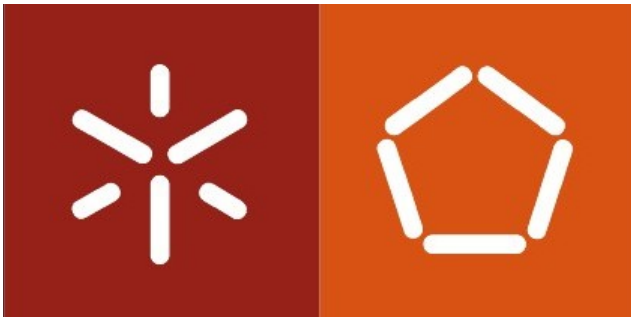


Universidade do Minho
Licenciatura em Engenharia informática

Projeto Laboratórios de informática 1º Fase

Departamento de informática

Universidade do Minho - novembro de 2023

Desenvolvido pelo grupo 45:

Afonso Pedreira (104537)

Dário Guimarães (A104344)

Hugo Rauber (A104534)

Índice:

1. Introdução

O presente relatório refere-se à primeira fase do desenvolvimento do projeto proposto na unidade curricular de Laboratórios de informática III do ano 2023/2024, que visa a criação de um programa de gestão e consulta de dados relacionados a utilizadores, voos e reservas. Esta fase inicial concentra-se na criação da arquitetura da aplicação, na implementação das funcionalidades básicas e na validação dos dados, conforme especificado nos requisitos fornecidos.

A proposta visa fornecer uma plataforma robusta e funcional que permita aos usuários consultar e extrair informações essenciais dos conjuntos de dados disponibilizados, bem como garantir a integridade e validade das informações processadas.

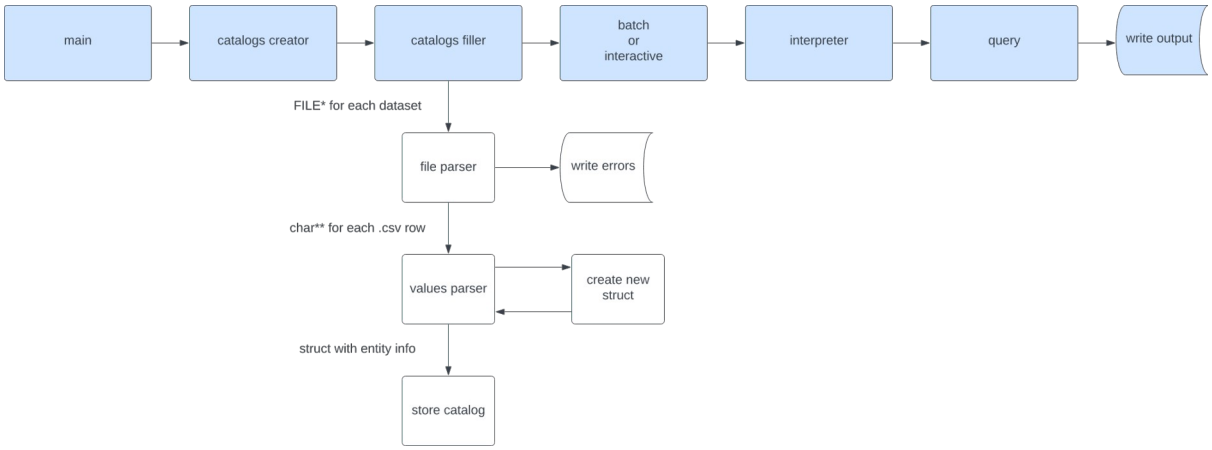
Durante esta etapa inicial, foi dado ênfase à definição de uma arquitetura modular, seguindo os princípios de modularidade discutidos nas diretrizes do projeto. Além disso, foram implementadas algumas das funcionalidades iniciais de consulta, cujos resultados foram submetidos aos testes dados, visando garantir a sua exatidão e adequação aos requisitos estabelecidos.

Este relatório, dividido em capítulos, aborda detalhes sobre a **arquitetura da aplicação**, os **testes unitários e workflow de trabalho**, **funcionalidades implementadas** até o momento e uma breve **conclusão**.

2.Arquitetura da aplicação:

Com o objetivo de atingir a maior modularidade possível no nosso projeto, dividimos as diversas tarefas em pequenas partes, acabando assim com uma estrutura que consideramos robusta e de fácil manutenção, tal como conseguimos provar ao longo desta primeira fase.

O nosso projeto ficou dividido em 5 módulos principais, que estão graficamente representados a baixo, e serão explicados.



O nosso módulo de entrada, o **catalogs creator**, é o responsável pela manutenção dos catálogos, vistos aqui como um espaço na memória, e abstraindo o seu conteúdo. É ele o responsável por alocar o espaço, criar a estrutura de dados, e libertar a memória alocada assim que possível. Este módulo irá retornar uma *struct* que acopla todos os catálogos

```
//! All the HashTables pointers in one struct
typedef struct catalogs {
  GHashTable* users;
  GHashTable* flights;
  GHashTable* passengers;
  GHashTable* reservations;
}* Catalogs;
```

Quanto à estrutura de dados escolhida para armazenar os nossos catálogos, optamos por Hash Tables, pois conjugam facilidade de criação e procura em um tempo constante, crucial para um tempo de execução aceitável e estável. A implementação escolhida, foi a da GLib, que disponibiliza todo um arsenal de funções e métodos poderosos que nos permitiram focar totalmente na implementação do projeto.

De seguida, para preencher o catálogo, temos o **catalogs filler**, responsável por todo o processo que começa com a leitura dos datasets e termina com os catálogos preenchidos. No entanto isto é um processo grande, e manter isto tudo dentro de um único módulo seria contra o paradigma, então optamos por dividir o processo em 3 etapas:

. o **file parser** recebe cada um dos ficheiros csv isoladamente, e cria um array, com o valor, como uma string, de cada uma das colunas, para cada linha. Caso detete algum erro em uma linha, recorre ao módulo **write errors** para o registrar.

. o **values parser** recebe o array de strings, valida cada uma individualmente e aloca e preenche as respectivas *structs* - tudo isto está também novamente modularizado em dois ficheiros separados)

. por fim, o **store catalog** recebe as *structs* que são armazenadas através das funções definidas pela GLib

Com os catálogos preenchidos podemos passar para a execução das queries. Aqui entra o modo **batch** que, de forma simples, vai passar como uma string, cada linha do ficheiro.

Para tentar criar uma solução realmente modular, ou seja, que seria compatível com o modo *batch* e com o modo *interactive*, entra o modo **interpreter** que recebe as queries como uma string, que terá sempre a mesma estrutura, e divide-a em um array de strings, com o primeiro elemento sendo o *id* da query e a *format flag*, e os restantes os parâmetros para a query.

Cada **query** vai ter o seu próprio ficheiro, e recebe os seus parâmetros de maneira uniforme - isto inclui os catálogos, a format flag e restantes parâmetros. Após calcular a sua resposta, a query passa ao módulo **write output** um array com uma *struct key field* que será útil caso a *format flag* esteja ativa.

Esta abordagem, de subdivisão do projeto em módulos especializados, revelou-se fundamental para a criação de um código menos suscetível a erros, proporcionando uma estrutura clara e bem definida. A modularidade permitiu isolar funcionalidades específicas, facilitando a identificação e correção de falhas, além de simplificar futuras atualizações. Além disso, a divisão de tarefas entre os membros da equipe tornou-se mais eficaz, permitindo um trabalho conjunto mais fluido e focado em áreas específicas, resultando em maior eficiência durante o desenvolvimento e manutenção do código.

3. Testes Unitários e Workflow de trabalho

Com o objetivo de evitar problemas que poderiam com o tempo escalar para algo incontrolável, decidimos tomar algumas boas práticas durante o projeto, como a criação de testes unitários e documentação. Faremos então uma pequena abordagem a cada uma das medidas:

3.1. Testes Unitários

Com o objetivo de desde cedo encontrar os erros que poderiam escalar, implementamos testes unitários às queries, através dos *datasets* fornecidos. Isto é para cada query corremos os seus exemplos fornecidos e verificamos se o output coincide com o esperado. Tudo isto pode ser acedido através do comando *make test*.

A mesma abordagem foi tomada para os memory leaks, que facilmente, através do comando *make check-memory*, podem ser prevenidos, e para formatar o código, com *clang-format*.

3.2. GitHub Actions e Review

Para prevenir que a main alguma vez contivesse erros, usamos o poder das GitHub Actions juntamente com os comandos do make. Assim, estipulamos que sempre que quiséssemos dar merge, era necessário que todos os quatro testes (compilação bem sucedida, boa formatação, testes unitários e testes de memory leaks) passassem. Para além disso tivemos o cuidado de ter o review de pelo menos um dos colegas como um requisito para o merge.

3.3. Documentação

Por fim, escrevemos uma documentação para todas as funções que estão expostas nos ficheiros de *headers* para que o trabalho em grupo fosse facilitado, estando assim sempre por dentro de como usar os outros módulos e que parâmetros estão eles à espera, bem como o que vão retornar. A documentação pode ser exportada, através do *Doxygen*, com o comando *make doxygen*.

4. Funcionalidades Implementadas

Aqui, abordar-se-ão, as **Funcionalidades Implementadas,** que, na sua essência, são as queries.

Assim, cada uma dessas funcionalidades está associada a uma query específica, acompanhada de funções auxiliares e podem ser explicadas da seguinte forma (apenas as queries implementadas nesta fase estão explícitas):

4.1. (Q1) Listar informações de um Utilizador, Voo ou Reserva

A função principal `query_1` recebe um identificador único e busca informações associadas nos dicionários de utilizadores, voos e reservas. Cada função de verificação (`is_user`, `is_flight`, `is_reservation`) valida a presença do identificador correspondente. As funções de escrita (`write_user_data`, `write_flight_data`, `write_reservation_data`) formatam e gravam as informações relevantes do utilizador, voo ou reserva num ficheiro de saída recém-criado, organizando os dados para facilitar a leitura e interpretação.

4.2. (Q2) Listar Voos ou Reservas de um Utilizador

`get_user_reservations_and_flights` procura e organiza as reservas e voos de um utilizador numa lista encadeada `user_reservations_and_flights`, tendo em conta o ID fornecido. A função `compare_dates` é um comparador utilizado para ordenar a lista encadeada por datas de início, priorizando reservas e voos.

A função principal `query_2` começa por verificar a existência do utilizador e o estado da conta. Em seguida, obtém e ordena as reservas e voos do utilizador. Posteriormente, itera através da lista resultante, escrevendo informações específicas no ficheiro de saída, dependendo de um argumento opcional.

4.3. (Q3) Apresentar Classificação Média de um Hotel

A função `calculate_average_rating` calcula a média das avaliações para um hotel específico identificado pelo hotel_ID. Para isso, percorre as reservas no catálogo, somando as avaliações e contando o número total de avaliações para o hotel em questão.

A função `query_3` cria um ficheiro de saída e calcula a média das avaliações para o hotel com o ID fornecido. Em seguida, formata esse valor como uma string e o escreve no ficheiro de saída. Por fim, o ficheiro é fechado e a função retorna.

4.4. (Q4) Listar Reservas de um Hotel

`compare_reservations_order` é um comparador utilizado para ordenar as reservas por data de início, priorizando a ordem decrescente. Se as datas de início forem iguais, as reservas são ordenadas por ID, em ordem crescente.

`get_Reservations_By_Hotel_Id` obtém uma lista encadeada de reservas de um determinado hotel a partir de um dicionário de reservas fornecido.

A função principal `query_4` obtém as reservas para um hotel específico usando `get_Reservations_By_Hotel_Id`, ordena essas reservas pela função de comparação `compare_reservations_order` e as escreve em um ficheiro de saída. Cada reserva é formatada como um conjunto de informações (ID, datas de início e fim, ID do utilizador, avaliação e preço total) e é gravada no ficheiro.

4.5. (Q5) Listar Voos de um Aeroporto entre Datas

`compare_flights_order` é um comparador utilizado para ordenar os voos por data de partida, priorizando a ordem decrescente. Se as datas de partida forem iguais, os voos são ordenados por ID, em ordem crescente.

A função `get_flights_by_airport_and_time` obtém uma lista encadeada de voos a partir de um dicionário de voos, filtrando-os por aeroporto de origem e um intervalo de datas específico.

`query_5` utiliza `get_flights_by_airport_and_time` para obter uma lista de voos que correspondem ao aeroporto de origem e ao intervalo de datas fornecidos. Em seguida, ordena esses voos pela função de comparação `compare_flights_order` e os escreve em um ficheiro de saída. Cada voo é formatado como um conjunto de informações (ID, data de partida, destino, companhia aérea e modelo da aeronave) e é gravado no ficheiro.

4.6. (Q8) Apresentar Receita Total de um Hotel entre Datas

A função `get_days_difference_inside_range` calcula a diferença de dias entre dois intervalos de datas dentro de um intervalo maior. Determina quantos dias estão dentro do intervalo especificado pelo `begin_date` e `end_date` e também pelos parâmetros `date1` e `date2`.

`get_hotel_total_revenue` calcula a receita total de um hotel a partir de um dicionário de reservas. Percorre todas as reservas no dicionário, verificando se cada reserva pertence ao hotel especificado pelo `hotel_id` e se está dentro do intervalo de datas fornecido. Para as reservas que atendem a esses critérios, ela calcula o valor total da reserva, considerando o preço por noite e a duração da estadia.

A função principal `query_8` utiliza `get_hotel_total_revenue` para obter a receita total de um hotel específico no intervalo de datas fornecido. O resultado é formatado como um conjunto de informações contendo apenas o valor total da receita, e é escrito em um ficheiro de saída.

Essa implementação permite calcular a receita total de um hotel com base nas reservas dentro de um intervalo de datas específico em um sistema de gestão de reservas de hotel.

4.7.(Q9) Listar Utilizadores por Prefixo de Nome

Lista todos os utilizadores cujo nome começa com um determinado prefixo, ordenando-os por nome.

`getUsersWithPrefix` recebe uma tabela de utilizadores (`user_table`) e um prefixo como entrada. Percorre a tabela de utilizadores, identifica aqueles com um nome que começa com o prefixo fornecido e adiciona-os a uma lista encadeada `result`, desde que o estado da conta seja verdadeiro.

A função `compare_users_order` é um comparador utilizado pela função `g_list_sort` para ordenar os utilizadores. Usa `strcoll` para comparar os nomes dos utilizadores. Se dois utilizadores tiverem o mesmo nome, usa o ID para desempate.

`query_9` utiliza as funções anteriores para encontrar os utilizadores com o prefixo fornecido, ordena-os e escreve os IDs e nomes correspondentes num ficheiro de saída.

5. Conclusão

Assim, para concluir, reforçamos que fase atual do projeto revelou uma estrutura sólida e modular, permitindo a gestão eficiente de funcionalidades distintas da aplicação, mas prevemos incorporar o sugerido encapsulamento durante a próxima fase, sendo que isto irá fortalecer ainda mais a estrutura do código.

Durante esta fase, enfrentamos alguns desafios. Embora a equipa tenha demonstrado coesão e colaboração, houve disparidades em termos de habilidades de programação entre os membros, e pensamos que esse tenha sido o mais significativo. Assim uma das dificuldades encontradas foi a disparidade de conhecimento técnico, onde um de nós não se sentia tão à vontade com a linguagem C e não estava tão alinhado com o conhecimento técnico dos restantes. Apesar disso, todos demonstramos empenho e dedicação, contribuindo de forma proativa para o progresso do projeto, o que se espera também que ocorra durante a próxima fase.