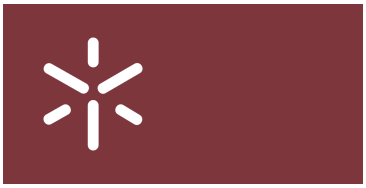

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Laboratórios de Informática III

Ano Letivo de 2025/2026

1ª Fase - LI3

Enrico Prazeres
A112881

Gonçalo Fernandes
A111855

November 13, 2025

LI3

Índice

1. Introdução

O presente relatório documenta a primeira fase do trabalho prático da unidade curricular **Laboratórios de Informática III**, lecionada no 2º ano da Licenciatura em Engenharia Informática da Universidade do Minho, durante o ano letivo 2025/2026.

Este projeto consiste no desenvolvimento de um sistema de gestão e análise de dados aeroportuários armazenados em memória, capaz de processar informação proveniente de ficheiros *.csv* relativos a aeroportos, voos, passageiros, aeronaves e reservas. O principal objetivo centra-se na implementação de uma solução eficiente que permita carregar, armazenar e consultar grandes volumes de dados, respondendo de forma rápida e precisa a um conjunto de queries predefinidas.

Ao longo deste relatório, será apresentada a arquitetura do sistema desenvolvido, as decisões de implementação tomadas, uma análise crítica do desempenho alcançado e as principais conclusões retiradas desta primeira fase do projeto.

2. Desenvolvimento

O sistema desenvolvido implementa uma solução completa para gestão e consulta de dados aeroportuários, estruturado em três componentes principais: o *parsing* e validação de dados de entrada, o armazenamento em estruturas de dados otimizadas, e o processamento de *queries* através do modo `batch`.

O modo `batch` permite a execução de comandos descritos num ficheiro *.txt*, cujo caminho é fornecido como argumento ao programa. Desta forma, o modelo de operação garante a uma fácil integração com a plataforma de avaliação automática, disponibilizada pelos docentes.

2.1. Arquitetura Geral

O sistema desenvolvido foi estruturado seguindo uma arquitetura modular que separa de forma eficaz as responsabilidades de cada componente. A Figura 1 ilustra a organização dos módulos e o fluxo de dados através do sistema.

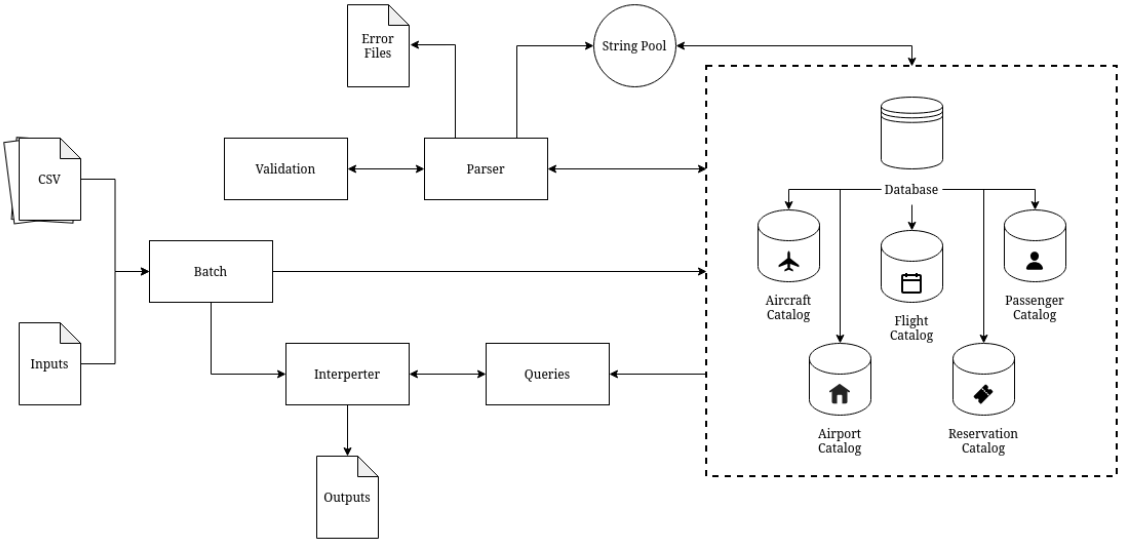

Figura 1: Diagrama da Arquitetura do Projeto

A arquitetura divide-se em três camadas principais:

Camada de I/O e Processamento: Responsável não só pela leitura e validação dos ficheiros *CSV*, mas também pela interpretação de comandos e escrita dos respetivos resultados. Esta inclui os módulos *Batch*, *Parser*, *Validation*, *Interpreter* e *Queries*.

Camada de Dados: Centraliza o armazenamento e gestão das entidades através de uma base de dados e os respetivos catálogos especializados, como *Aircraft Catalog*, *Flight Catalog*, *Passenger Catalog*, *Airport Catalog* e *Reservation Catalog*.

Camada de Utilitários: Disponibiliza funcionalidade auxiliares como *String Pool*, que otimiza o consumo de memória ao eliminar strings duplicadas.

2.2. Estrutura de Módulos

2.2.1. Parser e Validation

O módulo *Parser* é responsável pela leitura e interpretação dos ficheiros *CSV*, processando linha a linha e extraindo os campos relevantes. Este módulo trabalha em conjunto com o módulo *Validation*, que por sua vez, implementa as regras de validação sintática e lógica especificadas no enunciado.

Posto isto, as entradas inválidas são identificadas durante o *parsing* e redirecionadas para os ficheiros de erro (Error Files), mantendo o formato original dos dados rejeitados. Este processo garante que apenas dados válidos sejam armazenados na *Database*.

2.2.2. *Database* e Catálogos

A *Database* organiza-se em cinco catálogos especializados, cada um responsável pela gestão e armazenamento de um tipo de entidade:

Aircraft Catalog: Armazena e gere dados sobre aeronaves

Flight Catalog: Armazena e gere os dados de voos

Passenger Catalog: Armazena e gere dados de passageiros

Airport Catalog: Armazena e gere dados dos aeroportos

Reservation Catalog: Armazena e gere dados de reservas

Por sua vez, cada catálogo implementa estruturas de dados otimizadas para os padrões de acesso comuns das queries recebidas, utilizando por conseguinte, *hash tables* (*GHashTable*) para garantir pesquisas eficientes por identificador único (Id). Já para operações que requerem ordenação ou filtragem, são utilizados arrays dinâmicos (*GPtrArray*). Ambas as estruturas em questão são da *GLib*

2.2.3. *Batch*, *Interpreter* e *Queries*

O módulo *Batch* desempenha duas funções principais no sistema. Primeiramente, é responsável por iniciar o processo de carregamento de dados, enviando o *trigger* à *Database* que desencadeia o *parsing* dos ficheiros CSV.

Após o carregamento completo dos dados, o *Batch* coordena o fluxo de execução do modo principal, recebendo o ficheiro de comandos (inputs) e delegando o mesmo ao *Interpreter*. Este identifica o tipo de *query* solicitada em cada linha, trata os respetivos argumentos e invoca o módulo *Queries*, chamando a *query* correspondente.

O módulo Queries contém a lógica responsável por responder a cada interrogação, acedendo aos catálogos da *Database* e produzindo os resultados no formato especificado. Os resultados são finalmente escritos nos ficheiros de output através do módulo *Interpreter*.

2.3. Queries

2.3.1. Query 1

Na *query* 1 pretende-se que, dado um código IATA único, seja listado um resumo do respetivo aeroporto incluindo: o código, nome, cidade, país, e tipo do aero-porto.

Assim, para esta *query* acede-se diretamente ao *Airport Catalog* através de uma pesquisa por código IATA. A utilização de uma *hash table* garante acesso O(1), permitindo resposta imediata mesmo com grandes volumes de dados.

2.3.2. Query 2

Na *query* 2 pretende-se que sejam listados as top *N* aeronaves com mais voos realizados, sendo este *N* o número de aeronaves que devem constatar no output. Além disto, esta *query* pode (ou não) ainda receber um filtro opcional, o filtro de fabricante, sendo que quando presente só devem ser considerados aeronaves desse fabricante.

Para otimizar esta operação, durante o *parsing* dos voos, cada voo válido com estado diferente de *CANCELLED* incrementa um contador associado à respetiva aeronave. Este contador é armazenado numa estrutura *AircraftFlightCounts*, criada para o efeito, que mantém um mapeamento entre o identificador da aeronave e o número de voos realizados através de uma *hash table* (*GHashTable*). Durante o processamento da *query*, verifica-se se as aeronaves estão ordenadas através de um *bool* armazenado na estrutura, e procede-se à devida ordenação por ordem decrescente do número de voos através de um algoritmo de ordenação sobre o *array* de ponteiros (*GPtrArray*), se necessário. Posto isto, são depois selecionados os primeiros *N* elementos que satisfazem o critério opcional de fabricante (*manufacturer*).

2.3.3. Query 3

Na *query* 3 pretende-se listar o aeroporto com mais partidas entre 2 datas e o correspondente número de partidas associados. Para a contabilização de partidas de um aeroporto deverão ser considerados voos em que o origem é igual ao aeroporto em questão, e voos com estado diferente de cancelado. Por fim, em caso de empate, considera-se o aeroporto com menor identificador, isto é, o menor código em ordem lexicográfica.

Para otimizar esta operação, durante o *parsing* dos voos, cada partida válida é associada ao respetivo aeroporto de origem. Esta associação utiliza um *array* (*GArray*) que armazena as datas de partida reais (*actual_departure*) em cada entidade *Airport*. Durante o processamento da *query*, percorre-se a lista de aeroportos e, para cada um, verifica-se se a lista de datas está ordenada, através de um *bool* armazenado na estrutura, ordena-se se necessário e, finalmente, conta-se o número de partidas no intervalo através de um algoritmo de *binary search* para identificar os *lower* e *upper bounds* das datas especificadas.

2.4. Testes

A fim de facilitar o processo de confirmação de resultados, desenvolveu-se também um programa de testes capaz de percorrer as funções usadas nas *queries* e devolver informações relativas à sua velocidade, comportamento e uso de memória.

Este está dividido em essencialmente 3 partes:

- *PARSER STATISTICS*;
- *QUERY PERFORMANCE*;
- *GENERAL SUMMARY*;

É, numa primeira fase, realizada uma análise do tempo de execução relativo ao *parser*, sendo apresentado não só o tempo de execução total, como também o tempo relativo ao *parse* de cada tipo de entidade individualmente.

Por sua vez, para cada *query* o programa calcula a média de tempo (ms) por tipo de *query*, o tempo máximo gasto nesse mesmo tipo de *query*, o tempo total de execução de um grupo de *queries*, bem como uma classificação de sucessos e falhas. Para esta última, comparam-se os resultados obtidos com os resultados esperados.

Por fim, o resumo geral apresenta o tempo gasto na libertação de memória, o tempo global de execução do programa (ms) e a memória gasta (KB) durante a execução.

2.5. Análise Estatística

A avaliação de desempenho do sistema foi conduzida em duas fases complementares: testes preliminares em diferentes máquinas para validação inicial, e uma análise estatística rigorosa através de execuções automatizadas para caracterização detalhada do comportamento do sistema.

2.5.1. Equipamento

Computador 1

- OS: Arch Linux – Kernel 6.17.7
- CPU: Intel Core Ultra 7 155H
  - Cores: 16 cores / 22 threads
  - 1.7 GHz up to 4.8 GHz
  - Cache: L1 960 KB, L2 14 MB, L3 24 MB
- Memória: 32 GB DDR5-5600 MHz
- Armazenamento: 1TB SSD PCIe 4.0 NVMe

Computador 2

- OS: Arch Linux – Kernel 6.17.7
- CPU: Intel Core i7-1355U Gen 13
  - Cores: 10 cores / 12 threads
  - 1.7 GHz, up to 5 GHz
  - Cache: L1 928 KB, L2 6.5 MB, L3 12 MB
- Memória: 16 GB DDR4-3200 MHz
- Armazenamento: 512GB SSD PCIe 4.0 NVMe

2.5.2. Análise Estatística Básica

Para avaliar o desempenho da solução desenvolvida, foram realizadas 3 amostras de testes em dois computadores distintos (especificados acima), ambos configurados no modo performance e ligados à corrente elétrica, de forma a garantir resultados consistentes e representativos da máxima capacidade de processamento.

Os testes incluíram a medição dos tempos de *parsing* de cada entidade, a execução das *queries* implementadas e o consumo global de recursos do sistema. As tabelas seguintes apresentam as médias dos resultados obtidos nas 3 amostras de ambas as máquinas.

	Tempo de *parse* por entidade (ms)				
	Airports	Aircrafts	Flights	Passengers	Reservations
Computador 1	5.036	0.487	922.473	113.565	18.525
Computador 2	3.239	0.347	700.152	77.985	13.956

Tabela 1: Estatísticas do *parser*

	Tempo por *query* (ms)								
	Q1			Q2			Q3		
	Médio	Máximo	Total	Médio	Máximo	Total	Médio	Máximo	Total
Computador 1	0.009	0.0 35	0.344	0.676	0.699	27.027	1.980	47.008	79.193
Computador 2	0.008	0.056	0.330	0.530	0.673	21.205	2.046	38.235	81.832

Tabela 2: Estatísticas por *query*

	Tempo total de execução (s)	Memória utilizada (KB)
Computador 1	1.401	210508
Computador 2	1.309	210328

Tabela 3: Estatísitcas globais

2.5.3. Análise Estatística Detalhada

Para uma avaliação mais rigorosa e abrangente do desempenho do sistema desenvolvido, foi criado um script em Python que automatiza a execução de múltiplos testes e gera visualizações estatísticas detalhadas através da biblioteca *Matplot*. Neste caso o *script* foi corrido no computador 2, executando o programa 100 vezes consecutivas com o mesmo *dataset* (sem erros) e conjunto de *queries*, recolhendo métricas de tempo de *parsing*, tempo de execução de queries, consumo de memória e tempo total de execução.

Esta abordagem permite não só obter valores médios mais significativos, mas também analisar a variabilidade e consistência do desempenho, identificar *outliers* e compreender o comportamento do sistema sob execuções repetidas com condições idênticas.

2.5.3.1. Tempo total de execução:

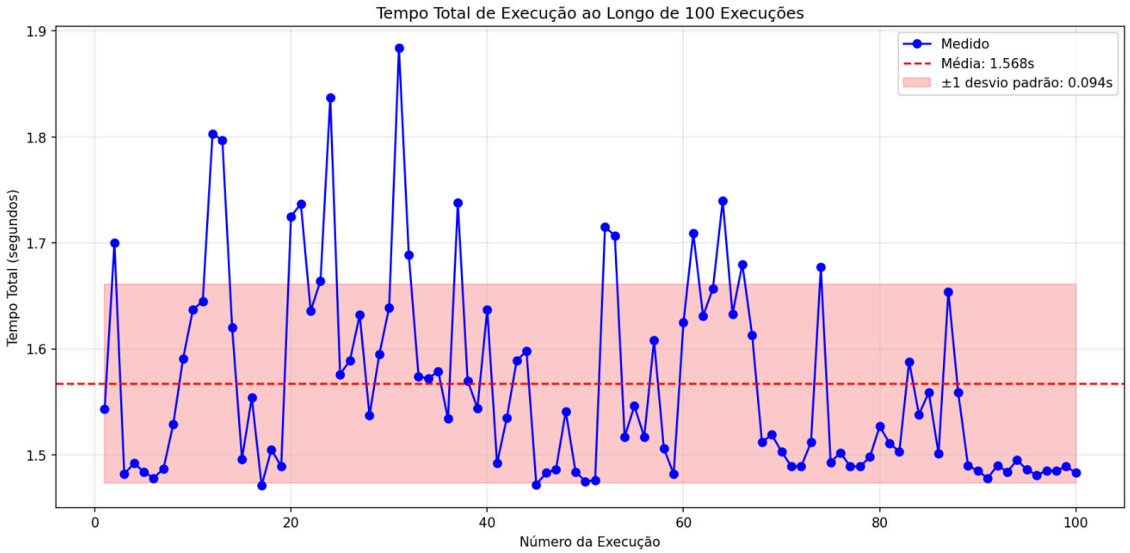

Figura 2: Tempo de execução total ao longo de 100 execuções

A evolução do tempo total de execução ao longo das 100 iterações apresenta uma média de 1.568s com um desvio padrão de apenas 0.094s (aproximadamente 6% de coeficiente de variação), demonstrando excelente estabilidade do sistema.

As primeiras 20 execuções apresentam maior variabilidade (1.48s - 1.88s), o que pode ser explicado pelo facto do sistema operativo ainda não ter os ficheiros em *cache* e de estar a adaptar possíveis otimizações automáticas. Entre as execuções 20-70, o sistema estabiliza dentro de um intervalo de ±1 desvio padrão, com picos esporádicos. Nas últimas 30 execuções, observa-se uma ligeira redução e maior estabilidade (média 1.48s), indicando que o sistema atingiu um estado otimizado.

2.5.3.2. Distribuição do tempo de *Parsing*

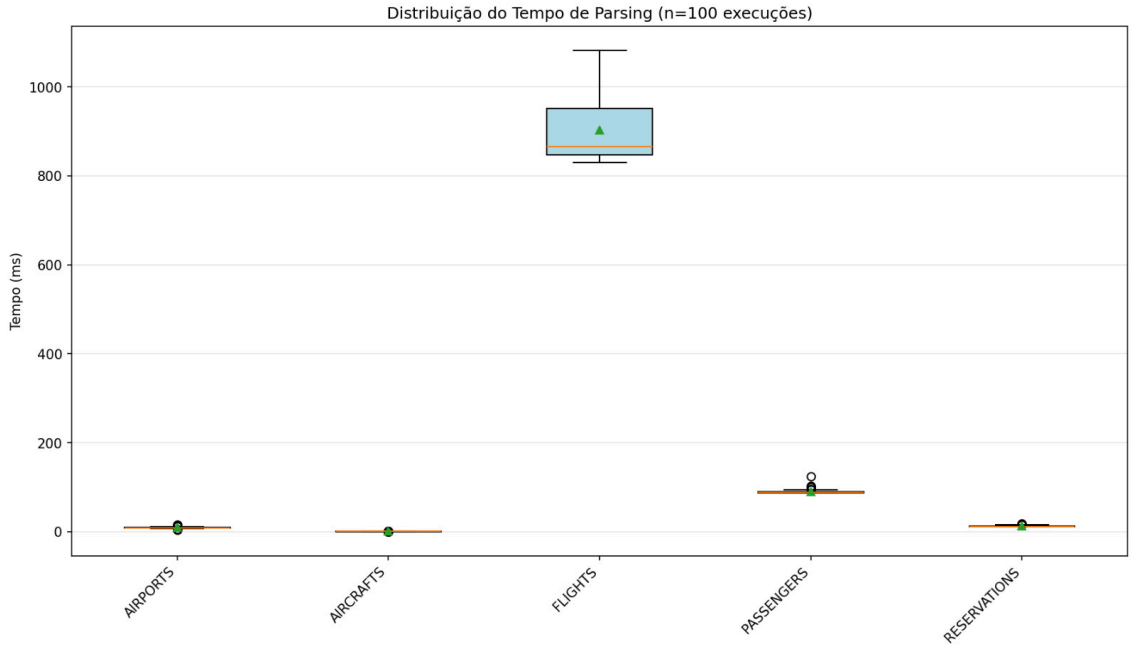

Figura 3: Distribuição do tempo de parsing em 100 execuções

A análise da distribuição do tempo de *parsing* por tipo de entidade ao longo de 100 execuções revela que o *parsing* de voos domina claramente o tempo total, representando aproximadamente 85-90% do tempo de *parsing*, com uma mediana em torno de 900ms e média de aproximadamente 910ms. Este resultado é diretamente proporcional ao volume de dados, dado que o *csv* de voos contém 1.552.180 linhas, isto é, mais do que 150 vezes superior ao segundo maior ficheiro (passageiros, com 280.002 linhas). Para além disso, as validações necessárias em cada voo, causam também impacto nestes valores.

A baixa variabilidade observada (desvio padrão reduzido, como evidenciado pelas barras de erro curtas na caixa de bigodes) indica que o processo de *parsing* é consistente e previsível, sem variações significativas entre execuções.

2.5.3.3. Tempo de execução de queries

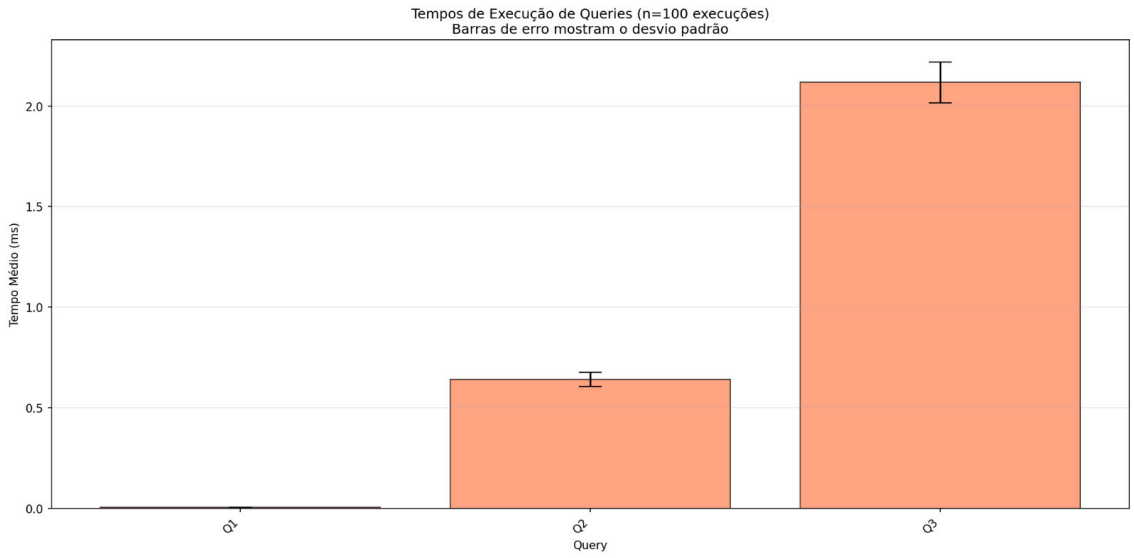

Figura 4: Tempo de execução de queries em 100 execuções

A análise do tempo de execução das queries revela características de desempenho dramaticamente distintas, refletindo a complexidade algorítmica de cada operação.

***Query* 1 (pesquisa direta por código IATA):**

- Tempo médio: 0.009ms
- Variação: praticamente inexistente (barras de erro invisíveis)
- Complexidade: *O(1)* graças à *hash table*

Este resultado confirma a eficiência da estrutura de dados escolhida, com tempo de resposta praticamente instantâneo independentemente do tamanho do catálogo de aeroportos

***Query* 2 (top *N* aeronaves por número de voos):**

- Tempo médio: 0.6ms, máximo 0.67ms
- Variação: baixa
- Complexidade: *O(n log n)* na primeira execução (ordenação), *O(n)* em execuções subsequentes quando já ordenado

O tempo ligeiramente superior reflete a necessidade de percorrer o *array* de aeronaves e, potencialmente, aplicar filtros de fabricante

***Query* 3 (aeroporto com mais partidas entre datas):**

- Tempo médio: 2.0ms, mas com alta variabilidade
- Tempo máximo observado: 47ms (23x superior à média)
- Desvio padrão: significativo (visível nas barras de erro extensas)

A elevada variabilidade da Q3 deve-se provavelmente à dependência direta do intervalo de datas fornecido em cada *query*. Intervalos mais alargados resultam em mais elementos a contar, o que pode explicar a variação entre 1ms (intervalos curtos, poucos aeroportos com partidas) e 47ms (intervalos com mais datas), bem como no facto de nas primeiras execuções, ser necessária a ordenação das *departures*.

2.5.3.4. Consumo de memória

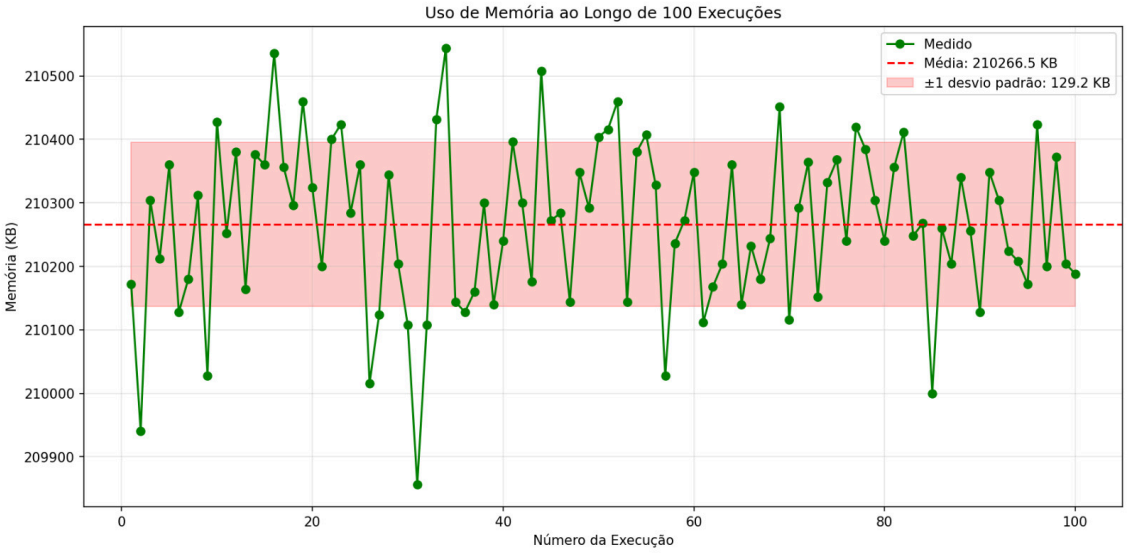

Figura 5: Consumo de memória ao longo de 100 execuções

A análise do consumo de memória revela uma das métricas mais estáveis de todo o sistema:

- Média: 210.266,5 KB
- Desvio padrão: 129,2 KB
- Coeficiente de variação: 0,06%

2.5.3.5. Variabilidade das métricas

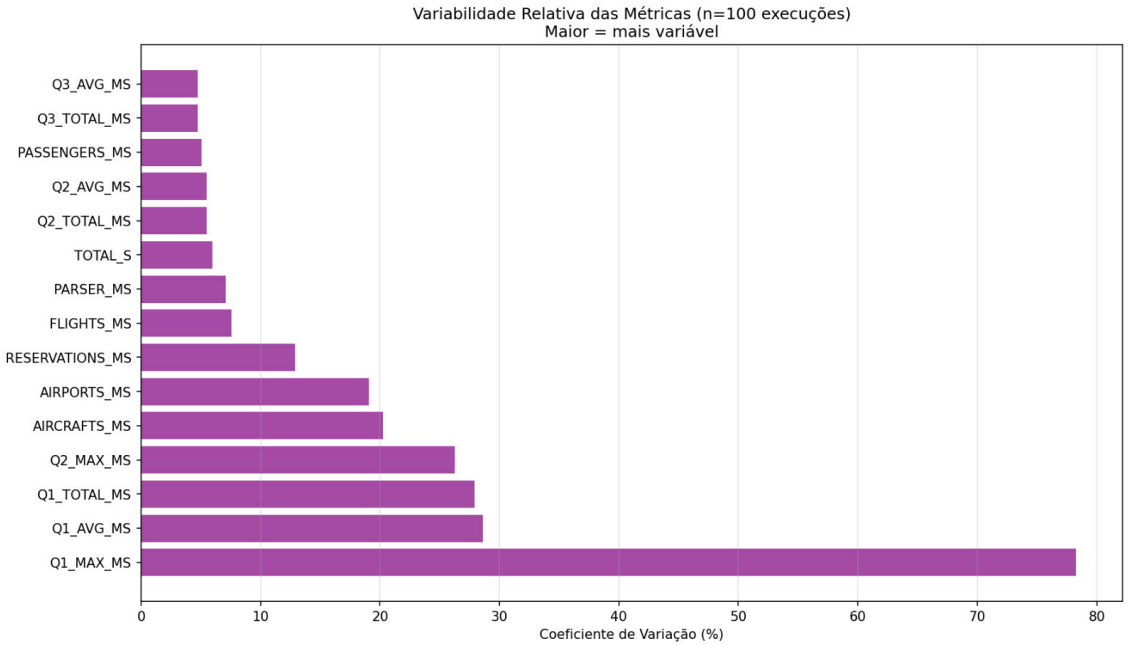

Figura 6: Variabilidade relativa das métricas nas 100 execuções

O gráfico de coeficiente de variação permite comparar a estabilidade de diferentes métricas do programa.

- **métricas com maior variabilidade (coeficiente de variação > 20%):**
  - *Q1_MAX_MS (83%)* e *Q1_TOTAL_MS* (29%): Estes são os casos com maior coeficiente de variação uma vez que a *query* 1 é tão rápida (0.009ms) que pequenas variações no sistema representam grandes percentagens.
  - *Q2_MAX_MS* (26%): No caso da *query* 2, as variações correspondem às diferenças entre execuções com e sem ordenação.
- **métricas estáveis (coeficiente de variação < 10%):**
  - Tempos médios de queries (4-7%): Operações previsíveis e consistentes
  - Tempo total de execução (6%): Excelente consistência global
  - Parsing de ficheiros grandes (6-8%): Pequenas variações

As métricas mais importantes (tempos médios, tempo total) apresentam excelente estabilidade, o que reforça a qualidade da nossa implementação. Tal como foi visto anteriormente, a variabilidade elevada observada em algumas métricas deve-se a valores absolutos muito pequenos, onde variações mínimas têm um impacto percentual maior.

2.5.3.6. Conclusão da análise

Em suma, a análise estatística de 100 execuções consecutivas confirma a consistência da estrutura atual, visto que os resultados evidenciam a viabilidade das escolhas tomadas para estruturas de dados e algoritmos. Para além disso, esta caracterização detalhada do comportamento do sistema, através de visualizações gráficas e métricas quantitativas, fornece uma base sólida para identificar oportunidades de otimização e melhorias que possamos fazer na segunda fase do projeto.

2.6. Ferramentas utilizadas

2.6.1. Memory Leaks

Para garantir a ausência de *memory leaks* e uma gestão correta de memória, utilizou-se o *Valgrind*, uma ferramenta que identifica acessos inválidos a memória, utilização de memória não inicializada e blocos de memória não libertados, permitindo detetar e corrigir problemas de gestão de memória durante o desenvolvimento.

2.6.2. Otimização de funções

Para identificar possíveis otimizações, recorreu-se ao *gprof*, uma ferramenta de *profiling* que analisa o tempo de execução de cada função.

Para isto, o programa é compilado com as *flags* `-g -pg`, com o intuito de gerar um ficheiro `gmon.out`. De seguida é executado o comando `gprof ./build/debug/programa-principal gmon.out > profile-data.txt` que escreve no ficheiro *profile-data.txt* todos os dados estatísticos relativos às funções do programa e a respetiva hierarquia das chamadas destas mesmas funções.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 41.97      0.60     0.60 15101866     0.00     0.00  consume_field
 20.99      0.90     0.30  4434796     0.00     0.00  is_datetime_valid
  4.90      0.97     0.07                             _init
  4.20      1.03     0.06 15101866     0.00     0.00  parser_strip_quotes
  3.50      1.08     0.05   294160     0.00     0.00  find_lower_bound
  2.80      1.12     0.04  1108699     0.00     0.00  flight_line_parser
  2.80      1.16     0.04   294160     0.00     0.00  find_upper_bound
  2.80      1.20     0.04   200000     0.00     0.00  is_email_valid
  2.10      1.23     0.03  4434796     0.00     0.00  parse_datetime
  2.10      1.26     0.03  1380793     0.00     0.00  get_airport_departure_dates
  2.10      1.29     0.03       40     0.00     0.00  query_three
  1.40      1.31     0.02  1108699     0.00     0.00  set_flight_status
  1.40      1.33     0.02        5     0.00     0.24  read_csv
  1.40      1.35     0.02                             compare_departure_dates
  0.70      1.36     0.01  4843150     0.00     0.00  string_pool_interning
  0.70      1.37     0.01  2224752     0.00     0.00  is_code_valid
  0.70      1.38     0.01  1108699     0.00     0.00  is_id_valid
  0.70      1.39     0.01  1108699     0.00     0.00  set_flight_airline
  0.70      1.40     0.01   200000     0.00     0.00  passenger_line_parser
  0.70      1.41     0.01                             free_flight
  0.70      1.42     0.01                             parse_phone_number
  0.35      1.43     0.01   200000     0.00     0.00  create_passenger
  0.35      1.43     0.01     1000     0.00     0.00  set_aircraft_range
  0.00      1.43     0.00  2195332     0.00     0.00  is_arrival_valid
  0.00      1.43     0.00  1128699     0.00     0.00  get_flight_catalog
  0.00      1.43     0.00  1109739     0.00     0.00  get_aircraft_catalog
```

Figura 7: `gprof` - Tempo de execução das funções e número de chamadas

2.7. Otimizações

Para além das otimizações específicas de cada *query*, foram implementadas melhorias transversais ao sistema visando reduzir o consumo de memória e melhorar o desempenho global.

2.7.1. Representação compacta de dados

A escolha adequada dos tipos de dados para representar informação permite reduzir significativamente o consumo de memória e melhorar o desempenho das operações. Dados com formato estruturado e previsível podem ser otimizados através de representações mais compactas e eficientes.

A título de exemplo, todos os tipo de datas foram convertidas de *strings* para representações numéricas: a data “2025-03-21” é armazenada como *long long* com o valor 20250321, reduzindo o espaço ocupado de aproximadamente 10-11 bytes (string + terminador nulo) para 8 bytes. Para além disso, este formato

permite comparações mais rápidas e diretas, uma vez que a ordem cronológica natural mantém-se na representação numérica (20250321 > 20250320 > 20250101).

De forma semelhante, categorias que se repetem frequentemente nos dados, foram convertidos de strings para enumerações (*enum*). Estados como “DELAYED”, “CANCELLED” e “ON_TIME” passam a ser representados por constantes inteiras (tipicamente 4 bytes), em vez de strings que ocupariam entre 8-11 bytes cada. Para além da redução de memória, *enums* permitem comparações e operações *switch* extremamente rápidas.

2.7.2. Alinhamento de estruturas

A ordem dos campos nas estruturas (*structs*) foi ajustada para minimizar o espaço ocupado através do alinhamento eficiente dos tipos de dados em memória.

2.7.3. *String Pool*

Ademais uma das otimizações mais impactastes foi a implementação de um *string pool*, que elimina a duplicação de strings em memória. Para este efeito, quando uma string é lida dos ficheiros CSV, verifica-se primeiro se já existe na *hash table* da *pool*. Se existir, retorna-se o ponteiro para a string já armazenada, caso contrário, a string é adicionada à *hash table* e o seu ponteiro é retornado.

Esta abordagem é particularmente eficaz para campos que se repetem frequentemente, como nomes de fabricantes, nacionalidades, e nomes de passageiros.

De seguida estão listados os resumos gerais de execução do programa, com e sem a utilização de *string pool*.

```
GENERAL SUMMARY

Free Time:                    0.387 s
Total Execution Time:         1.561 s
Memory Usage:                210016 KB
```

Figura 8: Resumos gerais da execução do programa com string pool

```
GENERAL SUMMARY

Free Time:                    0.101 s
Total Execution Time:         1.322 s
Memory Usage:                361060 KB
```

Figura 9: Resumos gerais da execução do programa sem string pool

Torna-se, portanto, evidente que a implementação deste mecanismo traz uma melhoria significativa de 41,84% no consumo de memória (de 361060 KB para 210016 KB), tendo contudo um custo de aproximadamente 18% no tempo de execução (de 1.322s para 1.561s). Esta troca (*trade-off*) é expectável, dado que o *string pool* requer operações adicionais de gestão e procura de strings. A redução substancial de memória justifica o custo temporal, especialmente no processamento de grandes volumes de dados, como esperado na próxima fase do projeto.

2.7.4. *Parsing* eficiente

Durante o *parsing* dos ficheiros de dados, foi implementada uma otimização significativa para campos não utilizados. A função `skip_field()` foi desenvolvida especificamente para campos com tamanho constante e conhecido que não necessitam de ser processados. Em vez de utilizar `consume_field()`, que percorre caracter a carácter campo, e que aloca memória, a função `skip_field()` simplesmente avança o ponteiro de leitura pelo número de bytes correspondente ao tamanho *hardcoded* do campo.

Esta abordagem elimina operações desnecessárias de cópia de memória e alocação dinâmica para dados que serão descartados, resultando numa melhoria substancial na velocidade de *parsing* dos ficheiros de entrada, especialmente relevante em *datasets* de grande dimensão.

2.7.5. *Flags* de compilação otimizadas

O *Makefile* foi configurado com *flags* de otimização avançadas, aproveitando o facto de que a máquina de testes compila e executa o programa localmente. Foi incluída a *flag* `-march=native`, que permite ao compilador gerar código otimizado especificamente para a arquitetura do processador da máquina onde é compilado.

Para além disso, foram aplicadas outras *flags* de otimização como `-Ofast` para ativar otimizações agressivas do compilador. Esta estratégia garante que o programa executa com o máximo desempenho possível no ambiente de avaliação.

3. Conclusão

A primeira fase deste trabalho prático permitiu consolidar conhecimentos essenciais sobre programação na linguagem C, com destaque para os conceitos de modularização, encapsulamento e gestão rigorosa de memória. O desenvolvimento do sistema exigiu não apenas domínio técnico da linguagem, mas também capacidade de planeamento e organização do código em módulos independentes e coesos.

A implementação dos mecanismos de *parsing*, validação de dados e resposta às queries propostas evidenciou a importância da escolha adequada de estruturas de dados para garantir eficiência máxima. Além de mais, a utilização de ferramentas como *Valgrind*, *GDB* e *Gprof* revelou-se fundamental para assegurar a qualidade e correção da solução desenvolvida, mostrando-se ainda relevantes para o desenvolvimento de projetos futuros.