`

Henrique Parola Meziara da Costa
José Pedro Dias Nascimento Fernandes
Marco Delgado Esperança

Sistema de Gestão de Recomendação

Relatório do Trabaho de Laboratórios de Informática 3
MIEI

Maio de 2020

Índice Geral

"Se eu fui capaz de ver mais longe
é porque estava de pé nos ombros dos gigantes"
Isaac Newton

Introdução

Conteúdo

Neste capítulo, faz-se uma apresentação, reflexão e descrição da perspetiva do nosso grupo face o trabalho. Pedimos categoricamente para ser tomada a devida **atenção** nessa introdução por ser nela em que começaremos por explicar as decisões e planeamento tomados ao longo do projeto.

1.1 Autorreflexão inicial

O processo *abstrato* inicial de *projetar uma estrutura global* para o projeto resultou numa abordagem do enunciado que acreditamos não ser a mais **intuitiva**. Reconhecemos que nossa perspetiva provavelmente não vai diretamente de encontro com aquilo que os professores tencionam encontrar. Porém, tomamos uma decisão e assentamos sobre um ponto de vista em que acreditamos ter bastante potencial e pode ser alternativamente tão válido como qualquer outro. Focamo-nos essencialmente em fazer o famoso **código genérico**, e acabamos por ter 85% do trabalho totalmente independente em si do enunciado. A base por trás dos mecanismos das *queries* e *interpretação de comandos* são totalmente independentes do projeto, sendo possível usar grande parte do código em qualquer outro projeto.

Sendo assim, era de se esperar que o código atingisse uma certa **complexidade**. Por conta disso não iremos abordar todos os detalhes aqui, pois existe muita informação. Em contrapartida, disponibilizamos tais **informações** através de 2 formas: ficheiros ***README*** dentro das pastas do projeto, ou repositórios independentes donde foram criadas, por nós, as **API'S** que utilizamos. Dentro da pasta *ROOT* do projeto temos um ***README*** com as informações relativas ao programa em si, sobre como utilizar e as posturas que um contribuidor do projeto deve ter. Dentro do *interpretador* e do módulo *table* existem outros ***README*** com mais informações específicas. Na secção de **Referências** encontram-se os links respetivos dos repositórios anteriormente mencionados.

1.2 Uma escolha um tanto diferente

Vamos direto ao assunto, nossa decisão para o projeto foi visualizar tudo como **TABLE**. Pegamos no conceito das **TABLES** que são os resultados das *queries* e vimos para além dele, generalizando-o como sendo a ***unidade de estrutura de armazenamento da informação*** do projeto, seguindo o mesmo padrão do tratamento de dados das base de dados como, **MySQL** e **MongoDB**. Óbviamente, o resultado de uma query é uma entidade diferente do tipo *SGR*. Porém o próprio *SGR* utiliza **TABLES** como forma de guardar informação. Seguimos o seguinte paradigma:

```
buscas em "TABLES" resultam em outras "TABLES", normalmente sendo
uma espécie de "sub-conjunto" da "TABLE" em que foi realizada a
busca.
```

A **TABLE** é uma estrutura de dados implementada nomeadamente a partir de uma *árvore balanceada de procura*, em que cada nodo é uma representação das **ROWS** de uma tabela. Com isso temos a possibilidade de realizar operações de procura com os benefícios trazidos por tal tipo de dados (árvore balanceada). A vantagem disto é poder continuar a ter as propriedades de *árvores binárias balanceadas* se quisermos fazer *buscas* a partir dos resultados das *queries*. Ou seja, em nosso trabalho, o resultado de uma *query* não serve apenas para ser "mostrado", como também para fazermos **buscas otimizadas sobre ele**, com as mesmas propriedades das *buscas feitas para ter obtido a própria TABLE em questão*. Isto porque afinal, tudo é guardado na forma de **TABLE**.

Ou seja, daí surge uma outra vantagem de tal organização: a manutenção do código. Vamos supor que o tipo **TABLE** encontra uma forma mais eficiente de guardar informação e subsitui a tal *árvore balanceada de procura* por outra estrutura de dados. Nada precisará ser feito a nível de manutenção de código das *queries*. Isto porque as *queries* apenas conhecem o tipo *SGR*; que por sua vez apenas conhece o tipo *TABLE*. As funções de *busca* sobre as **TABLES** continuarão a funcionar da mesma forma, pois o utilizador da **API TABLE** nem sequer tem noção de que a *árvore balanceada de procura* foi subsituída por outra entidade. O utilizador apenas aplica funções de *busca* sobre uma *TABLE* e a invocação de tais funções não muda. Com isso conseguimos tratar qualquer tipo de *ficheiro* de dados da mesma forma, seja ele *reviews, users, businesses*, etc. Armazenar a informação de cada ficheiros numa **TABLE** (estrutura implementada internamente por uma árvore binária já armazenando **ROWS** em cada nodo, mantendo o aspecto natural de uma "tabela em si") e após aplicar *queries* continuar a ter uma **TABLE**, foi a escolha um tanto "diferente" que tomamos.

Sendo assim, naturalmente tivemos de criar uma **API** para implementar a simulação de um **banco de dados**, possuindo funções de *busca* super otimizadas. Estas funções de *busca* são as unidade funcionais das *queries*, funcionando como **peças de um puzzle** para criar um grande **puzzle** que são as queries.

1.3 Pensando para além do trabalho

Como já citado anteriormente, criamos a **API** chamada ***Standart Table***. Tal **API** simula uma **base de dados** e possui essencialmente 3 partes distintas:

- `#include <table/table.h>` disponibilização de funções para criação de uma **TABLE** bem como inserção de elementos;
- `#include <table/search.h>` disponibilização de funções para **buscas** em **TABLES**;
- `#include <table/show.h>` disponibilização de funções para visualização de **TABLES**

As informações de tal **API** são dadas com mais detalhes na secção **TABLE**.

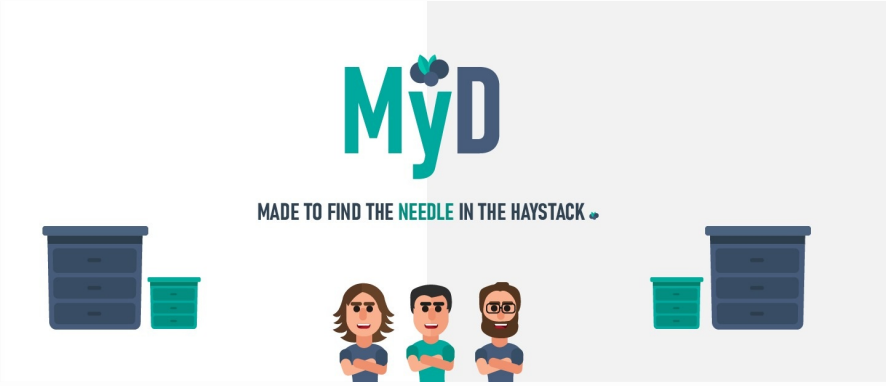
Para realizarmos o processo de **validação de comandos** por parte do utilizador, criamos a **API** chamada **Standart Interpreter**. Tal **API** fornece funcionalidades para tratar de diversas sintaxes de comandos (podendo aumentar para além do que o enunciado pede). O **Standart Interpreter** lida parcialmente com o **parsing dos comandos** pois há coisas que apenas seu programa por vezes está preparado para conhecer (como por exemplo uma das queries possuir como argumento strings como *EQ*, *LT* ou *GT*). Nesse caso a **API** não avalia se tais strings correspondem exatamente ao que a query pede … Porém o **Standart Interpreter** possui um mecanismo que para além de dar à volta a esse problema, interliga o resto de teu programa ao interpretador, realizando todo o fluxo do código **por trás das cortinas do interpretador**. Esse mecanismo é possível graças a possibilidade do usuário inserir as funções que são chamadas mediante cada comando do usuário. Isso é posivel graças ao conceito de **function pointer** que será abordado mais detalhadamente na secção do **Interpretador**.

“Um programa capaz de
ensinar o próprio utilizador”
Henrique Costa

Capítulo II

Guia do usuário

Conteúdo



Neste capítulo, faz-se uma breve visitada guiada nos aspectos mais importantes do programa, bem como algumas particularidades essenciais para uma melhor experiência.

2.1 Caro usuário

Caro usuário, seja bem-vindo ao **MyD**. O **MyDrawer** é uma espécie de *super armário* que possui suas *gavetas* devidamente otmizadas, de modo a extrair informação em pequenos espaços de tempo. Uma das coisas mais importantes em ter em conta é o mecanismo de **STATUS** do **prompt**. Como o Myd é um programa de *linha de comando*, o mesmo fornece uma validação visual dos comandos inseridos, através de mais de 7 **warnings** diferentes e **coloração** de acordo com o *parsing* do comando. Assim, se o *prompt* ficar vermelho siginifca que o comando inserido foi inválidado pelo *parsing*. Se estiver **verde** é o oposto, o comando foi processado com sucesso. Para além disso é sempre indicado o **tempo** de duração do comando em **segundos**.

2.2 Aprendendo a ler o manual

O manual ajuda a compreender a sintaxe dos possíveis comandos a serem feitos. Vale ressaltar que nessa primeira versão do projeto, **não pode haver espaços entre qualquer um dos carácteres inseridos na linha de comando**. Isso será exemplificado mais a frente, mas desde já fica essa informação. Para além disso, todas as queries e comandos (sem contar com os comandos informativos) necessitam do `;` no final.

O manual fornece nomeadamente 5 informações:

- **COMMAND**: indica o nome do comando;
- **ARGS NUMBER**: indica a quantidade de argumentos que a *função principal* associada ao comando recebe;
- **ARGS TYPE**: indica o **tipo** dos argumentos respetivamente. Vale a pena ressaltar como se escreve cada um dos tipos:

`s` Simboliza **string**. Deve ser escrito entre `""`. Por exemplo: `"Eren Yeager"`
`c` Simboliza **carácter**. Deve ser escrito entre `''`. Por exemlo: `'c'`
`d` Simboliza **número inteiro**. Não precisa de nada mais para além do número.
`f` Simboliza **float**. Não precisa de mais nada para além do número.

- **COMMAND TYPE**: indica se o comando retorna seu valor para uma variável ou não. Isto é:

  `x=business_info(sgr,...);` é do tipo **VAR**
  `show(x);` é do tipo **FUNC**
  `+` é do tipo **TEXT**

2.3 Sintaxes especiais

Devido a **API stditp** (que será explicada mais a frente) considerar o `;` e a `,` como carácteres especiais, tais carácteres não podem ser usados dentro dos argumentos dos comandos. Assim, quando quiser se referir ao `;` utilize `..`, enquanto para se referir ao `,`, utilize o `:`. Mostremos então algum dos comandos particulares:

`x=proj(x,{1:2:4})` Tal comando realiza a projeção das colunas de uma tablea. Escolhemos as colunas através de um **array**, indicado por `{}` e o carácter de separação é o `:`como citado anterioremnte.

`x=fromCSV("file", '..');` Tal comando faz **upload** do **file** para uma TABLE, utilizando como carácter de separação dos campos o `;`(pois como foi citado anteriormente, `..` representa o `;`.

2.4 O ficheiro de configuração

O **MyD** utiliza o ficheiro de configuração **mydConfig.cfg** para 2 efeitos:

- obter o **path** para os ficheiros que serão utilizados;
- indicar se o utilizador quer ou não ler o campo **fields** do ficheiro **users**.

Segue exemplo de ficheiro de config:

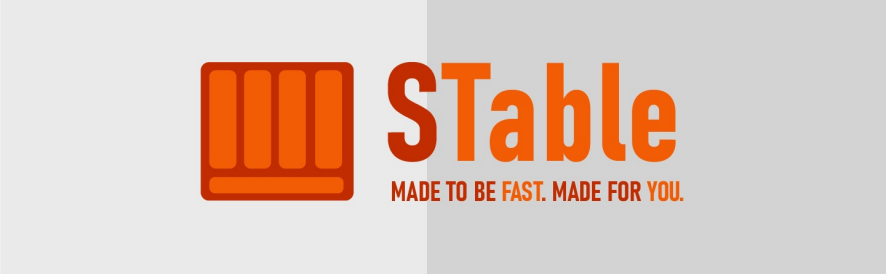
```
set path=/home/bekele/files/
set friends=yes
```

É importante respeitar a sintaxe do **set** separado por **path** (e escrever a partir do início da linha). O carácter = tem de vir junto do **set** e do início do seu valor inserido.

Capítulo III

Table

Conteúdo



Neste capítulo, faz-se uma breve introdução ao módulo **TABLE**, compost pela *API STable* **(Standart Table)**. Tal módulo encontra-se presente na secção **model** de nosso código.

3.1 O funcionamento interno da API

O **Standart Table** surgiu a partir do raciocínio que optamos inicialmente de utilizarmos **Tables** tanto para guardar a informação que vem do *upload* dos ficheiros, tanto para guardar a informação representada pelas **queries**. Assim teríamos de encontrar algo suficienteente genérico, tanto para ser capaz de armazenar qualquer tipo de informação, ser uma estrutura suficientemente otimizada (para buscarmos padrões rapidamente) e que de uma certa forma representa uma **tabela**.

Sendo assim, escolhemos representar internamente a **TABLE** sendo composta por:

```
struct table {
```

```
    int fields;
    int index;
    HEADER header;
    GTree **gtree;
};
```

O campo **fields** representa a quantidade de *colunas* que a **TABLE** terá, isto é, a quantidade de campos existentes em cada linha do ficheiro **CSV** em estudo. O campo **header** traz a informação das características de cada **field** (cada coluna). Para ser mais claro, tal **header** é um **array** de **INFO'S**:

```
struct info {
    int type;
    int size;
    int header_no;
    const char *field_name;
};
```

Assim, cada **INFO** armazena a característica de uma coluna, que é composta por: * **type** `indica se a coluna guarda informação do tipo int, float, string, char ou double através de algumas macros`; * **size** `indica o tamanho que o *type* ocupa no contexto na arquitetura do computador`; * **header_no** `indica o índice da coluna`; * **field_name** `indica o nome da coluna`

Já conhecendo então aquilo que o **header** armazena, voltemos a **TABLE**. Esta, como dizemos anteriormente, armazena informação em **árvores binárias balanceadas**. Isto está representado na estrutura através do `GTree ** gtree`.
A forma como conseptualizamos a table foi por forma a existir uma tree principal. Esta gtree principal alocava o conteúdo da table e estaria indexada pela coluna do identificador único (índice index). As outras gtrees estariam disponíveis se o utilizador quisesse um acesso mais rápido a uma dada coluna. Está disponível uma função que possibilita a indexação de outras colunas. O que esta função faz é criar uma gtree no índice dessa coluna. Estas trees são "secundárias". Isto porque não alocam outra vez o conteúdo. Apenas pegam no conteúdo da coluna e usam o conteúdo como uma *key*. Esta key estará associada a um array de **ROW**. Este array tem todas as **ROW**'s que têm o mesmo conteúdo na dada coluna.
Esta indexação permite uma acesso que não é possível ter em hashtables. Por exemplo em tops. Se um utilizador quiser saber um top pode aceder a uma coluna indexada e é possível aceder essa gtree de forma decrescente. Por outro lado numa hashtable não seria possível fazer este acesso decrescente sem ter que percorrer todos os elementos.

Em nosso projeto utilizamos maioritariamente apenas uma (a tree principal), mas por questões de otimização, há mais do que esta possibilidade.

3.2 Utilização da API no projeto

3.2.1 Criação de table

Para criar uma table apenas é necessário o número de colunas, a coluna que vai possuir a coluna com elementos únicos e por fim o nome das colunas.

```
TABLE * new = table_new(3, 0, "id", "Primeiro Nome", "Sobrenome")
```

Este código cria uma nova table com 3 colunas com os respetivos nomes das colunas.
O que esta função faz é fazer o parsing dos argumentos e enviar para uma função genérica que faz a criação da table.
Esta função recebe argumentos num formato já tratado.

```
char *colunas[3] = {"id", "Primeiro Nome", "Sobrenome"};
TABLE * new = table_new_generic(3, 0, colunas);
```

O resultado é o mesmo. A única vantagem de ter uma função genérica é que é possível automatizar criações de table sem saber de antemão o número de argumentos necessários. Como acontece na **table_new**.

3.2.2 Funções de procura

São disponibilizadas funções de procura que permitem fazer procuras à table. O intuíto principal delas é servirem como "building blocks" para que a partir delas seja possível fazer qualquer tipo de procura exigida. Desta forma o usuário da API deverá conseguir através da manipulação das funções de procura, chegar ao seu resultado pretendido.
As principais funções são:

- `ROW table_search_row_id(TABLE *t, void *key);`
  - Esta função devolve a **ROW** de um id único
- `TABLE *table_search_row(TABLE *t, char *header_in, char **headers_out, int h_quant, void *cont);`
  - Esta função permite a partir de um conteúdo, de qualquer coluna, dar uma table com todas as **ROWS**'s que possuem esse conteúdo. Esta table pode ser costumizada através dos headers_out para ter apenas um *subset* de colunas.
- `TABLE * table_join (TABLE * t1, TABLE * t2, char ** headers_out, int h1, int h2, int *types, tpointer (*func) (tpointer, tpointer, tpointer, tpointer), tpointer user_data);`
  - Dadas duas table esta função faz o join das duas
- `TABLE *table_search_letter (TABLE * t, char * header_in, char ** headers_out, int h_quant, char letter);`
- `void table_search_foreach(TABLE * t, int (*func) (tpointer, tpointer, tpointer), tpointer user_data);`
- `void table_search_foreach_inverse (TABLE * t, int (*func) (tpointer, tpointer, tpointer), tpointer user_data);`
- `int table_search_foreach_indexed(TABLE *t, int index, void (*func) (tpointer, tpointer), tpointer user_data);`

“Feito para se preocupar apenas
com o que importa em seu código”
Henrique Costa

Capítulo IV

Interpretador

Conteúdo

Neste capítulo, faz-se uma breve introdução ao módulo do **Interpretador**, composto nomeadamente pela *API stditp* **(Standart Interpreter)** , *funções primárias* e *função de load*.

4.1 Descrição da API

O ***stditp*** nasceu a partir de dois conceitos: **reutilização de código** e **interligação genérica modular**. De modo a criarmos um interpretador que possa ser utilizado em qualquer projeto e que tenha capacidade de se “comunicar” com qualquer parte do programa, dividimos o **parsing** dos comandos em 2 partes: * Análise sintática, funcional e paramétrica; * Análise dos **particular types** do programa em questão.

O primeiro **pilar** do parsing fica à cargo do **Standart Interpreter**, que disponiliza **estruturas** e **funções** preparadas para lidar com vários tipos de sintaxe de comandos - tais sintaxes são *features* que podem ser facilmente atualizadas.

As **Funções primárias** são as responsáveis pela segunda parte do parsing, e são elas que conhecem e manipulam as particularidades de nosso programa. A vantagem de utilizar tal separação de processos é poder noutros futuros projetos apenas criar *funções primárias*

novas de acordo com o objetivo do programa, sendo *75%* do **parsing** já realizado pela *API* anteriormente dita. Mas a estratégia envolvida vai muito além de "reutilização" … engloba também **interligação** (como veremos mais à frente).

4.2 Utilização da API no projeto

4.2.1 Inserção dos comandos

O **Standart Interpreter** controla majoritariamente o escopo da função ***main*** do programa. Inicialmente criamos uma **TABELA de opções**, a partir da função `init_opts_table()`, e armazenamos este tipo de dados numa `_OPTS_TABLE`. Tal tabela será responsável por todos os comandos do programa, bem como suas características e informações associadas. Tendo iniciado uma `_OPTS_TABLE` chamamos a função `insert_opt()` para inserir os comandos de nosso programa. Segue uma demonstração de como tal proceco foi feito:

```
/* criação da tabela de opções */
_OPTS_TABLE opts_table = init_opts_table ();


/* inserção do comando relativo a query 1*/
insert_opt (opts_table, "business_info", PRIMARY_business_info, 2,
        VAR, "busines info", varlist);


/* inserção dos restantes comandos ... */
```

Explicaremos agora o que signifca cada argumento da função `opts_table`:

- `1° ARG` _OPTS_TABLE onde vamos inserir a opção (comando);
- `2° ARG` nome do comando - forma com que ele é escrito pelo usuário;
- `3° ARG` function pointer da função associada ao comando
- `4° ARG` número de argumentos do comando;
- `5° ARG` tipo de sintaxe do comando;
- `6° ARG` informação associada ao comando;
- `7° ARG` **ipointer** para a informação, a nível de dados, trasnferida a função associada ao comando.

4.2.2 Funções primárias

O 3º argumento citado anteriormente, a ***função primária*** associada ao comando e o ***iponter*** (7º argumento) é que o permite a **interligação** modular da API do **stditp** com o seu programa. Isto porque usamos o conceito de **FUNCTION POINTER** para tal mecanismo de ligação. Para explicar este conceito no contexto do projeto, vamos usar como exemplo a query *business_info ()*. Conectaremos tal query com o **stditp** criando uma função intermediária a que chamaremos de `PRIMARY_business_info ()`. Essa função apenas deverá seguir as seguinte regras: * receber 1 parâmetro do tipo `_ARGS` disponilizo pela *API* * ter valor de retorno do tipo `(void *)`

A beleza disto é que o tipo `_ARGS` será uma struct que possuirá toda a informação necessária para utilizar em seu programa a partir do comando do usuário. Ela contém informação tanto vinda do usuário (a partir do **parse** do ***stditp***) tanto do programador que inseriu no tal ***ipointer*** referido anteriormente. Este ***ipointer*** não é nada mais que um `typedef` disponilizado pela *API* que é o mesmo que um `(void *)`. Assim, voltando ao exemplo com a query *business_info* (), sabendo que tal query recebe uma variável do tipo *SGR* e um *business_id*, o que fizemos foi inserir uma ***VAR_LIST*** como sétimo argumento da função `opts_table ()` (sendo o tal ***iponiter***). Uma ***VAR_LIST*** é um apontador para uma **estrutura opaca** de nosso programa que contém uma **Hash_Table** que armazena todas as variáveis de nosso programa, inclusive variáveis *SGR* se existir. Assim, se o usuário inserir na linha de comando:

```
x=business_info(sgr,"ishJIXOWQ0W9");
```

A função `PRIMARY_business_info` será chamada automaticamente e a partir dos ***getters*** do ***_ARGS*** poderei ter acesso aos:

- argumentos da query através do `get_args ()`;
- variável em que guardaremos o resultado da query através do `get_var ()`
- ***VAR_LIST*** com as variáveis existentes do programa com `get_user_data ()`
- ***_STATUS_PROMPT*** (status do último comando) com `get_status ()`;
- tempo que durou o *parse* com `get_time_opt ()`;
- número de argumentos com `get_argc ()`;
- e muito mais (…)

OBSERVAÇÃO: criamos uma função chamada `load_opts ()` para carregar numa ***_OPTS_TABLE*** toda informações dos comandos existentes em nosso programa.

4.2.3 Análise dos comandos

O **Standart Interpreter** fornece uma função chamada `n_o_readline ()` (***No Empty Readline***) que é uma espécie de *readline ()* melhorado, que armazena automaticamente os comandos **não vazios** no histórico e **retorna** a linha inserida pelo usuário. Utilizamos esta função para ler os comandos inseridos bem como a função `parse ()` que será o ***controle*** da analise dos comandos. Está função retorna o tal tipo ***_ARGS*** referido anteriormente com toda a informação a respeito do último comando inserido. Finalmente basta invocarmos a função `do_opt ()` e a função ***primária*** associada ao comando será executada. E assim, conseguimos de forma simples, organizada e eficiente, interligarmos nosso programa com a *API* ***stditp***.

OBSERVAÇÃO: chamamos **funcões primárias** anteriormente pois elas funcionam "antes" da função principal, nesse caso as *queries*. São nessas funções que invocamos as *queries* bem como fazemos o tal *parsing* dos *particular types*, isto é, vericamos se a variável inserida pelo usuário existe em nosso programa, verificamos se um dos argumentos inseridos é um `float` por exemplo no caso da query que recebe tal tipo de informação, etc.

4.2.3 Principais funções da API

`init_opts_table()` *Cria uma tabela de opções*
`insert_opt()` *Insere opções numa tablea de opções*
`parse()` *Lida com a análise de um comando e devolve o resultado do estudo*
`n_e_readline()` Lê uma linha inserida e disponibiliza atalhos como: **acesso ao histórico**
`do_opt` Invoca a função primária associada do parsing dos comandos
`get_args()` Devolve uma matriz contento os argumentos validados após feito o *parsing**
`get_var()` *Devolve a variável associada ao comando inserido*
`get_time_opt()` *Informa o tempo de duração do comando*
`get_status()` *Informa o **STATUS** do último comando*
`get_user_data()` *Devolve a informação adicional enviada para a função primária*

"Sem dados você é apenas
mais uma pessoa sem opinião"
W. Edwards Deming

Capítulo V

STATS

Conteúdo

Neste capítulo, faz-se uma apresentação e descrição do módulo **STATS**, responsável por interligar informação de *TABLES* distintas, otimizando consideravelmente alguma das queries.

5.1 Como interligamos os ficheiros?

Uma das partes essenciais para a otimização das **queries** é conseguir relacionar eficientemente as informações dos ficheiros **users**, **businesses** e **reviews**. Tivemos um *Trade-off* entre a quantidade de espaço que iríamos reservar para estas estruturas de otimização, e em que momento iríamos criá-las. Nossa decisão final foi a criação de *2 TABLES* para tal conceito de *interligação de informação*. Ambas possuem o mesmo *tamanho*, isto é, vão possuir o *número de ROWS* equivalente a quantidade de *business que possuem reviews*. E em termos de *momento de criação*, realizamos suas construções durante o **load_sgr()**. Mas o mais importante a sublinhar é que, uma dessas *TABLES* construímos durante o *upload do ficheiro reviews* e outra ocorre em *tempo linear* à quantidade de **ROWS** da *TABLE* business (numa operação de `table_join()`). Assim, em termos de *tempo de criação* e *espaço ocupado*, acreditamos ter encontrado um bom equilíbrio. Tais *TABLES* estão armazenadas na `struct sgr`, dentro de uma `struct`

`stats`. Para não haver dúvidas (pois reconhecemos a péssima escolha dos nomes), fica desde já esclarescido: a `struct stats` possui *2 TABLES*, uma chamada *STATS* e outra *STATS_2*. Assim, toda vez que ser dito ao longo do relatório que utilizamos o módulo *STATS*, estamos a nos referir a `struct stats`, e quando for dito que usamos a *TABLE STATS* e *TABLE STATS_2* estamos a nos referir as *TABLES* que estão armazenadas dentro da `struct stats`. Vejamos agora particularmente cada uma dessas *TABLES*.

5.2 TABLE STATS_2

A *TABLE STATS_2* é aquela que é preenchida durante o upload do ficheiro **reviews**. Para um melhor entendimento do contexto em que criaremos tal **TABLE**, temos de ter em conta quando preenchemos as **TABLES reviews, users e business**. Este processo é realizado dentro da query `load_sgr()`, nomeadamente pela função `fill_table()`. Porém, a função `fill_table()` já recebe a **TABLE** a qual será preenchida. Assim, antes de tal função, será encontrado algo do tipo:

```
TABLE * t = table_new(...);
```

E além do mais, a função `fill_table()` recebe 2 *TABLES*: uma que preencheremos com base no ficheiro que estamos a ler e outra que é justamente a *TABLE STATS_2*. Porém só chamamos tal função com um dos argumentos sendo a **TABLE STATS_2** quando formos ler o ficheiro **reviews**. Quando vamos fazer a `fill_table()` para qualquer outro ficheiro sem ser o de **reviews**, basta preencher a **NULL** o campo que seria da **TABLE STATS_2**. Assim, antes da `fill_table()` criamos a **TABLE STATS_2** da seguinte forma:

```
TABLE *t_stats = table_new(3, 0, "s | f | d ", "business_id",
"mean_stars", "quantity");
```

Como já foi explicado na secção **TABLE**, criaremos então tal **TABLE STATS_2** com 3 campos, com indexação com base em “business_id”. Agora restar invocar a `fill_table()` com a **TABLE reviews** e a **TABLE STATS_2** e tal TABLE será preenchida. Mas de que forma isso acontecerá?

A cada linha válida do ficheiro reviews, fazemos uma operação a qual resultará em algo que vamos inserir (ou atualização) na *TABLE STATS_2*. Após ler uma linha do ficheiro **reviews** e separar devidamente o conteúdo dela de acordo com os campos da *TABLE reviews*, filtramos da linha os seguintes dados:

- número de estrelas;
- business_id.

O que fazemos então é um `table_search_row_id()` do `business_id` na **TABLE STATS_2**. Esta **search** é o começo da “operação” citada anteriormente para inserir algo na *TABLE STATS_2*. Caso não for encontrado uma **ROW** indexada pelo *business_id*, o que faremos é inserir na **TABLE STATS_2** os campos filtrados no passo anterior (business_id e número de estrelas) e também o valor ***1*** no campo *quantity*, que simbolizará que nesse momento foi encontrado a primeira *review* desse *business_id*. A indexação usada nessa inserção será com base no campo *business_id*. O cenário muda quando a **search**, citada anteriormente, encontrar uma **ROW** indexada pelo *business_id*. Nesse caso, extraímos da **ROW** encontrada o valor do campo **mean_stars**, somamos ao

valor do "número de estrelas" filtrado da linha atual em análise do ficheiro **reviews** e dividimos pelo valor que está no campo "reviews" da **ROW** encontrada. Por fim, inserimos o valor obtido dessas operações matemáticas no campo **mean stars** e incrementamos em ***1*** o valor que está no campo **quantity**. Ou seja, o que está acontecer aqui é resumidamente uma atualização da média de estrelas de uma **ROW** que já existia na **TABLE STATS_2**. Assim, quando acabar o processo de `fill_table()` da **TABLE reviews**, acabaremos também por preencher a *TABLE STATS_2*, que será indexada por *business_id* e possuirá a quantiade de *reviews* correspondentes, bem como sua média de estrelas. Ter essa **TABLE** permitirá acessarmos em tempo ***O(log(n))*** a média de estrelas de qualquer *business_id*, que será útil em algumas queries.

5.3 TABLE STATS

O motivo pelo qual explicamos primeiro a **TABLE STATS_2** é que ela é criada e preenchida antes da **TABLE STATS** (nós já citamos que foi uma péssima escolha de nomes mesmo). Isto porque a **TABLE STATS_2** faz parte do processo de criação da **TABLE STATS**. Isto é, enquanto o foco da **TABLE STATS_2** é encontrar rapidamente a média de estrelas de um determinado *business_id*, o objetivo da **TABLE STATS** será muito mais ousado: agrupar continuamente *business_id's* de uma determinado cidade … E também por ordem crescente de **média de estrelas**! Criar tal **TABLE** foi sem dúvida um dos pontos altos do projeto, porque isto exigia uma certa complexidade e só iria compensar se fosse feito em um tempo aceitável.

Para conseguir criá-la, tivemos de disponibilizar na **API STable** uma função chamada `table_join()`. Esta função a priori foi concebida apenas para conseguirmos juntar **ROWS** de 2 **TABLES** que estejam igualmente indexadas. Passado um tempo ampliamos ela de tal modo que seria capaz escolhermos quais os **campos** de cada **TABLE** que seria "juntado" numa **TABLE final**. Assim já teríamos a chave para conseguir criar uma **TABLE** que seria o resultado de juntar as **ROWS** indexadas da **TABLE STATS_2** com as **ROWS** igualmente indexadas da **TABLE business**, escolhendo os campos `business_id, mean_stars, quantity, city e name` para fazer parte da **TABLE final**. Porém, apesar de já termos *média de estrelas, business_id e city* reunidas numa **TABLE** só, ainda faltava o famoso "algo a mais" … E é aqui que entra o conceito de **GKBDF** …

Generator **K**ey **B**y **D**iferent **F**ields

Tal conceito criado por nós (é o que julgamos pelo menos) surge para solucionar a problemática de como **indexar uma ROW numa TABLE com base em diferentes parâmetros**. A ideia consiste em precisar existir apenas um **campo de inexação único**. Por exemplo o *business_id* da **TABLE STATS_2** é único para cada **ROW**. Então pensamos, como seria possível deixar todas os *business_id's* de uma mesma cidade agrupados continuamente numa **TABLE**? Se usarmos o campo *city* como indexação teríamos o problema do **replace** (pois ao inserir algo com uma chave já existente na **árvore balanceada da glib**, acontece o "replace" desse nodo e não a adição). A solução que encontramos foi simples e bela: `concatenar a cidade com o business_id`. Assim, pelo facto do "business_id" ser único para cada **ROW**, automaticamente a string concatenada terá de ser única também. E repare, como numa árvore binária balanceada temos o invariante de `o nodo ser maior que todos os elementos da sub-`

`árvore da esquerda e menor que os da direita`, e como os conceitos de "maior e menor" aplicados a "strings" correspondem a ordem lexical por conta do código ASCII, conseguimos ter **ROWS** agrupadas por cidades, pois o começo da string concatenada citada anteriormente será igual para *business_id's* de uma mesma cidade. Mas não paramos por aqui, pensamos então: o que aconteceria se juntássemos a média de estrelas na concatenação dessa string? Bom, se a ordem for `cidade+média_de_estrelas+business_id` o que aconteceria é que, dado que o código ASCII "entende" a ordem dos digitos, teríamos a chave de indexação perfeita para agrupar *business_id's* de uma mesma cidade e por ordem crescente de média de estrelas. Para visualizar melhor:

`Imagine que tenha uma ROW com o business_id = abc, city = braga e média de estrelas = 3. Imagine agora outra com business_id = def, city = braga e média de estrelas = 4. Ao concatenar as strings de cada campo da ROW para formar uma chave de indexação, teríamos: braga3abc e braga4def. Ao comparar as 2 strings (em termos de ser maior ou menor com base no código ASCII) concluímos que braga3abc < braga4def justamente por 3 < 4 com base em ASCII. Podemos usar esse conceito para solucionar o problema de inserir ROWS com base na ordem crescente de média de estrelas!`

Tendo em conta isto, ampliamos a função `table_join()` de modo a receber um **function pointer** (e um **user_data** para carregar informação necessária para o function pointer) que será a **função de criação de chave de indexação**. No nosso caso, para a *TABLE STATS* tal função vai concatenar os campos **city + média de estrelas + business_id** e inserir a **ROW** (resultado do *join()*)numa *TABLE* com base na **key** criada.

Conseguimos fazer isso com uma travessia na *TABLE business* e por isso dizemos anteriormente que a criação de uma das *STATS TABLE* foi `em tempo linear a quantidade de ROWS da TABLE business`. Fazemos isto após já termos a *TABLE STATS_2* para fazer o `join()` com a *TABLE business* e tal *TABLE* mostra-se super essencial para a otimização da query *top_business_by_city()*.

"Tudo que uma pessoa pode imaginar,
outras podem tornar real"
Júlio Verne

Capítulo VI

Queries

Conteúdo

- **6.1** Pensando como um *puzzle*
- **6.2** Otimizações
- **6.3** Análise de complexidade

Neste capítulo, faz-se uma apresentação do **raciocínio** para construção de algumas queries, bem como suas **otimizações** e **análise de complexidade**.

6.1 Pensando como um *puzzle*

Tratamos a criação das queries assim como uma criança brinca com um *puzzle*. Em nosso caso as peças do *puzzle* são as funções de *search* disponibilizadas pela ***API STable***. Criamos as funções de *search* com o intuito de ser as ***mais genéricas possíveis*** para servir não exclusivamente para o nosso projeto e sim nosso projeto se servir delas.

6.2 Otimizações

Mostraremos agora algumas das otimizações mais relevantes que fizemos nas queries `3,4,5 e 6`.

6.2.1 Query 3

Declaração da função:

```
business_info (SGR sgr, char * business_id);
```

Para **otimizarmos** tal query recorremos ao módulo de **STATS**. Como já citado anteriormente, criamos 2 *TABLES* auxiliares com o intuito de servir de *ponte de ligação* entre **TABLES** distintas. Tais *TABLES* auxiliares foram concebidas de modo há, para além de relacionar *TABLES* diferentes, fazer tal relação de maneira otimizada. E um dos casos em que podemos ver isso é nessa query em análise. O que acontece nela é o seguinte:

```
ROW row = table_search_row_id (stats, &business_id);
```

Buscamos pela **ROW** indexada pelo *business_id* na *TABLE STATS_2*. Como sabemos, tal *TABLE* possui as seguintes informações: `business_id, média de estrelas do business_id, quantidade de reviews do business_id`. Como fazemos a busca pelo `business_id` e a *TABLE* está indexada justamente por tal campo, a busca é ***O(log n)***, onde **n** é o número de ROWS da *TABLE STATS_2*.

```
ROW business_row = table_search_row_id (business, &business_id);
```

De seguida (caso encontrarmos a **ROW** na busca anterior), fazemos uma nova busca, procurando pelo mesmo *business_id*, mas dessa vez na *TABLE business*. Isto porque precisamos preencher a *TABLE final* com alguns campos que a *TABLE STATS_2* não possui, e quem possui é justamente a *TABLE business* (esses campos são `name, city e state`). Como a *TABLE business* está indexada pelo campo *business_id*, a procura será ***O(log h)*** (onde ***h*** é o número de ***ROWS*** da *TABLE business*), tal como a anterior.

De resto basta juntarmos os dados anteriormente obtidos pelas procuras e ir acrescentando na *TABLE final*. Como as ***operações relevantes*** aqui podem ser reduzidas as 2 procuras anteriormente mencionadas, tal função possui uma complexidade ***O(log n)*** + ***O(log h)*** assimptóticamente.

6.2.2 Query 4

Declaração da função:

```
businesses_reviewed (SGR sgr, char * user_id);
```

Nessa query não recorremos ao módulo *STATS* na otimização. Fizemos proveito da *TABLE* armazenar informação numa **árvore binária balanceada** numa parte do código e percorremos algumas das *TABLES* noutras partes. Isto é, primeiramente criamos uma *TABLE* com todos os *business_id* em que o *user* dado como argumento fez *review*. Fazemos isso em:

```
TABLE * business_id = table_search_row (reviews, "user_id",
        headers_out, 2, &user_id);
```

A função `table_search_row()` disponibilizada pela **API STable** se enquadramente perfeitamente nesse primeiro processo de *filtrar*, isto porque com ela podemos escolher **o campo de comparação** para filtrar as *ROWS* (nesse caso é o campo "user_id") e também podemos escolher os campos que vão estar na *TABLE de retorno*, indicado por *headers_out* (escolhemos o *business_id* nesse caso). Como temos de percorer toda a *TABLE* para fazer tal processo de filtragem, o tempo de execução é ***O(n)***, onde ***n*** é a quantidade de ***ROWS*** da *TABLE reviews*.

De seguida temos de, para cada *business* da *TABLE obtida anteriormente*, procurar o seu *name* correspondente na *TABLE business*. Fazemos assim um *foreach* na *TABLE business_id* e para cada elemento, realizamos um `table_search_row_id()` na *TABLE business* e como tal *TABLE* está indexada por *business_id*, tal procura será feita em ***O (log z)***, onde ***z*** é a quantidade de ***ROWS*** da *TABLE business*. Tal processo de **fazer uma travessia na TABLE e realizar alguma operação** é feito através da linha:

```
table_search_foreach(business_id, businesses_reviewed_helper,
        (tpointer) user_data);
```

Onde **businesses_id** é a *TABLE* em que realizaremos a travessia, **businesses_reviewed_helper** é um *function pointer* que será responsável por realizar a operação de `table_search_row_id()` para cada **ROW** da *TABLE* em que se está a fazer a travessia e **user_data** é a informação adicional que o *function pointer* precisa receber para realizar seu propósito, como por exemplo a *return TABLE* que será preenchida durante a travessia.

Assim, podemos concluir que tal query é realizada em tempo de execução ***O(n) + O(y log(z))***, onde ***n*** é o número de **ROWS** da *TABLE reviews*, ***y*** é o número de ***ROWS*** da *TABLE* resultado da primeira operação de *search mencionada* e ***z*** é o número de ***ROWS*** da *TABLE business*.

6.2.3 Query 5

Declaração da função:

```
business_with_stars_and_city (SGR sgr, float stars, char * city);
```

Para otimizarmos tal função recorremos ao módulo de *STATS*. Primeiramente fazemos uma ***busca*** na *TABLE business* por todas as **os business que estão na cidade dada como argumento da função**. Utilizamos para tal a função `table_search_row()` disponibilizada pela **API Stable**. Com esta função podemos , para além de filtrar as **ROWS** a partir da comparação do campo *city*, escolher quais são os campos dessas **ROWS** filtradas que estarão na *TABLE de retorno*. Escolhemos então os campos `business_id e name` que são os campos que a *query* devolverá na *TABLE final*. Este procedimento dito é realizado em tempo ***O(n)***, onde ***n*** é o número de ***ROWS*** da *TABLE business* percorrida (como devemos filtrar todos os campos da *TABLE* a partir de um valor de comparação, obviamente temos de percorrer toda a *TABLE*). Este processo é realizado em:

```
TABLE * business_in_city = table_search_row (business, "city",
        headers_out, 2, &city);
```

O que temos agora é uma *TABLE* com todos os *businesses* da cidada dada no comando do usuário. Resta agora filtrar dessa *Table* apenas aqueles que possuem média de estrelas maior ou igual ao que foi dado no comando. E é aqui que entra a *TABLE STATS_2* para otimizar este processo. Isto porque tal *TABLE* já possui a média de estrelas de cada *business*, mas para além disso o que faz diferença aqui, é o facto de tal *TABLE* estar **indexada** por *businesses_id's*. Assim, para cada *business* da *TABLE business_in_city*, basta verificarmos sua média de estrelas na *TABLE STATS_2*. Como tal *TABLE* está indexada por **business**, a procura é ***O(log z)***, onde ***z*** é o número de ***ROWS*** da *TABLE STATS_2*. Este processo de *para cada ROW fazer alguma coisa* é realizado pela parte do código:

```
table_search_foreach(business_in_city,
        businesses_stars_and_city_helper, (tpointer) user_data);
```

O campo **businesses_stars_and_city_helper** é um **function pointer** que faz justamente algo com cada ROW que iremos visitar. O campo *user_data* é a informação que você julgar necessária para o *function pointer* anteriormente dito. Nesse caso enviamos diversos dados através de uma *struct auxiliar*, sendo que o mais relevante a saber é a *TABLE de retorno*, onde iremos preenchendo durante o *foreach*. Em geral, algumas das queries possuem uma estrutura semelhante, principalmente quando é realizado um *foreach*.

Sendo assim, dentro das opeações mais relevantes do código, temos a primeira *search* mencionada que é realizada em tempo de execução ***O(n)*** (onde ***n*** é o número de ***ROWS*** da *TABLE business*), e para cada elemento da *TABLE* resultante desta *search* faremos o *table_search_foreach* mencionado, em tempo ***O(y (log z))*** (onde ***y*** é o número de ***ROWS*** da *TABLE resultado do primeiro search* e ***z*** é o número de ***ROWS*** da *TABLE STATS_2*). Assim podemos concluir que está query é realizada em ***O(n) + O(y(log z))***.

6.2.4 Query 6

Declaração da função:

```
top_businesses_by_city (SGR sgr, int top);
```

Esta provavelmente é uma das queries em que conseguimos "melhor" **melhorar** nosso tempo. Na teoria tal query exige uma certa complexidade dado que temos de calcular os *top n businesses de cada cidade*. Para isso temos ter em conta que o número de estrelas e o campo de *cidade* estão em ficheiros distintos e ainda pior, inicialmente nem temos diretamente acesso a *média de estrelas* dos *businesses*. Vimos que essa ideia de **não relação direta** entre tais campos encerrou-se com a introdução do módulo *STATS*, onde a já referida *TABLE STATS_2* armazena a informação da média de estrelas, indexadas por *business_id*. Porém isso não era o suficiente para conseguirmos realmente otimizar a *query 6*. Faltava informação da *city* de cada business. O que nós fizemos então foi introduzir mais uma *TABLE* para o módulo *STATS*. Conhecida como *TABLE STATS* (diferente da *TABLE STATS_2* mencionada anteriormente), tal *TABLE* será o grande trunfo na execução de tal query. Como já foi explicado no capítulo **STATS**, a *TABLE STATS* possui indexação tripla a partir do conceito de **GKBDF** (generator key by diferent fields). Deixando de lado esta belíssima sigla criada por nós, usamos tal conceito de modo a já ter na *TABLE STATS* os businesses de cidades iguais seguidamente na *TABLE* e melhor ainda, em ordem crescente de *média de estrelas*. Assim, a única coisa que precisamos fazer é acessar o final da *TABLE* e fazer uma travessia de trás para frente, inserindo numa nova TABLE apenas os ***n*** businesses finais de cada secção de cidades iguais da *TABLE*. Fazemos isso em:

```
table_search_foreach_inverse (t, top_businesses_by_city_helper,
        user_data);
```

Onde ***t*** é a *TABLE STATS*, ***top_businesses_by_city_helper*** é o ***function pointer*** que vai realizar a parte lógica de selecionar apenas ***n*** businesses por cidade (os ***n finais***) e **user_data** é a informação adicional que o **function pointer** recebe, como por exemlo a *return TABLE* que será preenchida. Assim apenas fazemos uma travessia na *TABLE STATS* para realizar a query, por tanto realizamos ela em ***O(n)***, onde ***n*** é igual a quantidade de ***ROWS*** da *TABLE STATS* que acaba por ser a quantidade de *businesses que possuem reviews*.

6.3 Análise de complexidade

Nessa secção apresentaremos resumidamente a análise assimptótica, realizada anteriormente, dos temos de execução de cada **query** (para melhor informação das variáveis em questão, basta ver a seccão anterior):

- **Query 3**: ***O(log n) + O(log h)***
- **Query 4**: ***O(n) + O(y(log z))***
- **Query 5**: ***O(n) + O(y(log z))***
- **Query 6**: ***O(n)***

Capítulo VII

Testes de performance

Conteúdo

Neste capítulo, faz-se uma apresentação dos resultados obtidos em alguns comandos analisados.

7.1 Computador 1

Nas tabelas seguintes está exposto um breve resumo dos testes que fizemos para medir o desempenho do programa. Quando falamos em **desempenho** nos referimos ao **tempo de execução** de um comando face o **tamanho** dos argumentos envolvidos numa operação. Nesta secção trazemos as informações relativas a um dos computadores em que realizamos os *testes de desempenho*, assim como os resultados dos testes em função de alguns comandos possíveis de se fazer no programa.

Tabela 1: Características do <computador 1> usado nos testes de desempenho

Processor	# of Cores	# of Threads	Frequency (GHz)	Max Turbo Frequency	RAM (Gb)
AMD A6-7310 APU	4	4	2.0	2.4	8

Tabela 2: Resultados dos testes de desempenho do computador 1

Comando	Tempo de execução (seg)
load_sgr (*default files*)	**~ 62**
businesses_started_by_letter (*sgr*, *'A'*)	**~ 0.16**
business_info(*sgr*, *"0IWCmdWbv1w-5IgpKqtY8A"*)	**~ 0.000075**
businesses_reviewed (*sgr*, *"Jx5CzS0sVXf5oXbGKhhgHA"*)	**~ 0.0001**

businesses_with_stars_and_city(*sgr*,*3*,"*Jamaica Plain")	**~ 0.1**
top_businesses_by_city(*sgr*,*5*)	**~ 0.037**
international_users(*sgr*)	**~ 4.8**
top_businesses_with_category(*sgr*,*5*,*"Food"*)	**~ 0.25**
reviews_with_word(*sgr*,*"Carrots"*)	**~ 18**

7.2 Computador 2

Nessa secção trazemos as informações relativas ao segundo computador que utilizamos para testar o desempenho de nosso programa.

Tabela 3: Características do <computador 2> usado nos testes de desempenho

Processor	# of Cores	# of Threads	Frequency (GHz)	Max Turbo Frequency	RAM (Gb)
Intel i7-9750H	**6**	**12**	**2.60**	**4.50**	**8**

Tabela 4: Resultados dos testes de desempenho do computador 2

Comando	Tamanho do ficheiro (segundos)
load_sgr (*default files*)	**~ 22**
businesses_started_by_letter(*sgr*,*'A'*)	**~ 0.055**
business_info(*sgr*, *"0IWCmdWbv1w-5IgpKqtY8A"*)	**~ 0.00005**
businesses_reviewed(*sgr*, *"Jx5CzS0sVXf5oXbGKhhgHA"*)	**~ 0.000075**
businesses_with_stars_and_city(*sgr*,*3*,*"Jamaica Plain"*)	**~ 0.042**
top_businesses_by_city(*sgr*,*5*)	**~ 0.001**
international_users(*sgr*)	**~ 1.9**
top_businesses_with_category(*sgr*,*5*,*"Food"*)	**~ 0.1**
reviews_with_word(*sgr*,*"Carrots"*)	**~ 8**

“Planear é uma arte …
A arte de antecipar o futuro”
Henrique Costa

Capítulo VIII

Organização

Conteúdo

Neste capítulo, faz-se uma descrição de como nosso grupo de organizou, tanto a nível de código, tanto a nível comunicativo.

8.1 Organização estrutural de pastas

Seguindo o enunciado, utilizamos implicitamente (e explicitamente também) o conceito MVC de modularidade. Assim, estando dentro da pasta root do projeto, serão encontrado os seguintes ficheiros e pastas:

```
controller model view main.c Makefile mydConfig.cfg README.md
report
```

Indo agora dentro de uma das pastas `controller`. `model` ou `view` o seguinte padrão será encontrado:

```
.___.module1
|   |___.src
|   |   |
|   |   |__ file1.c
|   |   |
|   |   |__ file2.c
|   |
|   |___.includes
|       |
|       |__ file1.h
|       |
|       |__ file2.h
|
|___.module2
|   |___.src
|   |   |
|   |   |__ file3.c
```

```
|   |   |
|   |   |__ file2.c
|   |   (...)
|   |
|   |___.includes
|       |
|       |__ file1.h
|       |
|       |__ file2.h
|      (...)
(...)
```

Isto porque optamos por escolher o padão MF (Module First) de organização de pastas. Tal método estipula que os ficheiros de um mesmo tema devem ser agrupados dentro de um módulo e, cada módulo terá suas próprias pastas `src` e `includes`. Escolhemos proceder estruturalmente assim para poder ter uma visão mais independente de cada módulo a nível organizacional e funcional. Isto porque imagine o seguinte cenário:

`Foi decidido que o José Pedro será o responsável pelo módulo TABLE`

Assim, o José acabará por ter de criar tudo a respeito do tema TABLE de acordo com as regras anteriormente ditas. Se ele quiser pode até adicionar sua própria `main_table.c`, `Makefile` e `README.md` dentro da pasta TABLE, ressaltando ainda mais a modularidade e independência de seu código. E usando ainda este exemplo, vemos que o módulo TABLE acabou por ficar do seguinte modo:

```
.table
|___.src
|   |__ generic.c
|   |__ helpers.c
|   |__ memory.c
|   |__ search.c
|   |__ table.c
|
|___.includes
|   |__ generic.h
|   |__ helpers.h
|   |__ memory.h
|   |__ search.h
|   |__ table.h
|   |__ table_p.h
|
|___ TEST_table.c
|___ Makefile
|___ README.md
```

A beleza disto é que quem olha apenas para um módulo é capaz de ver um `projeto independente`. Assim, no fundo, temos vários `sub-projetos` dentro do grande `projeto`.

Separamos os módulos em:

- **sgr;**
- **table;**
- **pagination;**
- **messages;**
- **interpreter;**

Dentro do módulo `table` está a API STable e dentro do `ìnterpreter` está a API SInterpreter. O módulo de messages possui os warnigs e informações para promover a interação `usuário-programa`. O módulo `sgr` possui as queries, funções de parsing e declaração de `estruturas` específicas do trabalho pedido pelo enunciado. Por fim, o módulo `pagination` possui uma extensão da API STable, que foi separada do módulo *table* para manter a estrutura MVC.

8.2 Tratando bem o Github

Tendo em conta a devida atenção dada pelos professores a utilização do Github, nós retribuímos com mais detalhes ainda. Aqui entra a secção de comunicação da equipa. De modo a garantir um padrão ao longo do projeto e já manter uma postura mais profissional, utilizamos o conceito de commits semânticos. Para explicar esta parte, segue um excerto de nosso prórpio README.md:

`For best pratices, in order to help to track specified information and create a beautiful CHANGELOG, we following the Convential Commits: https://www.conventionalcommits.org/en/v1.0.0/ that is inspired by on the Angular Commit Guidelines: https://github.com/angular/angular/blob/22b96b9/CONTRIBUTING.md#-commit-message-guidelines`

Assim, todos nossos commits possuem a seguinte estrutura:

`<type>[optional scope]: <description>`

Um exemplo real foi quando o Henrique alterou o código das *time-functions* do interpretador, para uma nova versão:

`refactor(interpreter): change <time-functions>`

Para uma melhor leitura dos commits todos devem seguir a regra mais importante: serem consistentes semânticamente e sintáticamente com a frase:

`Ìf applied, this commit will <will your subject line here>`

Quanto ao nome dados as branches, seguimos o padrão:

`<branch-user-name>_<branch type>_<branch-name>`

Um exemplo disto foi quando o josé foi testar as queries:

`[BRANCH-NAME] jose_tests_queries`

Conclusão, seguimos regras, nos organizamos com *commits semânticos e atómicos*, assim como *branches por temas*, como foi tudo anteriormente caracterizado. Tais informações podem ser acessadas também em nosso README.md onde separamos uma secção para aqueles que um dia podem querer contribuir para o projeto, ter em conta as normas de dinamização de equipa. Assim, acreditamos ter respeitado o modelo MVC e ainda grantindo uma base bem sólida de organização tanto a nível de código, como entre os membros de equipa.

Reflexão final

O nosso grupo reconhece tantos aspetos positivos como negativos do trabalho. Estamos conscientes que faltou uma certa camada de abstração entre a API STable e os devidos *catálogos de users, reviews e businesses*. Entendemos isto, como também sabemos que é completamente acessível acrescentar tal camada em nosso projeto. Porém, este ponto que faltou em nosso trabalho, acreditamos ter compensado (de uma certa forma) com algo tão grandioso quanto o que foi pedido. Passamos desapercebidos pelos catálogos citados pois o que nós queríamos fazer era criar algo que não servia apenas para o nosso projeto. Sim, por um lado o catálogo de *users* (por exemplo) seria uma espécie de API para estudar e analisar ficheiros como o *users_full.csv*. Mas querendo ou não, existe uma certa limitação desta ideia cujo propósito se prende apenas ao nosso projeto. Acreditamos que o que fizemos ao criar a Standart Table tenha sido mais ousado e conquista um dos grandes pilares que estamos a aprender atualmente: `fazer código genérico`. Criamos algo que pode ser comparável (em sua devida proporção) as bibliotecas de GTree's ou HashTables da GLib. Mas na nossa, oferecemos um módulo para tratar de informações representadas na forma de tabelas (de forma super otimizada). Em qualquer outro projeto que precise simular uma *base de dados*, já temos algo feito e que sabemos que vai servir, independente dos dados que lidaremos. Por consequência disto acabos por tratar os ficheiros todos da mesma forma, causando a falta daquela camada de abstração citada anteriormente. Nossa conclusão e reflexão foi que, acreditamos ter proporcionalmente compensando a falta da *camada* citada com a criação da STandart Table. Querendo ou não, podemos adaptar nosso código para obedecer mais o enunciado e muito provavelmente não ter tratado os catálogos de forma específica seja algo penalizador, mas acreditamos ter criado algo com grande potencial que cumpre com nosso real objetivo: `pensar para além do nosso próprio trabalho`. E de bônus ainda criamos uma API que pode ser utilizada por

qualquer pessoa que queira fazer uma interface de linha de comando, e que é acessível a *updates* e melhorias, sem estar presa a um tipo específico de sintaxe de comandos.