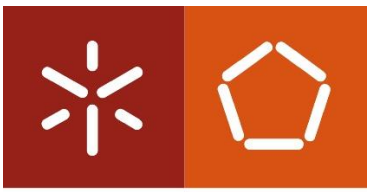


Universidade do Minho

Departamento de Informática

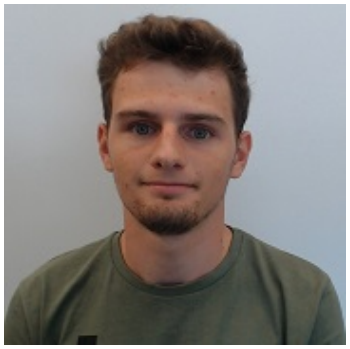
LI3 - Trabalho em Java

Sistema de Gestão de Recomendações

Grupo Nº 89

28 de junho de 2021

1 Autores



Bohdan Malanka
a93300

Diogo Rebelo
a93278

Henrique Alvelos
a93316

Conteúdo

1 Autores **2**

2 Introdução **3**
2.1 Decisões tomadas . . . 3
2.2 Testes de Desempenho . . . 3

3 Descrição da API **4**
3.1 Model . . . 4
3.1.1 IUser . . . 4
3.1.2 IBusiness . . . 4
3.1.3 IReview . . . 4
3.1.4 ICatalog<T> . . . 4
3.1.5 ITopBusiness . . . 4
3.1.6 IStates . . . 4
3.1.7 IStats . . . 5
3.1.8 IGestReviews . . . 5
3.2 View . . . 5
3.2.1 IView . . . 5
3.2.2 Table . . . 5
3.3 Controller . . . 5
3.3.1 IController . . . 5
3.4 Utilities . . . 5
3.5 Tests . . . 5

4 Complexidade das estruturas e Otimizações das Queries **5**
4.1 Query 1 . . . 5
4.2 Query 2 . . . 6
4.3 Query 3 . . . 6
4.4 Query 4 . . . 6
4.5 Query 5 . . . 6
4.6 Query 6 . . . 6
4.7 Query 7 . . . 6
4.8 Query 8 . . . 6
4.9 Query 9 . . . 6
4.10 Query 10 . . . 6

5 Apreciação crítica **7**

6 Conclusão **7**

2 Introdução

O presente trabalho tem como objetivo processar grandes volumes de dados de uma forma rápida e eficiente. De uma forma sucinta, o nosso programa socorre-se de três ficheiros de base essenciais (Ficheiros tipo .csv, que contêm os dados referentes aos utilizadores, aos negócios e às avaliações dadas pelos utilizadores aos negócios) e carrega-os para determinadas estruturas em memória, de modo a que a interação do Utilizador com o programa seja mais fácil. O desenvolvimento do projeto seguiu uma arquitetura de MVC (Modelo, Vista e Controlador), a qual surge descrita em detalhe numa secção posterior. Esta técnica permite uma melhor organização do código, o que se reflete numa sua melhor interpretação. Além disso, este projeto proporcionou um melhor conhecimento de uma linguagem recém aprendida (Java).

2.1 Decisões tomadas

Ao longo da realização do trabalho foram tomadas algumas decisões importantes e dignas de menção. Assim sendo, quanto à leitura, recorreu-se à utilização de um *BufferReader*, já que se mostra um método eficiente de acordo com a *Oracle*. Quanto à seleção de estruturas de dados, optou-se em seguir as que foram utilizadas no projeto em C, também de acordo com a arquitetura MVC.

2.2 Testes de Desempenho

Máquina	**Processador**	**Tipo de disco**	**Processo**	**Tempo (s)**	**Memória**
Acer Aspire (RAM : 8GB)	i5 8th Gen	SSD	Leitura de ficheiros	30.135	1816.494
			Q1	1.296	2.345
			Q2	1.187	0.002
			Q3	1.192	$5.493 * 10^{-4}$
			Q4	1.174	10.178
			Q5	1.172	0.076
			Q7	0.989	1.132
			Q8	6.889	330.792
			Q9	1.418	227.002
			Q10	0.786	0.007
MSI Prestige 15 (RAM : 16GB)	i7 10th Gen		Leitura de ficheiros	23.731	1913.655
			Q1	1.134	0.01399
			Q2	0.864	$1.526 * 10^{-4}$
			Q3	0.840	$6.104 * 10^{-4}$
			Q4	1.294	16.233
			Q5	0.840	0.068
			Q7	0.780	0.144
			Q8	2.163	495.227
			Q9	1.685	311.109
			Q10	0.910	0.0115
Lenovo IdeaPad (RAM : 12GB)	i5 8th Gen		Leitura de ficheiros	25.102	1881.629
			Q1	1.062	0.014
			Q2	0.904	$1.373 * 10^{-4}$
			Q3	0.915	0.002
			Q4	1.378	22.174
			Q5	0.955	0.067
			Q7	0.925	0.134
			Q8	2.255	382.272
			Q9	1.401	310.057
			Q10	0.849	0.006

Tempo médio de execução de queries									
Leitura	Query 1	Query 2	Query 3	Query 4	Query 5	Query 7	Query 8	Query 9	Query 10
26.323	1.164	0.985	0.982	1.282	0.989	0.928	3.769	1.501	0.848

3 Descrição da API

3.1 Model

3.1.1 IUser

Interface responsável por guardar a informação de um **USER**. Para além de ter o ID, Name e Friends (ainda sem o parcing) também contém um campo, responsável por guardar os *ID's de BUSINESS* aos quais o **USER** avaliou. Para isso, utilizamos um **List<>**.

3.1.2 IBusiness

Interface responsável por guardar a informação de um **BUSINESS**. Para além de ter o ID, Name, City, State e Categories (ainda sem o *parcing*) adicionamos dois campos: um **int** totalReviews, que guarda o número total de **REVIEW** deste *BUSINESS*, e um **float** totalStars para guardar o número total de estrelas que o *BUSINESS* recebeu. Assim, para calcular as estrelas médias do *BUSINESS* basta dividir totalStars por totalReviews.

3.1.3 IReview

Interface responsável por guardar a informação de um **REVIEW**. Não adicionamos atributos novos para além dos listados no enunciado do projeto em C.

3.1.4 ICatalog<T>

Interface genérica para os catálogos de User, Business e Review. Ou seja, as classes **UserCatalog**, **BusinessCatalog** e **ReviewCatalog** implementam esta interface pois, reparamos que são bastante similares entre si e por isso implementam os métodos comuns, já os métodos mais particulares para serem usados fazemos cast da respetiva classe. Fazendo assim o programa mais genéricas.

3.1.5 ITopBusiness

Interface responsável por representar um (**TopBusiness**) que resolvemos criar para a resolução da query 7. Assim, o único atributo desta classe é um Map em que cada key é uma String do nome de uma City e a value é um TreeSet de IBusiness, que são negócios que fazem parte da respetiva cidade, ordenadas por ordem decrescente de Stars.

```
//city name, b_id
private Map<String, TreeSet<IBusiness>> BusinessByCities;
```

Figura 1: Classe TopBusiness

3.1.6 IStates

Interface responsável por representar um (**States**) que resolvemos criar para a resolução da query 10. Assim, esta class tem dois atributos. O primeiro é composto por um Map em que a key é uma String do nome do State e a value é uma List<String> que corresponde á lista de cidades desse mesmo Estado. O segundo atributo é outro Map em que á key é uma String de State e a value é um Map com a key uma String de City e a value é uma lista de pares com Business ID e respetiva classificação média.

```
private final Map<String, List<String>> citiesOfState;
private final Map<String, Map<String, List<Pair<String,Float>>>> statesByCitiesByBusiness;
```

Figura 2: Classe States

3.1.7 IStats

Interface responsável pelas estatísticas do programa, exigidas no enunciado. Quanto aos anos e meses, decidimos utilizar três Maps's aninhados em vez de arrays tridimensionais. Assim, para cada ano temos um Map de meses, em que para cada mês tem Map de users id's, em que cada user id tem uma lista de reviews id's que fez.

```
//   ANO         MES       USER_ID   ARRAYLIST OF REVIEW_ID
private Map<Integer, Map<Integer, Map<String, ArrayList<String>>>> ageNmonth;
```

Figura 3: Classe Stats

3.1.8 IGestReviews

Módulo responsável pela base de dados do programa, sendo a Class agregadora do projeto. Faz a leitura dos ficheiros de input e guarda em *HashMap* os respetivos **USERS**, **BUSINESSES**, e **REVIEWS**, **TOPBUSINESS** e **STATES**. A API também contém a implementação de cada Query do programa.

3.2 View

3.2.1 IView

API responsável pela vista do programa, imprimindo as mensagens de erro e interface.

3.2.2 Table

Para o output das queries, decidimos mostrar o resultado numa JFrame que contém uma JTable.

3.3 Controller

3.3.1 IController

Módulo que tem o controlador do programa, ou seja, é responsável por comunicar com a View e a Model. Faz *parcing* do comando digitado e tenta perceber se é um comando válido ou não. Os comandos disponiveis são:

1. Queries;
2. Comandos como: help, save, load, quit;

3.4 Utilities

Modulo que contém algumas classes úteis como a classe Config que contém as variáveis constántes e a classe Pair genérico que guarda um par que qualquer tipo de classes.

3.5 Tests

Contén classes de teste e performace. A classe Crono é responsável pela contagem de tempo de execução dos comandos. A Classe Memory é responsável por calcular a memória gasta na execução das queries. E a classe Main é responsável por correr os testes

4 Complexidade das estruturas e Otimizações das Queries

4.1 Query 1

Para esta query, adicionou-se um atributo às *Stats* (**private TreeSet<String> BUSSnotReviewed**) que, como o nome indica, contém os ID's de negócios, por ordem alfabética. Cada ID é adicionado na função *fillStats*.

4.2 Query 2

Sendo que a classe *Stats* contém o atributo **private Map<Integer, Map<Integer, Map<String, ArrayList<String>>> ageNmonth**, que, como explicamos anteriormente, contém várias ID's de avaliações, de vários utilizadores, de vários meses e vários anos, basta um mês e um ano para calcular o tamanho das **Keys** e do tamanho de cada **Value** do **Map<String, ArrayList<String>**

4.3 Query 3

Voltando ao atributo usado para resolver a Query 2, basta buscar, para cada mês, de cada ano, os ID's de avaliações. Depois, para cada ID, procura-se na classe **Review**, a nota da avaliação. Depois de percorrer todos os anos, calcula-se a média para cada mês.

4.4 Query 4

Usando o mesmo atributo que foi usado para a Query 2 e 3, ao pesquisar cada ID de avaliação, verifica-se, nas avaliações, se corresponde ao dito ID do negócio dado. Se sim, adiciona-se o ID do utilizador a um HashSet, de modo a que não hajam repetidos. Além disso

4.5 Query 5

Para esta query o nosso pensamento foi tirando partido do atributo **businessReviewed** do **User** e ordenar por ordem decrescente de nº total de reviews. Porém, reparamos que acontece uma falha na leitura e essa lista é vazia. Portanto, a query 5 retorna sempre uma lista vazia.

4.6 Query 6

Esta query é a única que não tem nenhuma linha de código para a sua resolução, porque como fizemos uma má gestão de tempo decidimos dar prioridade ás outras queries.

4.7 Query 7

Aqui nos beneficiamos da Classe **TopBusiness** que contém por cada cidade a lista dos businesses que lhe pertence. Assim o output desta query é a lista **List<AbstractMap.SimpleEntry<String, List<String>>** em que a primeira entrada corresponde ao id do **User** e a segunda a respetiva lista dos 3 negócios com mais reviews feita. Durante a resolução desta query utilizamos um TreeSet para guardar os **IBusiness** aproveitando a ordem natural, visto que a Classe **Business** implementa Comparable<>.

4.8 Query 8

Para esta query queríamos uma **List<AbstractMap.SimpleEntry<String, Integer>** que a medida que percorremos **UserCatalog** vamos adicionando o id do utilizador e o respetivo nº total de reviews que fez. Depois, ordenamos essa lista e retiramos os top X users. No entanto, inicialmente fizemos o input desta query fixa, mas depois esquecemos de alterar isso e para executar basta escrever "*query8*", devolvendo o top 5 de users.

4.9 Query 9

O algoritmo para esta query ficou um pouco pesado, visto que nós percorremos o **UserCatalog** e para cada **User** percorremos os **Review** para encontrar todas as avaliações que ele fez e incrementar as estrelas que atribuiu. Assim, decidimos armazenar o output numa **List<AbstractMap .SimpleEntry<String, Float>**. Uma possível otimização é adicionar mais um atributo num **User** responsável por guardar o nº total de estrelas que ele atribuiu, que seria incrementado na leitura dos reviews. Infelizmente, a query retorna uma lista vazia e não conseguimos solucionar o problema.

4.10 Query 10

Por fim, na query 10, o nosso pensamento foi criar uma Classe **STATE** que contivesse dois Map com todos os States e respectivas listas de cidades que seria carregada na leitura, que posteriormente serviria para calcular o segundo atributo dessa Classe que é o output desta query. Infelizmente, embora a query dê um resultado, achamos que está errada.

5 Apreciação crítica

Em geral não estamos contentes com o resultados. Isto porque, devido à má gestão de tempo e imensa carga de trabalhos, não conseguimos gerir o trabalho. Nem todas as queries estão concluídas, nem todos os métodos estão documentados. Porém, podemos dizer que, mesmo com a má gestão, fizemos de tudo para que o trabalho estivesse minimamente prestável.

6 Conclusão

Em suma, aprendemos bastante com este projeto, pois fortalecemos o nosso conhecimento de Java. Além disso, também ficamos familiarizados com gestão e processamento de grande quantidade de dados que se aproxima muito da realidade atual. Para finalizar, se refizéssemos de novo ou se houvesse mais tempo, o trabalho estaria bastante melhor e a nota dada seria completamente diferente.