Laboratórios de Informática III

Guião 2

Universidade do Minho

João Gonçalo de Faria Melo, nº 95085

Cláudio Alexandre Freitas Bessa, nº 97063

Eduardo Miguel Pacheco Silva, nº 95345

Dezembro 2020

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 Guião 2, assim como nas estratégias e métodos utilizados.

2. Desenvolvimento do trabalho

Passando desta vez ao guião 2 no âmbito de Laboratórios de Informática III, deparamos agora com a necessidade de criar uma arquitetura para um API. Onde nos era aconselhado a separação em ‘Leitura’ e ‘Módulos de Dados’, como podemos observar na Figura_1.

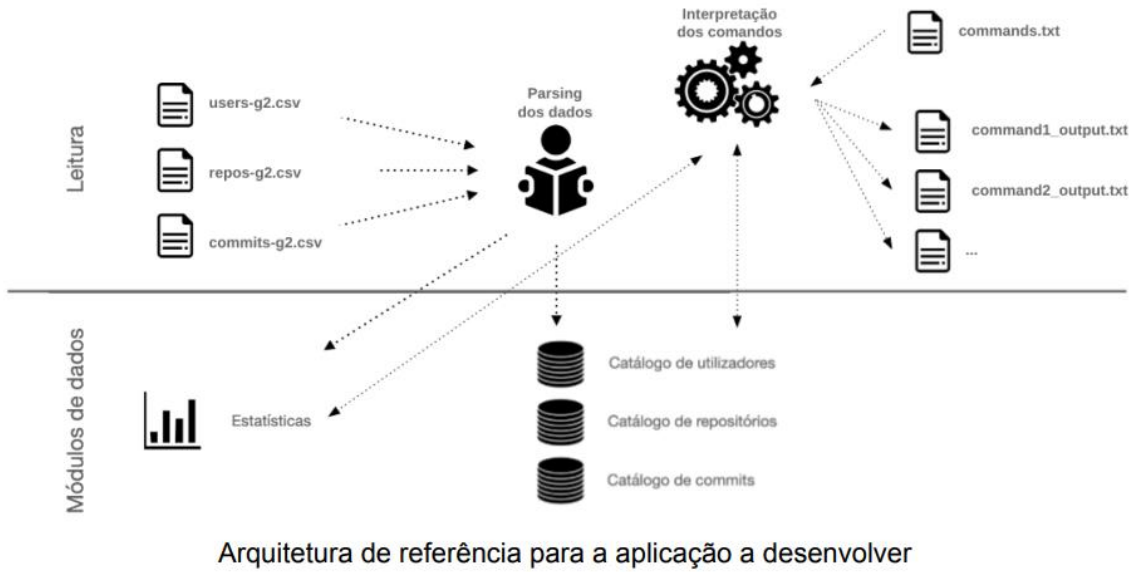

Figura 1 - Arquitetura de referência

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.

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 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.

2.4.1 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.

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.

2.5.3 Query 3

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

Tempo de execução – Constante.

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.

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.

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.

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.

Tempo de execução – 2N - Linear.

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.

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.

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.

3. Conclusão

Achamos que este foi o melhor método para a resolução das *queries* pois o custo de execução de cada query, provinha do esforço necessário para a criação das nossas estruturas de dados (*hashtables*) que era relativamente reduzido comparando com o esforço necessário caso fosse preciso usar algoritmos de ordenação, ou árvores binárias.

Estatisticamente o tempo de criação de uma tabela com 2 milhões de linhas era entre 3 a 4 segundos mais 2 a 3 segundos para a libertar, já tempo de percorrer a tabela era inferior a 0.5 segundo, tempo medio – 6,5s. Isto num computador razoável.

Comparando esses dados com o tempo de execução caso fosse por um outro método anteriormente referido o tempo de execução excedia os 10 segundos.

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 “arrebentar” por falta de memória, caso o número de linhas dos ficheiros seja muito, mas muito elevado; Tempo de execução normal caso os comandos sejam todos diferentes.

Relativamente a problemas de memoria libertamos a memória alocada não necessária sempre que possível, recorrendo ao *valgrind*.

No geral, consideramos que o grupo esteve bem organizado e trabalhou bastante em conjunto, estando várias vezes a discutir estratégias sobre os mesmo e opções a adotar para a resolução consoante o aparecimento dos problemas. Nesta parte 2 do trabalho prático no âmbito da cadeira de ‘LI3’ (Laboratórios de Informática III), notificamos o desempenho de Eduardo Silva. Achamos bastante simples a implementação e bastante interessante e útil a inserção dos módulos e das cápsulas, como mencionado anteriormente.