Universidade do Minho

Universidade do Minho

Licenciatura em Engenharia Informática

Laboratórios de Informática III

Relatório Fase 2 – Grupo 42

André Miranda – a104088

José Soares – a103995

Nuno Melo – a104446

Braga, Janeiro 2024

Índice

1 – Introdução

Tendo em conta o que foi afirmado no relatório da primeira fase, este projeto foi realizado e desenvolvido no âmbito da unidade curricular de Laboratórios de Informática III do ano 2023/2024 para abordar conceitos fundamentais como modularidade e encapsulamento, além de consolidar conhecimentos da linguagem C e de Engenharia de Software, nomeadamente, estruturas dinâmicas de dados, validação funcional, e medição de desempenho.

A primeira fase do projeto, consistia na implementação inicial de uma base de dados, tendo como objetivo a criação de um modo *batch*, responsável por fazer *parsing* e validação dos ficheiros de entrada, para que depois fosse possível iniciar um tratamento de dados adequado para a resposta de seis das dez *queries*, tendo sempre em conta os conceitos fundamentais necessários à execução do projeto.

Na segunda fase, foi exigida a manutenção dos requisitos anteriores e a implementação de melhorias. Os pontos chave incluem a continuidade de estruturas e módulos existentes, execução de todas as queries presentes na secção 4 do guião, a introdução do modo interativo, incluindo o menu de interação com o programa e um módulo de paginação para apresentação de resultados longos, análise de desempenho com testes de tempo/memória e testes funcionais. Vale também ressaltar que na segunda fase os *datasets* são significativamente maiores do que na primeira, redobrando assim a nossa atenção para ineficiências e perdas de desempenho no programa.

Durante a apresentação do nosso projeto na primeira fase, foram encontrados pontos fortes no nosso trabalho, como por exemplo, a modularidade e estrutura do programa, mas existiu também um *feedback* menos positivo, que foi fulcral para uma boa continuação do projeto e correção dos pontos fracos do nosso trabalho. Foram nos apontadas ineficiências na execução de várias queries, envolvidas com a procura de dados e output dos mesmo, e também no mau gerenciamento de memória, no qual eram guardadas informações desnecessárias sobre entidades. Dados estes pontos, segue-se agora para a discussão de como foi gerida e desenvolvida a execução do projeto.

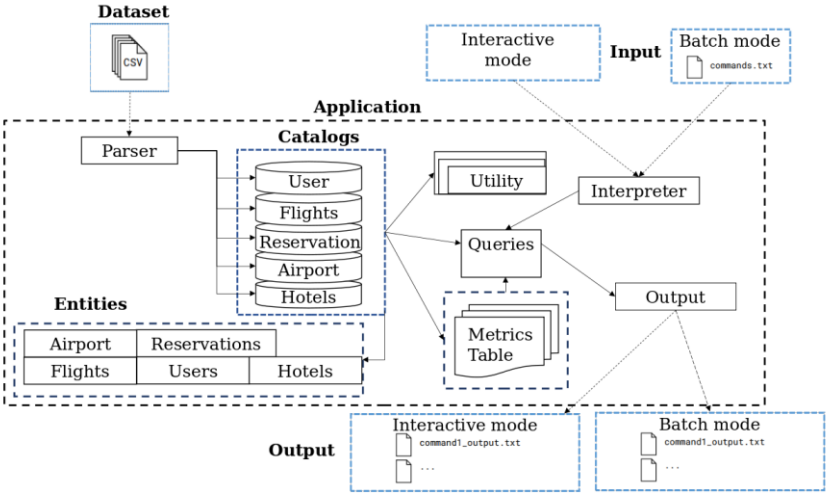
2 – Desenvolvimento

2.1 - Pipeline

Neste momento todo percurso do pipeline está estruturado do seguinte modo:

- Seleção do modo em que vai ser corrido o programa (batch ou interativo);
- Processamento dos ficheiros de entrada referentes às entidades do programa, seguindo-se o armazenamento em estruturas adequadas;
- Parsing e interpretação do ficheiro de input/comando respetivo às queries, e execução das mesmas;
- Organização e escrita dos outputs, dependendo do modo;
- Por fim, libertação de memória e encerramento do programa;

Consideramos este pipeline, uma maneira eficiente de organizar o projeto pois em cada momento é realizado apenas um trabalho, sequencialmente e que tem como objetivo otimizar ao máximo a execução das queries. Ao longo do projeto, a nossa ideia principal era incluir sempre no parsing o máximo de processamento e de "trabalho" efetivo, para que no final a saída de resultados fosse rápida e segura.

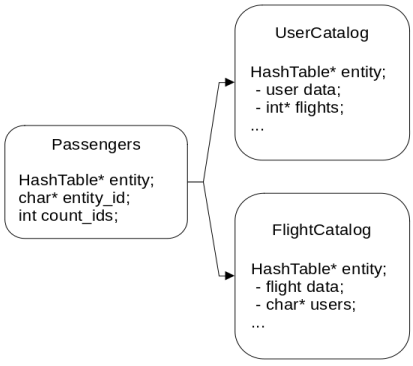


2.2 – Estruturas de Dados

Como foi referido na introdução, o reaproveitamento do código e estruturas é um ponto chave do projeto e assim, decidimos manter o modelo de catálogos dedicados a cada entidade.

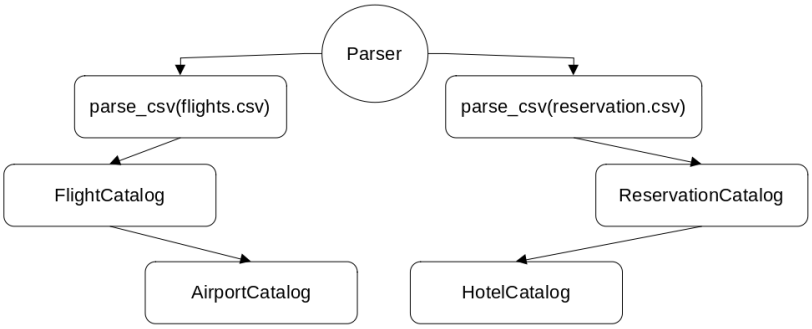
Na primeira fase, foram criadas quatro entidades sendo elas os *users*, *flights*, *passengers* e *reservations*, que estavam depois organizadas em quatro *hash tables* diferentes em que cada uma possuía um *array*, assim como um contador, das entidades inerentes a cada catálogo. Durante esta fase, e tendo em conta o desempenho que vai ser abordado a seguir neste relatório, optou-se por retirar o catálogo referente aos passageiros, uma vez que depois de uma análise profunda, foi denotada a sua extensividade, ineficiência e redundância. Como alternativa, decidiu-se reutilizar os catálogos já existentes como forma de apoio para guardar as informações dos passageiros.

No catálogo dos users, para cada elemento e além das informações do utilizador é guardada uma lista com os voos do mesmo, no catálogo das flights, é feita uma analogia semelhante e a cada voo corresponde uma lista de users desse mesmo voo.

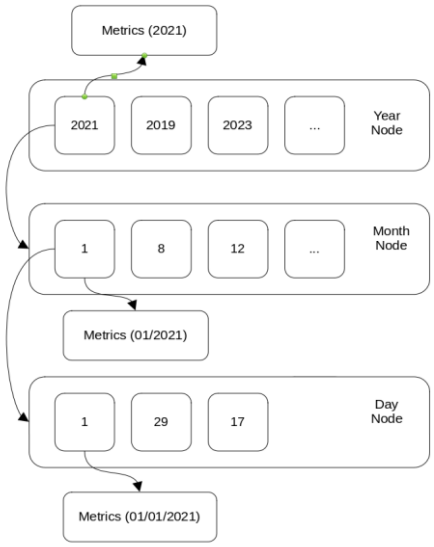


Outras implementações feitas, foram catálogos de hotéis e aeroportos que são uma componente essencial no contexto do projeto, pois proporcionam um meio organizado e eficiente de gerenciar informações mais especificas no contexto de algumas queries. Têm como objetivo e funcionalidades, armazenar e organizar as informações mais relevantes e trazer um melhor desempenho visto que disponibilizam mais facilidade no acesso, na atualização de informações e na integração com a resposta às queries. Estas estruturas de dados são formadas em paralelo com os catálogos principais e são também pelas tabelas de hash, onde através dos voos se constroem os aeroportos e através das reservas se constroem os hotéis.

Uma nota importante, assim como as entidades principais, estas também estão organizadas em entidade/catálogo, onde em cada módulo (.h) são diferenciadas operações exclusivas às suas respetivas partes.



Por fim, e com uma maior complexidade, foi implementada a *MetricsTable*, que no contexto do nosso projeto é a estrutura, ou conjunto delas, responsável por organizar e modular, em forma de árvore e de maneira hierárquica, o conjunto de dados relacionados a métricas do próprio programa, sendo elas número de utilizadores, passageiros, voos e reservas. Para esta estrutura, decidimos utilizar novamente tabelas de hash pois são capazes de executar buscas eficientes e no contexto do problema relacionado aos passageiros únicos, garantem um maior grau de precisão e rastreamento.



De uma forma simples, a MetricsTable nada mais é que um conjunto de tabelas de hash ligadas de forma granular (ano/mês/dia), isto é, a tabela de hash principal resume-se a nodos de anos, sendo que cada ano/nodo é novamente uma tabela de hash com o conjunto de meses e assim sucessivamente para os dias, cada nodo além do apontador para as próximas tabelas apresenta individualmente a sua estrutura de métricas. Em cada estrutura (ano/mês/dia) e para dar acesso às datas válidas, temos um array auxiliar e um contador, para assim a pedido de uma determinada data, termos acesso direto ao mapeamento da tabela.

Relativamente ao problema específico de passageiros únicos, utilizamos dentro das métricas uma outra tabela de hash, mas que desta vez apenas serve para perceber se um utilizador é único ou não numa determinada data. Caso não exista na tabela, é contabilizado como passageiro único, caso já pertença temos assim a confirmação de que para essa data o passageiro em questão não é único.

2.3 – Queries

Alterações queries primeira fase (2,3,4,7 e 9):

Query 2: Para esta *query*, através de um identificador de utilizador e o respetivo campo a analisar é necessário percorrer a lista de voos/reservas paras serem listadas, agora com a adição de uma lista de voos e reservas a um utilizador, em vez de percorrermos a lista toda para os dois campos, temos acesso imediato aos dados do utilizador em questão para serem ordenado e listados.

Query 3: Calcular a "classificação média de um hotel, a partir do seu identificador". Nesta query é utilizado o catálogo dos hotéis, que dado o identificador do hotel, retorna um apontador para uma *struct* Hotel onde estão guardados o somatório de todos as classificações assim como o número de classificações feitas pelos utilizadores, sendo então apenas necessário calcular a divisão entre estes dois parâmetros e devolver para o output.

Query 4: Do mesmo modo que na query 3, em tempo de parsing foi construída uma lista de reservas para cada hotel, e agora não sendo necessário percorrer a lista total de reservas, basta apenas ordenar a lista e retornar todas as informações para o módulo de escrita.

Query 7: Para listar o top N aeroportos com a maior mediana de atrasos utilizamos a nova estrutura *airportCatalog* na qual é guardado um campo, em cada aeroporto, uma lista de todos os atrasos, assim como o número de ocorrências para aquele aeroporto, sendo apenas necessário por parte da query ordenar a lista e obter a mediana.

Query 9: Nesta query, utilizamos dentro do catálogo dos users, uma *hash table* na qual através da chave, que correspondia ao número ASCII do primeiro caractere do nome do *user*, era fornecida uma lista de utilizadores (em forma de inteiro) na qual constavam apenas os que tinham um nome começado por dada letra. Esta alteração permitiu que em vez de ser percorrida toda a lista de utilizadores válidos, apenas uma pequena parte da lista fosse processada pela query e depois passada em forma de resultado para o output.

Em suma, e devido ao feedback dado pelos docentes, decidimos que devíamos ter em conta onde e como procuramos a informação, portanto para as queries implementadas na primeira fase apenas foi alterada a lógica de procura e propriamente a de processamento da mesma.

Na segunda fase do projeto, concluímos as queries 5, 6, 8 e 10:

Query 5: É pedida a listagem de voos com origem num dado aeroporto entre duas datas. Para o processamento e resposta a esta query fizemos uso da estrutura *AirportCatalog* onde constava para cada aeroporto uma lista de voos efetuados, dentro dessa lista a query limitava-se a percorrer todos os voos e restringir aqueles que tivessem acontecido entre o intervalo estipulado. No final era feita uma ordenação com o algoritmo *QuickSort* dos resultados, que eram depois devolvidos para o módulo de escrita.

Query 6: Através do catálogo dos aeroportos, e além dos tempos de atraso e listas de voo, existe um *array* em que cada posição corresponde a um contador de passageiros para um determinado ano, dentro de cada aeroporto. Esta query tira proveito do fácil acesso ao conteúdo de um array dado a posição, e por isso é efetuado o cálculo do índice para o ano passado à query, sendo apenas necessário repetir o processo para o N aeroportos e retornar a estrutura com os aeroportos já ordenados pelo número de passageiros.

Query 8: Dado um identificador de hotel, é utilizado o catálogo de hotéis para obter a estrutura associada a esse identificador. De seguida, com uma lista já formada de todas as reservas para esse hotel é feita uma comparação para saber se a reserva se enquadra no intervalo recebido pela query, caso esteja dentro do intervalo de tempo, é calculado o número de dias efetivos da reserva e é feita a multiplicação pelo preço por noite. No final, e com uma estrutura adequada armazenamos o valor da receita desse hotel e retornamos para o output.

Query 10: É pedido para "apresentar várias métricas gerais da aplicação" tendo em conta que estas métricas podem ser respetivas a cada ano, a cada mês de um dado ano, ou até para cada dia de um mês e um ano passados como argumento. Para a resolução deste problema, utilizamos a *MetricsTable* onde para cada data existe a respetiva estrutura de métricas associadas. Dado o *input* da *query*, é feita a interpretação do comando, para depois ser processada a data em que queremos operar. Com base nesta data é procurado o respetivo nodo na MetricsTable seguindo-se do mapeamento dos valores associados ao mesmo, no fim são guardados numa estrutura adequada os valores necessários, para então serem devolvidos para o módulo de escrita.

2.4 – Otimizações

As principais otimizações a destacar neste projeto são a diminuição da memória utilizada assim como o tempo de execução, e para isso foram feitas alterações de uma forma progressiva tendo como objetivo otimizar ao máximo cada estrutura de dados.

2.4.1 ID's de entidades

Anteriormente, e até ao fim da primeira fase, os identificadores para cada entidade tratavam-se de strings, e por isso, sempre que queríamos aceder ou atualizar a estrutura de dados relativa a cada entidade, era gasto mais tempo e também mais memória devido a alocações de memória para um maior número de bytes. Por isso, passamos a guardar como ID das entidades, com exceção dos users em que foi adaptada uma solução híbrida, um inteiro único para cada voo e reserva, fazendo assim com que a performance global do programa fosse maximizada.

No caso dos users, ainda é necessário guardar a *string* do identificador, no entanto e para contornar o problema da ineficiência, criamos outra *hash table* que para cada ID (*string*) associava uma chave única, e só depois a essa chave, era associado o user através de uma estrutura semelhante à anterior.

2.4.2 Alteração/remoção de campos das entidades

Verificamos ao longo do desenvolvimento do projeto, que existiam campos que não eram necessários ao funcionamento da aplicação e outros que poderiam ser abstraídos a inteiros em vez de strings, por isso fizemos uma limpeza geral nas entidades para que apenas constassem campos necessários.

2.4.3 Módulo de output

Durante a apresentação da 1ª fase, fomos alertados para o facto de termos implementado o módulo de output dentro das queries de uma forma que não é de todo eficaz. Por isso, criamos para cada query estrutura estruturas específicas e adequadas, que depois eram passadas como apontador (void *) para o respetivo módulo de output onde eram tratadas independentemente.

Dadas estas implementações, e outras já discutidas anteriormente, como os catálogos dos hotéis e aeroportos, fica presente uma tabela com os tempos de execução e respetivas otimizações ao longo do projeto, com médias de tempo (10 vezes) e memória utilizada. A máquina utilizada possui: 12th Gen Intel® Core™ i7-12700H × 20/ 16GB RAM.

Otimizações	Tempo	Memória
Queries completas, sem otimizações	79,4s	4,4GB
Mudança ID’s voo e reserva	70,2s	3,4GB
Implementação dos aeroportos, hotéis e métricas	64,3s	3,5GB
Alterações de campos de entidades	60,4s	2,9GB
Identificador inteiro dos utilizadores	55,7s	2,7GB
Remodelação do algoritmo de contagem dos passageiros únicos	46,8s	2,8GB
Formação da tabela de hash para query9	37,1s	2,9GB
Compilação -O2/O3	33,4s	2,9GB

2.5 – Testes e desempenho

O programa de testes no contexto do projeto foi desenvolvido para validar e avaliar o funcionamento do programa principal que lida com consultas em um conjunto de dados.

Esta componente do código é crucial para garantir a robustez e eficiência do programa principal, além de proporcionar uma análise do desempenho em diferentes máquinas e ambientes de teste. As medições de tempo e memória são fundamentais para avaliar o impacto das consultas em termos de eficiência.

Para comparar as diferentes performances e tirar proveito do programa-testes utilizamos um conjunto de medidas para ao longo do desenvolvimento do nosso trabalho conseguirmos ter uma ideia geral do ponto da situação. Para medir os tempos e picos de memória utilizamos a ferramenta /usr/bin/time, e as seguintes condições:

- Compilação com: -O3;
- Repetição de 10 vezes mais 2 de inicialização;
- Média das 10 vezes;

	Máquina 1	Máquina 2	Máquina 3
CPU	12th Gen Intel® Core™ i7-12700H	AMD® Ryzen 5 5500	11th Gen Intel(R) Core(TM) i3-1115G4
RAM	16GB	8GB	8GB
CORES THREADS	14 20	6 12	2 4
OS	Pop!_OS 22.04 LTS	Ubuntu 22.04.3 LTS	Virtual Machine Linux (2GB RAM)
COMPILADOR	gcc 11.4.0	gcc 11.4.0	gcc 11.4.0
DATA REGULAR 100 QUERIES	0.13s	0.17s	0,20s
DATA LARGE 500 QUERIES	33.7s	47.1s	215.6s

Na plataforma de testes, o tempo que consta atualmente é de 0.2/0.3 segundos para o dataset regular, e cerca de 70s para o dataset large. Um dos nossos maiores objetivos ao longo do projeto foi diminuir o tempo inicial e sem otimizações de 144s (na plataforma), pois apesar de não constarem critérios relativos ao tempo de execução no guião, sentimos que conseguíamos realmente otimizar ainda mais o código. Inicialmente colocamos a meta dos 100s e dos 3GB, mas felizmente alcançamos os 69-70 segundos com 2.9GB de memória.

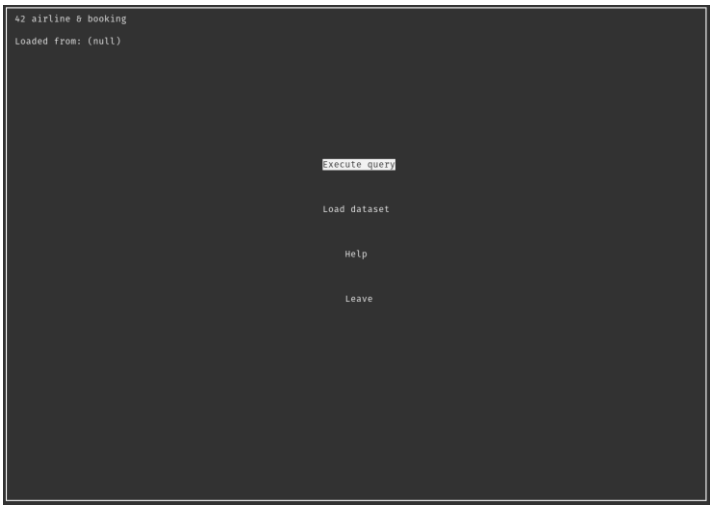
Em termos de memória, os resultados são: 24MB/2,9GB para as máquinas 1 e 2, e para a máquina 3, dado o ambiente ser mais limitado tem os seus picos de memória em 24MG/1,7GB.

Em suma, e face às implementações que fizemos, sentimos gradualmente que o projeto estava a ficar muito mais eficiente, nem tanto em relação ao tempo de parsing pois optamos por colocar maior parte do trabalho nessa parte, mas sim nas queries onde nenhuma alcança o limite de 0,06s (query 9).

2.6 – Modo Interativo

Como é descrito no guião, o modo interativo na nossa aplicação disponibiliza todo um menu interativo onde é possível carregar o *dataset*, a partir de onde este esteja guardado, executar comandos das dez queries, e é também disponibilizado acesso a um menu de ajuda para saber que tipo de comandos podem ser executados e como devem ser utilizados.

Dentro do que é a execução dos comandos, é então aberto um *display* onde ficam disponíveis os resultados dos comandos, caso estes sejam muito extensos, é feito então o modo de paginação onde é possível percorrer as diferentes páginas dos resultados.



2.7 – Aspetos a melhorar e dificuldades

Nesta fase, e também devido à complexidade acrescida, sentimos inicialmente algumas dificuldades em relação à estrutura do nosso projeto e ao grau de abstração presente no trabalho, relacionados ao facto de existir sempre maneiras de melhorar e opções mais "fáceis" de desenvolver.

Com o decorrer do projeto, fomos adaptando os problemas às nossas necessidades e no final obtivemos um bom resultado. Contudo, é preciso reconhecer aspetos mais fracos do nosso trabalho, como por exemplo:

- Modularidade: o nosso módulo *querie_services.c* correspondente às queries é muito extenso, e mesmo dividindo pela metade, torna-se difícil fazer a manutenção do mesmo. De resto, acho que cumprimos com esse requisito e todos os outros ficheiros apresentam essa característica.

- Complexidade estruturas de output: para contornar o problema das queries estarem a executar a escrita nos ficheiros de output, criamos estruturas relativas a cada query para ser as transportadoras dos resultados às funções de output. Na nossa opinião, apesar de nos darem segurança e flexibilidade no meio para escrever-mos as respostas, tornam-se também complexas e ocupam muito mais espaço no código, pois sendo este um projeto que visa o conceito fundamental de encapsulamento, decidimos preservar este conceito em todas as estruturas.

3 – Conclusão

Em resumo, a experiência com o projeto foi tanto enriquecedora quanto desafiadora, proporcionando um avanço significativo nas nossas habilidades em programação em C. Durante o processo, aprofundamos nossa compreensão em relação a várias estruturas de dados, reconhecendo a importância de analisar as características específicas de cada uma. Sentimos que em relação à primeira fase existiram grandes avanços em termos do nosso conhecimento, e tendo já algumas bases necessárias, o desenvolvimento do projeto tornou-se cada vez mais desafiador e interessante, tudo na medida correta.

Serviu também para criar boas práticas de programação, como a modularidade e o encapsulamento, pois estes conceitos foram enraizados no projeto desde cedo e, no geral, consideramos que estão bem implementados. Além disso, exploramos e melhoramos a forma como concebemos e executamos projetos em grupo, suscitando sempre o trabalho em equipa e boas discussões.

As valiosas lições extraídas deste projeto certamente serão aplicadas em futuros projetos, contribuindo para o progresso contínuo de nossa trajetória profissional na área de engenharia de software.