Universidade do Minho
Escola de Engenharia

Laboratórios de Informática III

Trabalho prático - Fase 2

janeiro,2025

Grupo 80

Pedro Manuel Macedo Rebelo, A104091

Eduarda Mafalda Martins Vieira, A104098

Margarida Cunha da Silva, A104357


A104091


A104098


A104357

Conteúdo

1. Introdução 3
2. Ajustes feitos 4
3. Sistema 4
4. Discussão 6
   4.1. Análise de desempenho 6
   4.2. Modularização e Encapsulmaneto 6
   4.3. Estruturas de dados 6
   4.4. Programa de testes 7
5. Queries 7
   5.1. Query 1: Listar o resumo de um aeroporto 7
   5.2. Query 2: Top N aeronaves com mais voos realizados 7
   5.3. Query 3: Listar o aeroporto com mais partidas entre 2 datas 8
   5.4. Query 4: Qual o passageiro que esteve mais tempo no top 10 de passageiros que mais gastaram em viagens durante um período? 8
   5.5 Query 5: Top N companhias aéreas com mais tempo de atraso médio por voo 9
   5.6 Query 6: Listar o aeroporto de destino mais comum para passageiros de uma determinada nacionalidade 9
6. Resultados obtidos 10
7. Conclusão 10

1. Introdução

O presente relatório apresentará informações relativas à 2ª Fase do Trabalho Prático da Unidade Curricular de Laboratórios de Informática III, pertencente ao 2º Ano da Licenciatura em Engenharia Informática da Universidade do Minho, realizada no ano letivo 2025/2026. O projeto consiste na implementação de uma base de dados em memória, utilizando ficheiros .csv fornecidos pela equipa docente, contendo dados de um sistema de gestão de voos (Aeroportos, Aeronaves, Voos, Passageiros e Reservas). O objetivo principal é armazenar estes dados de forma eficiente e implementar métodos de pesquisa e associações entre eles, aplicando conceitos como modularidade, encapsulamento, reutilização e abstração de dados.

Após a conclusão da 1ª fase, onde foi realizado o parsing e validação dos dados de entrada, o modo batch e as três queries iniciais, a 2ª fase focou-se na implementação das queries restantes, no desenvolvimento do modo interativo, na realização de testes funcionais e de desempenho e em manter modularidade e encapsulamento respondendo assim aos desafios propostos no enunciado desta nova etapa.

2. Ajustes feitos

Seguem-se algumas alterações realizadas no desenvolvimento da 2ª fase, tendo em conta o aconselhamento dos docentes durante a 1ª defesa, bem como a necessidade de garantir o funcionamento eficiente do programa com o novo dataset, significativamente maior do que o utilizado inicialmente. Destacam-se as seguintes modificações:

- Refatoração e reorganização de vários ficheiros do diretório trabalho-pratico, com destaque para a separação de responsabilidades entre módulos e a melhoria da legibilidade do código.
- Criação e/ou ajuste de estruturas auxiliares para suportar queries mais complexas, como o cálculo do top 10 de passageiros por semana e o tratamento de companhias aéreas com maior atraso médio.
- Adaptação do sistema de output, com alteração da lógica de geração dos ficheiros de resultados para garantir compatibilidade com o novo formato e requisitos do enunciado da 2ª fase.
- Expansão do modo interativo, permitindo ao utilizador executar todas as queries propostas e mudar dinamicamente o dataset em uso.
- Melhoria da documentação interna do código, com comentários mais detalhados e uniformização dos nomes das funções e variáveis, facilitando a manutenção e compreensão do projeto.

Estas alterações foram fundamentais para garantir a escalabilidade e fiabilidade do sistema, respondendo de forma eficaz aos desafios colocados pelo novo enunciado e pelo aumento do volume de dados.

3. Sistema

Na segunda fase do trabalho, o sistema foi significativamente expandido para cumprir os novos requisitos do enunciado. Para além do modo batch já existente, foi implementado o modo interativo, permitindo ao utilizador executar queries manualmente, visualizar resultados em tempo real e alterar o dataset conforme necessário. Esta funcionalidade responde diretamente ao pedido do novo enunciado, tornando a aplicação mais flexível e adaptada a diferentes cenários de utilização.

No início da execução, o sistema continua a criar todas as estruturas de catálogo necessárias (Aeroportos, Aeronaves, Voos, Passageiros, Reservas), agora otimizadas para lidar com datasets de maior dimensão. A leitura dos ficheiros .csv é realizada por um parser genérico, que invoca funções específicas para criação e validação de entidades como também inserção dos dados. Linhas inválidas são registadas nos respetivos ficheiros de erro, garantindo rastreabilidade e robustez.

O pré-processamento dos dados foi alargado: além da ordenação das aeronaves por número de voos, foram criadas estruturas auxiliares para calcular o top 10 de passageiros por semana, agrupar reservas por períodos e analisar atrasos médios por companhia aérea, etc. Estas operações intensivas são realizadas apenas uma vez, durante o carregamento inicial, permitindo respostas rápidas às queries subsequentes.

No modo batch, o sistema lê e executa sequencialmente as queries definidas no ficheiro de comandos, escrevendo os resultados nos ficheiros de output apropriados. No modo interativo, o utilizador pode escolher queries, fornecer argumentos, visualizar resultados e mudar o dataset sem reiniciar a aplicação.

Por fim, o sistema garante a correta gestão da memória, evitando leaks, e inclui melhorias na documentação interna e tratamento de erros, conforme recomendado pelos do-

centes e exigido pelo novo enunciado. Estas alterações tornam o sistema mais robusto, eficiente e preparado para desafios futuros.

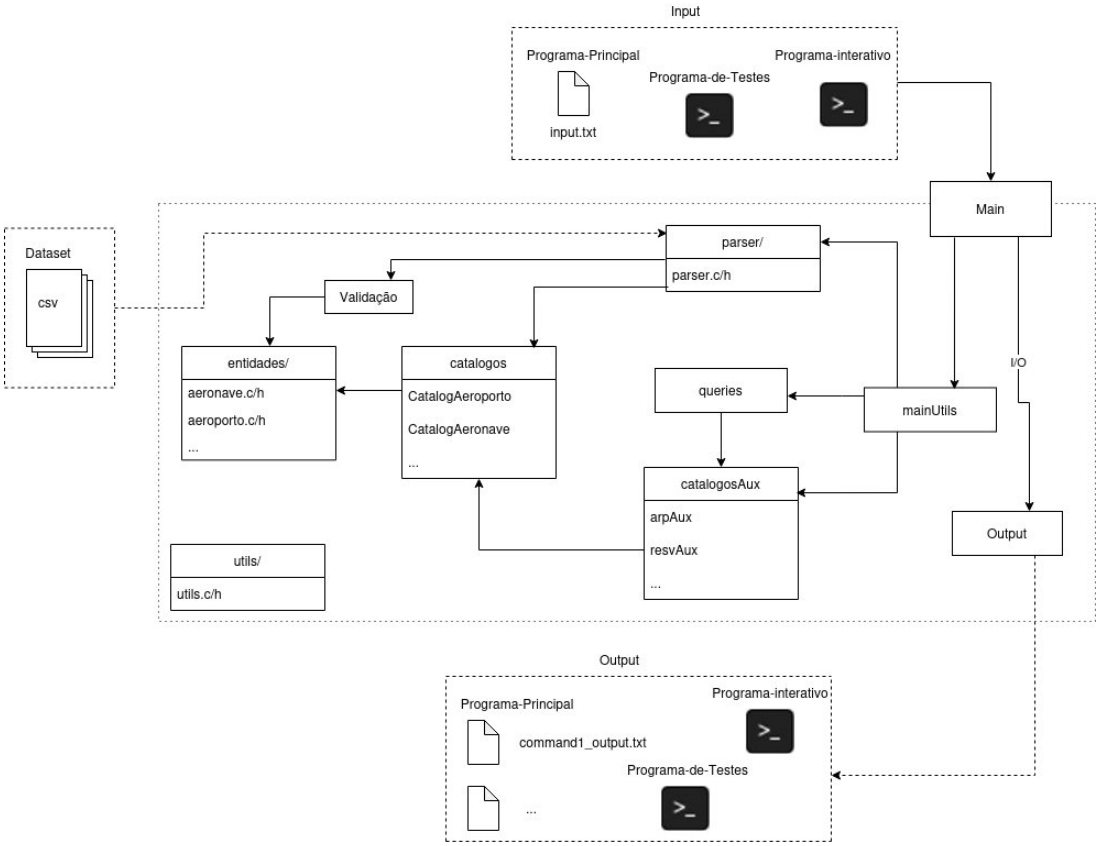

Figura 1- Arquitetura para a aplicaçã.

4. Discussão

4.1. Análise de desempenho

O desempenho do projeto foi avaliado com base na execução das queries, analisando o tempo de resposta e o consumo de memória em diferentes máquinas dos membros da equipa. Apesar das variações de hardware, os resultados mantiveram-se dentro dos parâmetros esperados, graças à utilização de algoritmos eficientes e estruturas como as HashTables da biblioteca GLib. Na segunda fase, o sistema foi otimizado para lidar com datasets de maior dimensão, recorrendo a pré-processamento e organização dos dados, o que permitiu manter a escalabilidade e a rapidez de resposta mesmo perante cenários mais exigentes, conforme solicitado no novo enunciado.

4.2. Modularização e Encapsulmaneto

A modularização foi aplicada de forma consistente ao longo do desenvolvimento do projeto, especialmente nesta segunda fase, onde o aumento da complexidade e do volume de dados exigiu uma estrutura ainda mais organizada. O código está dividido em várias unidades bem definidas e independentes, incluindo módulos para cada tipo de entidade principal (aeroporto, aeronave, voo, passageiro, reserva), módulos de gestão de catálogos, modulos auxiliares aos catálogos e módulos de responsabilidade única como parser, validação, queries e utils.

Cada módulo mantém uma responsabilidade clara, o que facilita a manutenção, testes e futuras expansões. Por exemplo, a lógica de validação de um aeroporto permanece isolada em validacao.c, enquanto a gestão dos dados está em aeroportos.c e a definição da estrutura em aeroporto.c. Esta separação foi fundamental para implementar novas queries e funcionalidades exigidas na segunda fase, como o modo interativo e o processamento eficiente de grandes datasets.

A interface de cada módulo é exposta através de ficheiros .h, ocultando os detalhes internos e promovendo o encapsulamento. A criação e manipulação das entidades é feita por funções específicas, como create_and_store_aeroporto, que têm como base setters e getters, garantindo a integridade dos dados e minimizando efeitos colaterais. Esta abordagem modular permitiu responder de forma eficaz aos desafios do novo enunciado e manter o projeto escalável e robusto.

4.3. Estruturas de dados

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

- **GHashTable (GLib):** Escolhemos GHashTable para os catálogos e auxiliares destes(por ex., CatalogAeroporto, CatalogAeronave, CatalogVoo) porque precisamos de procurar entidades pelo seu identificador único — por exemplo, código IATA para aeroportos ou ID para voos — de forma muito rápida. Em vez de percorrer listas, um hashtable dá-nos esse acesso quase, o que torna queries como a Query 1 (validações e pesquisas rápidas) muito mais rápidas e simples.
- **GPtrArray (GLib):** Usámos arrays de ponteiros dinâmicos da GLib para armazenar as coleções de dados de forma sequencial (ex: voos_array em voos.c).

Isto facilita a iteração sobre todos os elementos como também na ordenação, o que foi essencial para implementar as queries 2 e 3, que necessitam de processar *todos* os voos num determinado período ou de um determinado tipo.

- **Structs Auxiliares:** Para a maior parte das queries, criámos structs auxiliares temporária spara facilitar a contagem e ordenação das entidades, demonstrando uma solução otimizada para um problema específico. Essas structs são usadas apenas enquanto precisamos de agregar ou ordenar dados e depois descartadas

4.4. Programa de testes

Para garantir a correção e fiabilidade do nosso projeto, foi desenvolvido e aprimorado o programa-testes. Este programa automatiza a verificação dos outputs, sendo especialmente importante nesta etapa devido ao aumento do número de queries e à complexidade dos dados.
O funcionamento do programa-testes é o seguinte:

- Recebe três argumentos: o caminho para o dataset, o ficheiro de comandos (input.txt) e a pasta com os outputs esperados.
- Executa o programa-principal num diretório temporário, gerando os ficheiros de resultado para cada comando.
- Compara, linha a linha, cada ficheiro de output gerado (ex: command1_output.txt) com o respetivo ficheiro de output esperado, identificando rapidamente discrepâncias.
- No final, apresenta um relatório detalhado no terminal, indicando para cada query o número de testes bem-sucedidos (ex: Q1: 100 de 100 testes ok) e tempo usado para cada query e total, em caso de falha, aponta a primeira linha onde ocorreu a diferença.

Na segunda fase, este programa revelou-se ainda mais fundamental para validar a lógica das novas queries, garantir a compatibilidade com datasets maiores e detetar regressões durante as otimizações do código. Assim, assegurámos que todas as funcionalidades implementadas estão corretas e alinhadas com os requisitos do enunciado.

5. Queries

5.1. Query 1: Listar o resumo de um aeroporto

A Query 1 (query_aeroport_by_code) recebe um código IATA e realiza uma pesquisa eficiente na GHashTable de aeroportos através da função catalog_get_aeroporto_by_code. Se o aeroporto for encontrado, são recolhidos e formatados os seus dados (código, nome, cidade, país, tipo), juntamente com o número de passageiros que aterraram e partiram nesse aeroporto, que é calculado ao percorrer as reservas, considerando apenas reservas com voos com estado diferente de cancelado, conforme exigido no novo enunciado. O output é gerado no formato especificado (por omissão, separado por ponto e vírgula). Caso o identificador não exista, é retornada uma linha vazia.

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

A implementação da Query 2 (query_list_top_n_aeronaves_by_flightcount) foi dese-

nhada para uma performance elevada através de pré-processamento.

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

Assim, quando a query é efetivamente executada:

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

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

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

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

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

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

5.4. Query 4: Qual o passageiro que esteve mais tempo no top 10 de passageiros que mais gastaram em viagens durante um período?

A Query 4 (query_passageiro_top10_por_periodo) identifica o passageiro que esteve mais tempo no top 10 dos que mais gastaram em viagens durante um determinado período. Opcionalmente, pode receber um intervalo de datas como filtro, considerando apenas as semanas compreendidas nesse intervalo.

A implementação segue o enunciado da segunda fase:

- O top 10 é calculado semanalmente, com base no valor total gasto por cada passageiro (soma dos preços das reservas).
- Para cada semana (de domingo a sábado), é gerado o top 10 de passageiros.
- O campo que delimita as semanas é a data de partida estimada do voo (departu-

re).

- Para cada passageiro, é contabilizado o número de semanas em que esteve no top 10.
- Em caso de empate, prevalece o passageiro com o identificador (document_number) mais baixo.

A lógica foi implementada recorrendo a estruturas auxiliares (como passageiroAux), arrays dinâmicos (GPtrArray) e funções de ordenação e contagem, garantindo eficiência mesmo com grandes volumes de dados. O output segue o formato especificado: document_number;first_name;last_name;dob;nationality;count_top_10.

Esta abordagem modular e otimizada permite responder rapidamente à query, mesmo em datasets extensos, conforme exigido pelo novo enunciado.

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

A Query 5 (query_top_n_airlines_by_average_delay) recebe como argumento o número N de companhias aéreas a incluir no output. A implementação segue os requisitos do novo enunciado:

- Para cada companhia aérea, é calculado o tempo médio de atraso por voo, considerando apenas voos com estado Delayed.
- O atraso é contabilizado em minutos, como a diferença entre a data real de partida (actual departure) e a data estimada (departure).
- Para cada companhia, são guardados o nome, o número de voos atrasados (delayed_flights_count) e o atraso médio (average_delay), que no fim de tudo é arredondado a três casas decimais.
- Em caso de empate no atraso médio, as companhias são ordenadas alfabeticamente.
- O output apresenta os N resultados no formato especificado: airline;delayed_flights_count;average_delay.

A lógica foi implementada recorrendo a estruturas auxiliares e arrays dinâmicos (GPtrArray), garantindo eficiência e precisão mesmo com grandes volumes de dados.

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

A Query 6 (query_6) recebe como argumento a nacionalidade dos passageiros e segue os requisitos:

• O sistema utiliza a tabela de hash nationalitys_hash para associar nacionalidades a estatísticas de aeroportos, garantindo acesso eficiente mesmo com grandes volumes de dados.

• A função auxiliar get_top_aeroporto_from_nationality identifica o aeroporto de destino mais comum entre os passageiros da nacionalidade indicada, contabilizando apenas voos com estado diferente de cancelado.

• Para cada aeroporto, calcula o número de passageiros dessa nacionalidade que aterraram nesse destino.

• Em caso de empate, o aeroporto com o menor código IATA (obtido por get_aeroportoQ1_code) é selecionado, seguindo a ordem lexicográfica.

• O output apresenta o código IATA do aeroporto e o número de passageiros que ater-

raram, formatado por omissão como %s;%d\n ou, no formato alternativo, como %s=%d\n.
• Caso não existam passageiros com a nacionalidade indicada, ou não haja aeroporto correspondente, é retornada uma linha vazia.

6. Resultados obtidos

Os resultados obtidos nos testes são coerentes com as expectativas e confirmam a conformidade com o enunciado da segunda fase. Os algoritmos e estruturas de dados utilizadas (HashTables, arrays dinâmicos) demonstraram elevada eficiência, mesmo perante grandes volumes de dados. A modularização e a separação de responsabilidades facilitaram a manutenção e evolução do código, sem comprometer o desempenho global do sistema. Adicionalmente, o programa de testes automatizado permitiu validar todas as queries e detetar rapidamente eventuais regressões, garantindo a robustez e fiabilidade da solução desenvolvida.

7. Conclusão

O trabalho desenvolvido ao longo das duas fases do projeto cumpriu, de forma geral, todos os requisitos propostos no enunciado. O sistema foi desenhado para ser eficiente, modular e escalável, recorrendo a estruturas de dados como HashTables e arrays dinâmicos da GLib, que se revelaram fundamentais para lidar com grandes volumes de dados e garantir rapidez nas consultas.

Na segunda fase, o projeto foi aprimorado com a implementação do modo interativo, novas queries e otimizações para suportar datasets de maior dimensão. Foram realizadas correções e melhorias sugeridas pelos docentes, bem como adaptações para garantir a robustez e a conformidade com o novo enunciado. A documentação do código foi reforçada, seguindo boas práticas e facilitando a manutenção futura.

Durante o desenvolvimento, enfrentámos desafios como a resolução de memory leaks, a adaptação contínua das estruturas de dados e o equilíbrio entre encapsulamento e desempenho. A modularização e a separação de responsabilidades foram essenciais para manter o código organizado e eficiente, mesmo perante requisitos mais exigentes.

O programa de testes automatizado permitiu validar todas as funcionalidades e garantir a fiabilidade dos resultados. Embora haja sempre espaço para melhorias e otimizações, consideramos que o sistema final está sólido, eficiente e preparado para responder aos desafios propostos, refletindo o esforço e a aprendizagem adquiridos ao longo do projeto.