Projeto Laboratórios de Informática III

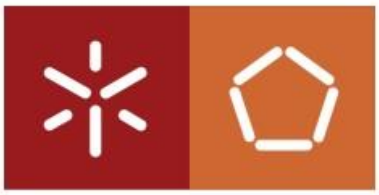
Fase 1

Projeto desenvolvido por:

Rafael Lopes Seara (A104094), Sara Catarina Loureiro da Silva (A104608) e Zita Magalhães Duarte (A104268)

Grupo 80

Licenciatura em Engenharia Informática



Universidade do Minho
Escola de Engenharia

Departamento de Informática

Universidade do Minho

Índice

Capítulo 1

Introdução

Foi-nos proposto, na unidade curricular de Laboratórios de Informática III, do ano 2023/2024, o desenvolvimento deste projeto, em que temos de implementar uma base de dados em memória que armazene dados fornecidos pelos docentes. Com este trabalho pretende-se desenvolver capacidades no que toca aos princípios fundamentais da Engenharia de Software - designadamente modularidade, reutilização, encapsulamento e abstração de dados.
O projeto está dividido em duas fases, e esta primeira fase consiste em implementar o *parsing* dos dados e o modo *batch*. Este modo consiste em executar várias *queries* sobre os dados de forma sequencial, estando esses pedidos armazenados num ficheiro de texto cujo caminho é recebido como argumento.

Assim sendo, este processo requer, não só a leitura e interpretação dos dados dos ficheiros, como o seu armazenamento em estruturas de dados versáteis e de rápido acesso.

Uma das vantagens deste projeto é termos acesso à *glib*- uma biblioteca que já possui implementações de várias estruturas úteis. Este grau de simplicidade e abstração convenceu-nos a usar esta biblioteca ao seu máximo potencial.

Capítulo 2

Desenvolvimento

Para uma maior facilidade na execução e organização do código, o nosso grupo começou por organizar e planear. Assim, conseguimos uma visão mais clara daquilo que temos e queremos fazer, para que consigamos evitar ter de refazer grandes partes de código devido a implementações ineficientes ou menos práticas. Começamos por imaginar uma arquitetura de como seria a ligação entre os diferentes módulos e a forma como iriam comunicar entre si. Tendo em consideração o mencionado, chegamos à seguinte arquitetura:

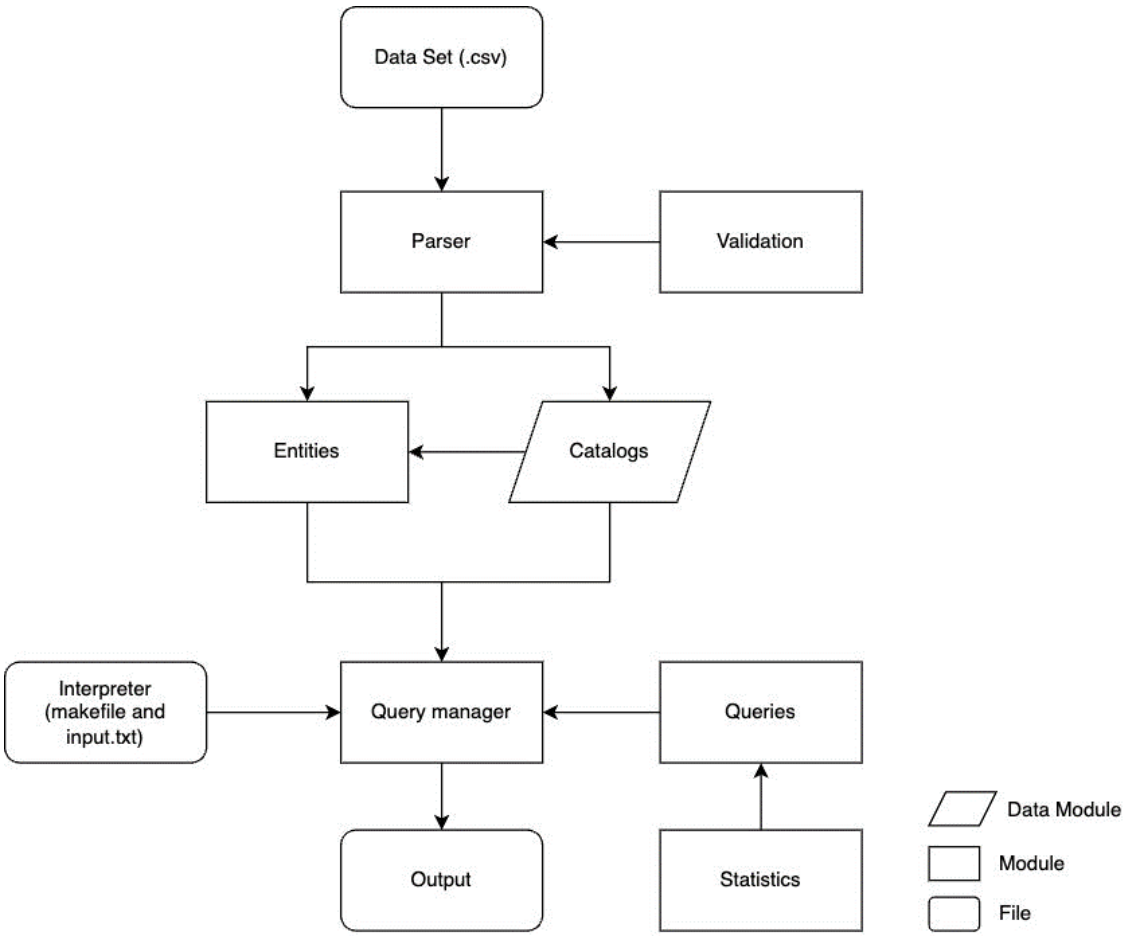


Figura 2.1: Diagrama da estrutura utilizada para o projeto

Em primeiro lugar, fazemos o *parser* dos ficheiros *.csv*, tal como a verificação dos *users*, dos *flights*, das *reservations*, e dos *passengers*. Nesta fase também verificamos se as *reservations* são válidas através do catálogo dos *users* e se os passengers são válidos através do catálogo *users* e do catálogo *flights*. Neste processo também adicionamos, aos catálogos previamente iniciados, as *entities* necessárias.

Outra implementação criada foram os *setters* e os *getters* para cada uma das *entities*, tal como funções adicionais para uso dos catálogos.

Finalmente, de forma a finalizar o solicitado nesta primeira fase, optamos por escolher responder às *queries* Q1, Q3, Q4, Q8 e Q9. Isto, pois, no nosso ponto de vista, eram aquelas cuja implementação era mais simples.

Temos também um modulo de estatísticas usado nas *queries* para ajudar a modularidade do código.

Assim, passamos a dar uma breve explicação de cada uma das *queries* e uma a forma como acabamos por abordar a sua execução:

Query1: A *query* 1 consiste em listar o resumo de um utilizador, voo, ou reserva, consoante o identificador recebido por argumento.

Assim, e tendo em atenção ao que nos é pedido, fizemos uma função que calcula o preço da estadia, uma que calcula o número de noites da estadia, uma que calcula o atraso do voo, uma que conta o número de passageiros no voo e a que executa a *query*.

Quando o argumento recebido é um *user_id,* começamos por verificar se o *user_id* recebido está na tabela. Se o *user_id* está na tabela, vamos buscar o nome do *user*, bem como o seu sexo, a data de nascimento, o código do país e o número do passaporte. Para além disso, criamos uma função que calcula a idade do *user*. De seguido, declaramos toda esta informação.

Por outro lado, quando o argumento recebido é um *reservation_id*, começamos por verificar se o *reservation_id* recebido está na tabela. Se o *reservation_id* está inserido na tabela, vamos buscar o nome do hotel, o id do hotel, a data de começo da reserva, bem como o seu fim. Para além disso, vamos ver se a reserva inclui pequeno-almoço e vamos buscar também o preço de cada noite e o *city tax.* Com esta informação criamos duas funções que calculam quer o preço da estadia e o número de noites. Depois de retirarmos toda esta informação, declaramo-la.

Em última instância, quando o argumento recebido é um *flight_id*, vamos verificar se o *flight_id* é diferente de zero e se este *flight_id* está presente na tabela. Se o *flight_id* está inserido na tabela, vamos buscar a *airline*, bem como o modelo do voo, a sua origem, a data esperada de partida, a data esperada de regresso, a data real de partida, a data esperada de chegada. Para além disso, vamos buscar o piloto e o copiloto.
Com os dados que retiramos, calculamos o número de passageiros desse *flight* bem como o atraso associado ao voo.

Query3: A *query* 3 consiste em apresentar a classificação média de um hotel, a partir do seu identificador.

A função *exec_query_3* recebe como parâmetros um identificador de um hotel, *hotel_id*, e um catálogo de reservas, *Reservations Catalog*. Para além disso, criamos um acumulador chamado de *number_of_reservations*, para contarmos quantas vezes encontramos o *hotel_id* desejado, iniciamos a variável *rating* a zero e acumulamos os

vários *ratings*, somando-os. De seguida, iniciamos um *loop* para percorrer todas as reservas do catálogo e verificarmos se o identificador do hotel da reserva é igual ao *hotel_id* fornecido. Quando o é, obtemos a classificação da reserva e acumulamos a classificação da reserva. Depois de iterar ao longo de toda a tabela e não encontrar mais nenhum *hotel_id* igual ao que recebemos como argumento, verificamos se há pelo menos uma reserva, através do *number_of_reservations* para evitar a divisão por zero, e se tal acontecer, calculamos a média, fazendo uma divisão entre o *rating* e o *number_of_reservations*.

Query4: Listar as reservas de um hotel, ordenadas por data de início (da mais recente para a mais antiga).

Para isso, fizemos uma função que compara as datas iniciais de duas reservas e ordena-as de forma crescente. Essa função recebe dois ponteiros *const void** que apontam para as reservas a serem comparadas, converte esses ponteiros para ponteiros do tipo RESERVATION para que possam ser utilizados e obtém as datas de início das reservas (por meio da função *getBeginDate*). As reservas são ordenadas primeiro pela data de início em ordem decrescente e, em caso de empate, pelo identificador da reserva em ordem crescente.

Posteriormente, implementamos a função que executa a *query*. Esta recebe o ID de um hotel, um *array* de estruturas *reservation*, que representa o catálogo de reservas e o tamanho desse catálogo, filtra as reservas que pertencem ao hotel específico com base no ID fornecido, usa um *loop* para percorrer o catálogo de reservas e copiar as reservas correspondentes ao hotel filtrado para um novo *array* chamado *filtered_reservations* e ordena esse novo *array* de reservas filtradas utilizando a função *qsort*, passando como argumento a função de comparação *compare_reservations*. Por fim, imprime os detalhes das reservas filtradas e ordenadas no console, usando informações como ID da reserva, datas de início e fim, ID do usuário, avaliação, preço total, nome e detalhes do hotel, entre outros.

Query8: A *query* 8 consiste em apresentar a receita total de um hotel entre duas datas (inclusive), a partir do seu identificador.

Para isso, criamos duas funções que nos auxiliaram na execução da *query*. A primeira converte a data para um número que representa a quantidade de dias desde uma referência fixa que, neste caso, é uma simplificação para fins de cálculo. Ela recebe uma data no formato "ano/mês/dia" e extrai o ano, mês e dia da *string* da data. A segunda recebe uma *RESERVATION*, uma data inicial e uma data final como parâmetros, obtém as datas de início e fim da reserva e o preço por noite. Seguidamente, verifica se as datas da reserva estão dentro do intervalo fornecido e converte-as para um formato numérico para facilitar a comparação, calcula o número de noites entre as datas de início e fim dentro do intervalo especificado e multiplica o número de noites pelo preço por noite para obter a receita daquela reserva dentro do intervalo. Para terminar, retorna o valor da receita.

A função que executa a *query* 8 recebe um *hash table (GHashTable)* de reservas, uma *string* de parâmetros e um inteiro que representado a linha. Esta função separa os parâmetros fornecidos (ID do hotel, data inicial e data final), calcula um caminho para criar um arquivo de saída para armazenar o resultado da *query*, inicializa um contador (receita) para calcular a receita total do hotel e itera sobre o *hash table* de reservas, procurando reservas associadas ao ID do hotel. Para cada reserva encontrada, utiliza a função ***busca_receita*** para calcular a receita considerando as datas fornecidas e soma o valor retornado pela função ***busca_receita*** à receita e, por fim, escreve o total da receita no arquivo de saída.

? f dá um formato de output diferente

Query9: Listar todos os utilizadores cujo nome começa com o prefixo passado pelo argumento, ordenados por nome (de forma crescente).

Fizemos uma função de comparação que é utilizada para ordenar as informações de usuário. Ela recebe dois ponteiros *gconstpointer* que apontam para estruturas e converte-os para *USER_INFO*(Estrutura usada para guardar os usernames e ids). De seguida compara os *usernames* dos usuários utilizando a *g_utf8_collate*. Se os *usernames* forem iguais, compara os *ID*s dos usuários e retorna um valor inteiro que representa a relação de ordenação entre os dois usuários para os ordenar de forma crescente.

De seguida, implementamos uma função que verifica se o utilizador possui um prefixo específico. Recebe um utilizador (*USER*) e uma *string* como argumentos, obtém o nome do usuário através da função *getName* e compara os primeiros caracteres do nome do utilizador com a *string* fornecida. No fim, retorna 1 se o prefixo estiver presente no nome do utilizador, caso contrário, retorna 0.

Por fim, implementamos a função que executa a *query*, que recebe um catálogo de usuários (USERS_CATALOG), uma *string* de argumentos (*args*) e o nome do arquivo de saída (output), abre um arquivo de saída especificado em modo de escrita. Seguidamente, inicializa um iterador para percorrer o catálogo de usuários e cria um *array* (*userArray*) para armazenar informações sobre os usuários que atendem aos critérios estabelecidos.

Esta função percorre o catálogo de usuários:

Para cada usuário ativo que tenha um nome com um prefixo correspondente aos argumentos fornecidos, cria uma estrutura *USER_INFO*, contendo o nome de usuário e o ID do usuário, e a adiciona ao *userArray*.

Ordena o *userArray* utilizando a função *compare_user_info* e escreve as informações dos usuários ordenados no arquivo de saída, no formato *user_id;username*. Para terminar, fecha o arquivo de saída e liberta a memória alocada para o *userArray*.

Capítulo 3
Dificuldades Sentidas

Relativamente às dificuldades sentidas, podemos afirmar que foram sentidas diversas dificuldades, inicialmente a nível de compreensão e, de seguida, ao nível de implementação de código. Para além disso, enfrentamos algumas complicações ao nível da divisão de trabalho e, consequentemente, em relação às colisões associadas ao GitHub.

Capítulo X
Conclusão

Apesar do referido no capítulo anterior e de não termos conseguido alcançar todos os objetivos propostos, pensamos que não ficamos muito aquém das expectativas e conseguimos consolidar os nossos conhecimentos de C com conhecimentos mais avançados relacionados a performance, gestão de memória, encapsulamento e implementação de *HashTables*, o que, para nós, é o mais importante.

Esperamos, na próxima fase, conseguir melhorar os aspetos menos positivos desta primeira fase e realizar com sucesso os próximos desafios.