Relatório Guião 2- Laboratórios de Informática III

Este documento tem como propósito explicitar o raciocínio presente na resolução dos desafios que nos foram propostos para o Guião 2. Procederemos, então, através da divisão por secções correspondentes a cada um dos módulos do nosso programa, à identificação e explicitação das estratégias que foram seguidas pelo grupo bem como as limitações que fomos encontrando ao longo da resolução.

USER:

De forma a recolher os dados do ficheiro "users-g2.csv", criamos o TypeDef **USER** que é representado por uma estrutura chamada user cujo o propósito é representar os tipos de dados que existem numa linha do ficheiro, referentes, portanto, a um só utilizador e posteriormente utilizamos um array de USER, para representar efetivamente os dados de todo o ficheiro referido.

Começamos, então, por criar uma função denominada ***start_users***, que nos permite a acumulação dos dados necessários (o id, login, type, followers, followers_list, following, following_list) para responder às queries. Para, de facto, adicionar os dados ao array mencionado utilizamos, inicialmente, a função ***associa_linha_users*** que converte uma linha no tipo USER e acrescenta o valor à posição final do array na função ***set_value_user***.
Em termos de alocação de memória, invocamos a função ***init_users*** (aloca memória para 30 elementos do tipo USER para uma estrutura USER) e ***realloc_users*** (recebe uma estrutura do tipo USER* e um inteiro n de 4 bytes e realoca 30*n elementos do tipo USER para a estrutura, realocando, assim, o array de forma a torná-lo dinâmico).
Com o objetivo específico de providenciar dados para a resposta das queries, esta função, à medida que lê e acumula os dados, também executa algumas outras tarefas importantes:

1. Armazena num um array os ids dos "amigos" de um dado id providenciado(a junção dos ids da sua *followers list* com os *ids* da sua *following list*) - necessário para a resposta à query 9.
2. com o auxílio da função ***incrementaTypes***, fizemos com que fosse possível verificar qual o tipo de utilizador ("Bot"; "Organization"; "User") presente em cada linha e assim com apenas uma leitura do ficheiro conseguimos saber ,também o total de utilizadores de cada tipo - necessário para a resposta à query 1.
3. no caso de estarmos efetivamente perante um utilizador do tipo "Bot" adicionamos o seu id a um array - necessário para a resposta à query 3.

De maneira a resolvermos as queries, foi também criada a função ***complete_triplo*** necessária para a resposta à query 5, query 6 e query 9.

A fim de administrarmos a memória de forma dinâmica, criamos também a função ***free_user*** que liberta a memória previamente alocada para todo o array, com o auxílio da função ***free_elem_user***.

COMMIT:

Com o intuito de armazenar os dados do "commits-g2.csv", criamos o Typedefs COMMIT que representa uma linha (ou seja um único commit). O Typedef COMMIT é representado por uma estrutura que chamamos commit, essa estrutura contém 3 inteiros de 4 bytes (integers) representativos dos campos repoId, authorId e commiterId da coletânea de commits, um elemento do tipo DATA, representativo da data do commit, 1 string chamada message que representa a mensagem do commit e por fim um inteiro não

negativo de 2 bytes (unsigned short int) onde armazenamos o tamanho, em caracteres, da mensagem do commit.

De modo a conseguir então efetivamente armazenar os dados do referido ficheiro na estrutura que fizemos, decidimos trabalhar com arrays de elementos da estrutura COMMIT, e para tal criamos a função ***start_commits***. Esta função utiliza outras funções como auxiliares de modo a facilitar este processo, no que toca a alocamento de memória, a primeira destas é a função ***init_commits*** que aloca memória para 30 elementos do tipo COMMIT, e quando necessário utiliza a função ***realloc_commits***, que recebe um array de estruturas COMMIT e um apontador para um inteiro de 4 bytes de modo a armazenar espaço para mais 30 elementos COMMIT a cada invocação, tornando este array dinâmico, já no quesito associar linhas do ficheiro "commits-g2.csv" aos campos de memória que alocamos, utilizamos a função auxiliar ***set_value_commit*** que através da função ***associa_linha_commit*** em que na qual se converte uma string do ficheiro em um elemento do tipo COMMIT e de um inteiro que representa o tamanho do array de COMMITS vamos associando esse valor ao fim do array de COMMIT e incrementamos uma unidade no comprimento do array.

Criamos também no ficheiro commits.c funções que retornam os valores de cada campo de um commit de modo a facilitar o nosso projeto: ***return_author_id*** (devolve o id do autor/dono do repositório em que o commit); ***return_repo_id*** (devolve o id do repositório onde o commit é feito); ***return_data*** - devolve a data do commit; ***return_commiter_id*** - devolve o id do utilizador que fez o commit; ***return_size_message*** - devolve o tamanho da mensagem do commit.

É importante destacar que elaboramos também formas de ordenar commits por via de quicksort (função ***qsort*** da ***stdlib***) dependendo de parâmetros distintos, nomeadamente da data de realização do commit e do tamanho da mensagem do commit, o que resulta na utilização de diferentes funções de comparação ao invocar ***qsort***, ***compare_datas_commit***, e ***compare_commit***, respetivamente.

Outras funções auxiliares para ajudar a resolver as queries no commits.c são:

- ***bot_repos*** - calcula o número de repositórios com commits feitos por bots;
- ***return_size_commits*** - devolve o número de commits
- ***return_colaborators*** - devolve o número de colaboradores
- ***commits_qtty*** - devolve a quantidade de commits feitos por um utilizador num intervalo de datas
- ***commiter_id_repos*** - cria um array de TRIPLO com a informação necessária para responder à query 6
- ***not_updated_repos*** - insere num array todos os repositórios que não têm commits depois de uma data
- ***updated_repos*** - insere num array todos os repositórios que têm commits depois de uma data
- ***triplos_parametrizavel*** - cria um array de TRIPLO onde guarda a informação para a query 9

Para libertar a memória que alocamos, criamos também a função ***free_commit*** que liberta a memória alocada para todo o array, com o auxílio da função ***free_elem_commit.***

REPO:

Para armazenar os dados do ficheiro "repos-g2.csv", criamos uma struct chamada REPO, em que a mesma representa os dados necessários para cada linha do ficheiro

referido e depois para representar todos os dados do tal ficheiro, tivemos por base usar um array de REPO.

Esta estrutura contém um inteiro de 4 bytes, para guardar o id do repositório, e 2 strings, uma em que guarda a linguagem do repositório e outra que guarda a descrição do mesmo.

Para termos os dados disponíveis, criamos uma função ***start_repo*** , que lê linha a linha do ficheiro e vai começar a acumular os dados necessários para conseguirmos responder às queries. Estes dados são o id do repositório, a descrição do repositório, a linguagem. Para então adicionarmos elementos ao tal array, primeiro criamos uma função denominada ***associa_linha_repo*** que converte a linha num REPO e depois adiciona este elemento na última posição do array na função ***set_value_repo*** . Para gerirmos obtermos uma memória dinâmica, também foi criada uma função chamada ***free_repo*** que dá free à memória alocada para todo o array, incluindo as strings associadas a cada elemento, usando a função ***free_elem_repo*** . Para acedermos aos campos do array REPO criamos 3 funções, em que cada uma retorna um elemento do array dado o índice a que queremos aceder. As 3 funções são:

- ***return_repo*** - retorna o campo “id” dado o índice do array e o array.
- ***return_desc*** - retorna uma cópia do campo “desc” dado o índice do array e o array.
- ***return_language*** - retorna uma cópia do campo “language” dado o índice do array e o array.

Neste módulo também foram criadas várias funções de auxílio para responderem ou ajudarem a responder às queries 6,7 e 8.

Para a query 6, foi criada uma função chamada ***repo_language*** que retorna um array de inteiros, com todos os ids dos repositórios, cuja linguagem associada seja igual à indicada por parâmetro.

Para a query 7, foi criada uma função denominada ***repo_no_updates*** que recebe um array de inteiros com todos os ids de repositórios que não tem commit até uma certa data, e esta função pega nesse tal array e cria um array de TUPLO com todos os repos ids mais a descrição associada ao repositório usando a função ***add_elem_tuplo*** para adicionar completar o array de TUPLO.

Para a query 8, foram criadas duas funções, ***repo_inf_data*** e ***repo_language_qtty***. A primeira função cria array de strings, definida no módulo stringarrays.c, e dado um array de inteiros com todos os ids de repositórios que tem commits a partir de uma certa data e em vai adicionando strings ao tal array de strings, em que cada string corresponde a uma linguagem. A segunda função depois de dado o tal arraystrings, invoca uma função chamada ***strarray_language_qtty*** definida no strarrays.c que retorna um array TUPLO e por fim, a função ***repos_language_qtty*** retorna este mesmo array TUPLO.

DATA:

Para nos ajudar na avaliação das datas, criamos um Typedef *DATA. Este tipo de dados representa um apontador para um zona de memória que contém uma estrutura de dados chamada data, que contém 6 inteiros não negativos de 2 bytes.

Para trabalhar com este tipo de dados criamos várias funções, que se encontram no data.c. A 1ª função chama-se ***init_data*** que aloca memória para 1 elemento do tipo *DATA. Também criamos uma função, denominada ***string_to_data*** que dada uma string vai lê-la de forma a separar os tais 6 inteiros e assim armazená-los na nossa estrutura de dados. Outra função chama-se ***string_to_ano*** que faz a mesma coisa que a função anterior, só que a

string só vem com 3 inteiros. 2 funções também muito similares são a ***dif_datas*** e ***dif_ano*** que recebem 2 elementos do tipo *DATA e devolve 1 se a 1ª for inferior à 2ª e 0 caso contrário. ***copy_data*** é uma função que criamos de modo a copiar os dados de um apontador de data para outro apontador do mesmo tipo. Por último, a função ***free_data*** é uma função que dá free à memória alocada inicialmente para uma dada data.

AUX:

Decidimos que como ao longo do projeto encontramos problemas comuns a vários módulos era pertinente a criação de um módulo em que escreveríamos funções auxiliares genéricas, que não trabalhassem com estruturas criadas por nós.

Neste módulo encontram-se então as seguintes funções:

- ***compareInts*** - função de comparação de dois pointers de inteiros;
- ***size_array*** - função que retorna o tamanho de um array de inteiros;
- ***pesquisaBinaria*** - função que faz procura binária de um elemento num array de inteiros;
- ***name_newfile*** - função que determina o nome de um novo ficheiro com a resposta a uma query;
- ***str_array*** - função de conversão de uma string *char para formato array de inteiros;
- ***concatena_arrays*** - função que concatena dois arrays mais pequenos num array maior ordenado;
- ***copyArray*** - função que copia o conteúdo de um array para um novo array.

ABIN:

Reparamos que em certos casos, podíamos em vez de usar um array que eventualmente resulte em algoritmos de tempo de execução linear ou quadrático, era mais vantajoso utilizar árvores binárias que resultem em algoritmos de tempo logarítmico. Então colocamos funções genéricas que trabalham em árvores binárias, estas funções iniciam árvores, procuraram elementos, adicionar elementos no ficheiro abin.c.

Este ficheiro contém um typedef *BTREE que corresponde à estrutura btree representativa de uma árvore binária que contém os campos nodo, que é um inteiro de 4 bytes que armazena o conteúdo de um nodo da árvore, e duas sub-árvores, BTREE esq e BTREE dir.

TUPLO:

Para nos ajudar a responder a certas queries criamos este módulo que contém uma estrutura com o mesmo nome. Esta estrutura contém um inteiro e uma string associada.

Nós para trabalharmos com este módulo tivemos por base trabalhar com arrays de TUPLO. Cada elemento deste array contém 1 inteiro de 4 bytes e uma string.

Definimos 2 funções que trabalham com a alocação de memória deste array, a ***init_tuplo*** que aloca memória para um array TUPLO de 30 posições e a ***realloc_tuplo*** que realoca a memória antiga de forma a aumentar a memória alocada para o tal array.

Para adicionar elementos este array, definimos 2 formas, uma que simplesmente adiciona e outra que vai adicionar só se o str a que queremos adicionar já existe em algum elemento desse array TUPLO. Relativamente à primeira, criamos uma função chamada ***add_elem_tuplo*** que verifica se é preciso realocar o array e caso não, adiciona então o elemento na última posição. A outra função que se chama ***add_tuplo***, primeiro invoca uma

outra denominada de ***exist_language_tuplo*** em que vai percorrer o array de forma linear e caso encontre uma string associada a um elemento for igual à string que queremos adicionar ao TUPLO retorna o índice onde ocorre e caso não ocorra devolve -1. Caso esta função que acabei de falar devolver -1, invoca a função ***add_elem_tuplo*** e adiciona um elemento no final, mas caso devolva um índice, acedemos diretamente a esse índice e incrementamos o inteiro associado uma unidade.

Uma forma de depois termos o array TUPLO ordenando decrescente em relação ao inteiro associado a cada elemento, criamos uma função que utiliza a função pré-definida *qsort* e criamos uma função auxiliar, ***cmpfunc_tuplo***, que compara 2 pointer de TUPLO e devolve a diferença dos inteiros associados a cada um dos TUPLOs.

Para ajudar depois quando queremos aceder aos campos de um elemento do array de TUPLO criamos 2 funções em que cada uma retorna um campo do TUPLO.

- ***return_tuplo_int*** - que retorna o inteiro associado a um tuplo que está no array.
- ***return_tuplo_str*** - que retorna a string associada a um tuplo que está no array.

Criamos também uma função para dar free a ao array TUPLO e a todos os elementos associado a cada TUPLO.

Para terminar, neste módulo criamos 2 funções que vão adicionar conteúdos aos ficheiros de forma diferentes. A primeira chama-se ***add_content_tuplo*** que adiciona ao ficheiro só os campos string associada a cada elemento do array TUPLO. A outra função chamada ***add_content_tuplo2*** que adiciona ao ficheiro ambos os campos associados a cada elemento do array TUPLO, no caso um inteiro e uma string.

TRIPLO:

Semelhante a outros módulos do projeto, este módulo tem como objetivo trabalhar com uma estrutura que criamos para responder a certas queries.

Esta estrutura é definida por um Typedef, cujo nome é TRIPLO. Este tipo de dados é constituído por uma string, login, e dois números, id e qtty, ordenados da seguinte forma: id, login, qtty, e utilizaremos arrays dinâmicos desta estrutura para escrever as respostas. Dependendo da query em questão, o campo qtty pode corresponder à quantidade de commits feitos num dado intervalo, ou à quantidade de caracteres da mensagem do commit.

As funções deste módulo são:

- ***init_triplo*** - função que inicializa um array dinâmico de TRIPLO;
- ***realloc_triplo*** - função que realoca memória para um array dinâmico de TRIPLO;
- ***add_elem_triplo*** - função que adiciona um elemento a um array de TRIPLO;
- ***cmpfunc_triplo*** - função que devolve a subtração associada a dois triplos diferentes;
- ***sort_triplo*** - função que utiliza ***quicksort***, para um array de TRIPLO, ordenando de forma decrescente o array com base no valor de qtty associado a cada triplo;
- ***return_id_triplo*** - função que retorna o inteiro correspondente id de um triplo;
- ***return_login_triplo*** - função que retorna uma cópia da string correspondente ao login de um triplo;
- ***return_qtty_triplo*** - função que retorna o inteiro correspondente qtty de um triplo;
- ***set_qtty_triplo*** - função que altera o valor do inteiro correspondente qtty de um triplo;
- ***set_str_triplo*** - função que altera a string correspondente ao login de um triplo;

- ***free_triplo*** - função responsável por libertar a memória de um array de TRIPLO.

Para terminar este módulo, criamos 2 funções que vão adicionar conteúdos aos ficheiros de forma diferentes. A primeira chama-se ***add_content_triplo*** que adiciona ao ficheiro todos os campos do TRIPLO, isto é, 2 inteiros e 1 string, de o array de TRIPLO. A outra função chamada ***add_content_triplo2*** que adiciona ao ficheiro os campos "id" e "login" associados a cada elemento do array TRIPLO, ou seja um inteiro e uma string.

QUADRUPLO:

Este módulo foi criado para auxílio da query 10. Este ficheiro contém uma estrutura com 3 inteiros de 4 bytes e uma string. Como já foi referido este módulo é um auxílio para a query 10, em que cada inteiro representa, um id de um user, a sua maior mensagem num dados repositório e o id do respectivo repositório. A string simboliza o login do usuário. Este módulo é muito similar ao módulo triplo e todas as funções neste módulo são iguais às que estão no módulo triplo, só que trabalham com QUADRUPLO em vez de TRIPLO.

STRARRAY:

Criamos este módulo para nos servir como módulo auxiliar em algumas queries. Este módulo tem por base uma estrutura de dados que vai representar um array de strings, ou então, uma matriz de char 's.

Para isto, começamos por criar 2 funções que trabalham na gestão de memória desta matriz, uma que aloca memória e outra que realoca. As funções são *init_strarray* e *realloc_strarray* respetivamente.

Outras funções definidas neste módulo:

- ***add_elem_strarray***, função que adiciona uma string ao tal array de strings.
- ***return_str_strarray***, função que devolve uma cópia da string numa certa posição do array.
- ***free_strarray***, função que dá free a todos os elementos do array de strings.
- ***str_language_qtty****,* função auxiliar da query 8. Esta função primeiramente cria um array de TUPLO e depois percorre todo o array de strings e em cada elemento utiliza a função add_tuplo para adicionar ou não a string na posição i do array de strings ao array de TUPLO.

COMMESSAGE:

Módulo que contém uma estrutura que contém um array de quadruplos. Esta estrutura é denominada COMMITMESSAGE e ela serve de auxílio para a query 10. Nós reparamos que na query 10 simplesmente pedia o top N de um array de TRIPLO no nosso caso, então decidimos criar uma estrutura que armazena o tal array triplos e quando for invocada a query 10, não temos que estar a calcular nada novamente.

Para isto definimos as várias funções:

- ***startcommitmessage***, função que vai buscar o tal array de TRIPLO para responder à query 10.
- ***free_cm***, função que dá free à estrutura definida neste módulo e ao array de TRIPLO associado.
- ***add_content_cm***, função que adiciona conteúdo a um ficheiro de uma estrutura COMMITMESSAGE.

Relatório Guião 2- Laboratórios de Informática III

TOTALSTATS:

Com o intuito de responder às queries estatísticas que necessitavam de valores relacionados com o total de certos tipos de dados criamos a typedef *TOTALSTATS, que é representada pela estrutura total_stats que contém 5 inteiros de 4 bytes que se referem ao número total de utilizadores do tipo bots, ao total de colaboradores, ao total de repositórios, ao total de commits e ao total de utilizadores existentes num ficheiro, respetivamente e um array designado bots_array_ids que armazena os ids dos utilizadores do tipo "Bot".

Começamos por inicializar esta estrutura através da função ***init_total_stats*** que aloca memória para a struct total_stats, inicializa os campos do tipo inteiro a 0 e aloca memória para o bots_array_ids.

Em seguida, começamos por preencher o bots_array_ids utilizando a função ***add_elem_to_array*** que é invocada na função ***start_user*** do módulo user.c.

Para obter o valor dos campos restantes desta estrutura, utilizamos a função ***set_total_stats****,* que, ao ser invocada no módulo guiao2.c, define o valor dos campos com o auxílio das seguintes funções: ***set_colaboradores****,* ***set_commits****,* ***set_repos****,* ***set_user*** e ***set_bot_ids***.

De forma a controlarmos a memória de forma dinâmica, criamos também a função ***free_TotalStats*** que liberta a memória previamente alocada para todo o TOTALSTATS.

TYPESSTATS:

Para conseguirmos responder às queries estatísticas que necessitavam de saber, o valor concreto do número de utilizadores de cada um dos tipos "Bot", "Organization" e "User" presentes no ficheiro criamos a typedef *UTYPESTATS, que é representada pela estrutura u_type_stats que armazena 3 inteiros de 4 bytes que se referem ao número de bot, organization e user respetivamente.

Começamos, então, por inicializar a estrutura através da função ***init_types_users*** que aloca memória para a u_type_stats e inicializa os campos da estrutura a 0.

Posteriormente, criamos a função *incrementaTypes* que, ao ser invocada no módulo user.c, durante a leitura do ficheiro "users-g2.csv" devolve um inteiro n que pode apresentar apenas os seguintes valores, de forma a distinguirmos o tipo do utilizador:

- n=0 -> estamos perante um user do tipo "Bot", pelo que incrementamos o campo bot da estrutura;
- n=1 -> estamos perante um user do tipo "Organization", pelo que incrementamos o campo organization da estrutura;
- n=2 -> estamos perante um user do tipo "User", pelo que incrementamos o campo user da estrutura;

Procedemos, também, à criação das seguintes funções para cumprir o propósito de responder à query 1 em específico: ***returnBots****,* ***returnOrganization****,* ***returnUser****.*

A fim de tornarmos a memória dinâmica, procedemos à criação da função ***free_Utypes*** que liberta a memória previamente alocada para todo o UTYPESTATS.

CONNECTION:

A fim de sermos capazes de responder de forma eficaz às queries parametrizáveis, optamos por criar uma Typedef CONNECTION que é representada por uma estrutura designada userConnections onde armazenamos o id de um dado *commiter*; um array de com os ids dos seus amigos (a junção dos ids da sua *followers list* com os *ids* da sua

following list) bem como o tamanho deste último array. Sendo que todos estes dados são correspondentes a um único commiter, posteriormente utilizamos um array de CONNECTION, para representarmos todos os commiters necessários.

Para efeitos de alocação e realocação de memória, começamos por criar as funções ***init_userConnections*** e ***realloc_usersConnections***, respetivamente.

Em seguida, decidimos criar a função ***add_elem_Connection***, de forma a efetivamente conseguirmos preencher os campos da struct com os dados necessários. Esta função, ao ser invocada na função ***start_user,*** vai preenchendo a struct. Todo este processo é efetuado, com o auxílio da função ***binary_search_amigos*** que verifica se o id do ***author_id*** do repositório está presente na lista de amigos do *commiter*, e só assim procedemos à anexação do seu id ao array. No módulo que trabalha com os commits é também invocada a função ***find_index_uc*** que procura o commiterId de um commit num array de CONNECTION.

Para administrarmos a memória de forma dinâmica, criamos também a função ***free_Connection*** que liberta a memória previamente alocada para todo o array, com o auxílio da função ***free_elem_Connection***.

QUERIES:

Para respondermos às querys criamos um módulo, para tal. A função principal chama-se *answer_queries* que recebe várias estruturas de dados que contêm a informação dos ficheiros de entrada, no caso recebemos um USERS que representa todos os users, um COMMITS que contém todos os commits, um REPOS que armazena todos os repos, um UTYPESSTATS e um TOTALSTATS que representam são as duas estruturas de dados que definimos para as queries estatísticas e um COMMITMESSAGE que vai responder a query 10.

A nossa função principal começa por aceder ao ficheiro que contém as perguntas, *commands.txt*, e vai começar a ler linha por linha. Para gerar o ficheiros .txt de resposta criamos uma função que denominada de *name_newFile* que se encontra no ficheiro aux.c que devolve uma string que é o nome do ficheiro da query à qual estamos a responder no preciso momento. Depois de devolver a string, criamos um ficheiro com esse nome e depois vamos analisar qual a query que queremos responder. Para saber qual a query que queremos responder, o nosso código analisa a linha que lemos do ficheiros *command.txt* e usa o strtol para converter para um número inteiro que é a query.

Depois de sabermos qual o número da query, criamos 10 funções static, em que cada uma simboliza 1 query, e todas recebem um parâmetro em comum que é o ficheiro para o qual irão escrever.

- **Query 1:**

A função associada a esta query chama-se ***query1*** e recebe como parâmetros a estrutura de dados UTYPESTATS. Para escrever a resposta criamos 3 funções no módulo *typessstats.c* em que cada uma devolve um parâmetro desta estrutura. Ao mesmo tempo que vamos tendo os valores, colocamos no ficheiro os devidos parâmetros.

- **Query 2:**

A função associada a esta query chama-se ***query2*** e recebe como parâmetros a estrutura de dados TOTALSTATS. Para responder a esta query esta função invoca 2 funções definidas no módulo *totalstats.c* que devolvem o inteiro associado ao campo “colaboradores_total” e “repos_total” e coloca estes 2 inteiros em 2 variáveis do tipo float.

Depois criamos outra variável do tipo float e coloca o resultado lá da divisão das 2 variáveis float e em seguida escreve no ficheiro o resultado da divisão.

- **Query 3:**

A função associada a esta query chama-se ***query3*** e recebe como parâmetros a estrutura de dados TOTALSTATS. Para responder a esta query esta função invoca returnBotsIds, função definida em *totalstats.c* que devolve o inteiro associado ao campo "bot_ids" e o guarda na variável do tipo inteiro r. Em seguida, escrevemos no ficheiro o valor guardado em r.

- **Query 4:**

A função associada a esta query chama-se ***query4*** e recebe como parâmetros a estrutura de dados TOTALSTATS . Para responder a esta query a função invocamos 2 funções definidas em *totalstats.c*, ***returnCommits*** e ***returnUsers***. A primeira função devolve um inteiro que contém o total de commits efetuados, guardado numa variável do tipo float e a segunda um inteiro com o total de utilizadores existentes, guardada numa outra variável do tipo float. Posteriormente, criámos uma outra variável do tipo float que armazena o resultado da divisão das 2 variáveis float e em seguida escreve no ficheiro o resultado da divisão.

- **Query 5:**

A função associada a esta query chama-se ***query5***, onde recebemos as coleções de users e de commits e o tamanho das mesmas, as strings correspondentes ao número desejado de outputs e aos limites superior e inferior do intervalo de datas, e o ficheiro onde o output será colocado o resultado da query. Nesta query começamos por transformar as strings associadas às datas em elementos do tipo DATA (função ***string_to_ano***) e por criar um triplo onde armazenamos as quantidades de commits feitos por cada utilizador (função ***commits_qtty***) que depois completamos com a função (função ***complete_triplo***). No fim colocamos os dados do array de TRIPLO no ficheiro de output (função ***add_content_triplo***).

- **Query 6:**

A função associada a esta query chama-se ***query6***. Primeiramente invocamos a função ***repo_language***. Depois de obtido o array com todos os ids de repositório. De seguida invocamos a função ***commiter_id_repos*** que vai ver os repos_ids dos commits e cria um array de TRIPLO com parte da informação para a resposta. Neste momento invocamos a função ***complete_triplo*** definida no users.c para meter o array de TRIPLO completo, isto é, adiciona a cada elemento do array o login do commiter_id. Por último invocamos a função ***add_content_triplo*** que adiciona o conteúdo do array a um certo ficheiro.

- **Query 7:**

A função associada a esta query chama-se ***query7***. Para obtermos a resposta para esta query, trabalhamos com a coleção dos repositórios, e a resposta é devolvida como um array de TUPLO. Para então criarmos esse array de TUPLO, começamos por invocar a função ***not_updated_repos*** definida no commit.c que vai buscar todos os ids de repositório em que não têm commit a partir de uma certa data. Depois invocamos a função ***repo_no_update*** que devolve um array de TUPLO com todos os ids do repositório e as suas descrições. Depois de termos a resposta, invocamos a função ***add_content_tuplo2*** que vai adicionar o conteúdo de um array de TUPLO a um ficheiro.

- **Query 8:**

A função associada a esta query chama-se ***query8***. Nós para obtermos a resposta para esta query, trabalhamos com a coleção dos repositórios, e a resposta é devolvida

como um array de TUPLO. Começamos por ir buscar todos os repositórios com commits a partir de uma certa dada utilizada ***updated_repos***. Em seguida, para obtermos o tal array correspondente, utilizamos 2 funções do módulo repos.c. A 1ª função é ***repo_inf_data***. A 2ª função utilizada é a ***repo_language_qtty***. Depois utilizamos a função ***strarray_language_qtty*** para obtermos o array de TUPLO para a resposta. Por último invocamos a função ***add_content_tuplo*** para adicionar o conteúdo do array de TUPLO ao ficheiro de resposta.

- **Query 9:**

A função associada a esta query chama-se ***query9,*** trabalhamos com dados armazenados no array de USER e o seu tamanho, um array de CONNECTION e o seu tamanho bem com a coleção dos commits e o seu tamanho. Começamos então por criar um triplo com os dados que necessitamos relacionados com o COMMIT e CONNECTION na função ***triplos_parametrizavel*** e posteriormente providenciamos o valor deste triplo à função ***complete_triplo*** que juntamente com os dados de USER completa o triplo. Por último, com a função ***add_content_triplo2*** adicionamos o conteúdo do array de TRIPLO ao ficheiro de resposta.

Esta query, infelizmente, foi a única que não conseguimos efetivamente implementar, pois na sua resolução deparamo-nos com erros relacionados com *memory leaks* e outros problemas que prejudicavam a gestão da memória e quando reparamos neste erro, optamos por não dar importância, pois ainda nos faltavam fazer outras partes relevantes do trabalho, tais como, este relatório e melhorar a modularidade e o encapsulamento.

- **Query 10:**

Para respondermos a esta query, pensamos como se ela fosse uma query estatística, em que antes de começarmos a responder às queries, utilizamos a estrutura de dados COMMITMESSAGE. A função associada a esta query é denominada de query10 e recebe como parâmetros a tal estrutura COMMITMESSAGE e uma string. Depois invocamos a função ***add_content_cm*** que adiciona o conteúdo do COMMITMESSAGE a um ficheiro .

CUSTO COMPUTACIONAL

Os arrays que usamos para armazenar as coleções, tanto a nível de tempo como de memória, trabalhamos com espaço e tempo linear. Em certas partes deste projeto, também usamos árvores binárias que trabalham com espaço linear (proporcional ao número de nodos) e tempo logarítmico. Quando invocamos certas funções do projeto podemos ter ciclos dentro de ciclos o que pode levar a tempos quadráticos ou O(Nlog(N)). Também temos funções que trabalham em tempo constante, como por exemplo as que retornam valores armazenados em campos específicos das nossas estruturas.

Uma forma de conseguirmos diminuir este custo computacional, em cada query que calcula ou um array de triplo ou um array tuplo, no final da resolução dessas queries, isto é, depois de escrever as respostas nos ficheiros de resposta, fazemos free ao espaço alocado dos respetivos arras e assim obtemos uma memória dinâmica. Quando não temos mais queries para responder o nosso programa liberta o espaço ocupado pelas coleções e também o espaço ocupado pelos dados auxiliares que nos ajudam a responder às queries, isto é, TOTALSTATS, TYPESSTATS, CONNECTION e COMMESSAGE. Nas funções que trabalham com STRARRAY, árvores binárias e arrays de inteiros auxiliares, quando não são preciso mais estes dados, libertamos também o espaço ocupado dos mesmos.