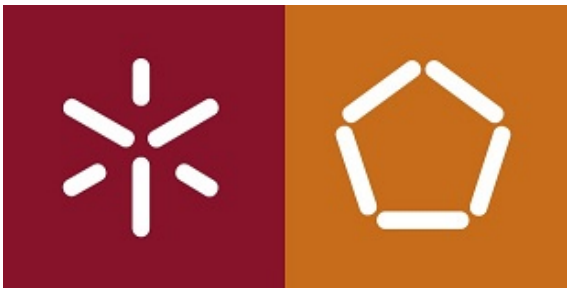

Universidade do Minho
Departamento de Informática

Sistema de Gestão de Recomendações

Laboratórios De Informática III

Grupo 54

5 de Maio, 2021


Mariana Rodrigues
(a93229)


Matilde Bravo
(a93246)


Pedro Alves
(a93272)

Contents

Chapter 1

Introdução

O presente relatório tem como objetivo apresentar o projeto realizado no âmbito da unidade curricular de Laboratórios de Informática III, ao longo do segundo semestre, do segundo ano, do Mestrado Integrado em Engenharia Informática da Universidade do Minho. Este consiste na criação de um programa capaz de ler , armazenar e gerir toda a informação válida contida em ficheiros *csv*(s), recolhida da base de dados do **Yelp**. Os dados são armazenados de modo a responder aos pedidos do utilizador da forma mais rápida e eficiente possível.

1.1 Descrição do Problema

Serão 3 os ficheiros *csv*(s) que deverão ser processados: *users.csv, businesses.csv e reviews.csv*, cada linha nesses ficheiros deverá conter um registo com a informação de um utilizador, business e review, respetivamente.

Um dos objetivos deste trabalho é armazenar toda a informação contida nesses ficheiros, nas estruturas de dados mais adequadas para cada fim e respondendo aos pedidos do utilizador sobre as mesmas.

1.2 Análise da Solução

Numa primeira fase averiguamos quais as estruturas de dados que melhor se adequavam à resolução dos problemas que enfrentávamos. A nossa escolha de organização teve sempre em vista a rapidez das queries, bem como, a menor utilização de memória possível. Durante este processo, tivemos também em atenção o encapsulamento e a modularidade do programa, bem como as boas práticas da linguagem.

Chapter 2

MODELO

A finalidade deste capítulo é expôr detalhadamente, e com a devida fundamentação, a nossa abordagem para o armazenamento, gestão de dados e resolução das **Queries** do utilizador. Esta componente constitui a maior parte da lógica do nosso programa.

2.1 SGR

Face à necessidade de responder a todos os pedidos do utilizador, foi criada a estrutura de dados opaca designada por *SGR*. Esta estrutura de dados armazena outras quatro, as quais iremos abordar posteriormente. Constitui, essencialmente, a "base de dados" do nosso programa.

sgr.c

```
struct sgr {
  UserCollection catalogo_users;
  BusinessCollection catalogo_businesses;
  ReviewCollection catalogo_reviews;
  Stats estatisticas;
};
```

2.2 Users

Este módulo é responsável pelo armazenamento de todos os utilizadores provenientes de registos válidos do ficheiro ***users.csv***.

user.c

```
struct user_collection {
  GHashTable *by_id; // <char* user_id, User user>
};
```

Cada um dos ***user***(s) deverá conter um *nome*, *id* e uma lista de *amigos*. Perante os desafios propostos e atráves de uma análise cuidada, decidimos que não iriamos guardar a informação relativa aos *amigos*. Uma vez que cada um dos ***user***(s) terá um único e exclusivo *id*, decidimos usar uma ***Hash Table***, onde a *key* seria o *id* correspondente a cada ***user*** e o *value* seria uma estrutura de dados do tipo *User*. Esta organização será conveniente, uma vez que, consultar o nome de

um utilizador dado o seu id será uma ação muito frequente, nomeadamente para a resolução de queries, e, desta forma, conseguimos realiza-la em tempo constante. Segundo a documentação da glib, as operações de inserção e remoção têm um custo amortizado O(1) e operações que impliquem a sua travessia em tempo O(n). Naturalmente, a estrutura de dados **User** conterá informação relativa a um user enquanto que a estrutura **UserCollection** corresponderá ao catálogo de Users que engloba todos estes na Hashtable. Perante a análise das queries relacionadas com os users, concluiu-se que não haveria mais nenhuma estrutura de dados necessária para as facilitar.

2.3 Businesses

Este módulo é responsável pelo armazenamento de todos os negócios e a sua respetiva informação dada pelo ficheiro ***businesses.csv***.

business.c

```
struct business_collection {
  GHashTable *by_id;
  PerfectHash by_letter;
};
```

À semelhança dos *users(s)*, e como já mencionado, um **Business** possui um identificador único que será frequentemente utilizado para consultar informação sobre um determinado negócio. Consequentemente e, pelos mesmos motivos, optamos por criar uma hashtable que estabelece a correspondência entre id e estrutura de dados **Business**. Perante a análise da Query 2, concluiu-se que seria oportuno ter também os negócios organizados por forma a facilmente obter todos aqueles começados por uma dada letra. Naturalmente, essa procura poderia ser feita em tempo de execução e a pedido mas não consideramos que faria sentido, uma vez que percorrer todos os negócios é muito custoso e lento. enquanto que adicionar à medida que é lido é bastante eficiente. A inserção de um pointer, neste caso, uma string correspondente à categoria, nesta estrutura de dados, é feita, no melhor caso, em O(1) e no pior caso em O(N);

Além da hashtable previamente mencionada, a **BusinessCollection** contém uma outra hashtable para facilitar a Query 2. Poderíamos ter utilizado mais uma hashtable da glib, contudo, esta ocuparia muita memória, nomeadamente devido à presença de mecanismos de tratamento de colisões. Por exemplo, as HashTables da glib guardam sempre as chaves na tabela em si, por forma a determinar se de facto ocorreu uma colisão ou se se trata de uma mera substituição. Escusado será dizer que armazenar todas as chaves e valores ocupa mais memória do que armazenar meramente os valores. Perante a análise desta Query , chegou-se à conclusão que seria possível desenvolver uma função de hash perfeita, isto é, uma função em que chaves diferentes terão garantidamente hashes diferentes. Consequentemente, garante-se que não ocorrerão colisões e , logo, não há necessidade de armazenar as chaves. Por esse motivo, foi criado um módulo cuja responsabilidade é disponibilizar a estrutura e funções para a perfect hashmap. Apesar das hashtables da glib funcionarem perfeitamente, consideramos que estas ocupavam mais memória e não traziam nenhuma vantagem para este **usecase** específico. Desta forma, conseguiu-se reduzir significativamente a utilização de memória do programa.

2.4 Reviews

Este módulo é responsável por armazenar todas as reviews provenientes do ficheiro ***reviews.csv***. ***review.c***

```
struct review_collection {
  GHashTable *by_id;
  GHashTable *by_user_id;
  GHashTable *by_business_id;
};
```

Cada ***review*** terá um *id* exclusivo e único, um ***user_id***, um ***business_id***, um número de ***stars***, inteiros ***useful***, ***funny***, ***cool***, uma ***date*** correspondente a quando foi realizada e um ***text***.

Com o intuito de armazenar todas as ***review***(s) válidas obtidas através da leitura, começamos por usar a mesma estratégia usada tanto no ***UserCollection***, como no ***BusinessCollection***. Usando uma ***Hash Table***, onde a *key* seria o *id* correspondente a cada ***review*** e o *value* seria uma estrutura de dados do tipo *Review*. Visto que o mesmo ***User*** poderá ter feito várias ***Review***(s) e que, o mesmo se pode aplicar a *Business*(s), isto é, a cada ***Business*** poderá ser atribuída uma lista de ***Review***(s), determinamos que a melhor maneira de guardar esta informaçao seria numa ***Hash Table***. Onde a *key* correspondente seria um ***id*** (ou um *business_id* ou um *user_id*), e o seu *valor* a guardar seria um ***GPtrArray ** com todas as ***Review***(s) equivalentes a esse mesmo *id*.(*GHashTable *by_user_id, GHashTable *by_business_id*, respetivamente).

2.5 Stats

Este módulo é responsável pelo cálculo e armazenamento das estatísticas das estrelas dos dados recolhidos dos 3 ficheiros. Este módulo, ao contrário dos três principais, não se limita a armazenar os valores lidos nas estruturas apropriadas, pois também realiza um cálculo da média à medida que a leitura é feita. Poder-se-ia ter lido todas as reviews e, posteriormente, calculado a média das estrelas do negócio e criado uma lista ordenada. Contudo, isso implicaria uma travessia por todas as reviews que achamos que poderia ser evitada mantendo um registo da média atual de estrelas com as reviews lidas até então e atualizando esse valor à medida que uma nova review desse negócio é lida. Desta forma, conseguimos diminuir o tempo de execução sem custos significativos a nível de memória. Cada vez que seja necessário obter o número de estrelas de um negócio, é necessário apenas fazer um lookup na primeira hashtable, business_id_to_stars e, desta forma, obtem-se, em tempo constante, este valor. Na query 5 é pedida uma ordenação dos negócios, quanto ao número de estrelas, numa dada cidade. Por esse motivo, foi criada uma hashtable em que a key é o nome de uma cidade e o value é uma lista ligada de negócios, organizada de forma decrescente quanto ao número de estrelas. Esta estratégia pode ser aplicada não só à query 5 como também à query 6. Optamos por utilizar listas ligadas porque o que se pretende não é uma mera procura de um valor, mas sim, recolher todos os valores até o número de estrelas pretendido ser encontrado. Desta forma, não são feitas travessias inúteis, pois todos os valores percorridos são valores que vão ser utilizados no resultado. A terceira hashtable serve um propósito semelhante à segunda, com a diferença de que a key é uma categoria ao invés de uma cidade.

```
struct stats {
  // uppdated as a new review is read, business_id to a StarsNode
  GHashTable *business_id_to_stars;
  // linked list of businesse sorted decrescently by stars
  GHashTable *city_to_business_by_star;
  // linked list of businesses sorted decrescently by category
  GHashTable *category_to_business_by_star;
};
```

2.6 Leitura

Este módulo tem como objetivo ler os três ficheiros CSV disponibilizados e chamar as funções necessárias para inicializar a SGR. Uma vez que o tamanho de uma linha não é conhecido previamente, optou-se por não criar nenhum buffer de tamanho fixo, uma vez que isso implicaria que o programa fosse, de alguma forma, falível. Portanto, optou-se por utilizar uma espécie de um array dinâmico de caracteres para guardar o conteúdo de cada linha à medida que a leitura é feita. Cada um dos três tipo de dados (Business, Review e User) tem a sua própria função de parsing que é responsável por processar uma linha de um dado CSV e criar a correspondente estrutura de dados com os valores lidos. Antes desta criação, cada linha é validada, tendo em atenção se, por exemplo, uma review não é referente a um User ou um a Business não existente. Além da leitura propriamente dita , por uma questão de eficência, são chamadas outras funções, como por exemplo a **call_update_Average_stars**, função responsável por criar ou atualizar os valores da hashtable das estatisticas relativos à média de estrelas. Naturalmente, o cálculo da média poderia ser feito previamente, mas, desta forma, poupamos várias travessias ao conjunto de reviews para realizar este cálculo, as quais teriam um impacto significativo na performance. No fim da leitura estarão criados os três catálogos necessários para inicializar a **SGR**.

2.7 Table

A estrutura table é uma parte crucial do programa e dita como a maior parte dos dados a ser apresentados pela view estão estruturados. Uma table contem sempre um header que indica o nome dos campos a ser apresentados. O conteúdo de cada linha está guardado num array de pointers, GPtrArray, o qual se assemelha a uma matriz, pois tem todos os seus elementos contíguos(char*) e é possível aceder a uma linha e coluna dado um índice, recorrendo à devida função de indexação. Uma table pode apresentar zero ou mais footers, os quais são essencialmente linhas com informação adicional, como por exemplo, o número total de elementos da tabela.

```
struct table {
  size_t number_fields;
  size_t number_footers;
  char **header;
  char **footer;
  GPtrArray *lines;
};
```

2.8 AST

Este módulo contém as estruturas de dados necessárias para fazer *parsing* dos comandos do programa.

O *parsing* é feito em duas fases. A primeira é a **tokenização**, que converte uma linha de texto em vários *tokens* - fragmentos que representam partes distintas de um comando. Os tipos destes *tokens* foram escolhidos com base na sintaxe da linguagem de comandos do programa:

```
typedef enum token_type {
    TOK_NAME,        // e.g. x, avg, fromCSV
    TOK_STRING,      // e.g. "business.csv"
    TOK_NUMBER,      // e.g. 0, 15
    TOK_FLOAT,       // e.g. 0.13, 20.4
    TOK_EQUALS,      // =
    TOK_COMMA,       // ,
    TOK_OPAREN,      // (
    TOK_CPAREN,      // )
    TOK_SEMICOLON,   // ;
    TOK_OSQ,         // [
    TOK_CSQ,         // ]
    TOK_OBRACKET,    // {
    TOK_CBRACKET,    // }
    TOK_FINISH       // Token falso que indica o final dos elementos
} TokenType;
```

É de notar que não existe nenhum *token* para espaços, pois estes são ignorados na sintaxe do programa.

A estrutura de dados para o *token* é uma simples estrutura que contém o seu tipo, o seu texto e a sua posição (para reportar erros). Estas estruturas são construídas através da função ***split_line***, que converte uma *string* numa lista de *tokens*, terminada pelo *token* de tipo TOK_FINISH.

Terminada a tokenização, chegamos à segunda fase, o *parsing* em si, em que convertemos a lista de *tokens* numa ***AST - Abstract Syntax Tree***. A AST é uma estrutura de dados que representa a linguagem numa estrutura de árvore, através de uma união de tipos:

```c
typedef enum ast_type {
  AST_NONE,          // AST vazia
  AST_FUNCTIONCALL,  // Chamada de uma funcao
  AST_ASSIGNMENT,    // Declaracao de uma variavel
  AST_VARIABLE,      // Referencia a uma variavel
  AST_NUMBER,        // Um numero inteiro
  AST_FLOAT,         // Um numero de virgula flutuante
  AST_STRING,        // Uma string
  AST_INDEX,         // Indexacao de um array/tabela
  AST_ARRAY          // Um array
} ASTType;

typedef struct ast *AST;

struct ast {
  ASTType type;
  union {
    FunctionCall function; // contem nome da funcao e array com
                           // AST dos argumentos
    VarAssignment assignment; // contem nome da variavel
                              // e AST da expressao
    Indexed index; // contem AST tanto do valor a ser indexado
                   // como do indice
    char *variable;
    char *string;
    int number;
    float float_num;
    GPtrArray *array; // array de AST
  } value;
};
```

2.9 State

Este módulo é responsável pela gestão das definições de variáveis do interpretador. A principal estrutura de dados deste módulo é a estrutura ***Variable***, que utiliza uma *tagged union* para poder conter vários tipos de dados:

```
typedef enum variable_type {
  VAR_NUMBER,
  VAR_FLOAT,
  VAR_SGR,
  VAR_TABLE,
  VAR_STRING,
  VAR_FUNCTION,
  VAR_ARRAY,
  VAR_OPERATOR,
  VAR_VOID,
  // O tipo ANY nunca e dado a uma variavel - serve apenas para
  // especificar que
  // qualquer parametro e valido ao definir uma funcao.
  VAR_ANY
} VariableType;

typedef union {
  int number;
  float float_num;
  SGR sgr;
  TABLE table;
  char *string;
  struct function *function;
  GPtrArray *array;
  OPERATOR operator;
} VariableValue;

struct variable {
  VariableType type;
  int references;
  char *name;
  VariableValue value;
};
```

Para poder libertar memória quando as variáveis já não são necessárias, é utilizada uma estratégia de contagem de referências. Quando as referências forem 0, significa que a variável pode ser libertada.

Uma coisa interessante sobre este módulo é que, as funções são também representadas como variáveis, que apontam para uma estrutura **function**. Esta estrutura contém informações como o número de argumentos, os seus tipos, e um apontador para a função de C a ser chamada. Esta estratégia permitiu reduzir bastante a complexidade à volta da chamada de funções, permitindo tratá-las como simplesmente mais um objeto do nosso interpretador, e não como um caso especial.

O estado em si é representado através de uma árvore binária, que faz um mapeamento entre o nome de uma variável e o seu valor. Esta estrutura foi escolhida para facilitar uma possível implementação de *tab-complete* no futuro, pois permite fazer uma procura eficiente a partir de um prefixo, mas infelizmente não foi possível implementar esta funcionalidade a tempo da entrega.

O estado interno do interpretador pode ser visualizado a qualquer momento na forma de tabela através da função **state()**.

2.10 Auxiliary

Este módulo engloba pequenas funções que são utilizadas com muita frequência e por diversos outros módulos. Foram criadas por forma a permitir a reutilização do código e , naturalmente, respeitam o encapsulamento.

2.11 Perfect Hash

Como outrora mencionado e, como o próprio nome indica, este módulo disponibiliza uma estrutura designada por perfect hashmap e funções para lidar com a mesma.

perfect_hash.c

2.12 Config

Este módulo tem como responsabilidade ler um ficheiro de configuração que indica o caminho para o ficheiro CSV de users, businesses e reviews. Esta config é um CSV pois é um tipo de ficheiro bastante conveniente para representar key value e pairs e, naturalmente, desta forma, podemos reutilizar todas as funções que leem e manipulam csv(s), incluindo o tipo de dados TABLE.

Chapter 3

APRESENTAÇÃO

3.1 Linha de comandos

A linha de comandos foi implementada utilizando o **readline**, uma biblioteca da GNU que fornece uma interface simples de linha de comandos utilizada, por exemplo, no bash. Esta biblioteca permitiu que tivéssemos, sem nenhum esforço adicional, funcionalidades como *tab-completion* para nomes de ficheiros e um histórico de comandos (acessível com as setas para cima e para baixo), utilizando também a configuração do readline do utilizador, se esta existir.

3.2 Paginação

O módulo da paginação é responsável pela visualização de tabelas numa interface de *pager*. Este *pager* foi implementado puramente em C, sem nenhuma biblioteca, utilizando os códigos de controlo de terminal ANSI, de modo a manter o máximo controlo possível para a visualização.

A tabela foi desenhada utilizando os caracteres de desenho de caixa, permitindo uma aparência contínua e uniforme.

Ao desenhar uma tabela, o programa primeiro obtém o tamanho do terminal, utilizando a função **ioctl**. Esta informação é utilizada para calcular o número de linhas a mostrar no ecrã. Se as linhas da tabela não couberem todas no ecrã, o módulo entra num modo de *paging* e configura o terminal para isso, utilizando as funções **termios** para desativar *buffering* ao mudar de linha (pois se não só conseguiríamos receber os eventos do teclado quando o utilizador carregasse no *enter*). Utilizamos, também, uma sequência de controlo de terminal para esconder o cursor.

Depois disto, imprimimos as linhas da tabela, e esperamos por um evento do teclado. Se for uma seta, mexemos o cursor para cima, de modo a ficar no início da tabela, apagamos o ecrã para baixo do cursor e imprimimos a tabela de novo. Se for a letra 'q', saímos do modo de tabela, repondo todas as configurações anteriores, e saímos da função.

Chapter 4

CONTROLADOR

4.1 Parsing

Este módulo é responsável por converter uma lista de *tokens* numa **AST**. Para o fazer utiliza o método *recursive descent*, em que temos várias funções que interpretam pequenas partes da linguagem a chamarem-se umas às outras para construir uma AST final:

```
SyntaxError parse_function(const TokenStream tokens, AST node,
                           int *consumed);
SyntaxError parse_assignment(const TokenStream tokens, AST node,
                             int *consumed);
SyntaxError parse_expression(const TokenStream tokens, AST node,
                             int *consumed);
SyntaxError parse_statement(const TokenStream tokens, AST node,
                            int *consumed);
SyntaxError parse_array(const TokenStream tokens, AST node,
                        int *consumed);
SyntaxError parse_index(const TokenStream tokens, AST node,
                        int *consumed);
```

Cada uma das funções devolve um **SyntaxError**, que serve para indicar um erro de sintaxe.
Utilizando esta técnica, o parser principal (`parse_statement`) funciona desta forma:

1. a) Tenta ler uma declaração de variável
   b) Se não encontrar uma, tenta ler a chamada de uma função
2. Tentar ler um ponto e vírgula (*token* **TOK_SEMICOLON**)

Se estes passos forem sucedidos, então devolvemos NULL, para indicar que não houve erros, devolvemos a AST correspondente à operação que encontrámos e indicamos quantos *tokens* consumimos.

Sabendo isto, é fácil ver como os outros *parsers* funcionam. Por exemplo, para o **parse_assignment**:

1. Tentamos ler um nome de uma variável (**TOK_NAME**)
2. Tentamos ler um igual (**TOK_EQUALS**)
3. Tentamos ler uma expressão

Esta estrutura permite-nos adicionar sintaxe à linguagem facilmente, e permite também fornecer erros úteis ao utilizador.

4.2 Execução

Este módulo consiste, essencialmente, de uma única função, que executa o código de uma AST, tendo em conta um estado do programa. Devolve uma **Variable**, do módulo de estado.

A função em si é bastante simples, vendo o tipo da AST e chamando-se recursivamente para calcular o valor de sub-árvores. A maioria da complexidade da função vem da parte que chama funções, pois esta tem de fazer toda a verificação de argumentos para os passar à função C correspondente.

Chapter 5

Conclusão

Perante os desafios propostos e a solução produzida, concluímos que cumprimos todas as tarefas dadas. Todavia, entendemos que existem alguns aspetos que, eventualmente, poderiam ser melhorados. Dever-se-ia ter analisado e estudado o nosso programa, em termos de desempenho face a diferentes tamanhos de ficheiros.

Para finalizar, consideramos que conseguimos alcançar todos os objetivos, bem como, implementar funcionalidades extra no nosso programa, obtendo resultados bastante satisfatórios nos testes por nós realizados.

Appendix A

Funcionalidades extra

- load_sgr como uma função que pode ser usada pelo utilizador dentro do programa. Pode ser chamada mais que uma vez e o seu resultado pode ser guardado numa variável. Desta forma é possível ter várias SGR ou até mesmo nenhuma. **load_sgr();**
- Módulo responsável pela leitura de um ficheiro de configuração(config.h), a qual indica o caminho para os csv, havendo na mesma a possibilidade de dar *override* aos mesmos dentro do programa.
- Help pages que indicam os argumentos, return value e propósito da função. **help(filter);**
- Possibilidade de visualizar todas as funções, variáveis e constantes disponíveis dentro do programa sob a forma de uma tabela. **state();**
- Criação de tipos de dados, a nível do interpretador, como por exemplo : **String**, **Float**, **Num**, **Array**, sendo possível manipula-los e guarda-los em varíaveis de outros tipos de dados
- Implementação de uma espécie de *Garbage Collector* que recorre ao método de *Reference counting*.
- Implementação dos comandos **count**, **join**, **max**, **min**, **avg**.
- Comando print que possibilita a impressão de qualquer tipo de dados implementado pelo programa. **print(5.0);**

Para assistir ao exemplo de uma utilização do programa e demonstração de algumas funcionalidades extra poderá clicar **aqui**.

Appendix B

Estatísticas de execução

B.1 Análise relativa aos Users

Face aos desafios propostos, verificamos que não iriamos necessitar da informação relativa aos friends de cada user. Perante isso, fomos analisar a performance da nossa aplicação sem e com a armazenação desses dados. Inicialmente, esses dados eram guardados a partir de um **gptr array**. (Nota: Estes dados foram obtidos no decorrer deste projeto.)

Comparação de Tempos	Texec (s)	Memória (GB)
Load SGR (com Friends)	8.4	4.01
Load SGR (sem Friends)	7.388	1.47
Load Users (com Friends)	4.08	2.79
Load Users (sem Friends)	2.55	0.28

Com isto, podemos observar que, em termos de tempo não se verifica grande alteração. Por outro lado, apercebemo-nos que a memória gasta pela nossa aplicação é reduzida em mais de metade.

B.2 Queries

Query 1
Ficheiros : Users (2.7 G), Businesses (17 M), Reviewns (700 M)

Texec (s)	Memória (GB)
8.352	1.31
8.395	1.30
8.366	1.24
8.617	1.29

Query 2

	Argumentos recebidos: sgr e ...		**Texec (s)**	**Número de dados obtidos**
Query 2	*Letra*	'a'	0.021940	11327
		'h'	0.007191	5889
		'z'	0.002016	796
		'Z'	0.001545	

	Argumentos recebidos: sgr e ...		**Texec (s)**	**Número de dados obtidos**
Query 3	*Bussiness id*	"ak0TdVmGKo4pwqdJSTLwWw"	0.000002	0 (Não existe)
		"6iYb2HFDywm3zjuRg0shjw"	0.000018	1
		"t35jsh9YnMtttm69UCp7gw"	0.000017	
		"Lye66-VKsO-npfyo3o7qeg"	0.000017	
Query 4	*User id*	"Invalido"	0.000002	0 (Não existe)
		"1RCRKuHgP3FskGUVnmFdxg"	0.000028	4
		"J9m48CKaTlXvkqFMs7L6Kg"	0.000043	13
		"I4cfBoRWEhB90HBdjh8z5Q"	0.000408	67
Query 5	*Burnaby | Estrelas:*	1	0.000788	294
		2	0.000706	281
		3	0.000268	214
		4	0.000252	91
		5	0.000091	18
	Atlanta | Estrelas:	1	0.026348	2196
		2.33	0.023025	1901
		3.72	0.010126	1093
		4.02	0.004792	765
		5.01	0.000376	0
	Portland | Estrelas:	1	0.043555	3231
		2.03	0.037823	3056
		3.62	0.014483	2282
		4.587	0.006969	1013
		5	0.002990	593
Query 6	*Top N*	1	0.005353	406
		43	0.009533	6544
		123	0.010484	10737
		450	0.013256	15046
		1000	0.033542	18924
		1500	0.057975	22033
		4000	0.125105	28085
Query 7	*Apenas sgr*		0.2929391	19318
			0.304538	
			0.315358	
Query 8	*Coffee&Tea | Top:*	5	0.000269	5
		23	0.000463	23
		391	0.001511	391
		873	0.006369	873
		1500	0.017034	1404
	Restaurant | Top:	1	0.002383	1
		123	0.002242	123
		2785	0.062247	2785
		5000	0.156701	5000
		10000	0.430300	9342
	Food | Top:	10	0.001157	10
		300	0.001809	300
		1500	0.020301	1500
		5000	0.139931	5000
		10000	0.160469	5271
Query 9	*Palavra*	"Uminho"	3.246196	0
		'hello'	4.743560	3411
		"HeLlO"	4.741797	
		"8"	3.002862	25148
		"the"	1.261971	906502

Todos os resultados aqui apresentados, foram obtidos através de um portátil com as seguintes especificações:

OS	Arch Linux x86_64
Host	20TACTO1WW ThinkPad E14 Gen 2
CPU	11th Gen Intel i7-1165G7 (8) @ 4.700GHz
Memory	SoDIMM de 16 GB DDR4 3.200 MHz
First Solid State Drive	512GB SSD NVMe