Universidade do Minho
Escola de Engenharia

LABORATÓRIOS DE INFORMÁTICA lll

GRUPO 96

TRABALHO REALIZADO POR:

A100754 – Rafael Peixoto

A95485 – Miguel Carvalho

A100711 - João Rodrigues

INTRODUÇÃO

O principal objetivo deste projeto é o bom uso do conceito de modularidade e a aplicação do encapsulamento de dados.

O encapsulamento de dados é um mecanismo onde a implementação dos detalhes de uma classe é mantida em segredo dos outros módulos garantindo que a inexistência de um vazamento da estrutura de dados ou dos dados em si.

As vantagens do encapsulamento são a própria ocultação de dados/ implementações. Uma maior flexibilidade nas restrições nas estruturas de dados e existe uma maior facilidade na adaptação/mudança a novos requerimentos.

Modularidade consiste na subdivisão do programa em subprogramas separados, o que permite uma maior facilidade na alteração de módulos sem que seja preciso fazer alterações nos restantes.

Como utilizamos estes dois conceitos no nosso projeto?

Para tirar proveito do conceito de modularidade utilizamos os ficheiros (“user.c”, ”driver.c” e ”ride.c”) para o armazenamento de dados específicos a um membro singular para cada um destes campos e funções para a gestão dos mesmos membros.

Utilizamos os ficheiros (“users.c”, “drivers.c” e “rides.c”) para o armazenamento do conjunto dos membros singulares (sem conhecer a sua estrutura interna) e funções para a gestão destes catálogos de dados.

Utilizamos o programa “stats.c” para a coleção geral dos catálogos de dados e funções de gestão necessárias aos mesmos. Note que o programa “stats.c” não conhece os tipos dos catálogos que são guardados garantindo o encapsulamento.

ESTRATÉGIA

“main.c”:

Este ficheiro é responsável pela geração do programa executável principal (“programa-principal”). E utiliza os seguintes módulos para a realização do mesmo:

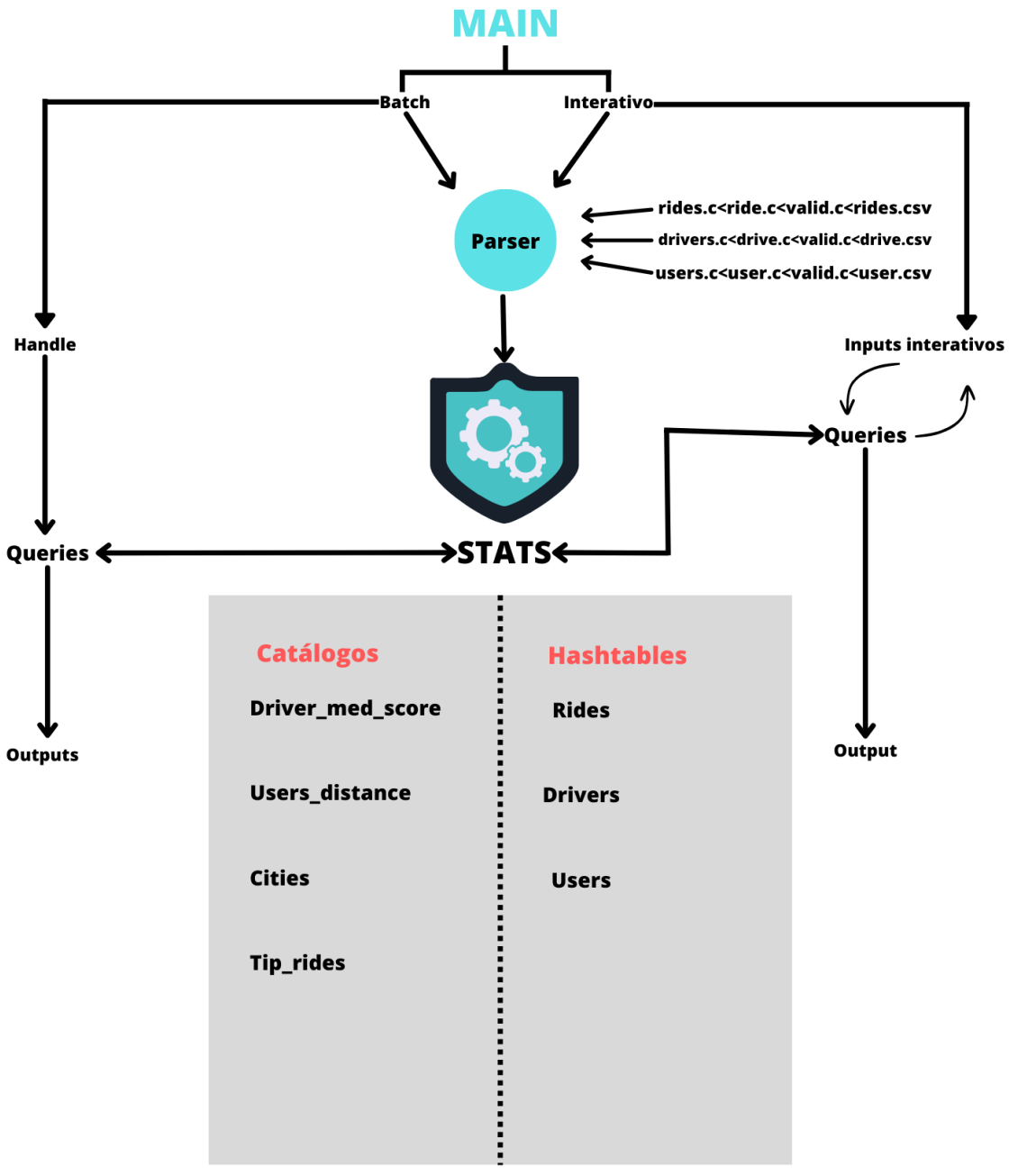
• **“parser.c”**: recebe como argumento a localização da diretoria que contém os ficheiros .csv e retorna o conjunto de catálogos de dados no tipo Stats;

• **“batch.c”**: recebe a localização do input.txt e a Stats retornada pelo “parser.c” e é responsável pelo parsing de comandos (através do “handle.c”), execução (“queries.c”) e escrita dos outputs das queries (função write_output localizada em “queries.c”).

• **“interact.c”**: no modo iterativo o utilizador usa o programa em via cli, é pedido-lhe para meter a localização do dataset e as queries são executadas via cli.

• **“valid.c”**: módulo que contem as funções de validação de dados.

ARQUITETURA DO PROJETO



VALIDACÃO DE DADOS

Nesta segunda fase do projeto foi adicionado um novo objetivo, a validação dos dados para os ficheiros csv.

A nossa estratégia foi adicionar funções de validação (valid.c) para verificar se uma dada cela é valida, se for é adicionada ao catálogo respetivo, se não é ignorada.

QUERY 1

Para a query 1 primeiro fazemos uma triagem do input, vê se o input é ou não um id, se for um id sabemos que queremos obter informações relativas a um condutor se não for queremos informações do utilizador.

Se estivermos a tratar de um condutor utilizamos funções auxiliares que iram buscar informações sobre esse condutor (como por exemplo get_driver_name, get_driver_age etc) que garantiram o encapsulamento de dados e damos output desses dados.

Se estivermos a tratar de um utilizador fazemos o mesmo processo do que como se fosse para um condutor, mas com funções auxiliares relativas ao utilizador.

QUERY 2

Para a query 2 criamos uma lista de Drivers (sorted_drivers_med_score) ordenada por menor avaliação media, em caso de empate aparecem primeiro os que têm a viagem menos recente, se o empate persistir aparecem primeiro os que têm id menor. Esta lista será guardada nas stats, assim sempre que a query for chamada apenas temos de ir buscar os últimos N condutores dessa lista, e esse será o nosso output.

QUERY 3

Na query 3 o objetivo é determinar os N users com a maior distância percorrida.

O método de pensamento desta query foi criar um catálogo para os users onde se ia aumentado a distância e calculando também a viagem mais recente à medida que se faz o parse das rides. Depois fazemos um quicksort para ordenar esse catálogo, em caso de empate na distância total usa-se o a viagem do user mais recente também calculada durante o parse das rides para desempatar, caso também empate usa-se o username para desempatar. Na query vai se apenas buscar os N primeiros users ao catálogo.

QUERY 4

Para a query 4 criamos um catálogo de cidades onde cada cidade terá o preço médio das viagens feitas nessa cidade que será colocado nas stats.

Assim quando a query 4 for chamada só precisamos de usar a função get_med_price_by_city que retornará o que queremos como output. Como essa informação está nas stats não precisamos de iterar as viagens todas sempre que a query for chamada.

QUERY 5

Para a query 5 iteramos todas as viagens, vemos se a viagem em questão se encontra entre a <dataA> e a <dataB> , se se encontrar acrescentamos o preço à variável total_price e acrescentamos 1 à variável nr_rides. No final de iterarmos todas as viagens guardamos na variável price_med a divisão do total de preço pelo número total de viagens e damos output a esse resultado.

QUERY 6

Para a query 6 iteramos todas as viagens, vemos se a viagem em questão se encontra entre a <dataA> e a <dataB> e se foi feita na cidade <city> , se se sim, acrecentamos a distância dessa viagem à variável distancia_total e acrescentamos 1 à variável nr_rides. No final de iterarmos todas as viagens guardamos na variável dist_media a divisão do total de distância percorrida pelo número total de viagens e damos output a esse resultado.

QUERY 7

Na query 7 o objetivo é determinar os N drivers com a melhor avaliação de uma certa cidade. O método de pensamento desta query foi quando se faz o parse das rides verifica-se se o driver está ativo se estiver coloca-se o driver no catálogo da cidade da ride na estrutura driver_aval. A estrutura driver_aval irá receber o id do driver, a avaliação da ride e número de viagens que esse driver fez nessa cidade. Quando se insere o driver na cidade se ele já existir vai-se apenas acrescentar a avaliação total e aumentar o número das rides do driver_aval. Se não existir cria-se um driver_aval novo. Depois de se fazer o parse das rides completo será feito um quicksort de todos os drivers_aval para todas as cidades ordenando-os pela avaliação média. Em caso de empate será ordenado pelo id do driver de forma decrescente. Na query apenas vai-se buscar os N melhores drivers_aval no catálogo da cidade em questão. Vai se buscar o nome do driver à hash table com o id do driver.

QUERY 8

Para a query 8 criamos duas listas, uma de homens e uma de mulheres durante o parse das viagens (estas serão inseridas na struct rides com o nome “male” e “female”), ambas estas listas estão ordenadas de forma que as contas mais antigas apareçam primeiro, mais especificamente, ordenar por conta mais antiga de condutor e, se necessário, pela conta do utilizador. Se persistirem empates, ordenar por id da viagem (em ordem crescente).

Quando é chamada a query 8 iteramos a lista que for necessária (males ou females) e se tanto o utilizador como o condutor tiverem mais idade do que o input dado acrescentamos a informação do id e do nome tanto do condutor como do utilizador à variável result que será o nosso output.

QUERY 9

Para a query 9 criamos uma lista de Rides (só com as que o passageiro deu gorjeta) ordenada por distância percorrida, no caso de ser a mesma, viagens mais recentes aparecem primeiro, se o empate persistir aparecem primeiro as viagens com id maior. Esta lista (sorted_tip_rides) é criada conforme dá o parse às viagens e é embutida nas stats. Assim não é preciso iterar as rides todas sempre que é chamada a query 9.

Quando a query 9 é chamada iteramos essa lista que está nas stats e vemos se a viagem se encontra entre a <dataA> e a <dataB>, se estiver acrescentamos a mesma à variável result que será responsável pelo output.

DIFICULDADES E MELHORIAS QUE PODIAM TER SIDO FEITAS

Uma das maiores dificuldades que tivemos neste projeto foi a implementação de funções que libertam espaço para as estruturas de dados quando realizamos a cópia das estruturas (através da função strdup), fazendo assim com que o nosso código tivesse bastante data leaks.

Outra grande dificuldade que sentimos está novamente relacionada à memória, desta vez não a libertação, mas sim o pico de memória a ser usada. Como criamos vários catálogos enquanto estamos a dar o parse de dados este pico tornou-se maior do que o disponibilizado pela máquina de testes automáticos, o que fazia as queries serem feitas nos nossos computadores, porém não passarem nos testes automáticos.

Uma coisa que poderíamos ter melhorado era fazer com que tanto nas queries 5 como na 6 não tivéssemos que iterar todas as viagens sempre que as queries são chamadas, fazendo assim com que o nosso código fosse mais rápido. Uma solução para isso seria usar balanced binary trees para diminuir o número de comparações, obtendo O(log_2(n)) em vez de O(n) sendo n o número de rides.