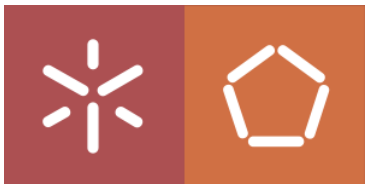

Universidade do Minho
Licenciatura em Engenharia Informática

Laboratórios de Informática III

2023/2024

Fase 2

Realizado por:

Ana Alves	A95128
Simão Antunes	A100597
Gonçalo Brandão	A100663

25 de janeiro de 2024

Conteúdo

Lista de Figuras

1 Introdução

Para a segunda parte deste projeto, foi necessária uma restruturação do projeto até então executado. Este relatório contem os passos e decisões tomados na realização da mesma. Desta forma, foi necessário ser capaz de criar coleções e estruturas de dados de forma a conseguir implementar estratégias eficientes e rápidas de realizar as queries propostas. Para este efeito, recorremos à biblioteca da linguagem C, GLib. Este relatório contem os passos e decisões tomados na realização das duas fases do projeto. Inicialmente é apresentada uma descrição detalhada de como realizamos o encapsulamento e garantimos modularidade, seguida de uma explicação da implementação do parsing dos ficheiros, bem como a forma como fizemos o a validação dos mesmos. Posteriormente temos uma explicação das estratégias implementadas no desenvolvimento das queries e do modo interativo proposto. Além disso, apresentamos os nossos resultados obtidos de forma a verificar a rapidez de cada query implementada, finalizando com uma breve conclusão.

2 Arquitetura do projeto

A arquitetura do trabalho foi baseada na arquitetura sugerida no enunciado disponibilizado. Desta forma, conseguimos entender o conceito de encapsulamento e modularidade e destacar os principais componentes, interações e dependências, o que é crucial para tomar decisões informadas sobre a implementação.

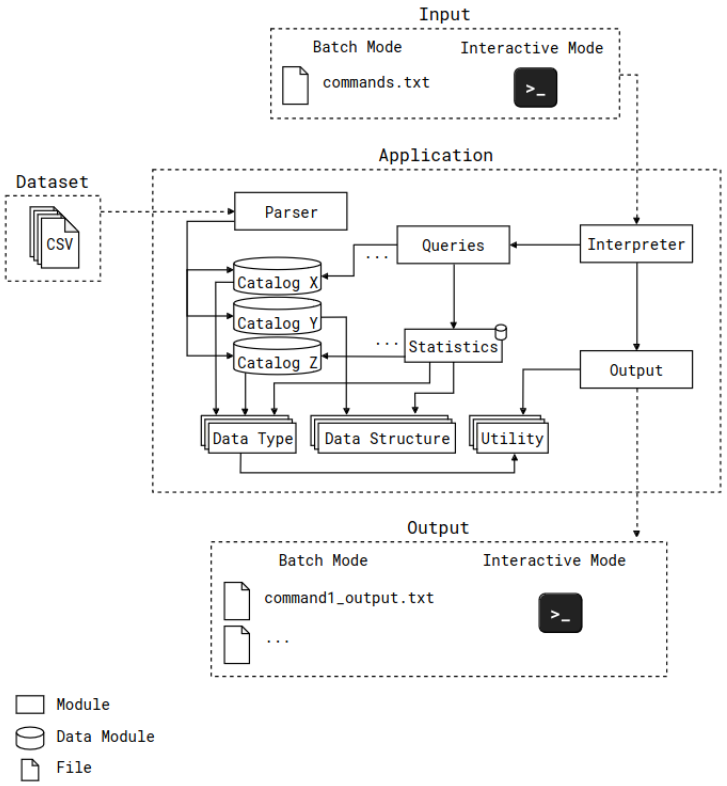

Figura 1: Sugestão da arquitetura para a aplicação a desenvolver

2.1 Encapsulamento e modularidade

De forma a garantir um bom encapsulamento e modularidade no trabalho, o mesmo foi estruturado para esse efeito, crinado módulos, onde as dependências apenas são dos níveis abaixo em relação aos mais acima, nunca no mesmo "nível"ou de cima para baixo.

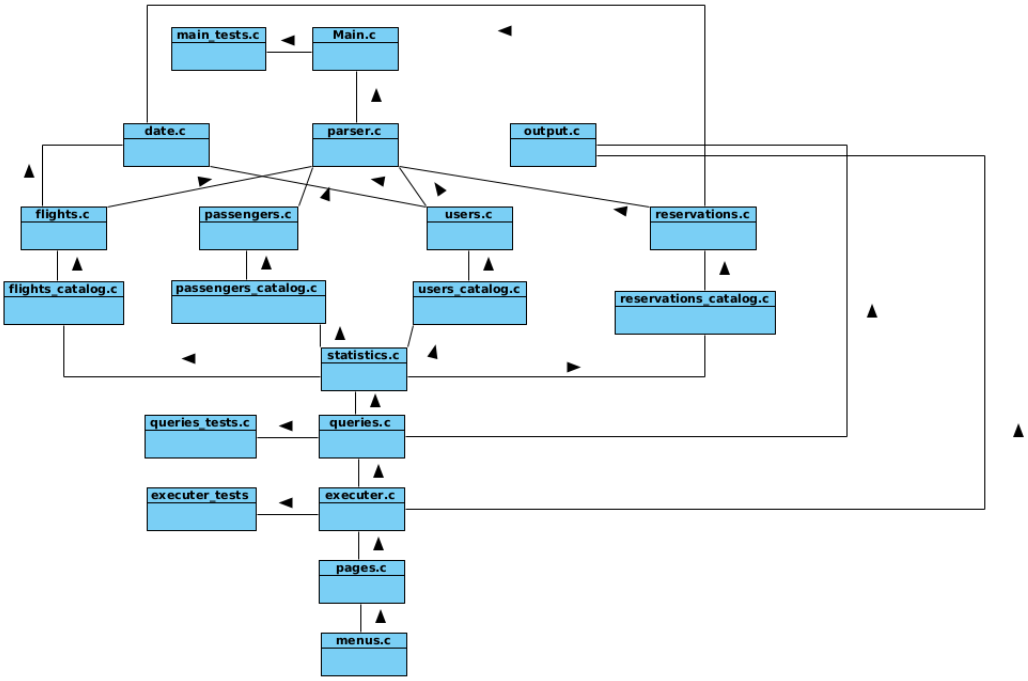

Figura 2: arquitetura para a aplicação a desenvolver

Posto isto, é nestes novos módulos que são implementadas coleções especificas, e todas as operações sobre estas, sendo estas estruturas privadas e as operações referidas públicas. Ou seja, apenas as funções no interior do módulo podem aceder a estas estruturas. No entanto, é sempre garantida a opacidade entre módulos através de cópias dos valores. Isto ocorre, por exemplo, com a estrutura User e as respetivas funções get que são as únicas com acesso à estrutura mencionada, devolvendo sempre cópias. Este encapsulamento está demonstrado na imagem que se segue. É de salientar que, no diagrama, estão apenas presentes os módulos gerais. Deste modo, de forma a tornar a imagem mais percetível.

```
typedef struct users {
    char * id;
    char * name;
    Date * birth_date;
    char sex;
    char * passport;
    char * country_code;
    Date * account_creation;
    int account_status;
} User;
```

Figura 3: Estrutura Users

```
char * get_user_id(User * user);

char * get_user_name(User * user);

Date * get_user_birth_date(User * user);

char get_user_sex(User * user);

char * get_user_passport(User * user);

char * get_user_country_code(User * user);

Date * get_user_account_creation(User * user);

int get_user_account_status(User * user);
```

Figura 4: Gets da estrutura Users

2.2 Estrutura de dados

```
typedef struct flights {
    int id;
    char * airline;
    char * plane_model;
    int total_seats;
    char * origin;
    char * destination;
    Date * schedule_departure_date;
    Date * schedule_arrival_date;
    Date * real_departure_date;
    Date * real_arrival_date;
} Flight;
```

Figura 5: Gets da estrutura Flights

A nível das estruturas de dados foram tomadas decisões de modo a diminuir o tamanho que elas ocupam. Para tal, a estratégia que utilizamos passou por utlizar o tipo de dados com menor tamanho possivel. Como no caso do id e no total seats, foram definidos como int uma vez que consideramos mais eficiente do que um char.

2.3 Parsing dos dados

Para fazermos o parsing dos ficheiros CSV, em vez de termos uma função de parsing específica para cada ficheiro, de modo a reutilizarmos código, criámos uma função geral para parsing de ficheiros. Deste modo, a função recebe como argumento uma função que vai ser específica de cada ficheiro. Posto isto, optamos inicialmente por criar um array de strings, composto pelos tokens, obtidos a partir de cada linha de dados dos ficheiros, os quais são de seguida tratados pela função passada como argumento.

2.4 Validação dos ficheiros de dados

De forma a fazer a validação de entradas de dados durante o parsing dos ficheiros CSV é passada como argumento uma função específica que tem, no seu corpo, uma função responsável pela validação dos dados de cada ficheiro. Assim, verifica-se token a token se o seu campo é ou não nulo e, no caso de estar preenchido, se está conforme o pressuposto. Caso seja válido, é então criada a coleção específica do ficheiro,

i.e Users, Flights, Passengers e Reservations, e adicionada à sua própria estrutura de dados, contida num catálogo exclusivo.

2.5 Queries

Para responder às queries foram criadas novas estruturas para servir como valor de return das várias funcões e também para melhorar o tempo de resposta. As estruturas de dados definidas no catálogo das estatísticas são apenas a GHashTable e o GPtrArray. A GHashTable foi escolhida para recolher informação nas queries, pois como para uma dada key a informação tem de ir sendo atualizada, é a estrutura mais rápida para detetar e devolver o campo que queremos atualizar. Já o GPtrArray foi a estrutura escolhida para que se pudesse ordenar as coleções que lhe são inseridas. Todas as funções que podem aceder ao catálogo das estatísticas são implementadas em estatistics.c respeitando sempre a opacidade e encapsulamento. Estas estruturas facilitam na resposta às querys uma vez que estar a percorrer um catálogo sempre que é feita uma interrogação é ineficiente. Dessa forma, o nosso objetivo era percorrer os catálogos necessários apenas uma vez e criar estruturas adequadas para responder a cada query. No entanto, essa implementação levou a um maior uso de memória, o que impediu que o programa fosse executado com o large-dataset.

```c
typedef struct cs{
    GHashTable * users;
    GHashTable * flights;
    GHashTable * reservations;
    GHashTable * hotels;
    GPtrArray * hreservations;
} CatalogoStats;
```

Figura 6: Catálogo de estatísticas

3 Leitura e Tratamento de Dados

De modo a conseguir responder às queries propostas de forma eficiente, primeiro organizamos o grande volume de informação presente nos flights,users, passengers e reservations. Desta forma, criamos catalogos diferentes para cada tipo de dados com a ajuda da biblioteca GLib. Para efectuar o parser, utilizamos uma função geral capaz de trabalhar qualquer estrutura de dados.

```c
void parser(char * path, void (* buildStruct)(char **, void *, void *, void *), void * structure1, void * structure2, void * structure3) {
    FILE * file = fopen(path, "r");
    if (!file) {
        perror("Erro ao abrir o ficheiro");
        return;
    }
    char * line = NULL;
    ssize_t read;
    size_t len;
    while ((read = getline(&line, &len, file)) != -1) {
        char * line_copy = strdup(line);
        char * formated_line = strsep(&line_copy, "\n");
        char ** tokens = parseLine(formated_line);
        buildStruct(tokens, structure1, structure2, structure3);
        free(formated_line);
        free(tokens);
    }
    free(line);
    fclose(file);
}
```

Figura 7: função para o parser

4 Modo Interativo

Para o modo interativo, foi criada uma coleção (Paginacao) que possui uma estrutura de dados com o intuito de armazenar a informação à medida que se vai formando um output nas queries, sendo escolhido o GPtrArray (dados), para além disso são utilizados também caracteres de modo a facilitar a paginação. Deste modo, no fim de uma query, a coleção encontra-se preenchida ao que se segue um modo interativo que permite ao utilizador ver até 40 valores de output simultaneamente, permitindo avançar e recuar de página, ir diretamente para a última página e voltar para a primeira. Enquanto o utilizador quiser, pode continuar a executar queries até o mesmo pretender encerrar o programa.

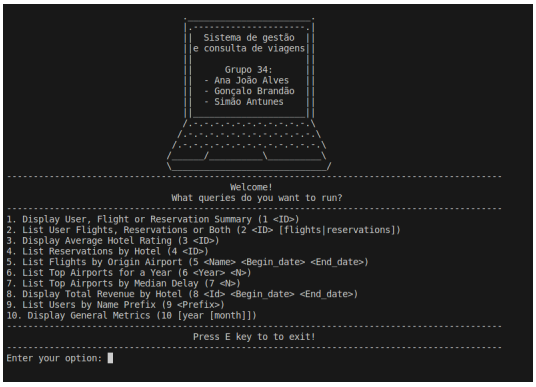

Figura 8: menu: página inicial

Figura 9: menu: paginação

5 Teste

De forma geral, a plataforma disponibilizada pelos docentes para testar o código tornou-se bastante útil na detenção e correção de erros ou código incompleto. Com base nos resultados obtidos, o código foi restruturado diversas vezes e com o auxílio do Valgrind estes foram resolvidos.

5.1 Memory Leaks

Na primeira fase do trabalho, o método utilizado gerava imensos erros e memory leaks devido às dependências de dados entre estruturas de dados que tornavam difícil libertar a memória, problema que foi corrigido através de uma reestruturação do mesmo. Ao longo do trabalho fomos sempre tentando prevenir a ocorrência das mesmas através da criação de funções para fazer free tanto das estruturas de dados como das coleções. Para além disso, sempre que foi necessária a utilização de memoria dinâmica, a implementação do código foi pensada de modo a poder libertar a memória alocada assim que possível. Atualmente o trabalho tem apenas as memory leaks associadas à utilização da biblioteca GLib, que são 0,019 MB.

5.2 Testes de Desempenho

Para esta parte do trabalho foram utilizadas as bibliotecas time.h para obter o tempo de execução e sys/resource.h para obter o maximum memory resident size das queries e do programa. Deste modo,temos uma função que compara os ficheiros criados pelas queries implementadas com os ficheiros com o output pretendido, lendo linha a linha os dois ficheiros e comparando as strings. Esta função vai então verificar se a query executou dentro do tempo estipulado e se os ficheiros são iguais, originando o output consoante os resultados obtidos.

Sistema Operativo	Processador	RAM(GB)	Memória (KB)	Tempo de Execução (s)
Ubuntu 22.04	i7- 10thGen	12	41600	1.111206
			41680	1.129149
			41472	1.140693
Fedora 39	AMD Ryzen 5 5600H	8	40704	0.695691
			40704	0.737732
			40832	0.796419
Ubuntu 22.04	AMD Ryzen 7 4800hs	32	41632	0.729595
			41604	0.699054
			41620	0.704132

Tabela 1: Desempenho do programa

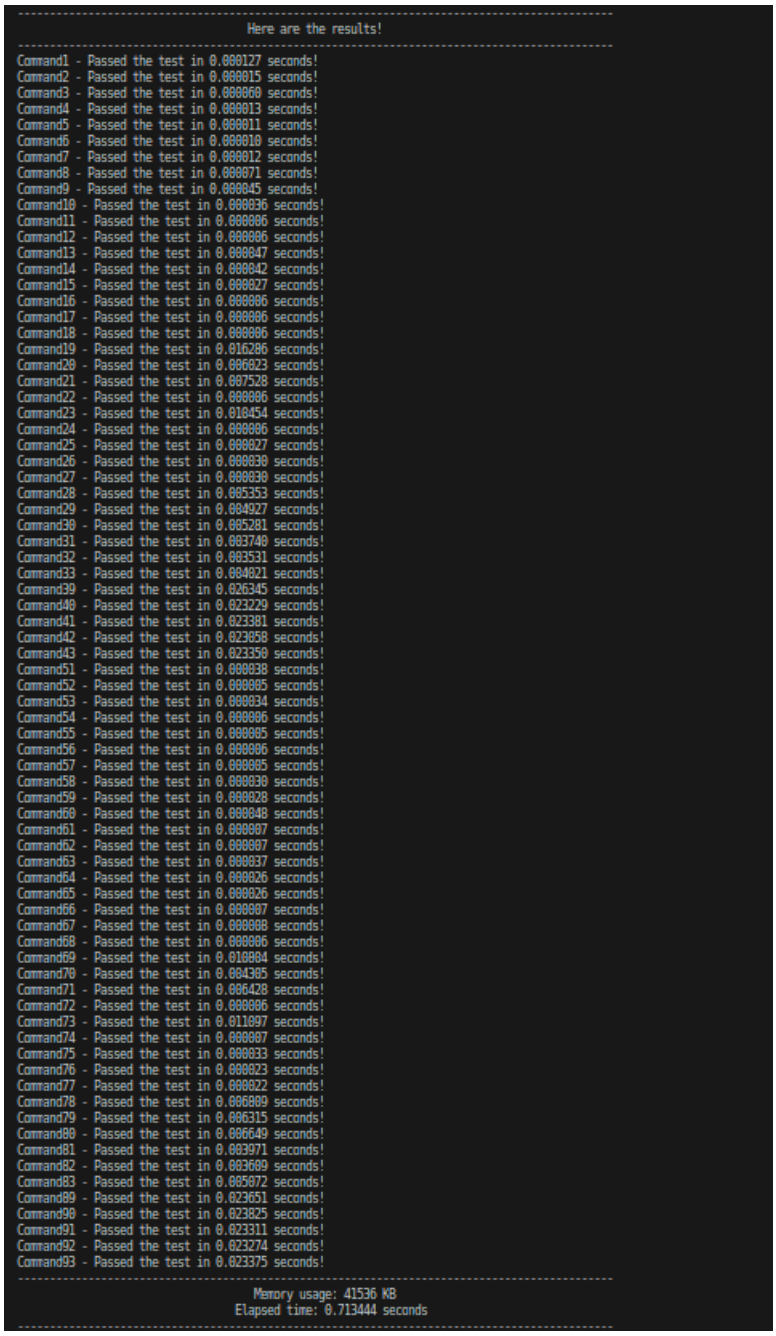

Figura 10: desempenho das queries no Ubuntu 22.04 (i7-10thGen)

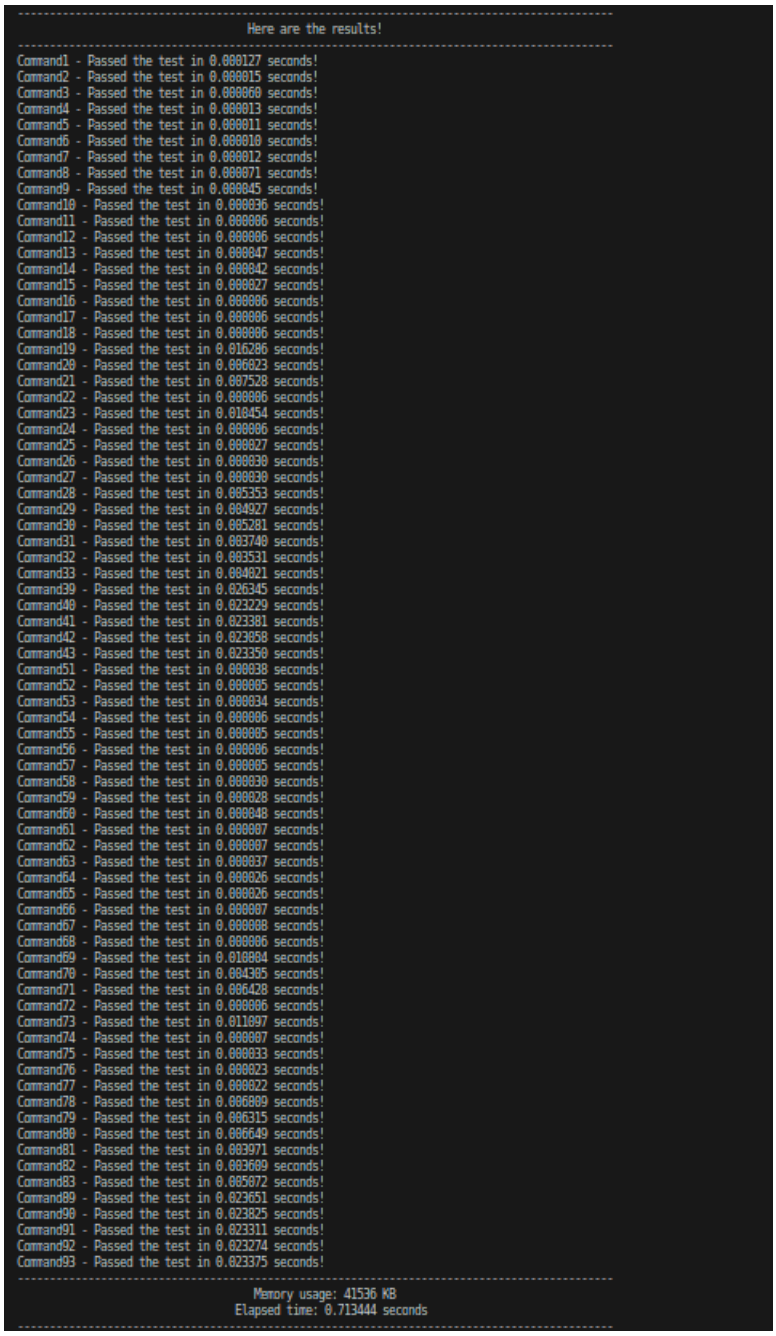

Figura 11: desempenho das queries no Ubuntu 22.04 (AMD Ryzen 7 4800hs)

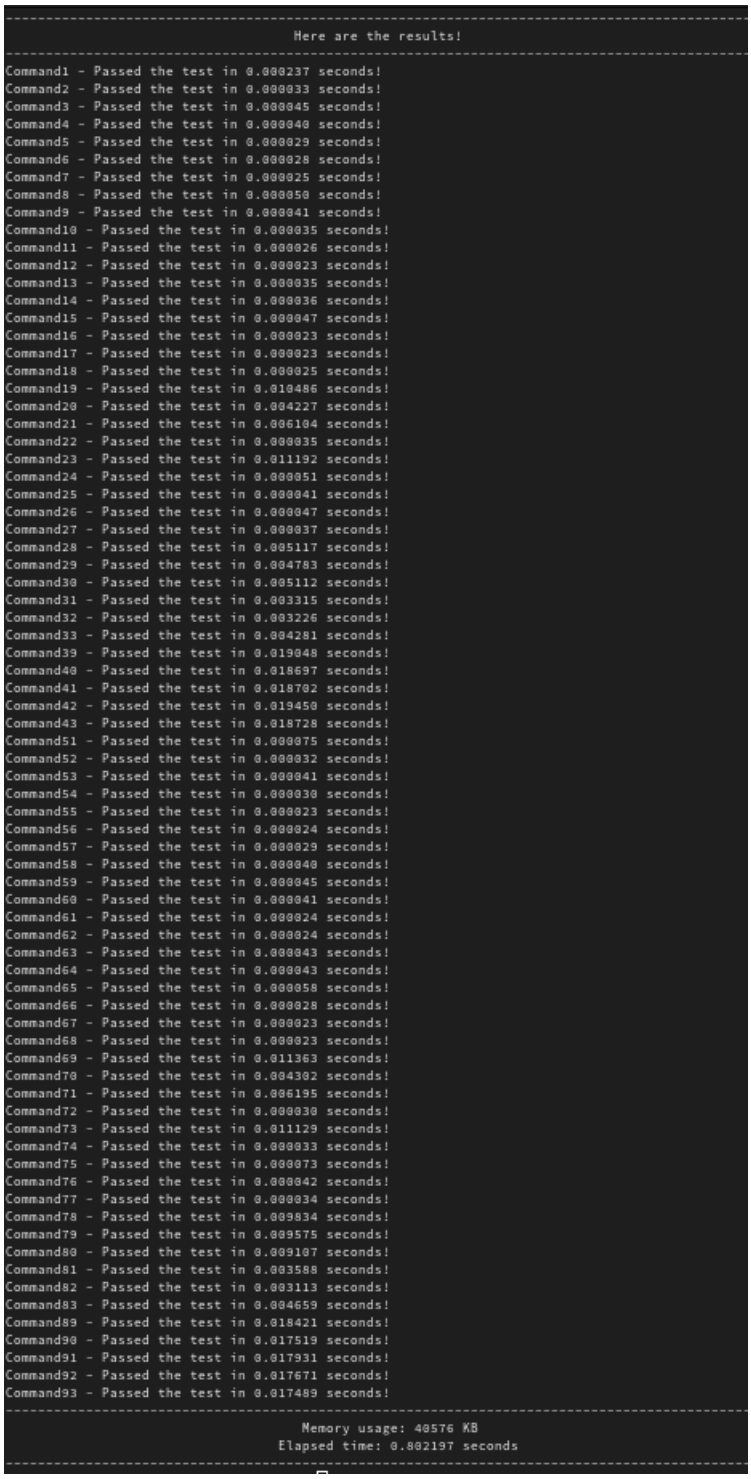

Figura 12: desempenho das queries no Ubuntu 22.04 Fedora 39 (AMD Ryzen 5 5600H)

Em termos de desempenho, observamos que nosso programa se destaca ao lidar com conjuntos de dados pequenos, apresentando resultados em menos de milissegundos na maioria dos casos. Essa eficiência inicial é promissora, sugerindo uma implementação otimizada para cenários menos desafiadores.

No entanto, é crucial considerar como esse desempenho se escala quando confrontado com conjuntos

de dados maiores. O comportamento eficiente em datasets pequenos não foi diretamente proporcional.

Além disso, ao lidar com dados em maior escala, notamos um aumento significativo no consumo de memória. O aumento no consumo de memória está intimamente ligado aos algoritmos utilizados no nosso programa. Tornou-se evidente que, ao lidar com volumes de dados mais substanciais, certos algoritmos podem não estar otimizados para gerenciar eficientemente os recursos disponíveis, resultando em um uso significativo de memória

Testar o programa em diferentes tamanhos de conjunto de dados foi fundamental para garantir uma eficiência consistente em diversas escalas, como é visível a semelhança dos resultados nas diferentes máquinas

6 Conclusão

Consideremos que o trabalho foi bastante desafiante principalmente com a nova implementação do dataset large. Mesmo após várias alterações da estrutura do código, não correu como planeavamos uma vez que para um elevado número de dados o trabalho ultrapassa o limite de memória.

Por outro lado, consideramos que conseguimos implementar um código que contempla boas práticas de encapsulamento, modularidade e opacidade. No primeiro, alcançamos um bom encapsulamento tendo sempre em conta uma hierarquia entre módulos, em que apenas existem dependências de baixo para cima e nunca no sentido inverso ou no mesmo "nível". Em relação à modularidade, garantimos que, apenas é possível gerir um determinado módulo com as funções do mesmo e, para além disso, tornámos os detalhes de implementação dos módulos privados e públicas apenas as funções necessárias a outros módulos. Já no que toca à opacidade, entendemos que esteja assegurada uma vez que nunca são fornecidos os valores reais dos dados, mas sim cópias ou clones dos mesmos.