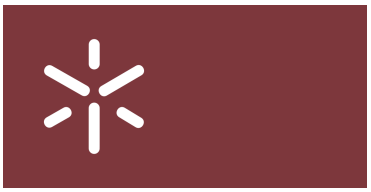


LI3 - Relatório da Fase II - Grupo 35

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

Janeiro de 2025

Resumo

Este relatório documenta a Fase 2 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 de um **programa-interativo** e de novas ***queries***.

1 Introdução

1.1 Contextualização

O presente relatório descreve a organização geral do sistema, a metodologia adotada para a execução das queries, o desenvolvimento de testes e a análise de desempenho. São ainda discutidas decisões de implementação, nomeadamente a escolha das estruturas de dados.

Esta segunda fase do projeto concentra-se no desenvolvimento de um programa interativo e na implementação de mais três *queries*, expandindo as funcionalidades existentes.

Adicionalmente, foram introduzidas alterações às três queries pertencentes à fase 1, nomeadamente ao nível do formato de apresentação dos resultados. Em particular, foi definido um novo delimitador entre os argumentos apresentados no output.

Em termos de formatação, as queries são representadas por um número, seguido dos respetivos argumentos. Nesta fase, o formato de output das queries pode assumir uma de duas variantes, consoante o número da query seja ou não acrescido do carácter 'S'. Por omissão, os resultados devem ser separados pelo delimitador **';'**, enquanto que, no formato alternativo, deve ser utilizado o separador **'='**.

1.2 Sistema

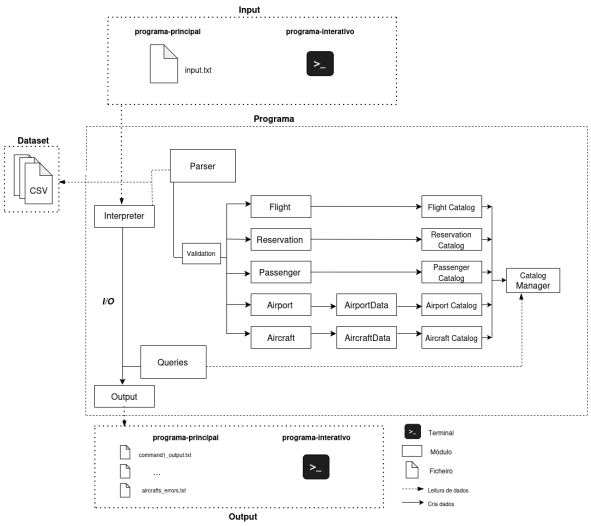


Figure 1: Diagrama da arquitetura do sistema

O diagrama apresenta a arquitetura global do sistema, evidenciando o fluxo de dados desde o input até ao output.

Tal como na fase 1, os dados iniciais são carregados a partir de datasets em formato CSV, que são lidos pelo Interpreter. Os comandos e dados passam depois pelo Parser, responsável pela análise e interpretação, e por um módulo de Validation, que garante a integridade dos dados antes da sua criação.

Cada tipo de entidade do sistema é tratado por um módulo específico (Flight, Reservation, Passenger, Airport e Aircraft). No caso dos aeroportos e das aeronaves, existe ainda uma camada intermédia (AirportData e AircraftData) para tratamento adicional da informação. As entidades são armazenadas nos respetivos Catálogos, que centralizam e organizam os dados.

A gestão e coordenação dos vários catálogos é assegurada pelo Catalog Manager, permitindo um acesso consistente à informação. As Queries utilizam estes dados para responder às solicitações do utilizador, e os resultados são enviados para o Output, seja sob a forma de ficheiros no programa principal ou diretamente no terminal no modo interativo.

Os componentes e funcionalidades que não se encontram explicitamente representados no diagrama serão explicados com maior detalhe nos tópicos seguintes.

1.3 Alterações

1.3.1 Catálogos e Entidades

Durante a fase 2 do projeto, foi realizada uma refatoração geral das entidades do sistema com o objetivo de separar os dados estáticos das informações dinâmicas utilizadas em consultas. Anteriormente, algumas estruturas de entidade armazenavam diretamente valores derivados ou contadores que seriam utilizados em queries, misturando dados essenciais com informações de suporte.

Para resolver essa mistura de dados foi criada uma estrutura auxiliar **<Entidade>Data**, que mantém um ponteiro para a entidade original e adiciona campos auxiliares, como contadores, listas de objetos relacionados ou outros valores derivados.

Em seguida, detalharemos o exemplo do `Airport`, mostrando como essa mudança foi implementada.

Na fase inicial, a estrutura `Airport` armazenava diretamente informações relacionadas às operações do aeroporto, como o array de voos de partida (departing_flights). Ou seja, valores que seriam utilizados nas *queries* estavam misturados com os dados essenciais da entidade (código, nome, cidade, país e tipo).

```
struct airport {
    char code[4];
    char* name;
    char* city;
    char* country;
    char type;

    GPtrArray* departing_flights;
};
```

```
struct airport_catalog {
    GHashTable* airport_data_by_code;
};
```

Na fase 2, a estrutura foi refatorada para separar os dados estáticos da entidade dos dados dinâmicos utilizados em *queries*. Criamos então a estrutura `AirportData`, que contém um ponteiro para a entidade `Airport` e adiciona campos auxiliares como `arrival_count`, `departure_count` e `departing_flights`.

```
struct airport {
    char code[4];
    char* name;
    char* city;
    char* country;
    char type;
};
```

```
struct airport_data {
    Airport* airport;
    int arrival_count;
    int departure_count;
    GPtrArray* departing_flights;
};
```

```
struct airport_catalog {
    GHashTable* airport_data_by_code;
};
```

Essa mudança traz várias vantagens:

- Separação de responsabilidades: a entidade representa apenas os dados estáticos, enquanto AirportData armazena informações calculadas ou utilizadas em queries.
- Facilita a manutenção: campos dinâmicos podem ser atualizados ou reinicializados sem afetar a entidade principal.
- Melhora a eficiência das queries: permite acessar rapidamente contadores e listas de voos diretamente na estrutura de dados usada para *queries*, sem interferir nos dados originais do aeroporto.

Em resumo, a refatoração introduz uma estrutura auxiliar (Data) que encapsula tanto a entidade quanto valores derivados, tornando o código mais modular e escalável para consultas complexas.

1.3.2 Escrita de resultados

Na defesa da fase 1 foi nos sugerido um mecanismo genérico de construção e escrita de resultados. Portanto na fase 2 procuramos atender a essa sugestão da melhor forma.

Para isso, foi criada a estrutura **QueryResult**, que representa o resultado de uma *query* como um conjunto de linhas, sendo cada linha composta por vários *tokens* (strings).

```
struct result_line {
    char** tokens;
    int num_tokens;
};
```

```
struct query_result {
    ResultLine** lines;
    int num_lines;
    int capacity;
};
```

Foram adicionadas funções para:

- Criar um resultado vazio, que inicializa a estrutura com capacidade dinâmica, permitindo adicionar linhas conforme necessário.

- Adicionar linhas ao resultado. Cada *query* pode construir o seu próprio resultado, adicionando linhas com os tokens correspondentes aos dados produzidos. Desta forma, cada *query* fica responsável apenas por produzir dados, e não por decidir como esses dados são apresentados.

A função ***write_result*** foi adicionada para centralizar toda a lógica de output. O seu objetivo é escrever o resultado para um ficheiro ou para uma janela **ncurses** utilizando um delimitador.

Desta forma, nenhuma *query* precisa de saber se o output vai para um ficheiro ou ecrã, se está em modo ncurses ou como formatar as linhas

1.3.3 Queries

Para além da implementação das novas *queries*, foram realizadas alterações nas lógicas das queries da fase 1 para melhorar o seu desempenho a respeito de tempo e uso de memória.

1.3.3.1 Query 1

Query 1: Listar o resumo de um aeroporto, consoante o identificador recebido por argumento

Além dos dados retornados na fase 1, passou a ser necessário devolver também o **número de passageiros que chegaram ao aeroporto** e o **número de passageiros que partiram do aeroporto**, considerando apenas voos com estado diferente de cancelado.

Para suportar essa contabilização, como já foi mencionado anteriormente, foi criada a estrutura `AirportData` no ficheiro airport_catalog. Essa estrutura mantém um ponteiro para a entidade Airport e armazena campos auxiliares, como `arrival_count` e `departure_count`, inicializados a zero quando o aeroporto é adicionado ao catálogo. Essa abordagem permite manter a estrutura da entidade limpa, como sugerido durante a defesa da fase 1.

Além disso, foi garantido que não existam **identificadores duplicados** entre entidades, evitando contabilizações incorretas. Inicialmente, não havia validação que impedisse reservas com o mesmo `reservation_id` de serem processadas múltiplas vezes, o que poderia inflar os contadores de passageiros.

Durante a validação no parser, cada reserva válida é processada verificando os flight_ids associados. Para cada `flight_id`, busca-se o voo correspondente com ***get_flight_by_flight_id_from_catalog***. Se o voo não estiver cancelado, **incrementa-se** o `departure_count` do aeroporto de origem e o `arrival_count` do aeroporto de destino na respectiva estrutura `AirportData`, utilizando a função ***airport_passenger_increment***.

Com essas informações pré-calculadas, a execução da query torna-se direta. Para cada aeroporto, basta acessar a `AirportData` através da hash table do airport_catalog e obter os campos necessários (airport_code, airport_name, airport_city, airport_country, airport_type, arrival_count e departure_count) para gerar o resultado final.

1.3.3.2 Query 2

Query 2: Top N aeronaves com mais voos

Anteriormente na fase 1, tínhamos adicionado um campo à entidade `aircraft` que armazenava o número de `flight` que tinha realizado.

No pior caso, na execução da *query* era construído um array temporário com todos os `flights` existentes. Que seria ordenado de forma descrescente e depois apenas seriam impressos os N `flights`.

Para melhorar a sua eficiência, foi decidido o uso de uma **min-heap**.

```c
struct aircraftcount {
    char* aircraft_id;
    int flight_count;
};
```

```c
struct MinHeap {
    AircraftCount* heap;
    int size;
    int capacity;
};
```

Essa estrutura de dados é criada na função `execute_query2` com capacidade igual a N, correspondente ao número de aeronaves que se deseja listar. Dessa forma, no pior caso, apenas os N maiores valores de voos são armazenados, garantindo eficiência tanto em tempo quanto em memória.

Toda a lógica de manipulação da min-heap, incluindo inserção, remoção e ordenação, está implementada no ficheiro `query2_utils.c`.

Em uma min-heap, existem duas propriedades às quais temos de colocar atenção:
- O menor valor está na raíz;

- Cada nó pai deve ser menor ou igual aos seus filhos.

Portanto, a cada valor inserido na min-heap, enquanto a heap ainda não atingiu sua capacidade N, o elemento é colocado no final do array e, em seguida, ocorre o processo de heapify-up para equilibrar a heap, garantindo que o menor valor permaneça na raiz e que a propriedade de heap seja preservada.

Após a heap estar completamente preenchida com N elementos, cada novo valor é comparado diretamente com a raiz. Caso seja maior que o valor da raiz, substitui-se a raiz pelo novo elemento e realiza-se novamente o processo de balanceamento. Se o valor for menor, ele é descartado. **Dessa forma, a heap mantém sempre os N maiores valores de forma eficiente, sem precisar armazenar todos os elementos analisados**.

1.3.3.3 Query 3

Query 3: Aeroporto com mais voos dentro de um intervalo de tempo

Para a Query 3, inicialmente, para cada aeroporto tínhamos um **GPtrArray** de voos ordenado pela sua data de partida (departure). Na execução original, o procedimento consistia em percorrer cada aeroporto e, dentro de cada aeroporto, iterar pelo array de voos, contando quantos deles estavam dentro do intervalo de datas fornecido. Desta forma, realizava-se uma busca linear.

O método anterior consistia em percorrer o array desde o início, identificando o primeiro voo cuja data fosse igual ou maior que a data inicial do intervalo. Em seguida, continuava-se a iteração até encontrar um voo cuja data fosse maior que a data final do intervalo, somando os voos correspondentes. Esse processo era repetido para todos os aeroportos.

Atualmente, adotamos uma abordagem utilizando **busca binária**. Com a busca binária que divide repetidamente o array ao meio para determinar se devemos seguir à esquerda ou à direita, localizamos o índice do primeiro voo dentro do intervalo, ou seja, aquele cuja data é maior ou igual à data inicial do intervalo. Chamaremos esse índice de i.

Em seguida, utilizamos outra busca binária para encontrar o último voo do intervalo, ou seja, aquele cuja data seja menor ou igual à data final. Para otimizar a busca, iniciamos a procura a partir do índice i. Esse procedimento retorna um índice f. Assim, o número de voos dentro do intervalo de datas é dado por f - i + 1.

2 Queries

2.1 Query 4

Query 4: Qual o passageiro que esteve mais tempo no Top 10 pasageiros que mais gastaram em viagens durante um período de tempo

O objetivo é devolver o **ID** do pasageiro e o **número de semanas** em que esse passagero aparece no Top 10.

Para tornar esta operação eficiente, a solução foi dividida em diferentes fases: a fase de recolha dos dados, fase de pré-processamento e a fase de consulta.

Abaixo são apresentadas as estruturas de dados auxiliares a este proceso.

```
struct weekly_top {
    char* week_key;
    char* top_ids[10];
    int count;
};
```

```
struct passenger_spend{
    char* id;
    double value;
};
```

```
struct passenger_catalog {
    GHashTable* passengers_by_document_number;
    GHashTable* weekly_stats;
    WeeklyTop** timeline;
    int timeline_size;
};
```

A estrutura `weekly_top` representa o resultado final de uma semana relevante para a *query* 4. A `week_key` identifica unicamente cada semana. O `top_ids` armazena os identificados dos até 10 passageiros com maior gasto nessa semana e count indica quantas posições do top 10 estão preenchidas caso existam menos de 10 passageiros.

A estrutura `passenger_spend` é utilizada apenas durante o pré-processamento. Associa um passageiro ao valor total gasto numa semana. As tabelas hash não são diretamente ordenáveis e portanto esta estrutura veio como substituição pois permite converter os dados semanais para um vetor temporário que pode ser ordenado facilitanto o cálculo do top 10 de cada semana.

O `PassengerCatalog` centraliza toda a informação. A hash table `weekly_stats` acumula gastos durante o carregamento, onde cada chave é um identificador de semana e o valor é outra hash table que associa passageiros aos totais gastos.

A `timeline` é um vetor ordenado cronologicamente de ponteiros para `weekly_top`, usado após o pré-processamento para responder às queries de forma eficiente, evitando percorrer *hash tables* complexas a cada query e permite limitar facilmente o intervalo temporal.

Na fase de recolha de dados, durante o carregamento das reservas, cada reserva contribui com o seu preço para um determinado passageiro. A semana relevante é determinada a partir da data de voo. Esse valor é acumulado em `weekly_stats` agrupado por semana e por passageiro. Nesta fase, não são calculados rakings, apenas se acumulam valores.

No pré-processamento, após todos os *datasets* estarem carregados, para cada semana registada em `week_stats`, os passageiros são ordenados por valor gasto. Apenas os 10 passageiros com passageiros com maior gasto são selecionados. Esses dados são armazenados na estrutura `weekly_top`. Todas as semanas são colocadas na `timeline` que é depois ordenada. A estrutra `weekly_stats` é libertada, pois deixa de ser necessária.

Na execução da query, o intervalo de datas dado é convertido para um intervalo de semanas e apenas as semanas relevantes da `timeline` são analisadas. Todos os passageiros que aparecem nos top 10 dessa semana são recolhidos e o passageiro com maior número de ocorrências é determinado.

2.2 Query 5

Query 5: Top N companhias aéreas com mais tempo de atraso médio por voo

Esta query recebe como argumento o número de companhias aéreas que devem constar no output.

Para facilitar o cálculo do atraso médio, utilizamos a estrutura auxiliar `AirlineStats`, que armazena o identificador da companhia (airline_id), o tempo total de atraso (total_delay), o número de voos com atraso (delayed_flights_count) e, posteriormente, a média de atraso por voo, calculada como **media = total_delay / delayed_flights_count**.

No catálogo de voos (`flight_catalog`), é mantida uma hash table (***airline_lookup***) que permite, dado um `airline_id`, obter um ponteiro para a estrutura `AirlineStats` correspondente. Esta estrutura é essencial durante o processamento dos voos e na execução de queries que requerem acesso direto às estatísticas de uma companhia específica.

Adicionalmente, é mantido um array dinâmico (**airline_array**) que contém ponteiros para as mesmas estruturas `AirlineStats` armazenadas na hash table. Este array permite a aplicação de operações de ordenação, que não são suportadas de forma prática por hash tables. O uso deste array simplifica também a libertação de memória associada às estatísticas das companhias.

Desta forma, a utilização combinada de uma hash table e de um array permite otimizar diferentes padrões de acesso à informação: acesso direto por identificador e iteração ordenada sobre o conjunto completo sem duplicação de dados.

Durante o parser, na validação de voos, para cada voo válido verificamos se houve atraso com a função ***is_delayed(status)***. Caso o voo esteja atrasado, utilizamos ***get_flight_delay*** para **calcular a diferença** entre `actual_departure` e `departure` em minutos.

Em seguida, buscamos a `AirlineStats` da companhia aérea correspondente no catálogo. Se ainda não existir, adicionamos com valores inicializados a zero, utilizando ***flight_catalog_add_airline_stats***. Caso já exista, atualizamos os valores da estrutura com ***airline_stats_increment***, **somando o atraso ao total_delay** e incrementando `delayed_flights_count`. Esse processamento durante o parser prepara os dados necessários para a execução da query.

Na execução da query propriamente dita, recebemos o argumento N e chamamos ***prepare_airline_stats_for_sorting***, que calcula a média de atraso para cada companhia utilizando ***set_airline_stats_media***, arredondando o valor para três casas decimais conforme especificado. Em seguida, a função ***airline_stats_sort_array*** ordena o array de `AirlineStats` com base na média de atraso, resolvendo empates em ordem alfabética pelo airline_id.

Por fim, percorremos os primeiros N elementos do array ordenado para gerar o resultado final, obtendo assim o Top N de companhias aéreas com maior tempo médio de atraso por voo.

2.3 Query 6

Query 6: Listar o aeroporto de destino mais comum para passageiros de uma determinada nacionalidade

A query recebe como argumento a nacionalidade dos passageiros.

Para suportar esta query, o catálogo de reservas mantém uma estrutura auxiliar designada por `nationality_stats`. Esta estrutura é implementada como uma **hash table** cuja chave é a nacionalidade e cujo valor é outra hash table. Esta hash table interna associa cada `airport_id` (código IATA do aeroporto de destino) a um contador que representa o número de passageiros dessa nacionalidade que chegaram a esse aeroporto.

Desta forma, a estrutura assume o seguinte formato lógico:

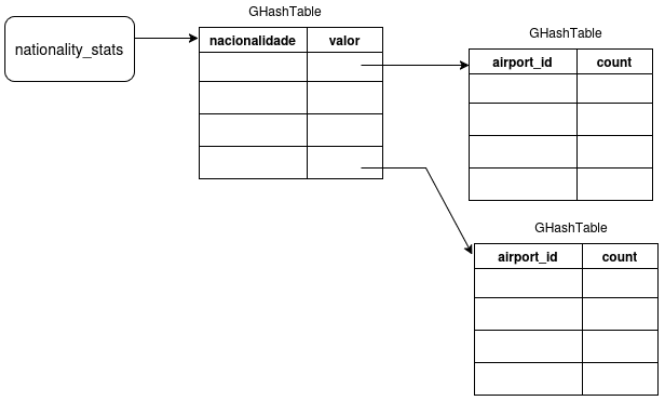


Figure 2: Estrutura Query 6

Esta abordagem permite um acesso eficiente às estatísticas associadas a uma nacionalidade específica, bem como a contagem incremental de passageiros por aeroporto de destino.

Durante o parser, no **processamento de cada linha válida de reservas**, são analisados os voos associados à reserva. Para cada **voo não cancelado**, obtém-se o aeroporto de destino. Em paralelo, é identificado o passageiro através do `document_number`, permitindo aceder à sua nacionalidade. Caso tanto a nacionalidade como o aeroporto de destino sejam válidos, é chamada a função ***reservation_catalog_add_nationality_increment***, que atualiza as estatísticas correspondentes.

Esta função começa por verificar se já existe uma entrada para a nacionalidade em `nationality_stats`. Se não existir, é criada uma nova hash table para armazenar os aeroportos associados a essa nacionalidade. Em seguida, é procurado o **contador do aeroporto** de destino; caso não exista, é criado e inicializado a zero. Por fim, o contador é incrementado, refletindo a chegada de mais um passageiro dessa nacionalidade ao aeroporto em questão. Este processamento durante o parser garante que, no final da leitura dos dados, todas as estatísticas necessárias para a query estão previamente calculadas.

Na execução da query, é criado um iterador específico através de ***reservation_catalog_create_stats_iter***, que permite percorrer todos os pares (`airport_id`, `count`) associados à nacionalidade fornecida como argumento. Caso não existam dados para essa nacionalidade, a query retorna um resultado vazio.

Durante a iteração, são comparados os contadores de passageiros de cada aeroporto, mantendo-se o aeroporto com o maior número de chegadas. Em caso de empate, o desempate é resolvido através da ordem alfabética crescente do código IATA, garantindo resultados determinísticos e consistentes com a especificação.

Após o término da iteração, o iterador é libertado e, caso tenha sido encontrado um aeroporto válido, o resultado final é construído contendo duas colunas: o código IATA do aeroporto de destino mais comum e o número de passageiros dessa nacionalidade que nele aterraram.

3 Otimizações

Para identificar os pontos críticos de desempenho do nosso código, utilizámos a ferramenta **Flame Graph** que permite visualizar quais as funções que estão a consumir mais tempo de CPU durante a execução do programa.

Com o Flame Graph conseguimos identificar funções candidatas a otimização, sem precisar adivinhar ou fazer alterações prematuras.

Para entender onde o nosso programa gastava mais tempo de CPU, gerámos um Flame Graph da execução inicial. A Figura 3 mostra o perfil original, onde é possível identificar claramente as funções críticas.

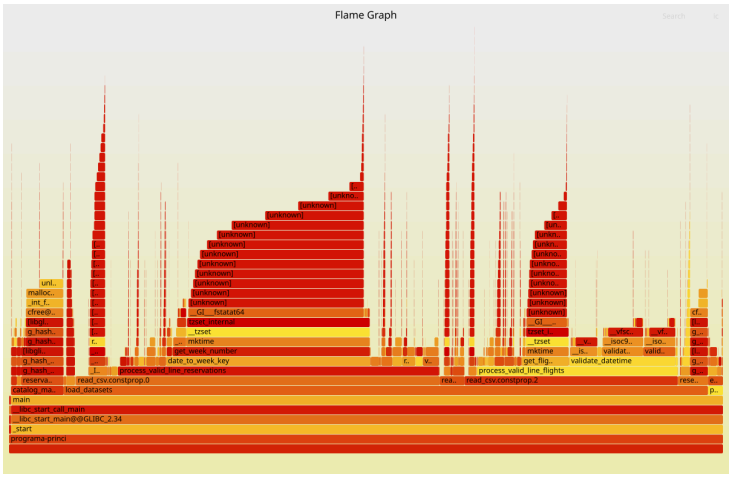


Figure 3: Estado inicial

Após implementar as otimizações sugeridas por esta análise, gerámos um segundo Flame Graph. A Figura 4 apresenta o estado após a melhoria, evidenciando a redução no tempo gasto nas funções críticas e a melhoria geral no desempenho do programa.

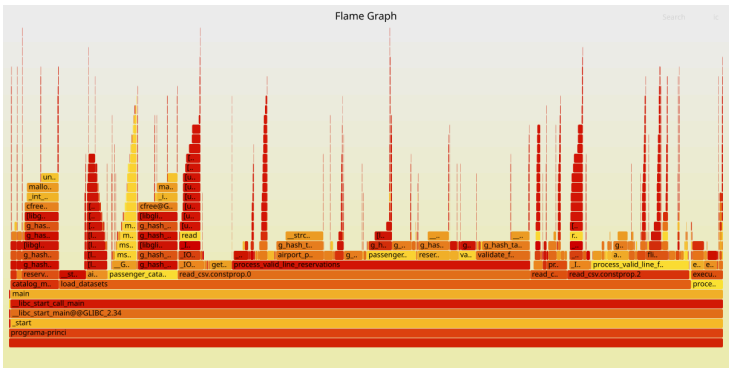


Figure 4: Estado otimizado

Uma das melhorias mais importantes foi a implementação de uma **String Pool**.

A análise do Flame Graph da versão inicial revelou que uma parte significativa do tempo de execução era consumida durante o processo de parsing do dataset, sobretudo ao elevado número de alocações dinâmicas e cópias de strings. Cada campo textual era duplicado individualmente através de **chamadas a strdup**, o que originava milhões de alocações redundantes para valores frequentemente repetidos, como nacionalidades, cidades, modelos de aeronaves e companhias aéreas.

Com o objetivo de ultrapassar este problema, foi introduzida uma String Pool. Esta estrutura é gerida pelo módulo `CatalogManager` e utiliza uma GHashTable para **armazenar apenas uma instância** de cada string distinta presente no dataset.

Durante o processo de parsing, sempre que uma string necessita de ser associada a uma entidade, o sistema passa a consultar a String Pool. Caso a string já exista, o ponteiro previamente alocado é **reutilizado**. Caso contrário, a string é duplicada uma única vez, registada na pool e o novo ponteiro é retornado. Desta forma, strings idênticas passam a ser partilhadas por múltiplas entidades, **eliminando a necessidade de alocações repetidas**.

A introdução da String Pool foi diretamente motivada pelos resultados observados no Flame Graph inicial, no qual as funções associadas à gestão dinâmica de memória surgiam como hotspots distribuídos ao longo de todo o pipeline de processamento. Considerando a elevada taxa de repetição de campos textuais no dataset, o modelo anterior de armazenamento revelou-se ineficiente tanto em termos de tempo de execução como de consumo de memória.

4 Modularização e Encapsulamento

O projeto foi desenvolvido seguindo uma abordagem modular, com o objetivo de maximizar a legibilidade, manutenibilidade e isolamento de responsabilidades. Cada módulo é responsável por um conjunto bem definido de funcionalidades, evitando dependências desnecessárias e acoplamento excessivo entre componentes.

A comunicação entre módulos é realizada exclusivamente através de interfaces públicas (ficheiros `.h`), garantindo um forte encapsulamento e ocultação de detalhes de implementação.

4.1 Arquitetura Modular

A estrutura do projeto reflete uma separação clara entre diferentes tipos de responsabilidades:

Entities (Entidades): Representam os dados do domínio da aplicação, como passageiros, voos, reservas e aeroportos. Estes módulos são responsáveis exclusivamente por armazenar dados e fornecer funções de acesso controlado, não contendo lógica de negócio complexa.

Catalogs (Catálogos): Funcionam como gestores de entidades, sendo responsáveis por armazenar e organizar coleções de dados, garantir unicidade e oferecer operações eficientes de pesquisa, inserção e atualização. Os detalhes da estrutura interna são totalmente encapsulados, expondo apenas *getters* e operações de alto nível para que módulos externos possam manipular estas estruturas de forma segura e controlada.

Queries (Consultas): Cada *query* é implementada de forma independente e modular, operando sobre os catálogos exclusivamente através das suas interfaces públicas. Esta separação permite adicionar, modificar ou remover queries sem impactar o resto do sistema.

Utils (Utilitários): Agrupam funcionalidades auxiliares e transversais ao sistema.

Validation (Validação): Centralizam toda a lógica de validação de dados.

IO (Entrada/Saída): Todas as operações relacionadas com leitura e escrita de dados, sejam ficheiros, consola ou outros dispositivos.

Interactive (Modo Interativo): Implementa a interface do programa interativo, isolando a lógica de apresentação da lógica de negócio.

4.2 Estratégias de Encapsulamento

O encapsulamento e a robustez foram assegurados através de estratégias complementares:

Ocultação de Dados (Opaque Structs): As estruturas internas (`struct`) são definidas exclusivamente nos ficheiros `.c`. O acesso aos dados é feito estritamente via *getters*, impedindo o acesso direto aos campos e protegendo a estrutura interna das entidades.

Imutabilidade e Cópias Defensivas: Para preservar a integridade da *String Pool* (onde múltiplos registos partilham o mesmo endereço de memória), os *getters* referentes a campos de texto retornam sempre uma cópia (`strdup`) dos dados. Esta abordagem desacopla a representação interna da sua utilização externa, garantindo que a manipulação ou libertação da string por parte do invocador não corrompe os dados partilhados armazenados no catálogo.

Gestão Centralizada de Recursos (Ownership): A arquitetura adota o padrão de desenho *Flyweight* para a gestão de strings. Ao invés de cada entidade ser responsável pela alocação e libertação dos seus próprios campos de texto, essa responsabilidade é delegada a um gestor central (no `CatalogManager`). Isto simplifica os destrutores das entidades e evita erros de *double free*, garantindo que a memória é gerida num único ponto controlado.

5 Testes

Para a secção de testes a estratégia manteu-se em comparação com a fase 1. 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***).

5.1 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

5.2 Estatísticas

Nas figuras seguintes são apresentadas as estatísticas das novas *queries* e também as estatísticas das *queries* da fase anterior tendo em conta as novas alterações.

	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6
Computador 1	31.3	13.0	415.5	2.6	3.0	18.6
Computador 2	24.3	9.6	204.1	1.2	1.3	9.6
Computador 3	31.7	14.3	413.8	2.9	3.1	18.1

Table 1: Estatísticas por query fase 2 (ms)

	Tempo total de execução (s)	Memória utilizada (MB)
Computador 1	23.5	1262
Computador 2	13.3	1262
Computador 3	17.5	1261

Table 2: Estatísticas gerais

5.3 Conclusão estatísticas

Pela observação das tabelas, é possível concluir que o Computador 2 voltou a apresentar o **melhor desempenho global**, registando os menores tempos de execução em todas as queries e também no tempo total de execução. Este comportamento confirma que o processador Intel i7-12650H continua a oferecer uma vantagem consistente face aos restantes sistemas.

Um ponto de destaque nos resultados obtidos é a estabilidade do consumo de memória, que se fixou aproximadamente nos 1262 MB em todas as máquinas testadas. Esta uniformidade demonstra que a pegada de memória depende do volume de dados carregado e das estruturas de dados definidas, sendo virtualmente imune a variações nas versões do Kernel ou na arquitetura dos processadores. Entre os sistemas equipados com processadores AMD, os resultados mantêm-se equilibrados verificando-se que o Computador 1 mantém uma ligeira vantagem sobre o Computador 3.

Em suma, os dados estatísticos validam não só a eficiência da solução desenvolvida, mas também a sua previsibilidade e estabilidade em diferentes ambientes Linux, garantindo tempos de resposta adequados mesmo nas operações de maior exigência computacional.

6 Programa interativo

Para esta fase, foi implementado um programa interativo utilizando a biblioteca **ncurses** O objetivo foi proporcionar ao utilizador uma experiência intuitiva na execução das *queries*.

A interface foi desenhada de forma modular, utilizando dimensões uniformes para garantir a consistência visual. A estrutura divide-se em dois níveis: Frames (contentores estruturais) e Pads (conteúdo interno). À semelhança do desenvolvimento web (*HTML/CSS*), onde elementos de layout organizam subelementos de conteúdo, esta hierarquia assegura a independência entre a gestão do posicionamento e a exibição de informação, otimizando a organização do código.

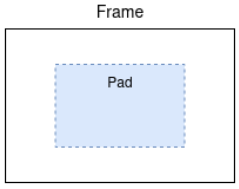


Figure 5: Hierarquia janelas ncurses

Principais funcionalidades

Exemplos visuais de cada funcionalidade encontram-se documentados na secção de **Anexos**.

1) Adaptação ao tamanho do terminal:

O programa verifica automaticamente se o terminal tem dimensões suficientes para desenhar todas as janelas definidas. Caso o terminal seja demasiado pequeno, é exibida uma mensagem instruindo o utilizador a aumentar o tamanho da janela, até que seja suficiente para uma apresentação correta da interface.

2) Janela de resultados *scrollable*:

As respostas às queries são apresentadas numa janela dedicada que suporta *scroll* vertical, permitindo navegar por resultados extensos sem perder a formatação ou truncar dados.

3) Validação de argumentos:

Cada query é validada de forma completa, incluindo:

- Número de argumentos: verifica se todos os argumentos obrigatórios foram fornecidos;
- Validação contexto: garante que os argumentos fazem sentido dentro do contexto da query (por exemplo, a Query 3 espera duas datas, sendo a primeira a data de início e a segunda a data de fim sendo assim, não é permitido que a data de início seja posterior à data de fim);
- Validação semântica: caso haja erros na introdução dos argumentos, são exibidas mensagens claras e opções para tentar novamente ou retornar ao menu principal.

4) Gestão de espaço em janelas de entrada:

Na janela específica onde o utilizador insere o caminho do dataset, a área de texto é limitada a um retângulo definido na interface. Quando o utilizador atinge o limite horizontal da caixa, a escrita continua na linha seguinte dentro do mesmo retângulo. Esta abordagem evita que o texto ultrapasse os limites visuais da janela, mantendo a interface organizada e legível.

5) Gestão do delimitador dos resultados:

Permite ao utilizador escolher o delimitador que quer que seja usado na escrita dos resultados.

6) Queries sem resultados

Para tornar a interface mais clara para o utilizador, foi implementada uma janela específica que exibe um aviso sempre que uma query introduzida não produz resultados.

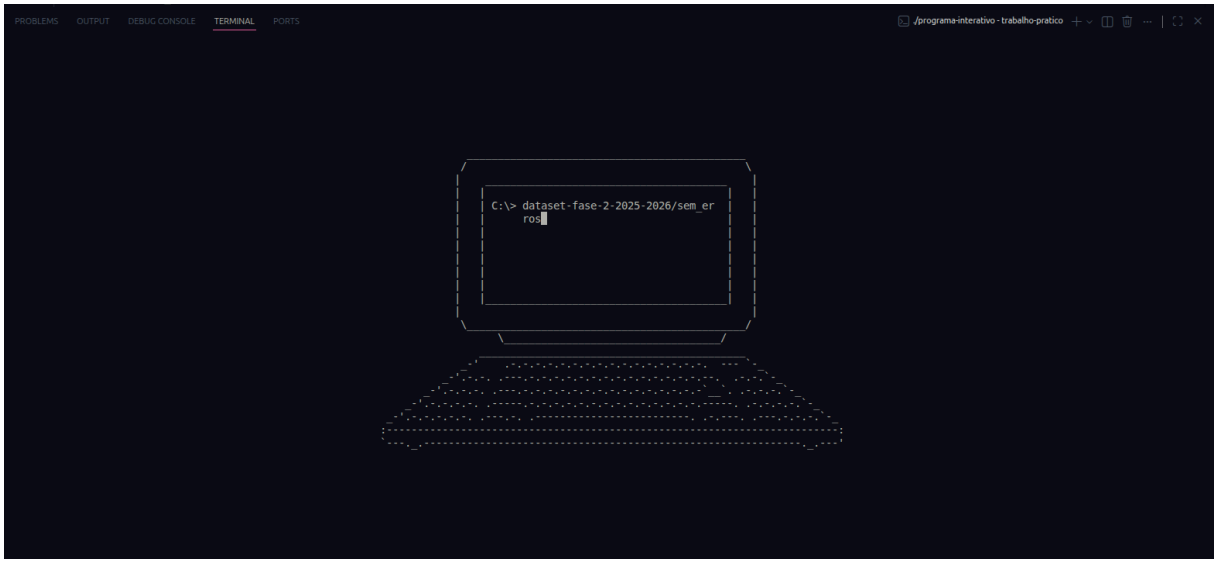
7 Conclusão

A segunda fase do projeto consolidou conhecimentos fundamentais em programação C, com especial foco em modularização, encapsulamento e gestão eficiente de memória. A **refatoração estrutural** através das estruturas Data separou dados estáticos de informações dinâmicas, **melhorando** significativamente a **clareza e manutenibilidade do código**. O mecanismo genérico de resultados (QueryResult e write_result) desacoplou a produção de dados da sua apresentação, tornando o sistema mais modular e preparado para futuras extensões. As otimizações de desempenho, guiadas pela análise com Flame Graph, trouxeram melhorias, nomeadamente a substituição de arrays por **min-heaps na Query 2**, a implementação de **busca binária na Query 3** e a **introdução da String Pool** que eliminou redundâncias e acelerou o processamento de dados. Entre os pontos de aprendizagem mais relevantes destacam-se a importância da modularização e encapsulamento para manter o código organizado e facilitar manutenção, a escolha adequada de estruturas de dados (GHashTables, GPtrArrays, heaps) para otimizar diferentes operações, o papel do pré-processamento estratégico na eficiência de queries complexas e a validação rigorosa como pilar da fiabilidade do sistema, evitando propagação de erros. Esta fase estabeleceu uma base sólida onde correção, organização e desempenho coexistem. A experiência adquirida em análise de desempenho, design de estruturas de dados e desenvolvimento de interfaces interativas constitui um alicerce valioso para projetos de maior complexidade.

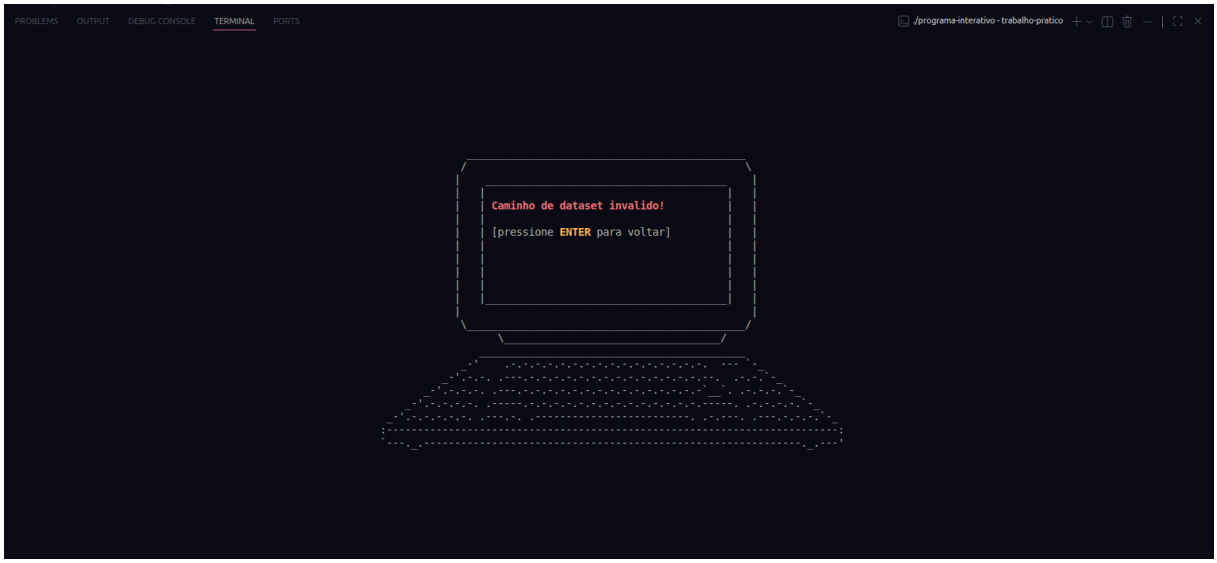
8 Anexos

Anexo I – Limite mínimo do terminal

Anexo II – Menu Inicial

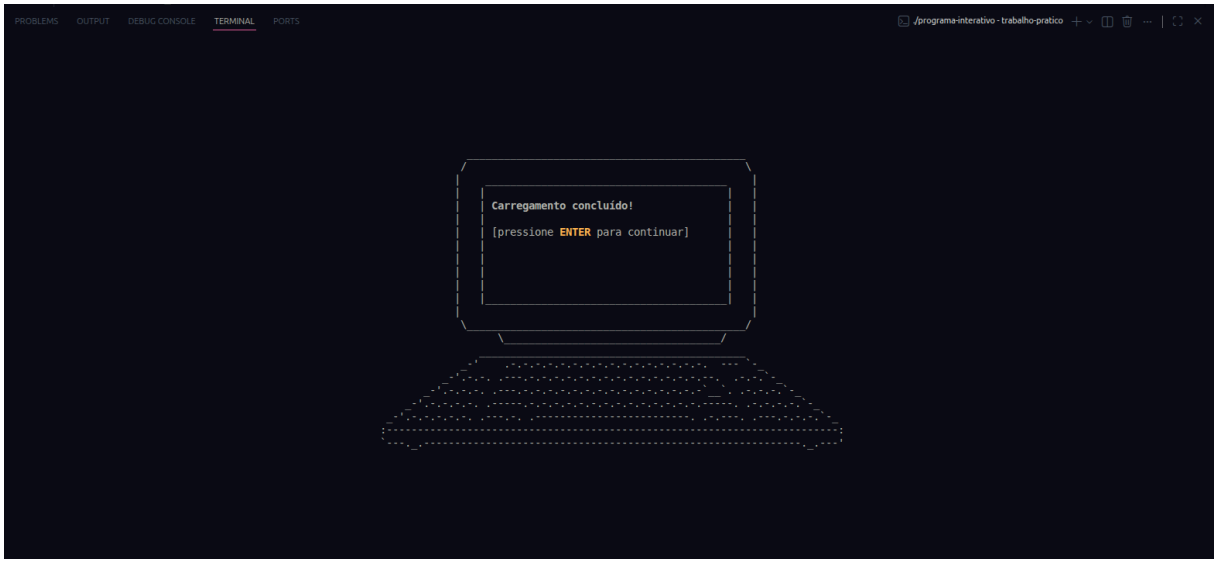


Anexo III – Limite escrita

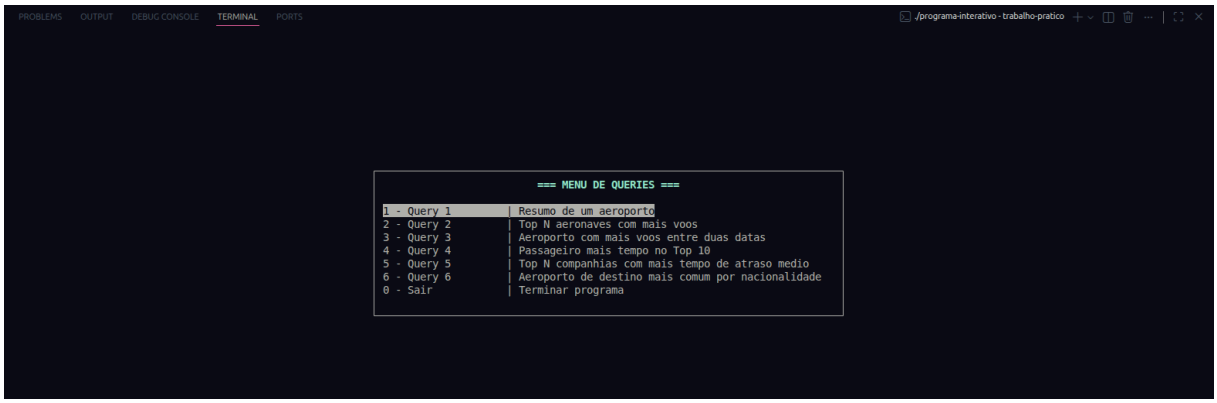


Anexo IV – Caminho *dataset* inválido

Anexo V – Menu carregar dados



Anexo VI – Carregamento de dados com sucesso



Anexo VII – Menu Queries

Anexo VIII – Menu delimitador

Anexo IX – Menu argumentos

Anexo X – Número inválido de argumentos

Anexo XI – Erro argumentos

Anexo XII – Semantica inválida de argumentos

Anexo XIII – Menu resultado queries

Anexo XIV – Menu queries sem resultado