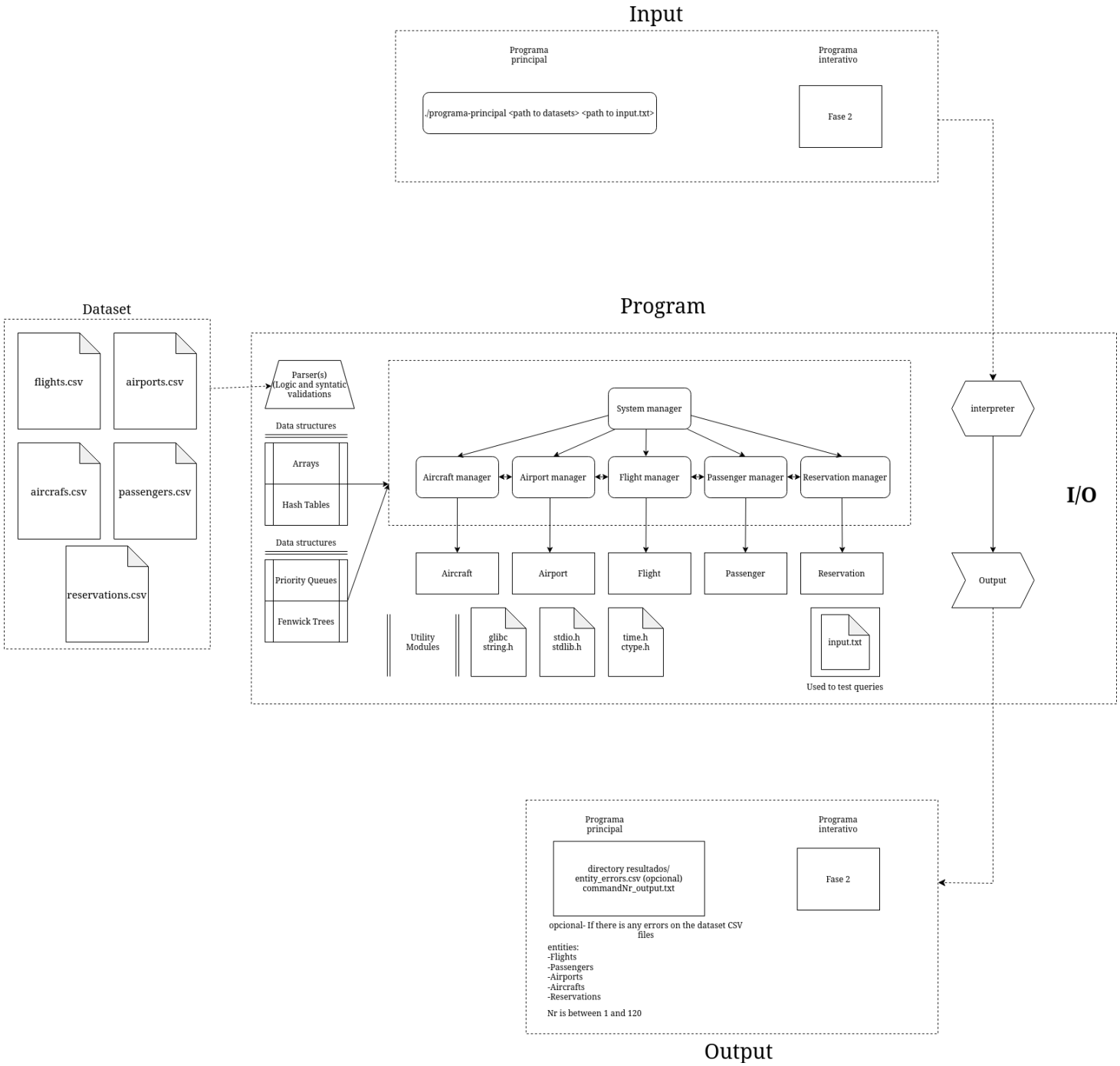

Figura 1 : Arquitetura da aplicação desenvolvida

2.2 | Descrição das Camadas

A justificação para esta arquitetura é pura e simplesmente a questão do isolamento. As queries (camada 3) não precisam de saber como os dados são lidos (camada 1) ou armazenados (camada 2). Da mesma forma, o parsing (camada 1) depende de um conjunto de funções de validação (camada 0) que por si não dependem de nada na aplicação.

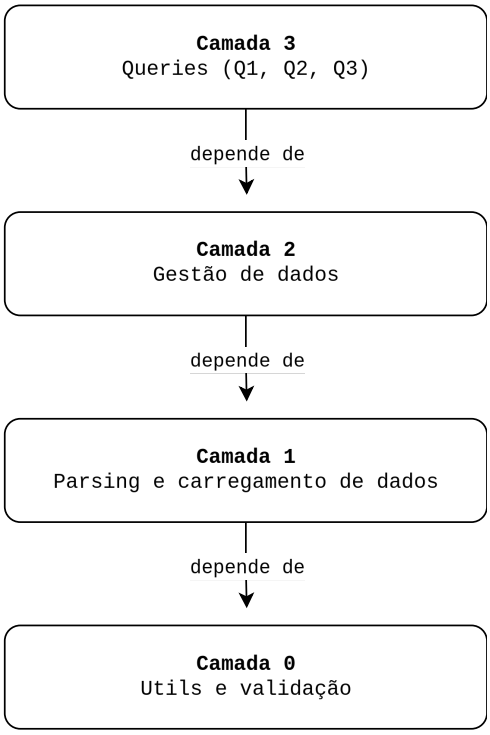

Figura 2 : Diagrama da arquitetura em camadas implementada. O fluxo de dados move-se de baixo para cima (da camada 0 para a 3), enquanto que o flow das dependências é de cima para baixo.

Nota: *Para evitar repetições, ao referenciar implementações vamos apenas mencionar os ficheiros '.c', pelo que a existência dos respetivos header files deve ser deduzida.*

2.2.1 | Camada 0: Validação e Utils

Esta é a camada mais baixa e fundamental do sistema. É composta por módulos que não dependem de qualquer outra parte do projeto.

`src/validation.c`: **Módulo central de validação**. Fornece todas as funções de verificação sintática (ex: `checkDatetime`, `checkAirportCode`, `checkEmail`) e lógica (ex: `checkDestinationOrigin`)

`src/core/utils.c`: **Módulo de utilitários genéricos**, como `trim` (para remover aspas) e `parseDatetime`, usados em todo o projeto.

2.2.2 | Camada 1: Parsing e Carregamento

Esta camada é responsável por I/O e pela transformação de dados textuais (ficheiros `.csv`) em estruturas de dados em memória. É o único ponto do sistema que lê os ficheiros do dataset fornecido.

`src/entities/*.c`: **Cada ficheiro de entidade** (`flights.c`, `airports.c`, `reservations.c`, `aircrafts.c`, `passangers.c`) apresenta a função de parsing correspondente (ex: `readFlights`, `readAirports`). Estas funções iteram sobre as linhas do CSV, fazem parsing manual para os flights e para os passangers, fazem o `g_strsplit` para as restantes entidades e utilizam extensivamente a **camada 0** (`validation.c`) para verificar a integridade de cada campo.

2.2.3 | Camada 2: Gestão e Armazenamento de Dados

Esta camada representa o "estado" da aplicação. É responsável por manter os dados carregados pela camada 1 em estruturas eficientes e prontas a trabalharmos "por cima" delas.

`src/core/dataset.c`: **Gestor de Dados** É ele que "possui" as cinco `GHashTables` principais (para voos, passageiros, etc.) e, de certa forma, gere-os (`initDataset`, `loadAllDatasets`, `cleanupDataset`).

`src/entities/*.c` **(structs)**: As definições das `struct` estão privadas dentro dos ficheiros `.c` das entidades. Esta camada expõe apenas os getters (ex: `getFlightOrigin`, `getAirportName`) nos ficheiros `.h`, garantindo o encapsulamento.

`src/core/fenwick.c`: Este módulo pura e simplesmente *é* uma estrutura de dados avançada e, contra-intuitivamente, pertence a esta camada (e não à de cima). Ele é inicializado pelo `queries.c` (camada 3) e armazena os dados de contagem de partidas de forma otimizada (`FTree`), prontos a serem consultados pela query 3.

2.2.4 | Camada 3: Queries

Esta é a camada superior, responsável por "responder" face aos inputs do utilizador. Ela usa os dados da camada 2 (através dos getters públicos) para produzir os resultados.

`src/queries/queries.c`: Atua como o **Interpretador de Comandos**. É responsável por ler o `inputs_fase1.txt`, fazer o `sscanf` de cada linha, abrir o ficheiro de output correto, e invocar a função de query apropriada.

`src/queries/query1.c`: Implementa a lógica da Q1. Efetua um lookup $O(1)$ na `GHashTable` de aeroportos (da camada 2).

`src/queries/query2.c`: Implementa a lógica da Q2. Utiliza os dados pré-calculados pelo `queries.c` e aplica a sua Min-Heap para encontrar o Top N.

`src/queries/query3.c`: Implementa a lógica da Q3. Interage com as Fenwick Trees (da camada 2) para obter as contagens de partidas no intervalo de datas.

3 | Discussão

Nesta secção, analisamos de forma crítica as decisões de implementação tomadas, justificando-as com base nos resultados de desempenho obtidos e no contexto dos requisitos do projeto.

3.1 | Análise de Desempenho

A metodologia de análise de desempenho baseou-se na execução do `programa-testes`, que não só valida a correção funcional de cada query contra um conjunto de resultados esperados, mas também mede o tempo de execução acumulado por tipo de query e o consumo máximo de memória do programa.

Correndo o programa de testes, obtemos o seguinte output (máquina do Salvador):

```
2526-G16/trabalho-pratico » ./programa-testes dataset-fase-1/
com_erros inputs_fase1.txt resultados-esperados
Loading datasets...
Flights loaded: 1088950 (58.635 seconds)
Passengers loaded: 200000 (3.818 seconds)
Airports loaded: 7354 (0.055 seconds)
Aircrafts loaded: 1000 (0.011 seconds)
Reservations loaded: 18037 (0.229 seconds)
3
Datasets loaded and validated.

Running and checking queries...
All done :)

Check errors.csv files for invalid lines.

Results:
Q1: 40 of 40 tests ok!
Q2: 40 of 40 tests ok!
Q3: 40 of 40 tests ok!


Mem used: 944 MB
Runtime (total acumulated):
Q1: 0.3 ms
Q2: 1.4 ms
Q3: 305.5 ms
Total time: 72091 ms
```

Os testes de desempenho foram executados nas nossas máquinas com as seguintes especificações:

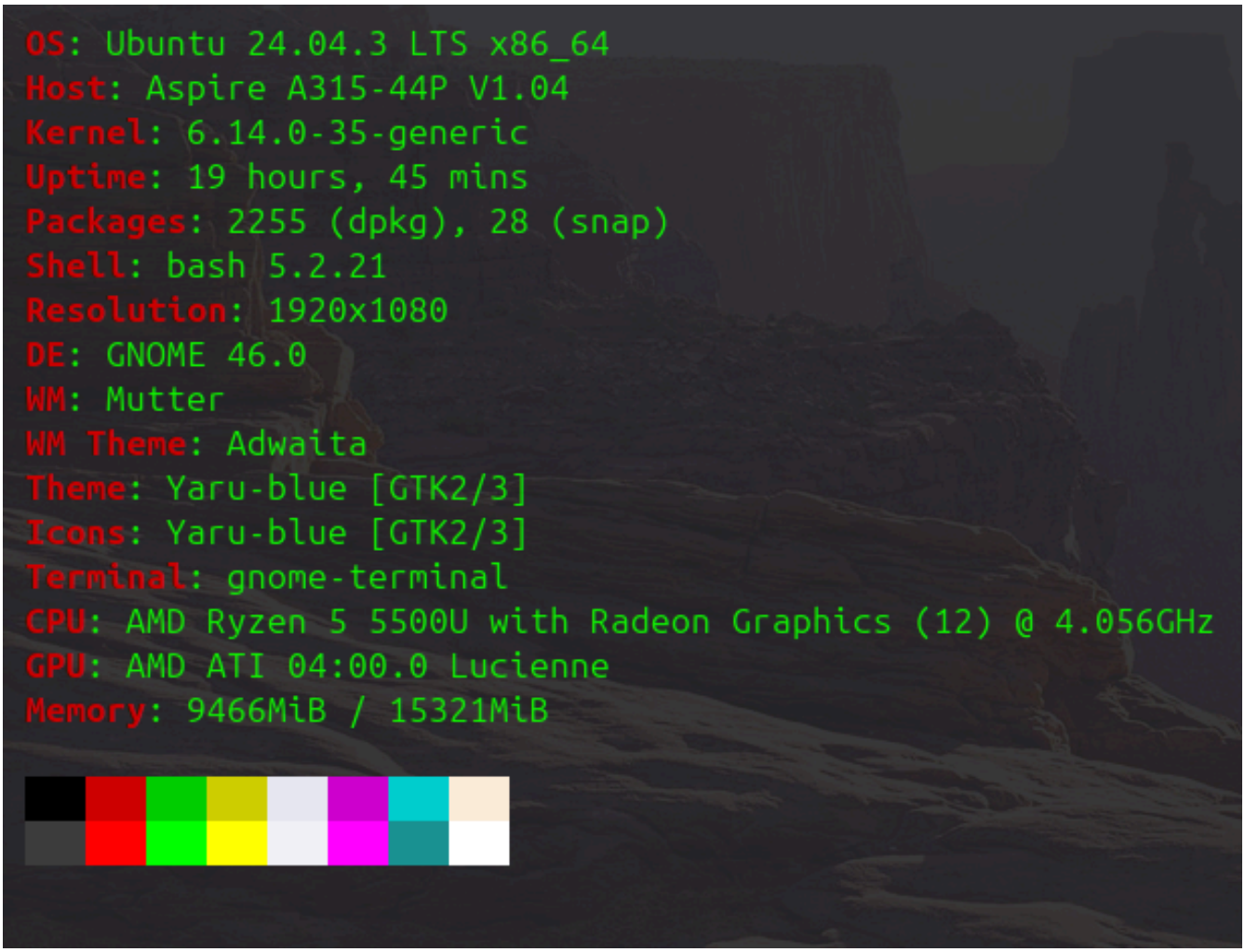

Figura 3 : `neofetch` na máquina do Eduardo

Figura 4 : `fastfetch` na máquina do Diogo

Figura 5 : `neofetch` na máquina do Salvador

3.2 | Técnicas de Modularização (e Encapsulamento)

O objetivo principal desta fase era o exercício de estruturação do programa em vários módulos, com mecanismos que garantissem o isolamento entre eles.

A nossa arquitetura com as 4 camadas, detalhada na secção anterior, é a nossa principal ferramenta de modularização. Ela impõe um fluxo de dependências estritamente descendente (ex: a Camada 3 pode depender da 2, mas a 2 nunca pode depender da 3), o que isola logicamente as responsabilidades.

Fisicamente, esta modularização é implementada através da separação do projeto nas directories que criámos. Os ficheiros .h em include/ definem os headers de cada módulo, enquanto os ficheiros .c em src/ contêm a implementação em concreto privada. Qualquer módulo que queira incluir outro (ex: queries.c a consumir dataset.c) fá-lo apenas incluindo o respetivo .h, permanecendo completamente "cego" aos detalhes da sua implementação.

Embora o enunciado especifique que o encapsulamento será avaliado em detalhe apenas na fase 2, a sua aplicação na fase 1 foi um pilar essencial para garantir que a modularização funcionava. Implementámos um forte encapsulamento de dados da seguinte forma:

- Estruturas de Dados Privadas: Todas as definições de struct (ex: struct flight, struct airport, struct passenger) residem exclusivamente nos ficheiros .c (da camada 2). Elas são expostas em zero headers.
- Getters: Os módulos incluídos são forçados a usar getters (que, como é óbvio, são públicos) que fornecemos. Isto deve-se também ao facto de que usamos typedef para expor as entidades.

Esta abordagem garante que, por exemplo, se a estrutura interna de um Flight for alterada, nenhum dos módulos de queries (camada 3, acima) necessitará de qualquer modificação, desde que a assinatura dos getters se mantenha.

3.3 | Justificação das Estruturas de Dados e Algoritmos

Apesar de este ser um tópico que só é esperado na segunda fase, achamos por bem incluir uma breve exposição do nosso desenvolvimento visto ser uma parte tão fulcral para todo o projeto.

Tendo em conta a natureza de cada entidade e a forma como as queries interagem com elas, decidimos usar certas estruturas para guardar os dados que são recebidos.

3.3.1 | GHashTable:

A decisão de maior impacto na performance geral foi a utilização da GHashTable, fornecida pela biblioteca GLib, como a estrutura de dados primária para o armazenamento de todas as entidades na camada 2.

Justificação: O caso de uso mais comum no sistema é o lookup de uma entidade através do seu ID (ex: um aeroporto, um voo, uma aeronave). Ao usar uma HashTable, garantimos que a query 1, que procura um aeroporto por código, tenha uma complexidade média de $O(1)$. Mais importante ainda, todas as validações lógicas (validateDataset em dataset.c) e lookups internos (ex: getFlight, getAirport) que ocorrem durante o carregamento e execução das queries partilham desta mesma eficiência, eliminando um potencial bottleneck sistémico.

3.3.2 | Query 2: Pré-cálculo e Min-Heap

Para a Query 2 (Top N aeronaves), uma abordagem ingénua (que considerámos inicialmente) implicaria iterar todos os voos, contar as ocorrências por aeronave e, finalmente, ordenar todas as aeronaves para encontrar o Top N. Esta abordagem seria altamente ineficiente.

A nossa solução é composta por duas fases de otimização:

Pré-cálculo (nas queries.c): Durante o arranque, iteramos O(F) sobre os voos uma única vez e preenchemos um array de contagens (flightCounts). Isto isola a operação mais time-heavy (processar os voos) num passo de inicialização.

A função da query recebe este array pré-calculado e utiliza uma Min-Heap de tamanho N. Ao iterar sobre as A aeronaves, cada uma é comparada com o menor elemento da heap. Se for maior, substitui-o e a heap é re-equilibrada. Esta seleção é drasticamente mais rápida do que uma ordenação completa, especialmente quando N é muito menor que A.

3.3.3 | Query 3: Fenwick Tree (Binary Indexed Tree)

A Query 3 foi aquela *menos imediata* para se desenvolver.

Para garantir uma resposta quase instantânea, implementámos uma estrutura de dados avançada na camada inferior à das queries:

1. Pré-processamento de Datas (no dataset.c): No arranque, a função getDatesTable processa todos os voos e cria um array ordenado de datas de partida distintas para cada aeroporto.

2. Construção da Árvore (em fenwick.c): A função getFTrees usa essa informação para construir uma Fenwick Tree (FTree) por aeroporto. Esta estrutura armazena as contagens de partidas por dia de forma conjunta.
3. Execução da Query (na query3.c): Quando a Q3 é chamada, ela apenas necessita de encontrar os índices das datas de início e fim (através de uma binary search) e, de seguida, chamar ftree_range_sum.

O resultado é que o tempo de resposta da Q3 é ínfimo, independentemente de possíveis milhões de voos num futuro dataset, justificando o seu desempenho superior nos testes.

3.3.4 | Otimização Seletiva de Parsing

Finalmente, o próprio tempo de carregamento (justamente na camada 1) foi otimizado. Reconhecemos que g_strsplit tem um overhead de alocação de memória... notável.

Para os ficheiros mais pequenos (airports.c, aircrafts.c), o g_strsplit foi mantido pela sua legibilidade e segurança. Para os ficheiros mais volumosos flights.c e passengers.c, no entanto, implementámos um parser manual com manipulação de pointers. Esta abordagem evita alocações de memória repetidas dentro do ciclo de leitura, reduzindo significativamente o tempo de arranque do programa.

3.3.5 | Multithreading (GLib)

Considerámos ativamente usar as capacidades de multithreading da GLib (como GThreadPool) para otimizar o programa, especialmente para carregar o dataset da fase 1 (camada 1), onde poderíamos processar os 5 ficheiros CSV em paralelo.

Descartámos esta ideia por uma razão puramente pragmática. Após questionar o professor Rui das aulas práticas, ele informou que a plataforma de testes online avalia os projetos num ambiente single-core (por grupo). A implementação de multithreading aumentaria a complexidade do código (gestão de locks, etc.) sem qualquer ganho de desempenho no cenário de avaliação, podendo até introduzir overhead e penalizar o tempo total!!

4 | Conclusão

O desenvolvimento da Fase 1 deste projeto foi um exercício prático interessante. O trabalho permitiu-nos consolidar os objetivos pedagógicos da unidade curricular - a aplicação de técnicas de modularização, a gestão de memória, a otimização de desempenho, entre outras.

A arquitetura que adotámos provou ser uma ferramenta de gestão de projeto essencial. Ao isolar as responsabilidades (validação, parsing, gestão de dados, queries, etc.), facilitou o desenvolvimento paralelo na nossa equipa, minimizou conflitos e permitiu-nos depurar o sistema de forma muito mais eficiente.

Como principal aprendizagem, destacamos a importância *crítica* da escolha da estrutura de dados correto. A diferença de desempenho entre uma abordagem, digamos de iteração, e a implementação de estruturas otimizadas foi possivelmente o fator determinante para o sucesso do projeto.

Apesar dos nossos cuidados, ainda temos muita margem de melhoria tanto nos tempos de execução como na memória utilizada.

Este desafio define claramente o "traçado" do nosso caminho para a fase 2 (para além da implementação das queries adicionais). Futuras melhorias passam por explorar técnicas de para evitar o uso de tanta memória e lazy parsing (para otimizar o tempo de carregamento), sem contar com otimizações das estruturas que já temos e talvez repensar nelas.

Concluímos a Fase 1 com todos os objetivos funcionais e de modularidade cumpridos, e com uma fundação de código com performance boa e bem estruturada, pronta para as otimizações e novas funcionalidades da próxima fase.