Laboratórios de Informática III

Guião 3

Universidade do Minho

João Gonçalo de Faria Melo, nº 95085

Cláudio Alexandre Freitas Bessa, nº 97063

Eduardo Miguel Pacheco Silva, nº 95345

fevereiro 2022

Índice

Índice de Figuras

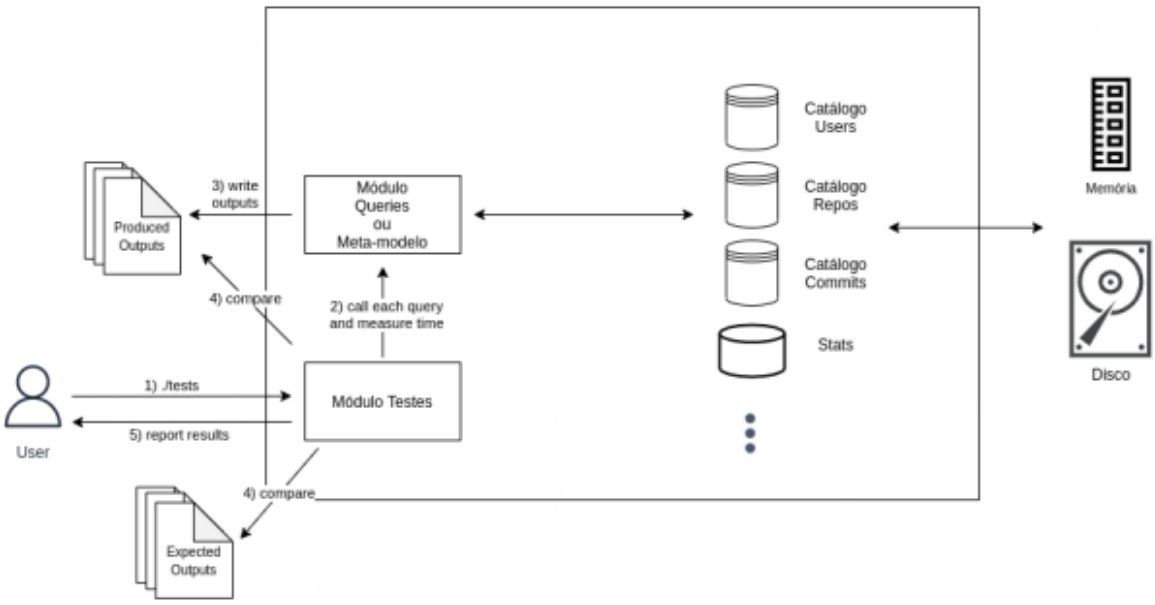
1. Introdução

Ao longo deste relatório é abordado o desenvolvimento do trabalho, centrando a atenção na resolução das etapas propostas pelos docentes, no 3, assim como nas estratégias e métodos utilizados. Sendo o foco, o encapsulamento e a modularidade, e obviamente a eficiência na execução das queries.

É pretendido agora, também ium conjunto de funções testes possíveis de executar através do executável *tests* e também de um interface quando não dado ficheiro de comandos como input.

2. Desenvolvimento do trabalho

Passando desta vez ao guião 3 no âmbito de Laboratórios de Informática III, deparamos agora com a necessidade de melhorar a arquitetura da nossa API. Como podemos observar na Figura_1.



Após a leitura dos ficheiros '*.csv' é necessário guardar os dados importantes para a realização das *queries*. Após a leitura necessitamos de armezar os dados, na qual situação optamos por utilizar vários tipos de *hashtable*, como explicado.

Existindo dois tipo de *queries*, as estatística e parametrizáveis, há obviamente dois processos diferentes a realizar. Para as estatíscas, iniciaremos a sua realização durante a execução de criação dos *catálogos*.

Finalizando assim com a intrepertação do ficheiro de comandos que resulta no output de vários ficheiros consoante o número de *queries* solicitadas, consoante o input.

2.1 Hashtable vs. Outros Métodos

Após a testagem de árvores binárias, listas ligadas, o tempo de inserção pós ordem, teria um custo muito mais elevado do que numa *hashtable*. Como o número de input de comandos não é assim tão elevado, não compensa a criação de uma árvore binária devido à sua organização demorada.

2.2 Encapsulamento e Modularidade

O encapsulamento e a modularidade apenas nos permite aceder aos módulos, ‘user.c’, ‘users.c’, ‘commit.c’, ‘commits.c’, ‘repo.c’, ‘repos.c’, ‘hashtable.c’, ‘queries.c’ e ‘parser.c’ através da sua API, promovendo o encapsulamento. O encapsulamento, desta vez no guião 3, for melhorado não utilizando qualquer tipo de manobra que obstruísse

2.3 Catálogos

Optando assim pela utilização como armazenamento de dados de uma *hashtable*, Criamos assim dois tipos de *hashtables*, uma delas para os ficheiro dos ‘users’ e ‘repos’ e outra hashtables ‘commits’.

A criação destes catálogos por este método torna o seu tempo de execução linear.

2.3.1 Catálogo dos ‘users’ e ‘repos’

Cada linha do ficheiro é diferenciada pelo seu respetivo *id* e é guardado na tabela consoante a função *hash* da *key*(*id*). Ficheiros em que as *keys* são diferentes e as *hash* tem o mesmo valor, é solucionado como se fosse uma espécie de lista ligada inserida na *hash*.

2.3.2 Catálogo dos ‘commits’

Como o ficheiro de ‘commits’ tem informação de repositorios consoante os ‘commits’, a *hash* guarda os *ids* dos repositorios consoante os ‘commits’. Guarda como repositório, mas dentro de cada *key* há uma lista ligada com todos os diferentes commits. Caso a *hash* da *key* seja a mesma, insere na ‘*hash* da *key* +1 (mais um)’ e assim sucessivamente, até encontrar um *slot* vazio.

2.4 Especificações

Como requerido no relatório, iremos anunciar as especificações dos computadores utilizados no teste e desenvolvimente deste projeto.

João Melo (1)

Processador: AMD Ryzen 5 3500U 2.1GHz

Placa Gráfica: Radeon Vega Mobile Gfx 2.1GHz

RAM: 12GB

Discos: SSD 512GB

Cláudio Bessa (2)

Processador: Intel Core i7 6700 2.6GHz

Placa Gráfica: NVIDIA GTX 950M / Inter HD Graphics 530

RAM: 2x8GB DDR4

Discos: SSD 256GB + HDD 1TB (Sistema Operativo a correr no HDD)

Eduado Silva (3)

Processador: Intel Core i7 -4510U 3.1GHz

Placa Gráfica: Intel Haswell-ULT

RAM: 2x4GB DDR3

Discos: SSD 240GB

Seguidamente iremos partilhar os ficheiros de comandos que utilizamos para fazer testes e testar tempos nesta fase do trabalho.

Commands.txt

5 5 2010-08-31 2013-08-31

6 10 Java

7 2010-04-25

8 5 31-08-2010

9 5

Commands2.txt

5 100 2010-01-01 2015-01-01

6 5 Python

7 2014-04-25

8 3 2016-10-05

9 4

2.5 Queries

As *queries* são comandos executados através de uma constante, *id* da *querie*, e algumas por uma constante e certos argumentos, como data, outra constante ou até linguagens de programação.

As primeiras 4 *queries* que são as estatísticas, foram as mais fáceis a ser realizadas, uma vez que a maioria é só aceder a certo catálogo e adicionar certo parâmetro a um acumulador associado.

As outras 6 *queries* foram, as parametrizáveis, são bastante semelhantes, de modo que foi basicamente atribuir o mesmo pensamento a todas, porém foram de facto mais difíceis.

2.5.1 Query 1

A primeira *query* é executada “paralelamente” com a criação do catálogo dos utilizadores. Enquanto o ficheiro e lido vai incrementando os contadores dos “*bots*”, “*orgs*” e “*users”*.

Tempo de execução – Linear

(1)=> 0.0002s (2) => 0.0001s (3) => 0.0006s

2.5.2 Query 2

Nesta segunda *query* é criada uma *hashtable* especifica onde insere todos os colaboradores e após ser criada, é lhe contada todos os elementos (colaboradores).

Tempo de execução – Linear.

(1)=> 0.243s (2) => 0.078s (3) => 0.078s

2.5.3 Query 3

É percorrida a *hashtable* dos colaboradores (query 2) e contado o número de *bots*.

Tempo de execução – Constante.

(1)=> 0.553s (2) => 0.219s (3) => 0.223s

2.5.4 Query 4

A *query 4*, tal como a *query 3*, é apenas aplicar a fórmula uma vez que os dados já foram obtidos ou para *queries* anteriores ou como cálculos auxiliares.

Tempo de execução – Constante.

(1)=> 0.0001s (2) => 0.0001s (3) => 0.0001s

2.5.5 Query 5

Para esta query iremos visitar cada *commit* e obter a sua data, se a data estivesse entre as pretendidas, lia-se o *commiter_id* e adicionava-se um ao acumulador desse id. Se o *'commiter_id'* já estivesse guardado na hashtalbe iria adicionar um ao acumulador caso contrário, criaria outra *hash key.*

Tempo de execução – 2N - Linear.

(1)=> 22.531s (2) => 140.734s (3) => 14.394s

2.5.6 Query 6

Para cada commit, verifica-se o seu repo_id, e verifica-se a linguagem desse repo. Se esse repo tiver a linguagem pretendida, adiciona-se um ao acumulador do commiter_id.

Tempo de execução –2N - Linear.

(1)=> 2.941s (2) => 1.797 s (3) => 1.672s

2.5.7 Query 7

Para cada repositório verifica-se se o repositório não teve nenhuma *commit* após a data referida, caso contrário printa-se a mensagem e o id do repositório. Nesta query, diferente das outras, não conseguimos realizar a 100% a parte da *interface* uma vez deparados com, *segmentation fault*, provavelmente por memória mal alocada, na qual não conseguimos descobrir onde ao certo mesmo utilizando o *valgrind*.

Tempo de execução – 2N - Linear.

(1)=> 12.543 s (2) => 75.406s (3) => 7.826

2.5.8 Query 8

Com a restrinção da data do commit, ser mais recente que a data dada verificamos o repo_id e procuramos no seu repositório qual é a linguagem que utiliza. Após isso adiciona-se um ao acumulador da respetiva linguagem.

Também 2N, linear.

(1)=> 13.023s (2) => 73.031 s (3) => 8.660

2.5.9 Query 9

Nesta query, verificamos o catálogo dos commits, para obter o committer_id e o repo_id. Esse commiter_id terá de ter amigos, ou seja, users simultaneamente na follower list como na following list, que sejam donos do repo_id do commit. Caso isso não aconteça o commit, é desprezado. Se for válido é adicionado um ao acumulador desse commiter_id.

Tempo de execução – 2N – Linear.

(1)=> 5.817s (2) => 3.860 s (3) => 3.042s

2.5.10 Query 10

Para esta query consultamos o tamanho da mensagem de um determinado commiter_id num determinado repo_id. Se o commit seguinte tiver uma mensagem maior e for o mesmo repo_id, substitui a maior mensagem por a última referida. Caso contrário mantêm. Ao fim verificamos quais o top n maiores mensagens. Apesar de feita, através dos testes comparando o *output* verificamos que não estava exatamente a fazer o pedido. Daí não apresentarmos tempos abaixo da mesma.

Tempo de execução – 2N – Linear.

(1)=> 0.000s (2) => 0.000s (3) => 0.000s

3. Conclusão

Estatisticamente o tempo de criação de uma tabela hash com 2 milhões de linhas era entre 9 a 10 segundos mais 6 a 7 segundos para a libertar, já tempo de percorrer a tabela era inferior a 0.5 segundo, tempo medio – 24s.

Algumas das tabelas necessárias para cada query podiam ser inicializadas no início do programa, mas optamos por não o fazer.

Desvantagens – Maior tempo de execução caso o número de comandos seja elevado.

Vantagens – Menor risco de o programa dar *crash* por falta de memória, caso o número de linhas dos ficheiros seja muito, mas muito elevado.

Relativamente a problemas de memória libertamos a memória alocada não necessária sempre que possível, recorrendo ao comando *valgrind* na linha de comandos.

Nesta fase final, era solicitado a utilzação de funções testes com objetivo de comparação entre os valores estimados das queries e os valores obtidos. Da mesma forma que era solicitado a cotação do tempo. Para a comparação dos dados utilizamos os dados obtidos da execução das queries com a base de dados do guião 2. Uma vez, que não alteramos a nossa API, apenas nos focamos a resolver a *interface*, *bugs* obtidos, melhorar as queries e faltas de cumprimento do encapsulamento.

No geral, consideramos que nesta fase o grupo não esteve bem organizado e trabalhou não tanto como gostariamos. Finalizamos por fim, esta fase do projeto e o projeto, não da forma mais eficiente mas cumprindo o objetivo principal da modularidade e do encapsulamento.