Laboratórios de Informática III

Trabalho prático - Fase 1

TRABALHO REALIZADO POR:

SAMUEL MACIEIRA FERREIRA (a100654)
JÉSSICA CRISTINA LIMA DA CUNHA (a100901)
DIOGO ANDRADE FERNANDES (a100746)

GRUPO 30

Introdução

No âmbito da unidade curricular de Laboratórios de Informática III foi-nos proposto, numa primeira fase, o desenvolvimento de uma aplicação com determinadas propriedades e funcionalidades descritas no enunciado do próprio trabalho, disponibilizado pela equipa docente da UC. Estas funcionalidades giram em torno de ficheiros de dados que contém diferentes informações relacionadas a *Users, Passengers, Reservations e Flights,* a partir das quais vamos gerar os respectivos catálogos. Para a realização deste trabalho, e como nos foi indicado inicialmente, tivemos em consideração a arquitetura proposta (Fig. 1).

Tendo em conta o propósito da UC mesmo que nesta primeira fase do projeto ainda não seja contabilizada para nota tentamos sempre ter em consideração as componentes relacionadas com modularidade e encapsulamento de dados.

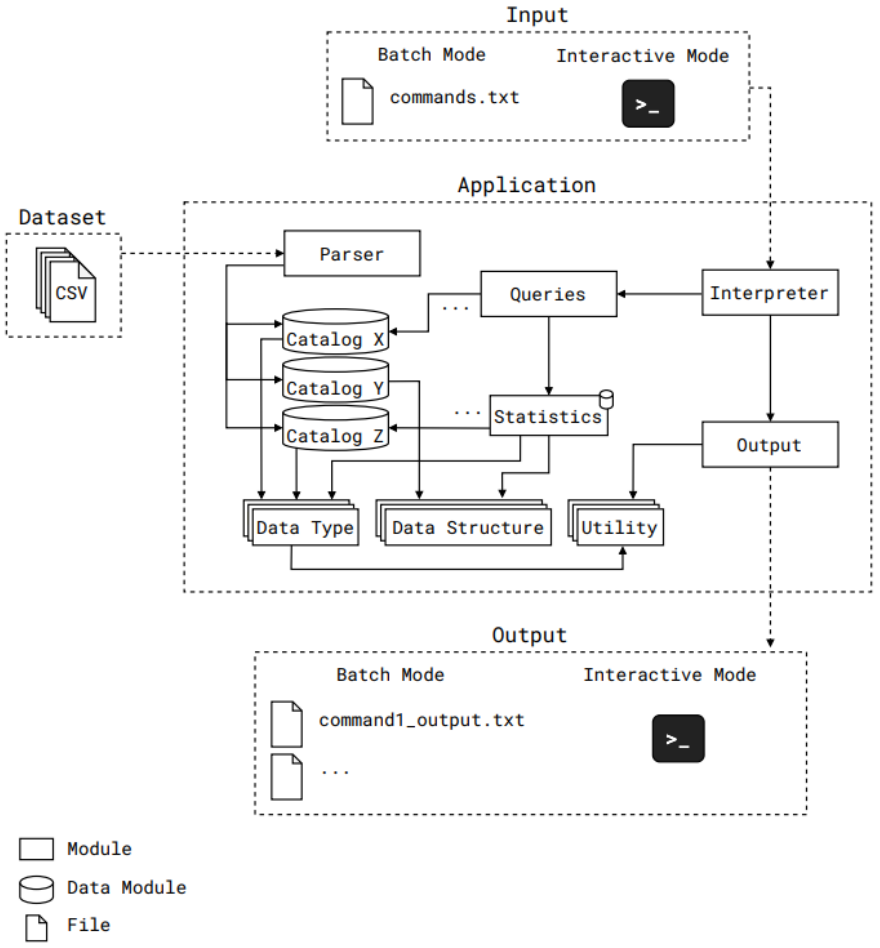

Fig. 1 - Arquitetura de referência para a aplicação a desenvolver

Estruturação

Em relação às estruturas utilizadas para os catálogos, tomámos diferentes decisões baseadas naquilo que consideramos ser vantajoso para o desenvolvimento do trabalho, nomeadamente para as *Queries* que definimos nesta fase do projeto.

```c
typedef struct catalog_users {
    GHashTable *users_hash;
} CATALOG_USERS;
```

```c
typedef struct catalog_passenger {
    GHashTable *passengers_hash;
} CATALOG_PASSENGERS;
```

```c
typedef struct catalog_reservations {
    GHashTable *reservations_hash;
    GHashTable *hotel_reservations;
    GHashTable *users;
}CATALOG_RESERVATIONS;
```

```c
typedef struct catalog_flights {
    GHashTable *flights_hash;
    GHashTable *airports_flights;
    GHashTable *users;
} CATALOG_FLIGHTS;
```

Fig. 2 - Estruturas do Catálogo de *users, passengers, reservations* e *flights*

Users

Para o catálogo dos *Users,* decidimos utilizar *hashtables,* tendo em conta que, como o nosso objetivo é procurar um determinado *user* a partir do seu *id,* este é o método mais eficiente que encontrámos. Para nos auxiliar, utilizámos a biblioteca *glib.*

Para além disto, e como se pode verificar na Fig. 2, adicionamos também novos parâmetros à estrutura dos *Users.*

```c
typedef struct user {
    char *id;
    char *name;
    char *email;
    char *phone_number;
    DATE birth_date;
    char *sex;
    char *passport;
    char *country_code;
    char *address;
    DATE_SECS account_creation;
    char *pay_method;
    enum account_status account_status;

    int age;
    int number_of_flights;
    int number_of_reservations;
    double total_spent;
} USER;
```

Fig. 3 - Estrutura *Users*

Passengers

Para o catálogo dos *Passengers* decidimos apenas utilizar o *id* do *Flight* e o *id* do *User*, esta estrutura visa facilitar o rastreamento preciso dos indivíduos que utilizam o serviço de voo, esta foi a melhor maneira que encontramos de combinar e organizar duas informações cruciais para rastrear passageiros.

```
typedef struct passenger {
    int flight_id;
    char* user_id;
}PASSENGER;
```

Fig. **4** - Estrutura *Passengers*

Reservations

Para o catálogo das *Reservations* utilizamos *hashtables* com o auxílio da biblioteca *Glib*. Esta escolha foi motivada pela necessidade de localizar reservas distintas por meio dos seus identificadores (*id*) de maneira semelhante ao que já foi utilizado. Decidimos incorporar variáveis adicionais à estrutura de reservas, as quais serão também devidamente atualizadas. Essa visa enriquecer a representação das *Reservations,* possibilitando um manuseio mais completo e eficiente dessas informações ao longo do código.

```
typedef struct reservation{
    char *id;
    char *user_id;
    char *hotel_id;
    char *hotel_name;
    int hotel_stars;
    int city_tax;
    char *address;
    DATE begin_date;
    DATE end_date;
    int price_per_night;
    char *includes_breakfast; //true or false
    char *room_details;
    int rating;
    char *comment;

    double total_cost;
    int nights;
} RESERVATION;
```

Fig. **5** - Estrutura *Reservations*

Hotéis e Airports

Os hotéis e aeroportos são estruturas criadas de forma a auxiliar o catálogo das *Reservations* e das *Flights*, respetivamente. Para além disso, os parâmetros destas estruturas também são atualizadas à medida que é adicionado uma reserva ou voo ao seu catálogo. Estas estruturas permitem responder mais facilmente a algumas das queries que implementamos.

```
typedef struct hotel_reservations {
    int rates_sum, rates_num;
    int reservations_total;
    int size;
    RESERVATION **reservations;
    GList *hotels_sorted;
} HOTEL_RESERVATIONS;
```

```
typedef struct airport {
    char *name;
    int num_passengers;
    GArray* delays;
    int size;
    int flights_total;
    FLIGHT **flights;
} AIRPORTS_FLIGHTS;
```

Fig.**6** - Estrutura *Hotels* e *Airports*

Flights

No caso dos *Flights,* voltamos a utilizar *hashtables* com o auxílio da biblioteca *Glib*. Isto porque de forma semelhante ao que foi feito na secção dos *Users* pretendemos encontrar os diferentes *Flights* através do seu *id,* pelo que a utilização destas *hashtables* nos pareceu ser a maneira mais vantajosa de o fazer.

Da mesma forma que fizemos com a estrutura de *Users* decidimos também adicionar variáveis à estrutura dos *Flights* que também serão atualizadas.

```
typedef struct flight {
    int id;
    char *airline;
    char *plane_model;
    int total_seats;
    char *origin;
    char *destination;
    DATE_SECS schedule_departure_date;
    DATE_SECS schedule_arrival_date;
    DATE_SECS real_departure_date;
    DATE_SECS real_arrival_date;
    char *pilot;
    char *copilot;
    char *notes;

    int delay;
    int numPassengers;
    GList *passenger_user_ids;
} FLIGHT;
```

Fig.**7** - Estrutura *Flights*

Validator

Para criar os elementos de cada catálogo tínhamos de percorrer quatro diferentes ficheiros de entrada, cada um com informação para criação dos diferentes tipos de dados constituintes de cada catálogo. No entanto, nem todos os dados nestes ficheiros eram entradas válidas de acordo com um conjunto de validações a executar.

Assim, para validar os *Users, Passengers, Reservations e Flights*, de forma a certificarmo-nos de que não guardamos dados errados em nenhum dos catálogos, criámos funções, guardando estas no ficheiro *validator.c* e de forma a "reutilizar" código entre catálogos, sendo que estas funções funcionam para todos os tipos de catálogo que desenvolvemos. Estas funções são então chamadas quando ocorre a criação dos dados e caso estes não sejam considerados válidos são descartados sem ser adicionados ao catálogo.

```
enum account_status verify_AccountStatus (char* token);

int verify_stars(char *token);

int verify_price(char *token);

char *verify_breakfast (char *token);

char *verify_countryCode(char *token);

int verify_rate (char *token);

int verify_date(char *token);

int verify_date_secs(char*token);

int verify_email(char *token);

int verify_airport(char *code);

#endif //VALIDATOR_H
```

Fig.8 - Funções presentes no ficheiro *validator.c*

Para as datas usadas nas diferentes estruturas decidimos criar duas diferentes *structs* pois precisávamos de uma que nos indicasse apenas o dia, o mês e o ano e outra que nos indicasse, para além disso, ainda as horas, os minutos e os segundos. Esta estrutura facilita a gestão de datas e horários, proporcionando uma organização mais clara e flexível para lidar com diferentes contextos e melhorando a legibilidade do código.

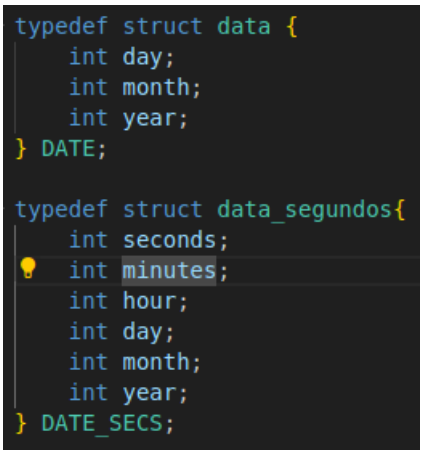

Fig 8 - *Structs DATE* e *DATE_SECS*

Queries

Após a estruturação do trabalho, e como objetivo definido no enunciado do projeto, criámos uma série de *queries* com o propósito de responder a determinadas questões propostas de modo a avaliar e validar o funcionamento e a eficiência do armazenamento e gestão da informação em memória. Nesta primeira fase do trabalho decidimos desenvolver, como era proposto, seis *queries* propostas: as *queries* 1, 2, 3, 4, 8 e 9.

Query I

Descrição: Listar o resumo de um utilizador, voo, ou reserva, consoante o identificador recebido por argumento.

Resposta: Para resolver esta query, começamos por verificar se o identificador é uma *flight*, *reservation* ou *user*, analisando se este é constituído apenas por números, se começa por "Book" ou outro, respetivamente. Depois deste processo, usamos a função da *glib g_hash_table_lookup*, encontramos a entidade correspondente na *hashtable* do catálogo correspondente através do seu identificador e guardamos todas as informações necessárias no formato definido e é feita a impressão dessas informações no ficheiro de output.

Query II

Descrição: Listar os voos ou reservas de um utilizador, se o segundo argumento for fights ou reservations, respetivamente, ordenados por data (da mais recente para a mais antiga). Caso não seja fornecido um segundo argumento, apresentar voos e reservas, juntamente com o tipo (flight ou reservation).

Resposta: Para a resolução da Query 2 a função Q2 é chamada com um catálogo e um ID como argumentos. Esta função, por sua vez, chama a função *flights_reservations_id_date_type*. A função *flights_reservations_id_date_type* inicialmente verifica se o usuário está inativo. Se estiver, retorna *NULL*. Caso contrário, a função com a ajuda de duas funções auxiliares (*get_flights_with_passenge*r e *get_reservations_with_id*) cria duas listas, uma delas com os voos que incluem o passageiro com o ID fornecido e outra de reservas que também incluem o ID. Em seguida, a função aloca memória para um array de estruturas *ID_DATE_TYPE*, que guardará as informações necessárias para a Query, um ID, uma data e um tipo (*reservation* ou *flight*). A função percorre a lista de voos, aloca memória para a nova estrutura *ID_DATE_TYPE* para cada voo, preenche os campos da estrutura e adiciona-a a um array. O mesmo processo é repetido para a lista de reservas. Finalmente, a função ordena o array de estruturas *ID_DATE_TYPE* por data e retorna o array como resultado final. A função *printQ2* é então chamada para imprimir as informações do array.

Query III

Descrição: Apresentar a classificação média de um hotel, a partir do seu identificador.

Resposta: Para resolver esta query verificamos inicialmente se o id do hotel era válido em seguida utilizamos a *g_hash_table_lookup* para obtermos a estrutura de reservas associada ao hotel específico é então calculada a média das avaliações do hotel. Posteriormente, é usada a *snprintf* para converter o valor de *int* para *char* para depois ser dado o output corretamente.

Query IV

Descrição: Listar as reservas de um hotel, ordenadas por data de início (da mais recente para a mais antiga). Caso duas reservas tenham a mesma data, deve ser usado o identificador da reserva como critério de desempate (de forma crescente).

Resposta: Para resolver esta query, primeiramente verificamos se o hotel especificado existe e se possui pelo menos uma reserva, em seguida organizamos as reservas utilizando a função *qsort*, cada reserva é formatada e armazenada em uma string, que é então atribuída a um array de strings (*result*), quando terminadas todas as reservas o array *result* é retornado para depois ser dado o devido output.

Query VIII

Descrição: Apresentar a receita total de um hotel entre duas datas (inclusive), a partir do seu identificador. As receitas de um hotel devem considerar apenas o preço por noite (price_per_night) de todas as reservas com noites entre as duas datas.

Resposta: Para resolver esta query começamos por converter as *string* para *DATE* usando a função *convertStringToDate*, em seguida, ela itera sobre as reservas associadas ao hotel especificando e calcula o total de receitas considerando apenas as reservas que se enquadram no intervalo de datas fornecido. O total de receitas é acumulado e é convertido para uma string usando *sprintf*, a string acaba então por ser retornada para depois ser dado o devido output.

Query IX

Descrição: Listar todos os utilizadores cujo nome começa com o prefixo passado por argumento, ordenados por nome (de forma crescente). Caso dois utilizadores tenham o mesmo nome, deverá ser usado o seu identificador como critério de desempate (de forma crescente). Utilizadores inativos não deverão ser considerados pela pesquisa.

Resposta: Para resolver esta query, primeiramente verificamos se o *prefix* é nulo, caso não seja prosseguimos com a função, usando a função *g_hash_table_iter_init* e verifica se o nome de cada usuário tem o prefixo especificado e se o status da conta é ativo, a lista é depois ordenada usando a função *compareUsers* é então preenchida uma string com a lista de resultados que é então retornada para ser dado o devido output.

Testes Funcionais

Para validar os resultados e assegurar o correto funcionamento do nosso projeto, desenvolvemos um executável de testes chamado *teste.c*. este executável opera de maneira semelhante ao programa principal, com a distinção de que compara os resultados obtidos com os resultados esperados, fornecidos pela equipa docente da disciplina. este processo é fundamental para garantir a precisão e confiabilidade do nosso software em conformidade com as expectativas estabelecidas.

Conclusão

Em suma, consideramos que a primeira fase deste projeto ocorreu de forma satisfatória, conseguindo implementar com sucesso as seis *queries* propostas. Além disso, achamos que conseguimos deixar preparado uma estrutura de catálogos que será capaz de suportar um grande volume de dados, proporcionando uma base sólida para a etapa seguinte.

No entanto, reconhecemos que há espaço para melhorias e refinamento. Enfrentamos dificuldades significativas no aspecto da validação de dados de forma a garantir a integridade e consistência das informações que não conseguimos implementar a 100% e a redução dos *memory leaks*. À medida que nos preparamos para a segunda fase, é imperativo abordar as limitações

atuais e tentar ultrapassá-las.

Em resumo, embora tenhamos enfrentado desafios substanciais, pretendemos direcionar os nossos esforços para melhorias significativas. A próxima fase será uma oportunidade crucial para abordar essas questões e elevar a qualidade geral do nosso projeto.