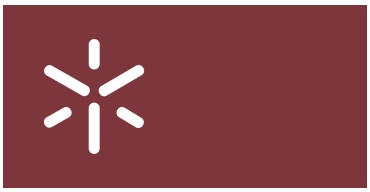

LI3 - Relatório da Fase I - Grupo 35

Débora Caetano (A112332)
Matheus Azevedo (A111430)
Francisco Martins (A111775)

Novembro de 2025

Resumo

Este relatório documenta a Fase 1 do trabalho prático de Laboratórios de Informática III. O projeto centra-se num sistema baseado em aeroportos e voos , com ênfase na aplicação de conceitos de **modularização** e **encapsulamento** em linguagem C. Esta fase detalha a implementação do **parsing** e da **validação** de um conjunto de dados, os datasets , garantindo a integridade dos dados para a posterior execução de um conjunto de **queries**.

1.1 Introdução

1.1.1 Contextualização

Este **trabalho prático** tem como **objetivo** desenvolver **competências** em programação estruturada em C, com especial foco em **modularização**, **encapsulamento** e **gestão de memória**. O projeto centra-se num **sistema de gestão de aeroportos e voos**, que permite carregar e validar datasets relativos a aeroportos, voos, aeronaves, passageiros e reservas, e **responder a queries** definidas pelo enunciado.

O relatório descreve a organização do sistema, a metodologia de execução das queries, a validação de dados, o desenvolvimento de testes e a análise de desempenho. São também abordadas decisões de implementação, como a escolha de estruturas de dados e estratégias de ordenação e acesso eficiente à informação.

Esta primeira fase concentra-se no parsing e validação dos ficheiros de dados em formato CSV, garantindo que apenas registos válidos são armazenados no sistema. Embora nem todas as entidades sejam armazenadas integralmente, uma vez que ainda não é necessária (por exemplo, os passageiros são guardados apenas como uma hash table em que a chave é o document number e ainda não armazenamos as reservas), todos os datasets são totalmente validados tanto sintaticamente como logicamente.

Além disso, a fase 1 inclui a implementação de três queries principais:

Query 1 (Q1): Listar o resumo de um aeroporto dado o seu identificador;

Query 2 (Q2): Listar as N aeronaves com mais voos realizados, podendo aplicar um filtro opcional por fabricante;

Query 3 (Q3): Identificar o aeroporto com mais partidas dentro de um intervalo de datas.

O sistema foi desenhado de forma a suportar estas queries de forma eficiente, utilizando arrays ordenados para travagem antecipada em intervalos de datas e hash tables para acesso rápido às entidades, garantindo tanto a correção como a eficiência na execução das queries.

1.1.2 Pontos relevantes

Nesta fase são implementados diversos componentes essenciais para o bom funcionamento do projeto.

Tais como:
- **Sistema de parsing**: processa os ficheiros *.csv*;
- **Validação**: para cada entidade é feita uma validação dos dados;
- **Estruturas de dados eficientes**: como, por exemplo, **GHashTables** e **GPtrArray** (biblioteca GLib);
- **Pré-processamento de dados**;
- **Queries implementadas**.

1.2 Sistema

A arquitetura do sistema foi desenhada segundo princípios de modularidade e encapsulamento garantindo o acesso aos dados apenas ao módulo que as controla.

O diagrama seguinte demonstra como é feita essa organização.

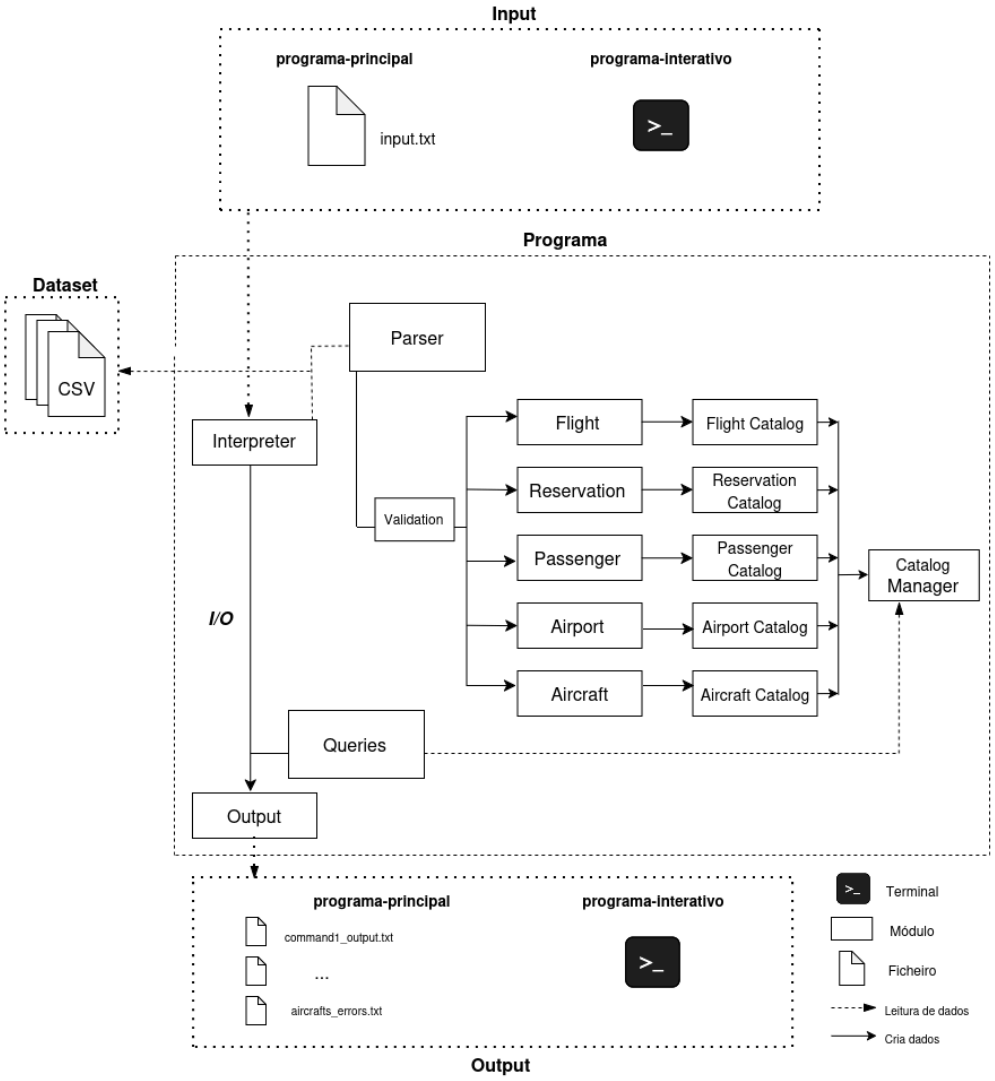

Figure 1: Diagrama da arquitetura do sistema

O sistema está organizado em módulos, seguindo um fluxo de dados sequencial da entrada até à saída. Esta arquitetura foi escolhida pelos seguintes motivos:

- **Separação de Responsabilidades**: Cada módulo tem uma função específica, o parser lida com a leitura dos ficheiros CSV, a validação verifica a conformidade dos dados, os catálogos gerem o armazenamento e as queries executam análises. Esta separação facilita a manutenção e permite alterações isoladas sem afetar outros componentes.
- **Fluxo de Dados**: Os dados fluem numa única direção (Input -> Parser -> Validação -> Entidades -> Catálogos -> Catálogo Geral -> Queries -> Output), criando assim uma hierarquia.
- **Reutilização**: O parser é genérico, usando callbacks para todos os CSVs. Funções de validação e utilitários de datas são partilhados entre módulos.
- **Tratamento de Erros Centralizado** : Registos inválidos são automaticamente gravados em ficheiros de erro específicos, garantindo rastreabilidade.
- **Agrupamento de dados**: Uso de um Catalog Manager com todos os catálogos das entidades, permitindo acesso rápido aos dados e que todos os componentes acedam aos catálogos da mesma forma.

1.2.1 Carregamento de dados

O carregamento de dados orquestra a leitura de todos os ficheiros do dataset, sendo o ponto de partida para a inicialização dos logs de erro e dos catálogos. Todo o processo começa na função `load_datasets`, que lê cada ficheiro *.CSV* através da função genérica `read_csv`.

A ordem de carregamento é crucial devido às dependências entre entidades. Assim, os ficheiros são processados na seguinte sequência: **Airports** -> **Aircrafts** -> **Flights** -> **Passengers** -> **Reservations**.

Esta ordem assegura que todas as referências cruzadas podem ser validadas durante o carregamento, permitindo que, por exemplo, as reservas referenciem passageiros e voos já existentes no sistema.

1.2.2 Parsing

O módulo de *parsing* é o componente usado para ler, validar e converter os dados nos ficheiros de dataset (.csv).

O sistema de *parsing* desenvolvido adota um sistem baseado em *callbacks*, permitindo o processamento específico de diferentes tipos de ficheiros. A função `read_csv` constitui o ponto de entrada principal, recebendo como parâmetros:

- o número de campos esperados para o ficheiro em questão;
- o caminho do ficheiro a processar;
- uma função de callback específica para validação e processamento dos dados;
- estruturas de dados auxiliares;
- um ficheiro para registo de erros.

Sendo assim, esta abordagem permite, por exemplo, processar o ficheiro **airports.csv** invocando `read_csv` com 8 campos e a função callback `process_valid_line_airports`, que implementa a lógica de validação específica para aeroportos.

1.2.3 Processamento do parsing

Para todos os ficheiros o processamento é o mesmo. Cada linha é lida sequencialmente e decomposta em campos específicos de cada ficheiro. Os *callbacks* são, como já foi referido, responsáveis por verificar a validade dos campos. A decomposição dos campos é feita pela função `parse_csv_line` que encapsula todos os campos entre aspas duplas, separados por vírgulas. Registos válidos são armazenados nas estruturas de dados correspondentes (e.g., **Airport**) e armazenados nos catálogos associados (i.e., **AirportCatalog**). Registos inválidos são descartados e registados em ficheiros de erros específicos (e.g., **aircrafts_errors.csv**)

1.2.4 Entidades

O sistema define cinco entidades principais, apoiadas por módulos de estruturas de dados auxiliares. Nesta primeira fase, o foco recaiu nas entidades **Airports**, **Aircrafts** e **Flights**, por serem fundamentais para a execução das queries e validação dos dados.

1.2.5 Catálogos

Os catálogos constituem a estrutura central de gestão das entidades, sendo responsáveis pelo armazenamento, acesso e manipulação dos dados. Cada catálogo utiliza uma GHashTable como estrutura principal, garantindo consultas e inserções eficientes.

Cada catálogo fornece algumas operações básicas como as seguintes:

- **create**: inicialização do catálogo;
- **destroy**: libertação de memória;
- **add**: adição de uma entidade ao catálogo;
- **get**: consulta de uma entidade a partir da sua chave única.

1.2.6 Estrutura comum

Todos os catálogos seguem um padrão semelhante: struct **entity_catalog** {**GHashTable*entities_by_key;**};

1.2.7 Queries

1.2.7.1 Query 1

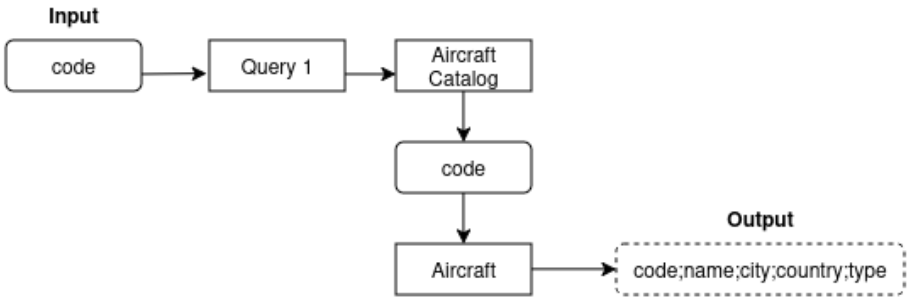

Figure 2: Diagrama da query 1

Esta query tem como objetivo listar o **resumo de um aeroporto** a partir do seu identificador. As entidades estão armazenadas em catálogos, onde cada entidade é indexada numa **GHashTable** pelo seu identificador único, garantindo **acesso rápido e eficiente**.

A execução da query consiste nos seguintes passos:

1. Procurar a entidade **Airport** correspondente ao identificador fornecido no catálogo.
2. Se encontrada, extrair e apresentar os campos solicitados: `code`, `name`, `city`, `country` e `type`.
3. Caso o identificador não exista, retornar uma linha vazia.

Esta abordagem permite que a query seja eficiente, com tempo de acesso próximo de **O(1)**, aproveitando a estrutura de hash do catálogo.

1.2.7.2 Query 2

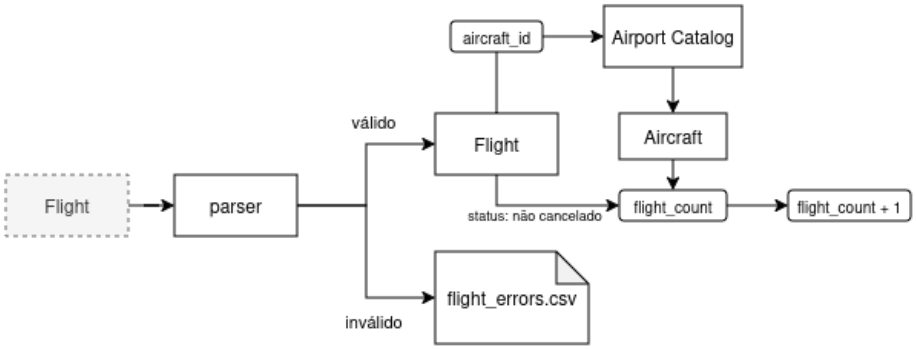

Figure 3: Parsing e validação flight

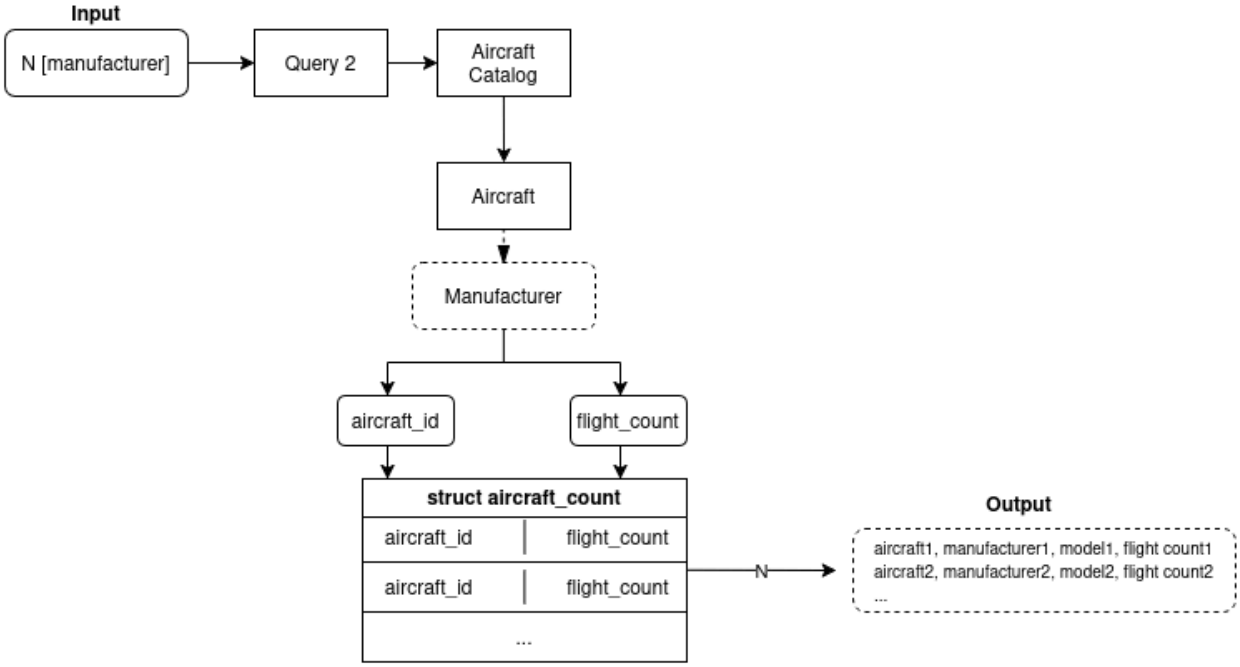

Figure 4: Diagrama da query 2

Esta query tem como objetivo **listar as Top N aeronaves com mais voos**, podendo receber opcionalmente um filtro pelo **manufacturer**. Quando o filtro está presente, apenas aeronaves desse fabricante específico são consideradas.

A execução da query consiste nos seguintes passos:

Durante o parsing e validação:

1. Para cada voo válido (não cancelado), identifica-se a aeronave correspondente através da função `get_aircraft_id_from_flight(flight)`.
2. Incrementa-se o contador `flight_counts` da aeronave usando a função `aircrafts_counter_increment`.

Durante a execução da query:

1. Se o filtro de fabricante estiver presente, constrói-se um array temporário contendo apenas as aeronaves desse fabricante, armazenando o `identifier` e `flight_counts`.
2. Ordena-se o array em ordem decrescente de `flight_counts`. Em caso de empate, utiliza-se a ordem lexicográfica crescente do `identifier` (posição a posição, i.e., "XB-NIQ0" vem antes de "XB-OIQ0").
3. Imprime-se até N entradas no ficheiro de saída, incluindo os campos: `identifier`, `manufacturer`, `model` e `flight_count`.

Esta abordagem permite contabilizar eficientemente os voos, aplicar filtros opcionais e garantir ordenação consistente, utilizando o campo `flight_counts` previamente incrementado durante o parsing dos voos.

1.2.7.3 Query 3

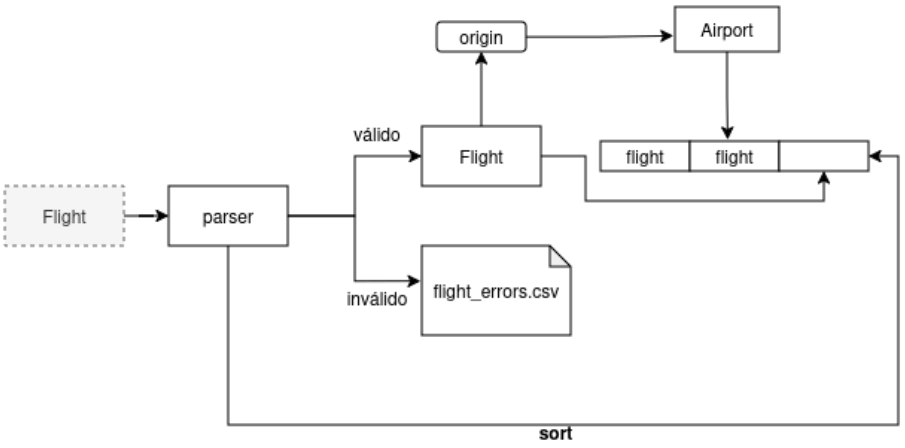

Figure 5: Parsing e validação flight

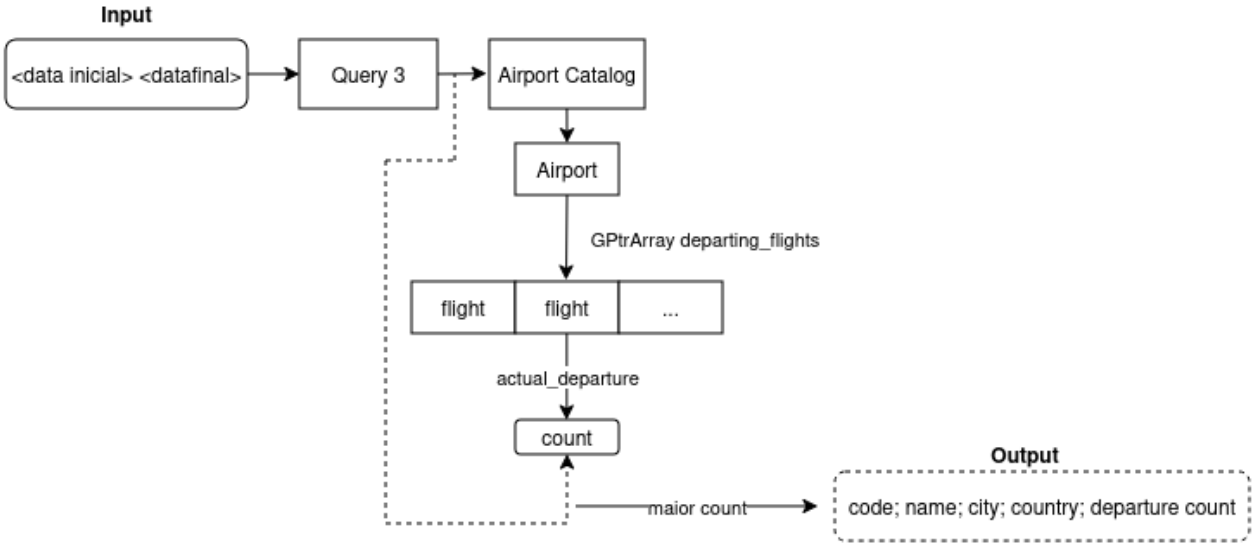

Figure 6: Diagrama da query 3

O objetivo desta query é identificar o aeroporto com o maior número de partidas dentro de um intervalo de datas.

Durante o parsing e validação:
1. Para cada airport é usado um GPtrArray `departing_flights` que armazena todos os voos que partiram desse aeroporto.
2. Para cada voo válido (não cancelado), identifica-se a sua origem correspondente através da função `get_flight_origin(flight)`.
3. Cada voo é armazenado no array de voos do aeroporto de origem correspondente.
4. Após carregar todos os voos, os arrays são ordenados por `actual_departure` em `parser.c` para otimizar consultas posteriores.

Durante a execução da query:

1. Percorrer a **hash table** de aeroportos.
2. Para cada aeroporto, percorrer o seu array de voos ordenado.
3. Contar apenas os voos que estão dentro do intervalo de datas especificado.
4. Interromper a contagem assim que um voo ultrapassa a data final do intervalo.
5. Manter registo do aeroporto com mais partidas; em caso de empate, escolher o aeroporto com menor código lexicográfico.
6. Retornar o aeroporto com mais partidas e o respetivo número de voos, apresentando os campos `code`, `name`, `city`, `country` e `departure_count`.

Esta abordagem garante eficiência, combinando o acesso rápido através da **GHashTable** com a travagem antecipada proporcionada pelos **arrays de voos ordenados**.

1.3 Discussão

1.3.1 Testes

Para a secção de testes, foi implementado um conjunto de funções que mede o desempenho do sistema e verifica a correção dos resultados. A abordagem adotada envolve a invocação direta das funções já existentes para carregar os datasets (`load_datasets`) e processar os comandos (`process_commands`).

Metodologia:

1. **Carregamento de Dados**
   - Os datasets são carregados inicialmente, e o **tempo de carregamento** é registado.
2. **Agrupamento de Comandos**
   - Os comandos de input são agrupados por query e armazenados em ficheiros temporários.
   - Permite contabilizar o número de comandos de cada query.
3. **Execução das Queries**
   - Cada grupo de queries é executado de forma sequencial.
   - Regista-se o tempo de início e fim de execução para calcular o tempo individual de cada query (em milissegundos).
4. **Verificação de Resultados**
   - Os resultados gerados são comparados com os resultados esperados.
   - Eventuais discrepâncias são identificadas, incluindo linha e ficheiro onde ocorreu o erro.
5. **Estatísticas por Query**

- Calculam-se métricas como:
  - Número de testes executados
  - Testes passados
  - Tempo total de execução
  - Memória utilizada pelo programa

Dessa forma, podemos validar os resultados e ver como cada query se comporta em termos de desempenho.

O diagrama abaixo apresenta uma visualização de um resumo deste processo:

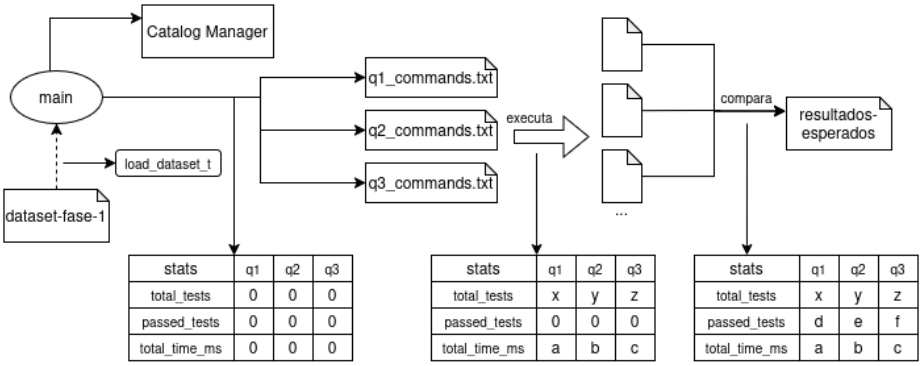

Figure 7: Diagrama execução de testes

1.3.2 Especificações

Computador 1

- OS: Ubuntu 24.04.3 LTS
- Processador: AMD Ryzen™ 7 5700U with Radeon™ Graphics × 16
- Kernel version: Linux 6.14.0-35-generic

Computador 2

- OS: Ubuntu 24.04.3 LTS
- Processador: 12th Gen Intel® Core™ i7-12650H × 16
- Kernel version: Linux 6.14.0-35-generic

Computador 3

- OS: Ubuntu 24.04.1 LTS
- Processador: AMD Ryzen™ 5 5600H with Radeon™ Graphics × 12
- Kernel version: Linux 6.14.0-33-generic

1.3.3 Estatística

	Query 1	Query 2	Query 3
Computador 1	2.6	7.2	576.3
Computador 2	0.5	4.2	302.2
Computador 3	3.7	12.6	557.0

Figure 8: Estatísticas por query

	Tempo total de execução (s)	Memória utilizada (MB)
Computador 1	3.6	300
Computador 2	1.3	300
Computador 3	6.6	299

Figure 9: Estatísticas gerais

Pela observação das tabelas, conseguimos chegar à conclusão de que o computador **2** (Intel i7-12650H) foi o mais rápido em todos os testes. A diferença de desempenho é maior nas queries simples, como a query 1, e diminui nas queries mais complexas, como a query 3, o que sugere que em operações complexas outros fatores além do processador também influenciam o desempenho.

Entre os processadores AMD, como esperado, o Ryzen 7 5700U (Computador 1) teve melhor desempenho que o Ryzen 5 5600H (Computador 3) em todas as queries.

O uso de memória foi praticamente igual nos três computadores (cerca de 300 MB), o que indica que o processador é o fator mais importante para o desempenho nestes testes.

1.4 Conclusão

Nesta primeira fase do projeto, foi possível consolidar detalhes fundamentais em programação em C, com especial foco em modularização, encapsulamento e gestão de memória.

O desenvolvimento dos catálogos e das estruturas de dados associadas, como GHashTables e GPtrArray, demonstrou a importância de escolher estruturas eficientes para acesso rápido e manipulação de grandes volumes de dados. As queries implementadas permitiram aplicar estas estruturas, de modo a testar o desempenho e a corretude do sistema de forma prática.

Entre os pontos de aprendizagem mais relevantes destacam-se:

1. A importância da modularização e do encapsulamento para manter o código organizado e fácil de manter;
2. A necessidade de pré-processamento e ordenação eficiente de dados para acelerar a execução de queries;
3. O papel central da validação de dados na fiabilidade do sistema, evitando propagação de erros para etapas posteriores.

Como melhoria futura, poderia ser explorada a otimização do uso de memória na query 2 através do uso de uma min heap em vez de um array completo, permitindo armazenar apenas as N aeronaves com mais voos. Esta abordagem reduziria a memória necessária, embora possa implicar algum aumento do tempo de execução devido às comparações e reorganizações na heap.

No geral, esta fase foi ótima para criar uma base sólida: tudo funciona de forma correta e organizada, deixando espaço para optimizações futuras em termos de tempo de execução e memória.