Trabalho prático - LI3

Relatório da 2ª fase

Ana Sá Oliveira, a104437
Inês Silva Marques, a104263
José Rafael De Oliveira Vilas Boas, a76350

LEI - 2023/2024

Índice

Arquitetura do projeto

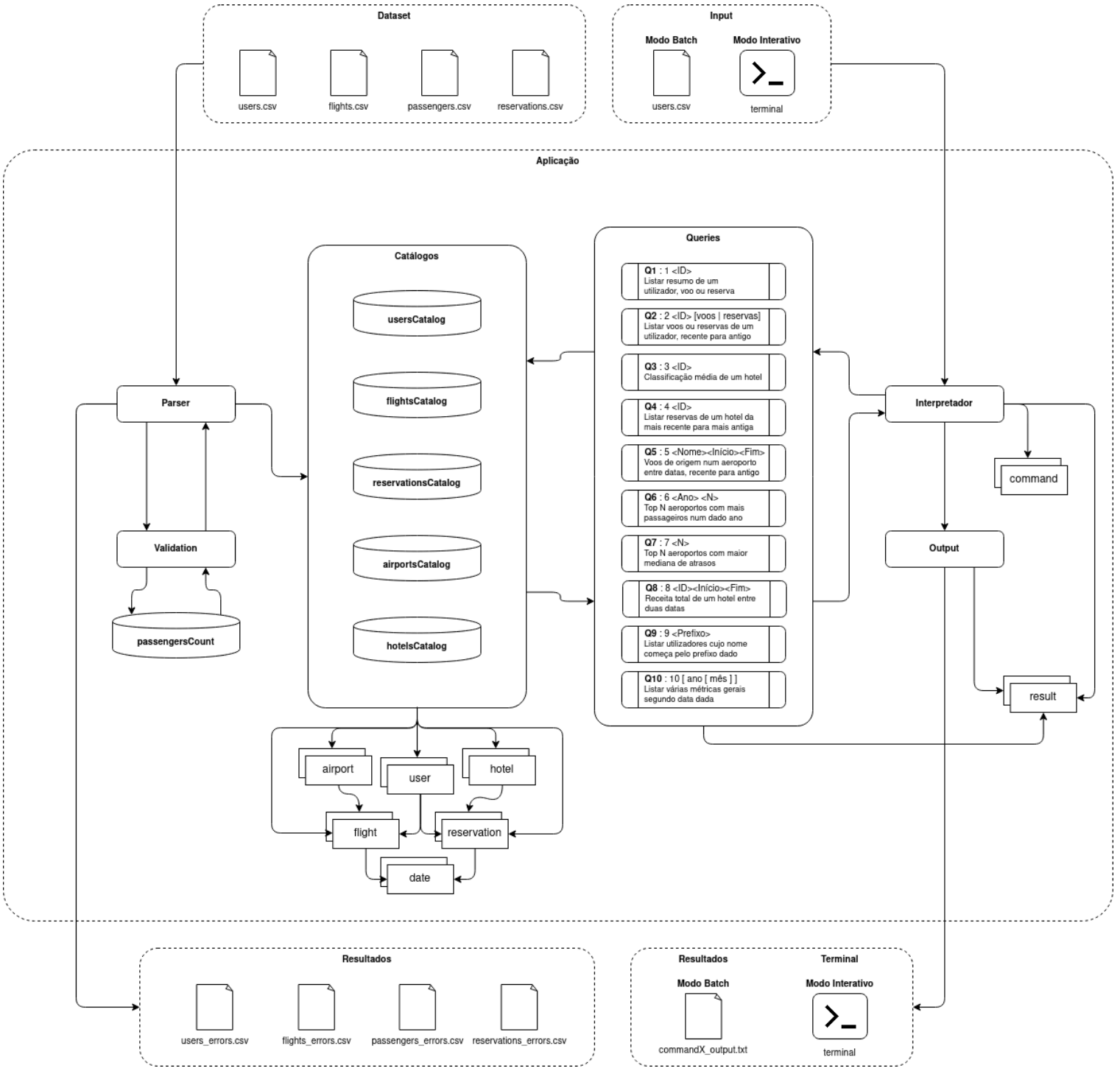


Figura 1 - esquema da arquitetura do programa

Tipos e estruturas de dados

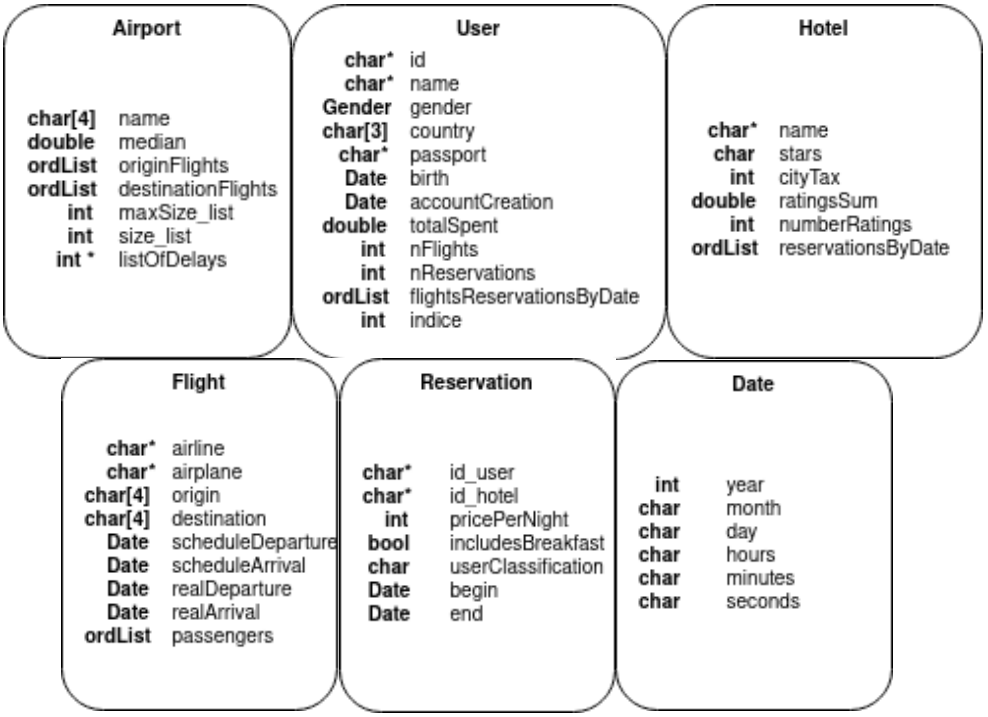


Figura 2 - estruturas de dados

Catálogos de dados

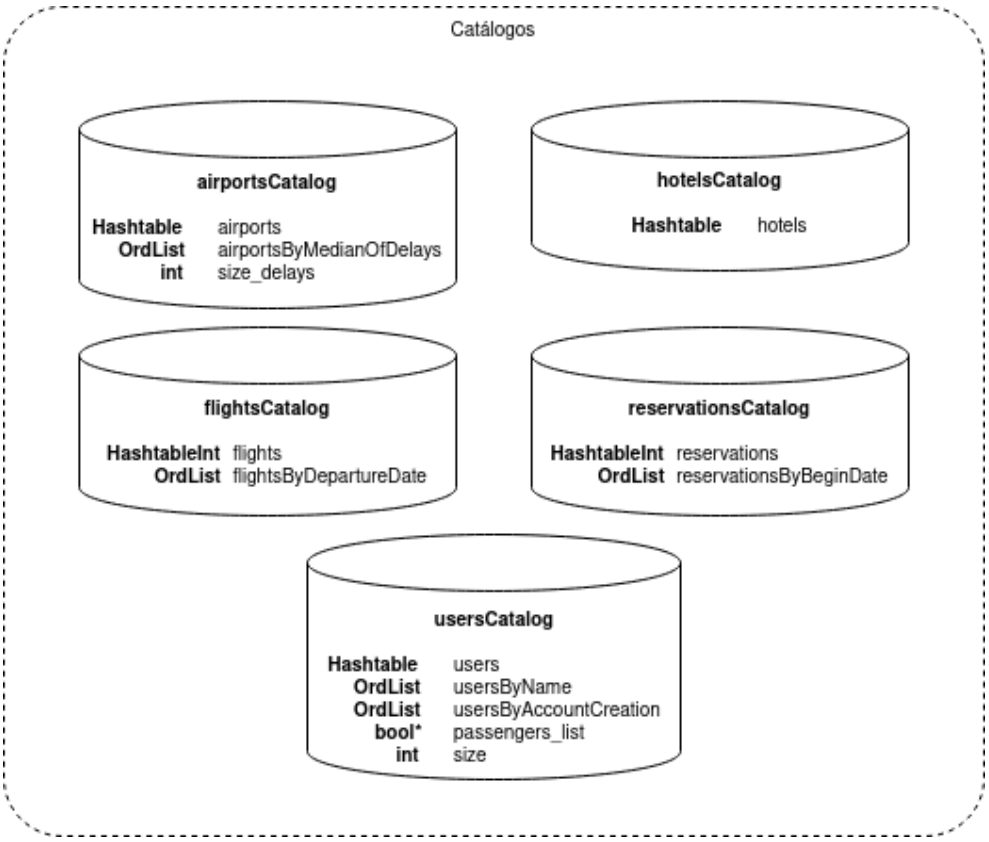


Figura 3 - catálogos de dados

Encapsulamento e Modularidade

Nesta fase do projeto, tínhamos que encapsular tudo aquilo que faltava. Nas funções gets, que retornassem uma string (que já tínhamos construído), passamos a usar a função strdup para duplicar a string, de modo a não dar acesso à string original. Em muitas identidades, tínhamos apontadores para outras identidades de outros catálogos, o que correspondia a uma quebra no encapsulamento. Assim, substituímos estes apontadores por ids das identidades que queríamos guardar ali. Nas queries, tentamos ir buscar as informações que necessitamos a cada catálogo, ou trazer do catálogo até as queries. Todas as identidades e estruturas de dados tornamos opacas. Para além disso, continuamos a dividir o programa em diferentes módulos, por exemplo, adicionamos o módulo interativo, o módulo aerport, o módulo aerportsManager, etc...

Parser

No parser, transformamos 4 funções que faziam parsing de 4 ficheiros diferentes em apenas 1 função que consegue fazer o parsing de todos os tipos de ficheiros. Paramos de copiar linhas (para ter a linha completa caso a entidade fosse inválida) e passamos a usar a função fseek se precisássemos de ler a linha novamente. Paramos de alocar memória e copiar string literals e passamos a usá-las diretamente ou com o sprintf. Também paramos de usar funções que abriam e fechavam constantemente o ficheiro de erros, e simplesmente passamos a abrir e fechar uma vez o ficheiro de erros na função parse_file, o que melhorou o tempo de execução do programa.

Interpretador de comandos

Utilização do novo tipo dos resultados e correção no parsing que assumia que o id de uma query só tinha um caracter - o que não acontece na query 10.

Output

No módulo de output foram feitas mudanças com o objetivo de generalizar as funções, não tendo assim que ter uma função que imprime o output de cada query. Para isso foi criado um tipo QueryResult, que armazena tanto os resultados calculados como o nome do campo a que se referem, para que os resultados possam ser escritos diretamente, independentemente de qual query foi executada.

Modo interativo

O modo interativo ocorre quando o programa tem apenas 1 argumento (./programa-principal). Após clicar Enter, pedimos ao utilizador para ajustar a janela do terminal de modo a respeitar os limites mínimos (154 x 26) e as suas preferências. Também avisamos que este é o único momento no programa em que o utilizador pode ajustar a janela do terminal. Após clicar Enter e se a janela tiver tamanho apropriado, entramos no modo interativo. Aqui o utilizador é avisado que não pode mais ajustar o tamanho da janela do terminal, ou então o programa irá terminar imediatamente. Depois é pedido o path para o dataset. Se for colocado um path inválido, é pedido novamente que coloquemos o path do dataset correto (3 tentativas). Se colocarmos um path correto, irá aparecer o menu com

todas as querys e com a opção de sair do programa. Cada uma das querys irá pedir aquilo que for necessário. Se pedir um número N e o utilizador escrever texto, este será avisado para poder corrigir o seu erro. Também será pedido o formato do output que desejamos. Temos duas opções: CSV ou campo por campo. Depois será apresentado o resultado da query que selecionamos. Para visualizar resultados longos temos paginação. Para virar de página basta clicar nas setas (direita ou esquerda). Para voltar ao menu basta clicar Enter. Tentamos fazer um modo interativo intuitivo, fácil de entender e amigo do utilizador, que avisa os erros do utilizador mas também as limitações do próprio programa (não se poder ajustar o tamanho do terminal). Demos um tamanho mínimo à janela do terminal, para o programa ser sustentável de se usar (para vermos o menu inteiro, todas as opções, etc…). Como não lidamos com a mudança de tamanho da janela do terminal, decidimos encerrar o programa sempre que o utilizador altere este tamanho, para que o utilizador não experiencie uma má apresentação dos resultados, etc.

Módulo de testes

O modo testes recebe 4 argumentos (./programa-testes, path para o dataset, path para o ficheiro de input e path para a diretoria com os outputs esperados). Para realizar os testes, comparamos os ficheiros criados pelo programa com os ficheiros de output esperados, através da função compare_files, para vermos se são iguais ou não. Se não forem iguais, esta função retorna a linha onde ocorreu a primeira incongruência. Depois também contamos o tempo que cada comando demorou, e posteriormente contamos o tempo que todos os comandos de uma query demoraram em conjunto. Apresentamos assim, no terminal, cada comando, a query que realiza, o seu input, o tempo que demorou e se passou no teste. Se não passou no teste também apresentamos a linha onde ocorreu o primeiro erro. Também apresentamos, para cada query, o tempo que todos os comandos dessa query demoraram em conjunto e quantos desses comandos passaram no teste (em fração e em percentagem). Se nem todos passaram no teste, então é indicado o primeiro comando onde ocorreu o erro. No fim, apresentamos o tempo geral de execução do programa e a memória total usada pelo programa.

Utilidade

No módulo utility encontram-se várias funções e pequenas estruturas de dados auxiliares que são usadas em várias partes do programa.

Temos várias funções que lidam com ficheiros, verificam se certas directorias e ficheiros existem, se são válidas e que removem um new line de um ficheiro.

Funções que utilizam strings, convertem strings para inteiros, tipos Date e vice versa, comparam duas strings com tipo void* (para utilização em estruturas de dados abstratas, como as hashtables) e que convertem as letras minúsculas de uma string em maiúsculas.

Estrutura de dados opaca do tipo Date que armazena uma data com dia, mês, ano, horas, minutos e segundos, e funções para criar, converter, comparar, aceder a campos, libertar espaço em memória e calcular dias destas datas.

Por fim, temos um outro tipo de estrutura de dados opaca que armazena um par inteiro string e funções para criar, aceder, comparar e libertar o espaço em memória destes pares.

Queries e Análise do desempenho

Na **query 1**, apenas acessamos a um elemento da hashtable, logo esperávamos tempo constante, ou aproximadamente, constante - O(1).

Dataset pequeno: 36 comandos

```
Query  1 total duration: 0.001767
Query  1 total duration: 0.001670
Query  1 total duration: 0.001717
```

Ryzen 7 5700u
Intel Core i7-7500U
Ryzen 7 5700U

Dataset grande: 180 comandos

```
Query  1 total duration: 0.014751
Query  1 total duration: 0.014013
Query  1 total duration: 0.013768
```

Ryzen 7 5700u
Intel Core i7-7500U
Ryzen 7 5700U

Na **query 2**, acessamos um elemento da hashtable, O(1) e ordenamos a sua lista de reservas e/ou voos. Ordenamos, primeiro, por id com heapsort, O(nlog(n)), e depois por data com o radixsort, O(n). No fim, percorremos novamente a lista, O(n), para acessar cada elemento da mesma O(1).

Dataset pequeno: 12 comandos

```
Query  2 total duration: 0.000732
Query  2 total duration: 0.000649
Query  2 total duration: 0.000742
```

Ryzen 7 5700u
Intel Core i7-7500U
Ryzen 7 5700U

Dataset grande: 50 comandos

```
Query  2 total duration: 0.005789
Query  2 total duration: 0.006311
Query  2 total duration: 0.005255
```

Ryzen 7 5700u
Intel Core i7-7500U
Ryzen 7 5700U

Na **query 3**, apenas acessamos a um elemento da hashtable (a soma dos ratings e o número de ratings já foi calculado) e fazemos uma divisão, logo esperávamos tempo constante, ou aproximadamente, constante O(1).

Dataset pequeno: 6 comandos

```
Query  3 total duration: 0.000350
Query  3 total duration: 0.000276
Query  3 total duration: 0.000371
```

Ryzen 7 5700u
Intel Core i7-7500U
Ryzen 7 5700U

Dataset grande: 50 comandos

```
Query  3 total duration: 0.003954
Query  3 total duration: 0.004218
Query  3 total duration: 0.003678
```

Ryzen 7 5700u
Intel Core i7-7500U
Ryzen 7 5700U

Na **query 4**, acessamos a um hotel numa hashtable, O(1), e, se ainda não está ordenado, ordenamos a lista de reservas desse hotel. Primeiro ordenamos por id usando heapsort, O(nlog(n)), e depois ordenamos por datas com radixsort, O(n). Depois percorremos novamente a lista, O(n), para acessar cada elemento da mesma O(1).

Dataset pequeno: 6 comandos

```
Query  4 total duration: 0.006894
```
Ryzen 7 5700u

```
Query  4 total duration: 0.009796
```
Intel Core i7-7500U

```
Query  4 total duration: 0.006971
```
Ryzen 7 5700U

Dataset grande: 5 comandos

```
Query  4 total duration: 1.028847
```
Ryzen 7 5700u

```
Query  4 total duration: 1.077637
```
Intel Core i7-7500U

```
Query  4 total duration: 1.039996
```
Ryzen 7 5700U

Na **query 5**, acessamos a um aeroporto numa hashtable, O(1), e, se ainda não está ordenado, ordenamos a lista de voos de origem e a lista de voos de destino. Primeiro ordenamos por ids com heapsort, O(nlog(n)), e depois ordenamos por datas com radixsort, O(n). Depois percorremos a sua lista de voos ordenados até encontrarmos um voo que corresponda a data inicial ou superior, depois continuamos a percorrer a lista até aparecer um voo que corresponda a data final ou superior, O(n), (pesquisa linear).

Dataset pequeno: 6 comandos

```
Query  5 total duration: 0.000965
```
Ryzen 7 5700u

```
Query  5 total duration: 0.000988
```
Intel Core i7-7500U

```
Query  5 total duration: 0.000983
```
Ryzen 7 5700U

Dataset grande: 50 comandos

```
Query  5 total duration: 0.110078
```
Ryzen 7 5700u

```
Query  5 total duration: 0.153914
```
Intel Core i7-7500U

```
Query  5 total duration: 0.113368
```
Ryzen 7 5700U

Na **query 6**, fizemos uso de uma min heap com tamanho N aeroportos, dado pelo comando, calculando para cada aeroporto o número de passageiros no ano dado e adicionando à heap caso o número seja maior que o mais pequeno que se encontra lá. Depois de percorrido o ano fazemos remove da heap até esta estar vazia, obtendo assim os aeroportos ordenados por ordem crescente de número de passageiros.

Foram para isso utilizadas duas listas de ids de voos (uma de origem e outra de destino), que se encontram nas estruturas de cada aeroporto, ordenadas por data, para o cálculo do número de passageiros daquele aeroporto num dado ano, fazendo uso de pesquisa binária para encontrar o ano e percorrendo os voos até não estarmos no ano dado, usando gets para obter o número de passageiros de cada voo. Também foi usada uma estrutura auxiliar (de um par inteiro e string) para guardar o número de passageiros e o nome do aeroporto na heap.

A utilização de uma min heap permite inserções em O(log(N)) e caso o número de passageiros não seja maior que o menor que se encontra na heap, essa inserção é feita em O(1), mas como temos que percorrer todos os voos de cada aeroporto num dado ano, esperávamos um tempo maior relativamente às outras queries, devido ao cálculo do número de passageiros de todos os voos, o que se verifica na execução da query, no dataset grande.

Dataset pequeno: 6 comandos

```
Query  6 total duration: 0.000794
```
Ryzen 7 5700u

```
Query  6 total duration: 0.001027
```
Intel Core i7-7500U

```
Query  6 total duration: 0.000822
```
Ryzen 7 5700U

Dataset grande: 50 comandos

```
Query  6 total duration: 1.094449
```
Ryzen 7 5700u

```
Query  6 total duration: 1.449957
```
Intel Core i7-7500U

```
Query  6 total duration: 1.129183
```
Ryzen 7 5700U

Na **query 7**, começamos por ordenar os delays de cada aeroporto, usando quicksort $O(n^2)$. Depois calculamos a mediana de cada aeroporto. Depois ordenamos todos os aeroportos em função dessa mediana, usando quicksort novamente $O(n^2)$. Aqui não usamos uma min heap, pois uma vez ordenada a lista dos aeroportos, o trabalho já está feito para os próximos comandos com query 7, independentemente do N escolhido. Assim, só se ordena tudo uma vez, e depois basta apenas ir buscar os resultados a esta lista já ordenada.

Dataset pequeno: 4 comandos

```
Query  7 total duration: 0.000301
```
Ryzen 7 5700u

```
Query  7 total duration: 0.000253
```
Intel Core i7-7500U

```
Query  7 total duration: 0.000317
```
Ryzen 7 5700U

Dataset grande: 10 comandos

```
Query  7 total duration: 0.125504
```
Ryzen 7 5700u

```
Query  7 total duration: 0.231900
```
Intel Core i7-7500U

```
Query  7 total duration: 0.125623
```
Ryzen 7 5700U

Na **query 8**, testamos várias estratégias:

1 - Ordenar uma lista com reservas pela data inicial, percorrer a lista calculando o preço de cada reserva entre os limites dados, parando quando a data inicial for maior que o limite final, neste caso temos uma ordenação do array das reservas de um hotel que é feita com um heapsort para ordenar por id com O(nlog(n)) em todos os casos, seguida de 3 ordenações com um radix sort para ordenar por data com O(n), e finalmente o cálculo do custo de cada reserva que é feito no pior dos casos n vezes

2 - Ordenar uma lista com reservas pela data final, pesquisa binária da data limite inicial, percorrer a lista calculando o preço de cada reserva entre os limites dados, parando quando chegar ao fim da lista, aqui temos novamente a ordenação do array com heapsort seguida de 3 ordenações com um radix sort, depois pesquisa binária da primeira reserva a ser calculada O(log(n)) e finalmente o cálculo do custo de cada reserva até ao fim da lista que é feito no pior dos casos n vezes também

3 - Não ordenar a lista de reservas, apenas percorrer a lista e calcular o preço da reserva entre os limites dados, neste caso não ordenamos a lista e apenas percorremos a totalidade da lista, calculando o preço de cada reserva entre os limites dados, sendo necessário apenas percorrer a lista uma vez O(n)

Como o número de queries a serem executadas não é muito elevado, é preferível percorrer o array das reservas de cada hotel em cada query pois o tempo necessário para ordenar os arrays não compensa o ganho no cálculo dos preços. Caso o número de queries fosse muito elevado seria melhor a ordenação do array de cada hotel (pois esta é guardada na estrutura), e aí o ganho já compensaria o tempo da ordenação

Tempos médios de cada caso:

Caso	Tempo médio (seg) (com ordenação)
1	13.2175
2	13.9375
3	1.9325

Tempos médios com o dataset grande medidos num processador r7 5700u

Dataset pequeno: 10 comandos

```
Query  8 total duration: 0.000923
```
Ryzen 7 5700u

```
Query  8 total duration: 0.001021
```
Intel Core i7-7500U

```
Query  8 total duration: 0.000912
```
Ryzen 7 5700U

Dataset grande: 50 comandos

```
Query  8 total duration: 0.734522
```
Ryzen 7 5700u

```
Query  8 total duration: 0.729550
```
Intel Core i7-7500U

```
Query  8 total duration: 0.750370
```
Ryzen 7 5700U

Na **query 9**, é feita uma ordenação dos utilizadores por nome no catálogo dos utilizadores com um heapsort O(nlog(n)), após o parser ser concluído, depois é usada essa ordenação para pesquisa dos nomes com o prefixo dado, utilizando pesquisa binária para encontrar o primeiro nome com esse prefixo e percorrendo a lista até um nome não conter esse prefixo.

Como fazemos muitas comparações com strcoll e devido à ordenação feita pelo strcoll e à utilização de caracteres utf-8 utilizamos várias funções para verificar se um certo prefixo é prefixo de um nome na pesquisa da lista, apesar da pesquisa na lista ser uma pesquisa binária, o peso das comparações com caracteres utf-8 é grande, logo esperávamos um tempo de execução relativamente maior que o das outras queries, o que se verifica nos tempos de execução da query, no dataset grande.

Dataset pequeno: 8 comandos

```
Query  9 total duration: 0.001497
```
Ryzen 7 5700u

```
Query  9 total duration: 0.002578
```
Intel Core i7-7500U

```
Query  9 total duration: 0.001502
```
Ryzen 7 5700U

Dataset grande: 50 comandos

```
Query  9 total duration: 2.302704
```
Ryzen 7 5700u

```
Query  9 total duration: 2.271628
```
Intel Core i7-7500U

```
Query  9 total duration: 2.354599
```
Ryzen 7 5700U

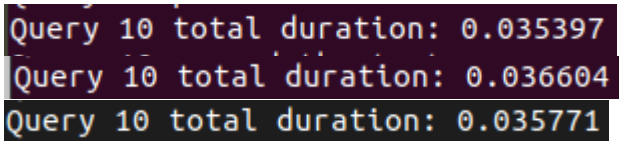
Na **query 10**, são pedidas várias informações sobre o dataset ao longo do tempo, para as calcular agrupadas por dia, mês ou ano, dependendo dos argumentos, utilizamos a função catalogs_compute_Q10, que consulta esses dados para uma determinada data, mês ou ano, e chamamos essa função tantas vezes quanto necessário, por exemplo, se forem pedidas as informações relativas a 2022, chamamos a função para cada mês desse ano.

Para calcular os utilizadores que criaram conta numa data criamos uma lista com os ids dos utilizadores ordenados por data de criação de conta, a função getNewUsers pesquisa (com a função searchDataOrdList que utiliza pesquisa binária) o primeiro índice que corresponde à data que queremos e, a partir daí, enquanto estiver dentro dessa data, é incrementado o número de utilizadores.

O número de reservas é calculado da mesma maneira que os utilizadores, com uma lista ordenada por data de início e pesquisa binária.

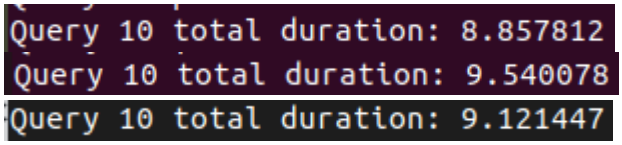
O número de voos é calculado da mesma maneira que os utilizadores, com uma lista ordenada por data de partida e pesquisa binária, e para calcular o número de passageiros, ao encontrar um voo na data pedida, para além de somar um ao acumulador de voos, também é somado o número de passageiros desse voo aos passageiros dessa data. Para calcular o número de passageiros únicos criamos uma lista ordenada com os ids destes passageiros (com os ids repetidos). Com cada id, acessamos a cada utilizador, que contém um número que corresponde ao seu respetivo índice numa lista de bools. Com o índice, acessamos a posição deste utilizador nesta lista. Se tiver 0 (false) então colocamos a ser 1 (true), pois este utilizador é passageiro e incrementamos o número de passageiros únicos. Se não tiver 0, não fazemos nada. Assim contamos todos os passageiros únicos. Depois percorremos a lista toda de novo para colocar tudo a 0s (para a próxima query 10), usando memset para ser mais rápido. Para esta query demorosa ser mais rápida (e também para o resto do programa ser mais rápido), decidimos colocar a flag -O2 na makefile, o que levou, no dataset grande, o programa ir de 14 segundos para 11 segundos. Depois usei bools ao invés de inteiros nesta lista (menos memória a alterar logo mais rápido) e o programa foi de 11 segundos para aproximadamente 9.5 segundos.

Dataset pequeno: 6 comandos



Ryzen 7 5700u
Intel Core i7-7500U
Ryzen 7 5700U

Dataset grande: 5 comandos



Ryzen 7 5700u
Intel Core i7-7500U
Ryzen 7 5700U

Adequação da aplicação para um dataset maior

Considerando um dataset de ordem superior, tivemos que começar a tentar diminuir a memória usada pelo programa (para não exceder o máximo de 5GB). Primeiramente, as nossas hashtables tinham tamanho fixo. Então fizemos com que as hashtables aumentassem de tamanho, consoante aumentava o número de elementos das mesmas. Assim, o tamanho delas seria o adequado para qualquer dataset. Outra estratégia que tivemos foi substituir os ids de voos e reservas (que ocupavam 10 e 14 bytes respetivamente) por números inteiros longos (que ocupam apenas 8 bytes).

As listas ordenadas que algumas estruturas têm estavam a ser criadas com um tamanho mínimo, alteramos esse tamanho para 1, pois em algumas ocasiões essas listas teriam poucos elementos, o que também resultou numa diminuição da memória.

As mesmas listas utilizavam apontadores void para ter abstração de tipo, mas quando mudamos os ids de voos e reservas para inteiros ao guardar na lista tínhamos de guardar um apontador para um inteiro o que ocupava mais memória do que ter apenas um inteiro numa lista de inteiros, então alteramos as listas para poderem ter um array de inteiros, caso esse fosse o tipo de data a ser usado, o que fez uma grande diferença na diminuição da memória do programa, pois existem muitas reservas e voos em listas.