Universidade do Minho
Escola de Engenharia

# Relatório - Fase 2
# Laboratórios de Informática III


Martim Redondo
A100664

Rodrigo Castro
A100694

Tiago Moreira
A100541


18 de janeiro, 2024 - Grupo 95

# Conteúdo

# 1 Introdução

Apesar de já ter sido apresentado um relatório sobre a primeira parte do projeto, este relatório irá descrever o projeto como um todo, incluindo a segunda parte. Isso ocorre porque o relatório final deverá fazer comparações entre as duas partes e explicar com detalhes o código e as ideias por trás dele.

Em resumo, este relatório irá explicar o que foi feito de novo para a segunda fase, bem como as mudanças feitas na primeira fase e os novos temas que foram abordados.

# 2 Modelação

## 2.1 Arquitetura de Aplicação

A arquitetura do nosso projeto é constituída principalmente por quatro componentes principais, sendo elas:

- **Parse**, responsável tanto pela leitura do ficheiro, como do tratamento de dados e a sua inserção de forma válida nas HashTables. Este ficheiro utiliza alguns outros ficheiros auxiliares (parseFicheiroFlights, parseFicheiroPassagers, parseFicheiroReservations, parseFicheiroUsers), estes ainda têm acesso aos ficheiros das HashTables específicas (exemplo: hotel).

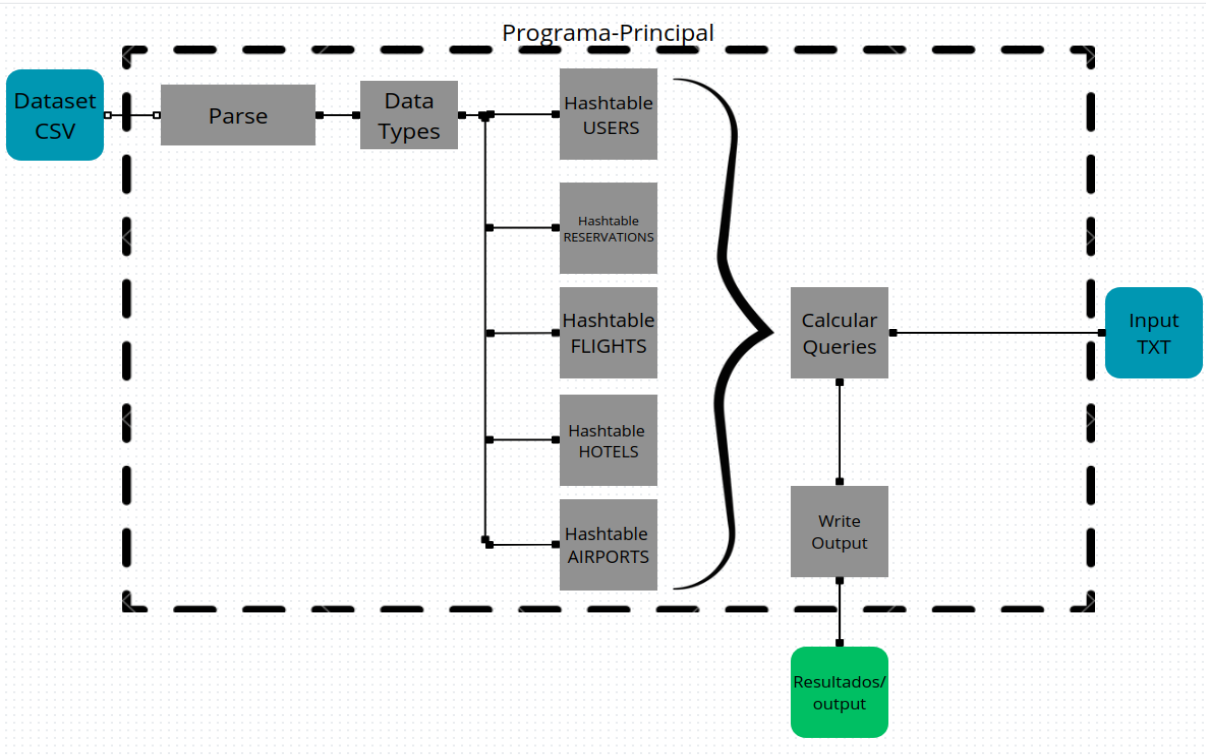
Algumas *mudanças* foram feitas desde a fase 1. A sugestão dos professores foi criado um **catálogo**, onde estariam as HashTables guardadas. Ainda,no parse, houve a preocupação da organização da HashTable aeroportos tendo em conta alguns parâmetros, o motivo para tal implementação é para depois haver uma maior facilidade na implementação das queries.

- **CalcularQueries**, responsável por ler o input do utilizador e perceber o que o programa deve fazer com aquela informação. A única mudança feita é o facto de em vez de esta função receber as HashTables todas, recebe o **catálogo** onde elas estão guardadas.

- **WriteOutPut** é a função responsável por executar a query certa (tem acesso a todas as queries implementadas).

A partir da fase 1, foram implementadas algumas *mudanças* com o objetivo de melhorar a **modularização** e a **reutilização de código**. Essas mudanças são evidentes em alguns aspectos da nossa implementação, como a remoção da função WriteOutPut para cada query. Antes, cada query era responsável por escrever o seu próprio resultado no arquivo. Agora, a consulta retorna o resultado em formato de **Glist** e a função WriteOutPut usa uma nova função, **escreveFicheiro**, para escrever o resultado no arquivo.

-**Menu** é a classe onde foi construído o menu iterativo. Para o utilizador conseguir ter acesso a este menu é preciso que ele dê, unicamente, o caminho para o ficheiros .csv.


Programa-Principal
Dataset CSV
Parse
Data Types
Hashtable USERS
Hashtable RESERVATIONS
Hashtable FLIGHTS
Hashtable HOTELS
Hashtable AIRPORTS
Calcular Queries
Input TXT
Write Output
Resultados/ output

## 2.2 Organização dos Dados

### 2.2.1 Estrutura de Dados

Todas as nossas estruturas de dados principais são HashTables, todas estão a ser criadas e guardadas no nosso **Catálogo** que se encontra no *utils.c*, porêm todas são as informações são devidamente tratadas num ficheiro parse específico para cada HashTable (exemplo: parseFicheiroFlights).

Ao todo foram criadas 5 HashTables, sendo elas, **HashTable users**, **HashTalbe voos**, **HashTable reservations**, **HashTable tablehotel** e **HashTable aeroportos**.

Fora as HashTables existem, também, as subestruturas que compõem as estruturas referidas em cima. Passo a citá-las:

-**Users**, guarda os seguintes dados de um user: id, nome, email, telemóvel, data de nascimento, sexo, número de passaporte, código do País de origem, data da criação da conta e status da conta (ativa/inativa).Guarda também uma lista de voos em que este já fez parte e também das reservas.

-**Flights**, guarda os seguintes dados de um voo: id, companhia aérea, modelo do avião, número de lugares, origem, destino, data marcada de saída, data marcada de chegada, data real de saída, data real de chegada, nome do piloto, nome do copiloto e número de passageiros, que é incrementado cada vez que há uma entrada válida no passengers.csv com este voo.

-**Reservation**, guarda os seguintes dados de uma reserva: id da reserva, id do user, id do hotel, nome do hotel, estrelas do hotel, imposto da cidade, data de início da estadia, data de fim da estadia, preço por noite, inclui pequeno-almoço, e avalição do user.

-**Hotel**, guarda todos os dados referentes aos hotéis: id do hotel, nome do hotel, preço por noite, imposto da cidade, número de avaliações, avaliação total, e uma lista de todas as suas reservas.

-**Aeroporto**, guarda todos os dados referentes aos aeroportos: nome do aeroporto, lista dos voos atrasados e lista de voos.

**Nota importante:** como forma de melhorar o encapsulamento as estruturas foram passadas para os seus respetivos *.c*, ficando no *.h*, unicamente, a chamada da estrutura. Algo que tinha sido ignorado por nós na primeira fase.

# 3 Implementação

## 3.1 Nota

O output de todas as queries tem duas formas de ser impreso, de uma maneira formata, ou não formatado.

Todas as querys passaram a devolver GList em vez de void, pois, na primeira fase, cada query era responsável por escrever a resposta que calculou no ficheiro.

## 3.2 Query1

A query1 pode usar a informação do input de forma diferente, dependendo do seu tipo. Na primeira fase, haviam sido implementadas três funções diferentes para escrever no arquivo, dependendo do tipo de input. No entanto, agora, temos uma função principal que leva o input para uma das funções auxiliares correspondentes, com a respetiva HashTable. Cada função auxiliar retorna uma Glist, que é usada pela função WriteOutPut para escrever no arquivo. Isso permite haja o acesse ao arquivo apenas uma vez e melhor a modularidade. Esta query é extremamente rápida, uma vez que é só preciso dar lookup na HashTable e calcular pouca informação.

## 3.3 Query2

Como a 1ª query, esta devolve informação diferente dependendo do input. No caso de apresentar só reservas, esta percorre a lista de reservas de um user, ordenando-a, e de seguida retira a informação. Na apresentação só de voos, o processo é o mesmo, mas como o user só guarda o ID dos voos, temos de criar uma lista com os Voos em si. Já quando é pedido a informação de voos e reservas, fazemos o que se faria nos dois outros casos mas juntamos a informação numa struct que diz se é voo ou reserva e a sua data, toda numa GList. Depois ordenamos a lista conforme os requisitos dados no enunciado. Como é de esperar este método é o mais lento dos 3, sendo o mais rápido o das reservas.

## 3.4 Query3

Esta query recebe a hashtable de hotéis e usa o ID do hotel para aceder o hotel correto na hashtable. Esta query foi resolvida de forma simples, pois a estrutura de dados está organizada de uma maneira que facilita o cálculo da média das avaliações. Uma vez que é só preciso dar lookup, esta é a query mais rápida do nosso programa.

## 3.5 Query4

Esta query recebe a hashtable de hotéis e usa o ID do hotel para acessar todas as informações do hotel na hashtable. Em seguida, essas informações são organizadas de acordo com os parâmetros do enunciado. Como temos de percorrer a lista toda para organizar, e mais uma vez para retornar o resultaado, isto acaba por afetar o tempo de execução.

## 3.6 Query5

Esta query recebe sempre a HashTable de aeroportos. Primeiramente, o programa usa a informação que já está disponibilizada na struct do aeroporto e cria uma lista de voos, esta lista é depois ordenada por data de voo mais antigo para voo mais recente. Apesar de seguir o mesmo pensamento da Query 4, esta é mais rápida uma vez que um aeroporto tem menos voos, do que um hotel tem reservas.

## 3.7 Query6

Para auxiliar a resolução da query 6, foi criada uma estrutura auxiliar, **nomeEpassageiros**, com os dois elementos necessários na saída da função: o nome do aeroporto e o número de passageiros. A query recebe uma tabela hash de aeroportos e os parâmetros de entrada. Ainda nesta query, uma lista vazia, nomeEpassageiros, é preenchida com elementos da HashTable que correspondem aos parâmetros. No entanto, essa lista precisa ser ordenada de acordo com os critérios do enunciado. Para isso, é usada a função *organizarPorQuemTemMais*. Apesar de iterar por todos os voos de todos os aeroportos, ela não chega a afetar muito a performance uma vez que o número de voos não é grande suficiente, se isto mudasse, a performance iria ser severamente afetada.

## 3.8 Query7

Como em outras querys, esta utiliza uma estrutura auxiliar, **nomeEatrasos**, para armazenar dados sobre os aeroportos e seus atrasos. A estrutura é preenchida com dados da tabela hash de aeroportos, de acordo com os parâmetros da query. Em seguida, a lista é ordenada de acordo com a mediana dos atrasos, com os aeroportos com maior mediana aparecendo primeiro. Em caso de empate, a lista é ordenada pelo nome dos aeroportos. Como na Q6, a performance depende da quantidade de voos, esta query tem um tempo médio de execução próximo da Q6.

## 3.9 Query8

Esta query recebe, exclusivamente, a HashTable hotel. O programa obtem o hotel correto, através de um look up, e como a struct hotel guarda a lista de reservas, este só precisa de correr cada reserva e ir adicionando a receita de todas elas, o número de noites é calculado a partir das datas especificadas. A performance desta query é dependente do número de reservas que um hotel tém, uma vez que a lista de reservas é iterada para calcular a receita total.

## 3.10 Query9

Esta query recebe a HashTable users e vai guardar numa Glist todos os users que tenham o prefixo dado no input, depois disso organiza a Glist por nome de user, e se for preciso desempata através do uso do ID. O desempenho desta query depende do número de utilizadores registados ativos e também quantos destes têm um nome com o prefixo dado. Uma vez que iteração por listas acontece 3 vezes, sendo uma delas para fornecer o resultado da query, esta acaba por ter a sua performance afetada.

## 3.11 Query10

Não implementada.

# 4 Desempenho e Explicações

Iremos apresentar uma tabela onde serão apresentados os dados sobre os testes. É de ter em conta que os resultados que constam da tabela é a média de três testes consecutivos. Ou seja, em cada máquina o código foi testado três vezes e a média desses resultados é a que consta na tabela, tal decisão foi tomada para evitar ao máximo qualquer desvio. Quanto às maquinas usadas:

**LeNovo**: AMD Ryzen 7 4800U 1.80 GHz e 8 GB de RAM

**MSI**: Intel I7-11800H @ 2.30 GHz e 16 GB de RAM

**MAC**: Apple M1 Max e 64 GB de RAM (8 núcleos de desempenho e 2 núcleos de eficiência)

| | LeNovo | MSI | MAC |
|---|---|---|---|
| Parse | 39.533774 | 37.32481 | 123.482311 |
| Q1 | 0.000091 | 0.000087 | 0.000162 |
| Q2 | 0.000097 | 0.000094 | 0.000493 |
| Q3 | 0.000070 | 0.000063 | 0.000111 |
| Q4 | 9.047006 | 8.974910 | 21.556163 |
| Q5 | 0.004710 | 0.004703 | 0.084622 |
| Q6 | 0.025851 | 0.021924 | 0.027174 |
| Q7 | 0.022881 | 0.021712 | 0.026549 |
| Q8 | 0.027575 | 0.027573 | 0.818930 |
| Q9 | 0.243093 | 0.239417 | 0.275878 |
| Programa(tempo) | 113.384553 | 109.30461 | 305.373506 |
| Programa(memória) | 2432 KB | 2432 KB | 5292032 KB |

Figura 1: Tabela de Testes

A grande disparidade do MAC em relação às outras máquinas não está relacionada ao projeto, mas sim aos métodos que o usuário do MAC teve que adotar para conseguir compilar o projeto.

# 5 Memory Leaks

Tendo em conta, unicamente, à primeira fase, o programa, de acordo com os testes da plataforma de LI3, contém, apenas, 0.01884 MB de memory leak, visto ser o espaço ocupado pela biblioteca GLIB-2.0, sendo impossível reduzir mais o número de MB de memory leak.

# 6 Porquê de algumas decisões (ainda não explicadas)

Uma das mudanças simples que influenciou significativamente o nosso desempenho foi a mudança da função **glist append** para a função **glist prepend**. A complexidade de **glist append** é *O(n)*, enquanto a complexidade de **glist prepend** é *O(1)*. Na primeira fase, essa diferença não foi significativa, pois a base de dados era pequena. No entanto, na segunda fase, a diferença foi grande o suficiente para que a mudança fosse suficiente para colocar o código a funcionar.

# 7 Conclusão

Na conclusão gostariamos de realçar os pontos fortes do nosso trabalho, como a atenção máxima para o encapsulamento e modularidade, tal como para a redução dos memory leaks. Sabemos que o projeto ainda tem espaço para diversas melhorias, pois não implementamos a query10 e a query8 não está totalmente correta, por exemplo, contudo sentimos que fizemos um bom trabalho na cadeira de LI3 e apresentamos um trabalho bastante satisfatório.