Laboratórios de Informática III

Trabalho prático

Grupo 53

Luiz Rodrigues A100700
Michelle de Morais A98576
Carolina Pereira A100836

Índice

1. Introdução.

Este projeto foi realizado no âmbito da unidade curricular Laboratórios de Informática III, tendo como objetivo principal desenvolver um programa capaz de ler, armazenar e executar operações sobre dados. Ao longo deste relatório, esclareceremos as escolhas feitas e as dificuldades que surgiram durante a execução desta primeira fase do projeto.

Inicialmente, dedicamos tempo à leitura atenta do enunciado e à escolha da forma de representação dos dados. Em seguida, definimos a sequência de desenvolvimento do projeto. Optamos por iniciar pela estruturação geral, seguida pela implementação da leitura dos arquivos, depois pela criação das queries e, por fim, a verificação de eventuais vazamentos de memória e validações do conjunto de dados.

Este relatório oferecerá uma visão geral das etapas e das considerações que moldaram o desenvolvimento do projeto, e uma visão mais detalhada da versão de entrega do projeto.

2. Versão de entrega.

2.1. Estruturação dos dados.

Para conseguirmos entender melhor como deveríamos prosseguir com o trabalho fizemos as estruturas gerais para o projeto, sendo definidas em C como uma struct no ficheiro .c e tendo um typedef no ficheiro .h.

Depois de termos atribuído a informação cada entrada será introduzida em uma GHashTable, escolhemos esta estrutura pois possibilita inserção e procura de um elemento (com sua chave) em tempo constante O(1) e não nos interessa a ordem das entradas.

Seguem as definições das estruturas e as respectivas chaves utilizadas para serem inseridas na tabela de hash:

```c
struct user{
    char* id;
    char* name;
    char* email;
    char* phoneNumber;
    char* birthDate;
    char sex;
    char* passport;
    char* countryCode;
    char* address;
    char* accountCreation;
    char* payMethod;
    char* accountStatus;
};
```

User, key = id

```c
struct flight{
    char* id;
    char* airline;
    char* planeModel;
    int totalSeats;
    char* origin;
    char* destination;
    char* scheduleDepartureDate;
    char* scheduleArrivalDate;
    char* realDepartureDate;
    char* realArrivalDate;
    char* pilot;
    char* copilot;
    char* notes;
};
```

Flight, key = id

```c
struct reservation{
    char* id;
    char* userId;
    char* hotelId;
    char* hotelName;
    int hotelStars;
    int cityTax;
    char* address;
    char* beginDate;
    char* endDate;
    int pricePerNight;
    char* includesBreakfast;
    char* roomDetails;
    char* rating;
    char* comment;
};
```

Reservation, key = id

```c
struct passengersInFlight{
    GList* userList;
    char* flightId;
    int totalPassenger;
};
```

PassengerFlight, key = flightId

```c
struct passenger{
    GList* flightList;
    char* userId;
    int totalFlights;
};
```

Passenger, key = userId

O motivo de termos duas estruturas para o ficheiro dos passageiros é para facilitar no funcionamento do programa, a estrutura passengersInFlight guarda uma lista de utilizadores associados à um voo e o total de passageiros nesse voo, já a estrutura passenger tem uma lista de voos associados à um utilizador. Para guardar essa lista utilizamos uma lista duplamente ligada (GList) pois não nos interessa a ordem em que as informações são guardadas na lista. Poderíamos substituir por uma lista ligada simples se fosse

necessário diminuir a memória utilizada pela lista, mas como não apresentou ser um problema muito grande nessa primeira parte utilizamos a dupla pois é mais versátil.

2.2. Leitura e atribuição.

A leitura é realizada pelo ficheiro parser.c, que tem a função geral de parsing. A função parte, que recebe o nome do ficheiro que queremos ler (ex. “flights.csv”), a tabela em que será guardada as instâncias de cada estrutura, uma função fillTable, um apontador para o ficheiro de entradas erradas, e a estrutura em que guardamos todas as tabelas.

```
void parse(char* inputFileName,GHashTable* table, parseLine fillTable, FILE* errors, Tables* t);
```

Estrutura da função parse.

Primeiramente, pelo uso da função fopen(), vamos abrir o ficheiro para leitura, caso o ficheiro não exista terminamos a execução com um exit(EXIT_FAILURE). Em seguida declaramos um buffer chamado line com tamanho 2048 (LINESIZE) e utilizamos a função fgets() para ler de linha à linha o ficheiro, neste caso pode ser que seja mais desejável utilizar a função getline para evitar casos em que o buffer não suporta a linha toda mas isso pode ser mudado na versão final do projeto. Se o line estiver vazio na primeira leitura assumimos que o ficheiro está vazio e novamente terminamos a execução. Após as verificações que fizemos, inicia-se um ciclo while, que executa até não haver mais linhas para a leitura. Dentro do ciclo e executada a função fillTable.

```
typedef void (*parseLine) (GHashTable* table, char *line, FILE *errors,Tables* t);
```

Definição do tipo de uma função parseLine.

Há uma função fillTable para cada estrutura (ex. fillFlightTable), que cria uma nova instância da estrutura por meio de uma função atribute (ex .

atributePassenger). Para atribuirmos as informações utilizamos a função strsep() que será executada em um ciclo while até chegarmos ao fim da linha, dentro desse ciclo temos um switch para simplificar a atribuição em que cada case do switch equivale a um campo da estrutura (ex. User: case 1 : id, case 2: name, case 3: email, etc). A função retorna uma instância da estrutura ou retorna NULL se durante a validação algum campo estava inválido. Ao fim do parsing fechamos o ficheiro de entrada e continuamos com a execução do programa.

2.3 Queries.

Após termos atingido um estado aceitável do programa partimos para a criação dos módulos das queries. Decidimos tentar implementar o número máximo de queries para esta primeira fase, sendo que as 6 primeiras foram as mais focadas, e as outras poderiam ser feitas caso tivéssemos tempo.

Uma dúvida que surgiu durante a criação foi se a lógica das queries deveria ser implementada no próprio módulo, ou se deveria ser feita nos módulos das estruturas (ex. na query 2 procuramos informações nos módulos flights.c e reservations.c), para resolver esse dilema perguntamos à um professor da equipe docente que nos disse que deveríamos seguir a segunda opção, e então, por isso, seguimos a instrução.

O módulo individual das queries possui apenas um formatQuery que, com a linha lida no ficheiro input.txt, separa as informações necessárias. Também possui um runQuery que liga-se aos módulos externos para obter as informações necessárias, e chama as funções de escrita do resultado.

Em alguns módulos também estará presente estruturas auxiliares (ex struct query2Aux) e as funções atribuídas a essa estrutura.

Após a execução temos que chamar a função printQ ou printQF dependendo se no input temos um F ou não, essa função escreve para um

ficheiro com a função fprintf(). Em alguns casos temos a estrutura FAC (FileAndCount) que é útil para as queries que têm mais de um resultado.

Para executarmos as queries é necessário chamar a função interpreter definida no módulo handlequery.c, essa função recebe a estrutura das tabelas e o ficheiro que devemos ler as queries (modo batch) caso o ficheiro não exista terminamos a execução do programa.

Para cada linha do ficheiro input.txt criamos um novo ficheiro de resultado em que escreveremos o resultados da query, para lermos as linhas utilizamos a função fgets(). Após terminarmos a execução fechamos o arquivo input.txt.

2.4 Validação e vazamentos de memória.

Estes dois temas estão juntos, pois foram as últimas coisas que fizemos, até termos feito a validação utilizávamos os datasets clean. Grande parte dos vazamentos estavam na atribuição e validação das estruturas, porém também haviam casos em que algumas queries tinham leaks por falta de atenção durante sua criação.

A validação é feita durante cada fillTable que fizermos, como cada fillTable retorna uma nova instância das estruturas se retornarmos um NULL sabemos que algum campo estava inválido, logo escrevemos no ficheiro de erro a linha que foi utilizada para a atribuição. Nas funções atribute, onde são feitas as validações, utilizamos a função de libertação daquela estrutura (ex. destroyPassenger) antes de retornarmos NULL, caso seja necessário remover a entrada inválida de outro ficheiro (caso dos users e flights) chamamos as funções de remoção com o id da estrutura inválida. As condições para invalidar foram as descritas no enunciado.

Os vazamentos foram encontrados com ferramentas como o valgrind e corrigidos com as libertações necessárias. Grande parte da origem desses vazamentos foi a falta de atenção durante o desenvolvimento do projeto.

2.5. O Makefile.

Para criar o Makefile buscamos uma abordagem equilibrada entre boas práticas convencionais e a personalização necessária para atender aos requisitos específicos do projeto, nos baseamos no guião que foi proporcionado, em exemplos que encontramos em pesquisas, e por perguntas aos nossos colegas.

Seguimos um padrão mais geral de compilação em que os arquivos fonte são primeiramente compilados em ficheiros objetos e linkados juntos para criar os executáveis finais.

As flags utilizadas -Wall, -Wextra e -Wno-unused-parameter foram utilizadas para garantir a qualidade geral do código e utilizamos o pkg-config para conseguirmos utilizar a biblioteca glib

3. A ordem de execução.

Após termos explicado como estruturamos o programa agora vamos ver a ordem em que o programa-principal é executado.

Iniciamos pelo ficheiro main.c que possui a nossa função de entrada main() que verifica se o número de argumentos está correto, se não estiver terminamos a execução. Depois disso fazemos a atribuição com a função atributeTables(),executamos as queries com o interpreter e por fim liberamos a memória associada às tabelas com a função destroyTables().

O ficheiro table.c é onde está a estrutura que guarda todas as tabelas que precisamos, nele está definida a função de atribuição que chama cada hashAtribute e depois atribui o valor de retorno dessa função para cada entrada da estrutura.

```c
struct tables{
    GHashTable* flightTable;
    GHashTable* passengerFlightTable;
    GHashTable* reservationTable;
    GHashTable* userTable;
    GHashTable* passengerTable;
};
```

Estrutura que guarda as tabelas

4. Representação gráfica do projeto.

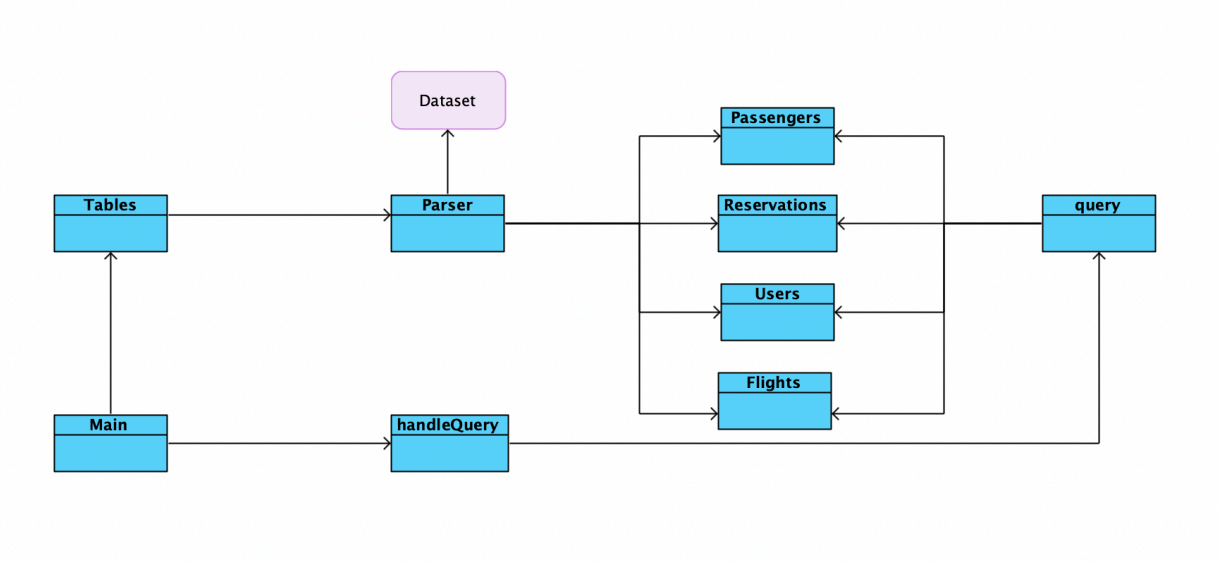

Diagrama que representa o fluxo definido no ponto 3

Nessa imagem conseguimos ver de modo geral como os módulos estão relacionados, sendo que nem todas as ligações estão estabelecidas.

Esta imagem visa simplificar o entendimento da estruturação.

5. Conclusão e trabalho futuro.

Após essa primeira fase conseguimos nos aprofundar no conhecimento e nas funcionalidades da linguagem C, além disso os desafios propostos nos levaram a desenvolver um pensamento mais crítico no desenvolvimento.

Mesmo após termos melhorado a versão inicial ainda há muitos pontos do projeto que devem ser aprimorados.

Uma das melhorias mais importantes é a adequação das estruturas, principalmente nas queries, para o tamanho atual do dataset a velocidade não é afetada pelas estruturas escolhidas, porém com um aumento do tamanho pode ser que algumas estruturas sejam alteradas, por exemplo as listas ligadas, que podem ser substituídas por árvores binárias (GTree).

Devemos garantir que o programa seja escalável para um aumento significativo do tamanho.

Também seria interessante definir a nossa estrutura Tables como um singleton, para que não seja possível instanciar mais de uma vez a estrutura.

Pode ser que seja necessário introduzir novos módulos para lidar com outras partes do projeto, por exemplo as escritas de resposta das queries.

Podemos alterar o modo como os ficheiros são lidos.

Utilizar o módulo statistics, que nessa fase não teve nenhuma função implementada.

Além disso ainda falta a implementação total das queries, o modo interativos e o módulo de testes e uma melhoria geral do código escrito.