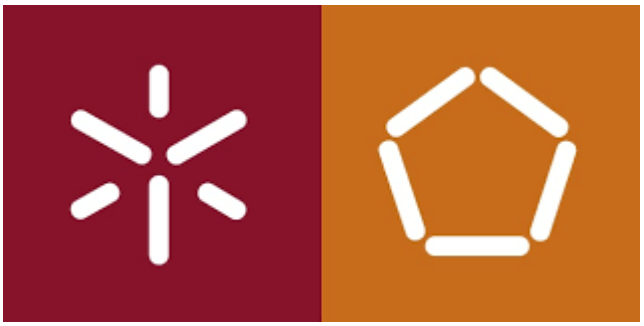

UNIVERSIDADE DO MINHO

LICENCIATURA EM ENGENHARIA INFORMÁTICA

LABORATÓRIOS DE INFORMÁTICA III

GUIÃO III

GRUPO 30

José Carvalho (a94913)

Miguel Silva (a87031)

Ana Rita Poças (a97284)

ÍNDICE

Introdução

Neste guião, o nosso objetivo é conseguirmos trabalhar com dados de dimensões de ficheiro muito superiores aos que tínhamos manipulado nos guiões anteriores, conseguirmos testar a eficiência de cada uma das nossas queries (através do desenvolvimento de testes que que avaliam o desempenho de cada query requerida no guião 2) e desenvolver um mecanismo de interação (através da implementação de um menu) de forma a conseguirmos criar um programa interativo visualmente apelativo com o utilizador.

As principais questões que nós levantamos, específicas deste guião, foram:

- Como devemos lidar com o facto de termos muitos mais dados e que alterações podemos fazer de modo a aproveitar ao máximo o trabalho que realizamos nos outros guiões?
- Como iremos criar o programa interativo?
- O nosso programa é eficiente?
- Que medidas poderíamos adotar para melhorar a sua eficiência?
- Como iremos verificar se as nossas respostas estão sempre a devolver o mesmo resultado?
- Qual será a melhor forma de comparar resultados?

Este guião também tinha outros fatores importantes a ter em conta, tais como a correta implementação do encapsulamento e da modularidade bem como conseguir dar uso ao que fizemos para o guião-1, só que agora para os ficheiros do guião-3. Outro fator que demos atenção foi também a complexidade do código, pois chegamos a um ponto em que tínhamos várias funções muito compridas e tentamos comprimi-las, embora algumas ainda tenham ficado muito extensas.

Neste guião trabalhamos em vários fatores com ficheiros, embora ainda continuemos a trabalhar com a memória RAM em certos casos. Utilizamos a biblioteca “ncurses” para o programa interativo. Tentamos otimizar certas queries, nomeadamente a query 5, 6 e 10 e tentamos encontrar uma solução para a query 9, query que não conseguimos fazer no guião-2.

Principais mudanças no código:

Dados

Como foi já referido, os ficheiros .csv nesta parte do trabalho têm muito mais linhas, o que implica maior número de bytes necessário para representar em memória RAM.

Nós, para este problema, decidimos que em vez de estarmos a trabalhar com estruturas na memória principal, devíamos criar os nossos .csv para cada ficheiro com a informação necessária de cada catálogo para a resolução das queries. Qualquer função que, anteriormente percorria o nosso catálogo ia visitar os respetivos campos em memória que eram armazenados em estruturas auxiliares, agora, em vez disso, para todas as funções fazemos fopen dos nossos catálogos .csv e percorremos o ficheiro de forma a retiramos a informação que pretendemos.

Com este método, como a memória secundária, ou simplesmente disco, tem muito mais Bytes para armazenar informação do que a memória RAM conseguimos evitar o problema de não termos memória suficiente. A única parte em que nós continuamos a trabalhar com a memória RAM, são as nossas estruturas que criamos para auxílio nas respostas às queries. Eventualmente quando estas estruturas não são mais necessárias, acabamos por libertar o espaço ocupado por estas estruturas e assim conseguimos obter uma memória dinâmica.

Queries

Nós fizemos algumas alterações nas queries em relação ao guião-2. A principal é por uma questão de eficiência e de forma a tirarmos partido da memória secundária, nós adaptamos o nosso código para que, quando é invocada uma query estatística ou uma ou uma query parametrizável em que já calculamos anteriormente, em vez de fazer o processo de cálculo todo outra vez, acedemos a um ficheiro que contém a resposta e simplesmente transferimos linha a linha para o ficheiro de output da resposta. Para isto, criamos pastas para cada uma das queries para guardar os resultados das devidas queries e criamos, também, uma função ***where*** que faz o processo de procura do ficheiro que contém a informação. A função ***where*** devolve uma string com o nome do ficheiro que contém a resposta ou o ficheiro em que temos de escrever a resposta.

Ao diminuir o número de vezes que repetimos o processo de cálculo para os mesmos dados, tornamos o programa mais rápido para comandos repetidos.

Programa interativo

Para o programa interativo utilizamos a biblioteca **ncurses** e só tivemos por base em usar as funções ***printw***, para printar a informação, e a função ***clear***, para limpar a janela de ncurses para apresentarmos nova informação. Para obter informação a partir da janela de nurses, tal como, qual a query a ser executada, e no caso de ser parametrizável, os seus parâmetros, quantos utilizadores queremos ver por página e etc, utilizamos a função ***getstr***.

A informação que é apresentada no ecrã, é obtida a partir do ficheiro que contém a resposta e depois usamos o fgets para ir buscar a informação linha a linha.

Nós definimos alguns comandos próprios do nosso programa que são, P : avança para a próxima página em relação à atual, A : volta para a página anterior à atual, S <número> : Saltar para uma certa página que é indicada no <número> e por último 'Q' que é a forma de finalizar a query ou até mesmo sair do programa, dependendo onde se encontra o programa. Outro comando que nós definimos foi, quando o programa pede uma query, o utilizador pode inserir a letra 'M' que é apresentado no ecrã um menu, em que explica o que cada query faz / responde. Nós definimos por base os comandos com letras maiúsculas, mas achamos por bem o nosso programa aceitar também as letras minúsculas, por exemplo, o comando 'm' faz a mesma coisa que o comando 'M' e assim sucessivamente o que facilita a interação entre o utilizador e o programa embora estes comandos alternativos não estejam explicitamente visíveis no ecrã.

Possivelmente a utilização de teclas mais intuitivas para percorrer as respostas poderia ter sido uma boa opção, por exemplo as setas do teclado no lugar de 'P' e 'S', mas optamos por seguir o modelo disponibilizado pelos professores no enunciado do guião, fazendo algo semelhante.

Por último, decidimos que nas queries em que é necessário data ou linguagem o nosso programa filtra ambos os fatores. Para a data, simplesmente criamos uma função no data.c que verifica se a data é válida, por exemplo: se o mês está entre 1 e 12, se a data têm 3 parâmetros e etc. Para a linguagem, criamos uma função no aux.c denominada ***validate_language*** que já irá ser explicada aqui no relatório quando falarmos deste módulo.

Módulo menu.c

Este módulo tem todas as funções auxiliares respetivas ao programa interativo. Como já foi referido, para esta parte de interatividade utilizamos a biblioteca ncurses.

Primeiramente criamos uma função denominada ***read_file_queries*** que dado um ficheiro que contém a resposta a uma certa query, vamos perguntar qual o número de linhas por página e de seguida começamos então a apresentar as páginas. Para o comando P, não definimos nenhuma função, simplesmente o ciclo avança normal. Para o comando A, criamos uma função que avança para a página anterior do ficheiro, utilizando "**fseek**" e um ciclo até chegar a um certo número de iterações em que cada iteração ele faz um fgets. Para o comando S, criamos uma função para tal, em que se a página for anterior da página atual, o procedimento é muito similar à função do comando A, mas se for posterior, a função para este comando simplesmente continua a avançar no ficheiro até chegar à página pretendida. É de ressaltar que para todos estes nossos comandos de transitar de páginas quando o utilizador quer ir para o página inexistente, o nosso programa permanece na mesma página, apresentando uma mensagem de "Página Inválida". Quando o utilizador escreve um comando inválido, tal como no caso anterior, permanecemos na mesma página mas a mensagem passará a ser "Página Inválida".

Para apresentarmos as respostas das queries criamos uma função chamada ***print_some_lines*** que vai printar várias linhas de um certo ficheiro e depois dependendo da query, invoca uma certa função que printa a linha. Nós definimos várias funções, pois o formato das queries varia consoante a query que escolhemos, umas têm mais campos relevantes que outras e as estatísticas têm um formato diferente das parametrizáveis. Criamos uma função muito genérica para printar uma certa string no ecrã, dado o espaço

que a string tem que ocupar, isto é, se a string tiver valor inferior ao tamanho que pretendemos, é acrescentado espaço até ficar com o espaço completo, caso tenha tamanho superior, é só apresentado parte da string.

Por último criamos um função que ajuda a perceber o formato das nossas respostas, denominada ***format*** que devolve uma string com o formato específico de uma certa query e funções responsáveis pela parte estética do programa iterativo, nomeadamente funções que apresentam o menu no terminal (***printMenu_iterativo*** e ***printMenu***) e funções auxiliares que escrevem um determinado número de traços de modo a separar partes independentes do programa e na formatação geral do nosso programa (função ***print_traco*** e as suas variantes que tal como as funções que apresentam o menu, diferem entre si consoante se usam ncurses para o programa iterativo ou não, e variam também se colocam ou não um caracter '\n' no final da escrita dos traços).

Alterações gerais dos módulos

Para este guião criamos um novo módulo denominado *hashint.c* . Resumidamente, este módulo tem por base trabalhar uma hashtable em que vai armazenando inteiros e foram só definidas funções básicas, tais como uma que devolve uma chave, uma outra que inicializa a hashtable, uma que insere um elemento na hashtable, outra que verifica se um inteiro já ocorre na hashtable e por último uma função que dá free à hashtable. O objetivo deste módulo foi para melhorar a eficiência das queries, 5,6 e 10 que estavam próximas ou que excediam o tempo aceitável (5 segundos).

No *aux.c* adicionamos 2 funções de forma a verificar quando for invocada uma query que já tenhamos calculado previamente, vamos buscar a resposta ao ficheiro em vez de realizarmos o cálculo todo novamente, sendo escudado. Neste módulo também foram criadas 2 funções para verificar se os resultados são os mesmos das queries. 1 delas é a principal, que se chama ***validate_files*** e verifica se os ficheiros existem e caso existam, compara linha a linha para verificar o resultado, mas caso o ficheiro de comparação não exista, o nosso código simplesmente cria um novo ficheiro copiando para lá todas as linhas do nosso ficheiro de resposta. Esta última parte é trabalho de outra função, chamada ***copy_info_files*** . Outra função que já foi referida, a ***validate_language***, que é uma função que valida a linguagem, basicamente esta função dada uma string, verifica se esta é nula ou um "\n" ou um "\0" ou não e se for um destes casos extremos retorna 0, se não retorna 1, anteriormente até se se for verificar versões passadas do repositório onde elaboramos o projeto, consideramos tornar o nosso programa não case sensitive, transformando inputs como por exemplo "java" em "Java", mas acabamos por abandonar essa ideia e assumir que os inputs do utilizador são válidos pois da forma como implementamos a validação acabamos por não conseguir abranger todos os casos dado que existem imensas linguagens.

Uma alteração importante foi na função ***int_qtty*** na função de triplo.c, em que o ciclo for percorre de ordem inversa. Decidimos isto pois reparamos que era uma otimização feita dada ao formato dos ficheiros, porque quando geralmente quando aparecia um id, aparecia mais algumas linhas em que o id era igual, no que resulta o id estar no final do array triplo e assim fazemos menos iterações.

Outra alteração também relevante foi no user.c. Antes para completarmos algum triplo ou quádruplo com os logins dos utilizadores, usávamos por base o array que representa a coleção de users. Adaptamos o nosso código, pois agora não trabalhamos

com catálogos em arrays. Para isto usamos por base um array de tuplo que percorre o catálogo .csv do user e cria um tal array de tuplos com o id do utilizador e o seu login. Depois de fazer isso ordenamos esse tal array com base no inteiro, neste caso, o id do utilizador. Quando for preciso completar alguma estrutura com o login, criamos este array desta forma, depois como está ordenado usamos uma pesquisa binária para procurar o login correspondente e colocamos uma cópia da string do login do array de tuplo na tal estrutura em que precisa do login.

Como mencionado previamente na secção anterior, adaptamos algumas das nossas funções de modo a trabalharem com hashtables e não com árvores binárias o que diminuiu tempo de execuções, estas alterações são notórias no módulo commit.c, onde embora as funções alteradas se mantenham semelhantes às originais, exceto pelo facto de agora o catálogo de commits está armazenado num ficheiro e não numa estrutura, a passagem de árvores binárias a hashtables não foi muito complexa devido ao encapsulamento e modularidade que tivemos como objetivo preservar ao longo do projeto.

É importante salientar que, tal como no guião 2, tentamos também gerir a memória da forma mais correta e eficiente possível e como tal libertamos com funções de free tanto pré-definidas como da nossa autoria a memória alocada a estruturas assim que acabamos de trabalhar com elas.

Por último, neste guião alteramos a estrutura da pasta saida dividindo-a em subpastas, uma para cada query com os outputs totais da query quando executada com dados parâmetros e uma pasta para os outputs dos comandos executados.

Testes

Para esta parte dos testes criamos um módulo parecido ao **guiao3.c**, só que na ***answer_queries*** iremos passar um argumento para informar ao módulo de queries a dizer que estamos a executar o **./tests** e não o **./guiao-3**. Para isto passamos um inteiro em que se for igual a 0, estamos a executar o **./guiao-3** e se for 1 estamos a executar o **./tests**. De seguida criamos uma variável global teste em que colocamos lá esse tal argumento.

Para medir os tempos, usamos o código disponibilizado pelos professores no guião.

Nos nossos testes, todas as queries que conseguimos resolver ficaram dentro do tempo ideal, exceto a query 9 que supera um pouco o tempo útil. Porém, tendo em conta que, esta query não estava funcional no guião anterior, consideramos que fizemos um bom progresso neste guião uma vez que, pelo menos, conseguimos implementá-la, embora seja a única query instável, no sentido em que, o seu resultado varia consoante a iteração para o mesmo comando. Infelizmente, não conseguimos resolver este erro, uma vez que não sabemos o que o esteja a causar.

Para facilitar o teste das nossas queries, o executável **./tests** pode ser invocado com um ficheiro de commands.txt e também pode não ser invocado com nenhum argumento, tal como no executável ./guião-3. No caso de ser invocado sem nenhum ficheiro commands, o nosso programa vai começar a filtrar os ficheiros e depois deste processo pergunta no terminal se o utilizador quer ir para o programa interativo ou se quer realizar os testes que foram pré-definidos por nós. No caso do menu interativo, depois quando é apresentada a resposta a uma query que o utilizador invocou e o mesmo saia da resposta, o tempo da query é apresentado na mesma página em que o nosso programa pede outra query.

Para comparar os resultados, invocamos a função já explicada ***validate_files*** que faz o trabalho de verificar se as respostas são iguais. Para a execução desta função, criamos a pasta **expected_files** onde os ficheiros de comparação, caso existam, estão lá armazenados. Caso contrário, como explicado na secção acima, o nosso programa encarrega-se da criação de um novo ficheiro.

Na tabela da página seguinte, apresentamos os tempos de execução que nós registramos para as queries conforme as testamos.

	José Carvalho	Ana Rita Poças	Miguel Cidade Silva
Query 1	(0.49) (0.49) (0.49) = 0.49	(0.45) (0.52) (0.49) = 0.49	(0.56) (0.55) (0.55) = 0.55
Query 2	(2.18) (2.20) (2.18) = 2.19	(2.28) (2.78) (2.60) = 2.55	(2.88) (2.87) (2.95) = 2.90
Query 3	(1.39) (1.35) (1.36) = 1.37	(1.27) (1.39) (1.37) = 1.34	(1.95) (1.94) (1.96) = 1.95
Query 4	(0.38) (0.38) (0.38) = 0.38	(0.39) (0.37) (0.39) = 0.38	(0.49) (0.48) (0.49) = 0.49
Query 5	(3.00 | 3.01 | 3.00) = 3.00 (3.60 | 3.61 | 3.60) = 3.60 (3.88 | 3.88 | 3.90) = 3.89	(2.98 | 2.94 | 3.05) = 2.99 (3.65 | 3.56 | 3.74) = 3.65 (3.83 | 3.79 | 3.99) = 3.87	(4.02 | 4.13 | 3.94) = 4,03 (5.12 | 5.34 | 5.04) = 5.17 (5.62 | 5.80 | 5.57) = 5,66
Query 6	(3.43 | 3.44 | 3.43) = 3.43 (3.46 | 3.47 | 3.46) = 3.46 (3.38 | 3.37 | 3.37) = 3.37	(3.50 | 3.44| 3.67) = 3.54 (3.50 | 3.58 | 3.99) =3.69 (3.41 | 3.45 | 3.64) =3.50	(4.85 | 4.93 | 4.97) = 4.92 (4.90 | 4.97 | 5.03) = 4.97 (4.79 | 4.74 | 4.91) = 4.81
Query 7	(1.22 | 1.22 | 1.22) = 1.22 (1.47 | 1.47 | 1.47) = 1.47 (2.87 | 2.87 | 2.87) = 2.87	(1.22 | 1.23 | 1.22) = 1.22 (1.43 | 1.62 | 1.48) = 1.51 (2.96 | 3.00 | 2.95) = 2.97	(1.96 | 1.97 | 2.02) = 1.93 (2.25 | 2.20 | 2.24) = 2.23 (4.51 | 4.34 | 4.43) = 4.43
Query 8	(2.91 | 2.92 | 2.93) = 2.92 (2.39 | 2.39 | 2.38) = 2.39 (1.03 | 1.03 | 1.03) = 1.03	(3.02 | 3.56 | 3.23) = 3.27 (2.46 | 2.82 | 2.57) = 2.62 (0.95 | 0.98 | 0.97) = 0.97	(4.58 | 4.36 | 4.47) = 4.47 (3.84 | 3.65 | 3.70) = 3.73 (1.59 | 1.56 | 1.55) = 1.57
Query 9	(5.30) (5.32) (5.28) = 5.30	(5.53) (5.30) (5.49) = 5.44	(7.41) (7.17) (7.15) = 7.24
Query 10	(3.96) (3.95) (3.94) = 3.95	(4.10) (4.04) (4.08) = 4.07	(6.24) (6.28) (6.22) = 6.25

Tabela 1 - Tabela de análise de tempos médios da resolução das queries registados pelos membros do grupo (a análise desta tabela deve ser acompanhada pela leitura da secção Caraterísticas dos nossos computadores)

É importante realçar que estes testes são os pré-definidos por nós e foram feitos por todos os elementos do grupo com os ficheiros respetivos ao guião-3, com todos os computadores ligados à corrente e com poucas coisas abertas no computador que possam consumir recursos que podem afetar os tempos medidos (por exemplo: correr o código apenas com o ficheiro do relatório e o terminal abertos) sendo então compreensível que os tempos que medimos podem variar de execução a execução na mesma máquina dependendo de diversos fatores.

No geral, as queries foram sempre testadas no pior caso, isto é, não foram invocadas queries repetidas, se assim fosse haveria tempos menores que 10 milisegundos a contar para a média. Isto acontece pois implementamos o nosso código de forma a ser mais eficiente neste tipo de casos.

Também convém frisar que os tempos de execução das queries foram sempre arredondados para duas casas decimais e foram medidos em segundos. Nos testes que definimos para as queries parametrizadas, utilizamos os seguintes parâmetros de modo a, na nossa perceção termos uma bateria de testes diversificada capaz de testar vários cenários:

5 : 10 2010-01-01 2015-06-15 | 20 2015-06-15 2019-12-31 | 30 2010-01-01 2019-12-31
6 : 10 Python | 20 Java | 30 C#
7 : 2010-01-01 | 2015-06-15 | 2019-12-31
8 : 10 2010-01-01 | 20 2015-06-15 | 30 2019-12-31
9 : 15
10 : 30

Analisando os resultados da tabela, concluímos que o fator que mais afetou os resultados obtidos foram as especificações dos nossos computadores, primeiramente, como mencionamos, consideramos que tivemos bons resultados, mas um dos membros no nosso grupo registou tempos de execução muito superiores (sobretudo nas queries parametrizáveis), isso deve-se à utilização de uma virtual machine em windows por parte desse colega e como tal os tempos que o Miguel obteve não podem ser considerados como muito relevantes quando comparados com os obtidos pelo José e pela Rita, que possuem máquinas com melhores especificações que vão mais em conta com o standard do mercado, a não ser que o programa tivesse como objetivo ser desenvolvido para máquinas menos dotadas a nível de processador e de memória, onde nesse caso teríamos de ter especial cuidado no desenvolvimento do código utilizando técnicas de programação mais eficientes como as mencionadas na secção da crítica, ou outras que ainda não conhecemos.

Características dos nossos computadores

Listamos as características das nossas máquinas segundo a seguinte legenda:
P : processador
R : Memória RAM / principal
M : Memória Disco / secundária
SO : Sistema Operativo
F : Frequência do Processador

José Carvalho, A94913:

- P : I7-7500 U, F: 2,70 GHz
- R : 7,70 GB
- M: 30 GB alocados para o Ubuntu de um disco SSD de 512 GB
- SO: Ubuntu 20.04.3 LTS (64-bit)

Miguel Cidade Silva, A97031:

- P : AMD Ryzen 5 4600H, F: 3.00 GHz
- R : 8,00 GB
- M: 55 GB reservados para a máquina virtual

- SO: Ubuntu 20.04 (Máquina Virtual), ou seja processador a funcionar a 4 cores com RAM efetiva de 4 GB

Ana Rita Poças, A97284:

- P : I5-10300H, F= 2.50GHz
- R : 7,6 GB
- M: 40 GB alocados para o Ubuntu
- SO: Ubuntu 20.04.3 LTS (64-bit)

Crítica

No geral, acreditamos que o nosso programa está bom, visto que todas as queries são respondidas dentro do tempo ideal. É importante realçar que, conseguimos, no decorrer da resolução deste guião, resolver o erro que estava associado ao mau funcionamento da query 9 no guião 2 e agora ela funciona. No entanto, nós temos algumas ideias do que fazer para potencialmente otimizar o nosso programa. Uma delas seria, nas partes de código pesado, isto é, naquelas funções em que têm ciclos com muitas iterações, poderíamos ter optado por tirar partido da programação paralela usando o OpenMP e assim aproveitamos o multicore dos nossos processadores. Outras otimizações poderiam passar por, eventualmente, adaptarmos parte do código para ser vetorizado e em certos ciclos tentar fazer loop unrolling de forma a criar menos dependências de dados entre as iterações.

A nível de parâmetros cumpridos sentimos que o nosso código vai de acordo com os princípios abordados nesta unidade curricular de modularidade e encapsulamento, tentamos também apresentar código legível e com boa documentação, apesar de neste guião devido à complexidade acrescentada pela parte interativa tenhamos funções efetivamente mais compridas.

Visto que este é, também, o último guião da disciplina achamos por bem fazer uma apreciação geral do projeto como um todo e podemos afirmar que este trabalho de grupo embora trabalhoso e desafiante foi bastante interessante para nós como estudantes de engenharia informática e possíveis futuros trabalhadores/investigadores na área pois com ele obtivemos conhecimento sobre boas práticas de programação e ganhamos experiência no que diz respeito às dinâmicas de grupo e de gerir o trabalho em equipa.