# Projeto de Laboratórios de Informática 3
# 2ª Fase
# Grupo 34

Alexandre de Freitas Ferreira Pacheco A80760
Diogo José Cruz Sobral A82523
José Pedro Milhazes Carvalho Pinto A80741

12 de Junho de 2018

A80760

A82523

A80741

**Resumo**

Como já foi visto, nesta unidade curricular foi-nos proposta a implementação de um sistema de resposta a *queries* sobre um *dump* da base dados do site *Stack Overflow*.

Era pretendido que esta segunda fase do projeto fosse desenvolvida em *Java*. Pelo que o esperado era uma maior facilidade na implementação das soluções para qualquer problema que surgisse e em atingir objetivos como o o encapsulamento das estruturas de dados e a abstração de código.

Ao longo desta segunda fase do projeto não pudemos deixar de notar a grande diferença, em termos de ferramentas disponíveis e "comodidade", entre trabalhar em *Java* e em *C*.

# 1 Ferramentas Utilizadas

À senelhança do que aconteceu na primeira fase do projeto, servimo-nos de uma biblioteca para efeturar o parsing dos ficheiros *xml*, que foi o *Java SAXParser*.

Por outro lado, para organizarmos as nossas estruturas de dados, desta vez nem sequer considerámos implementar versões nossas das estruturas, visto que o java já apresenta uma API riquíssima em ferramentas para este propósito.

# 2 Tipos e Estruturas de Dados

De modo a conseguir responder às *queries* propostas de forma eficiente, é natural que o mais importante é a forma como organizamos o grande volume de informação presente nos *dumps* disponibilizados.

Primeiro, analisámos a informação contida em cada um dos ficheiros *xml* disponibilizados por comunidade (e.g. *android*, *ubuntu*...) e o que era pedido nas *queries*, a fim de perceber quais os dados que seriam recorrentemente necessários.

## 2.1 Tipos de Dados: *Users* e *Posts*

As entidades elementares na forma de descrever cada comunidade, à volta das quais gira a grande maioria da informação, são os *users* e os *posts*.

Naturalmente, e semelhantemente ao que fizemos na 1ª fase, para cada um destes objetos criámos uma classe, `MyUser` e `MyPost`.

Para representar as *tags* bastou implementar uma correspondência entre os seus nomes e `Ids`, como veremos à frente.

## 2.2 Organização dos Objetos de Posts e Users

Para organizar a informação utilizámos sobretudo a interface *Map* do *Java*.

Organizámos os *users* num `LinkedHashMap` em que a chave são os `Ids`.

Organizámos os *posts*, tal como na primeira fase em duas estruturas separadas, para os poder procurar quer por data quer por `Id`. Assim, ficaram num `LinkedHashMap` semelhante ao dos *users* e noutro `LinkedHashMap` com as `LocalDate` como chave e uma lista de *posts* por cada data.

Esta lista de *posts* foi implementada numa classe que definimos fazendo uso da interface `List` e um contador de perguntas e respostas que é preenchido aquando do *loading*.

A informação das *tags* consiste num `LinkedHashMap` cujas chaves são nomes das tags e os valores são os `Ids` das mesmas.

## 2.3 Outras Estruturas Auxiliares

Para além da organização essencial referida, decicidmos implementar, ainda, estruturas auxiliares, que são úteis sobretudo quando a resposta a determinadas *queries* envolve calcular um conjunto de perguntas mais respondidas ou um conjunto de utilizadores com maior reputação.

Estas estruturas consistem em dias listas, mais propriamente, `ArrayList`. Um deles contém os `Ids` de todos os *users* ordenados pelo número de posts que o respetivo `user` efetuou. O outro segue a mesma lógica, mas o critério de ordenação é a reputação dos *users*.

# 3 Modularização e Abstração de Dados

Como temos aprendido ao longo dos últimos tempos, são boas práticas as de manter um código modular e abstraído do tipo de dados com que trabalhamos.

### Encapsulamento

Em *Java* é bastante fácil de encapsular os dados, uma vez que a própria linguagem já tem mecanismos que facilitam esta prática, como a simples declaração de variáveis de instância como `private`. A fim de impossiblitar a propagação de apontadores da estrutura interna da nossa comunidade, todos os *sets* e *gets* das classes que criámos trabalham com clones. Os sets clonam a informação que recebem, e os gets clonam a informação interna para devolver o clone.

A utilização das APIs do *Java* tornou o código bastante abstrato, visto que a qualquer momento podemos mudar, por exemplo, o nosso programa para implementar os mapas num `TreeMap` em vez de `LinkedHashMap` e pouco ou nada termos de mudar. Este comportamento verifica-se na maior parte do código.

# 4 *Queries*

Nesta secção passamos a explicar a abordagem que tomámos em relação a cada uma das *queries*, note-se que muitas delas funcionam como na primeira fase.

### *Query* 1 - Título e autor da pergunta

O procedimento que tomamos é bastante simples.

1. Procuramos o post com o `Id` fornecido e verificamos se se trata de uma pergunta.
   (a) Se for uma pergunta, devolvemos o seu título e o `DisplayName` do autor (procurado o `OwnerUserId` na árvore de users).
   (b) Se for uma resposta, repetimos o ponto **1** com o `ParentId` do post, que será o `Id` da pergunta correspondente.

### *Query* 2 - Top N *users* com mais posts

Esta query toma partido da estrutura auziliar em que, aquando do *load*, ordenamos os users pelo seu número de *posts*. Copia-se desta estrutura os N primeiros utilizadores (ou todos, caso N seja maior que o tamanho desta coleção).

### *Query* 3 - Número de perguntas e respostas ao longo de um período

O `LinkedHashMap` que organiza os posts por data tem como chave a `LocalDate` referente a cada dia, e como valor um objeto que contém uma lista de `Ids` de *posts*, e dois contadores, que indicam quantas perguntas e respostas ouve nesse dia. Tomando partido desta organização:

1. Percorre-se um ciclo em que se acrescenta a cada iteração um dia à `LocalDate` que corresponde ao início do período.
2. Em cada entrada do *map*, soma-se ao total de perguntas o número de perguntas na lista de *posts* nessa entrada, e faz-se o mesmo para as repsostas.
3. No fim do ciclo, devolve-se as variáveis que acumularam o somatório ao longo do ciclo.

### *Query* 4 - Perguntas com determinada *tag* feitas num período

1. Efetua-se um ciclo semelhante ao descrito na *query* 3 (no que toca ao modo de iteração).
2. Em cada iteração, obtém-se e percorre-se a coleção de *posts* correspondentes a cada data.
3. Se um *post* tiver a *tag* a verificar o `Id` do mesmo é adicionado a uma lista resultado, a retornar no final.

### *Query* 5 - Informação de um *user* e os seus últimos *posts*

Esta *query* consiste em simplesmente procurar um utilizador na nossa estrutura e devolver a sua informação.

1. É procurado o *user* em questão.
2. É percorrida a sua lista de *posts*, e cada um deles é adicionado a um `TreeSet` ordenado pela data de criação dos posts.
3. Armazena-se os `Ids` dos 10 primeiros (ou menos, se forem menos de 10) *posts* do *set* referido numa lista-resultado.
4. Retorna-se a biografia e a lista resultado construída.

### *Query* 6 - N respostas mais votadas ao longo de um período

A diferença entre os *upvotes* e *downvotes* de uma resposta equivale ao seu parâmetro `Score`, pelo que este já se encontra calculado a partir do momento em que o carregámos do ficheiro *xml*.

1. É efetuado um ciclo semelhante aos referidos nas *queries* 3 e 4.
2. Para cada data, percorre-se os *posts* correspondentes e adiciona-se os que forem perguntas a uma lista auxiliar.
3. Esta lista de *posts* é ordenada segundo os *scores* (recorrendo a um `Comparator`), e devolve-se os `Ids` dos primeiros N (ou menos, caso não hava N) *posts*.

### *Query* 7 - N perguntas com mais respostas ao longo de um período

A nossa *query* 7 toma partido do atributo `AnswerCount` dos posts, pelo que a solução é bastante semelhante à da *query* 6. As únicas diferenças são que os `Ids` devolvidos são de perguntas e não de respostas, e que a ordenação feita na lista auxiliar recorre a um `Comparator` que tem em conta o número de respostas em vez do *score*.

### *Query* 8 - N perguntas mais recentes com determinada *tag*

Na *query 8*, o critério que utilizámos para averiguar se um título contém uma dada palavra foi o método `String contains(String str)` da API do *Java*.

1. São percorridos todos os posts do *map* que os organiza.
2. Se um post contiver no título a palavra a procurar, este é adicionado a um `TreeSet` ordenado segundo um `Comparator` que ordena cronologicamente os posts.
3. Retorna-se numa lista resultado os `Ids` dos N primeiros (ou menos de N, caso não existam) posts do *set* referido.

### *Query* 9 - N perguntas mais recentes em que dois *users* participaram

Esta é outra query em que se tira bastante partido do facto de termos em cada objeto que representa um *user* uma lista com os `Ids` dos seus posts.

1. Cria-se um `HashMap<MyPost, Integer>` auxiliar, onde se insere como chave as perguntas correspondentes a cada *post*.
2. Percorre-se os *posts* de um *user* e insere-se a pergunta correspondente no `map` auxiliar, com o valor de 0.
3. Repete-se o processo para o outro *user*, mas se uma dada pergunta já estiver no mapa, insere-se com o valor 1.
4. Para cada entrada do mapa auxiliar, se o seu valor for 1, insere-se o `post` num `TreeSet` ordenado cronologicamente recorrendo a um `Comparator`.
5. Retornam-se os `Ids` dos N primeiros *posts* deste `TreeSet`.

### *Query* 10 - Melhor resposta

Esta *query* é relativamente simples, devido ao facto de no `load` registarmos em cada *post* post os `Ids` das suas respostas (caso seja uma pergunta).

1. É percorrida a lista de respostas do post dado, e para cada uma calcula-se uma pontuação mediante a fórmula dada no enunciado.
2. Retorna-se o `Id` da resposta para a qual tenha sido observada uma maior pontuação.

### *Query* 11 - N *tags* mais usadas ao longo de um período pelos N *users* com maior reputação

Esta *query* faz uso de grande parte das estruturas que temos montadas.

Nesta query, é criada um `HashMap` com as ocorrências de cada *tag*, que depois são ordenadas num `TreeSet` para poderem ser devolvidas. É ainda importante mencionar que, na nossa interpretação do que foi pedido, utilizámos os N utilizadores com maior reputação de sempre (mesmo que não tenham postado nesse período de tempo). No resultado da query, `Ids` de tags com o mesmo número de ocorrências estão em ordem crescente.

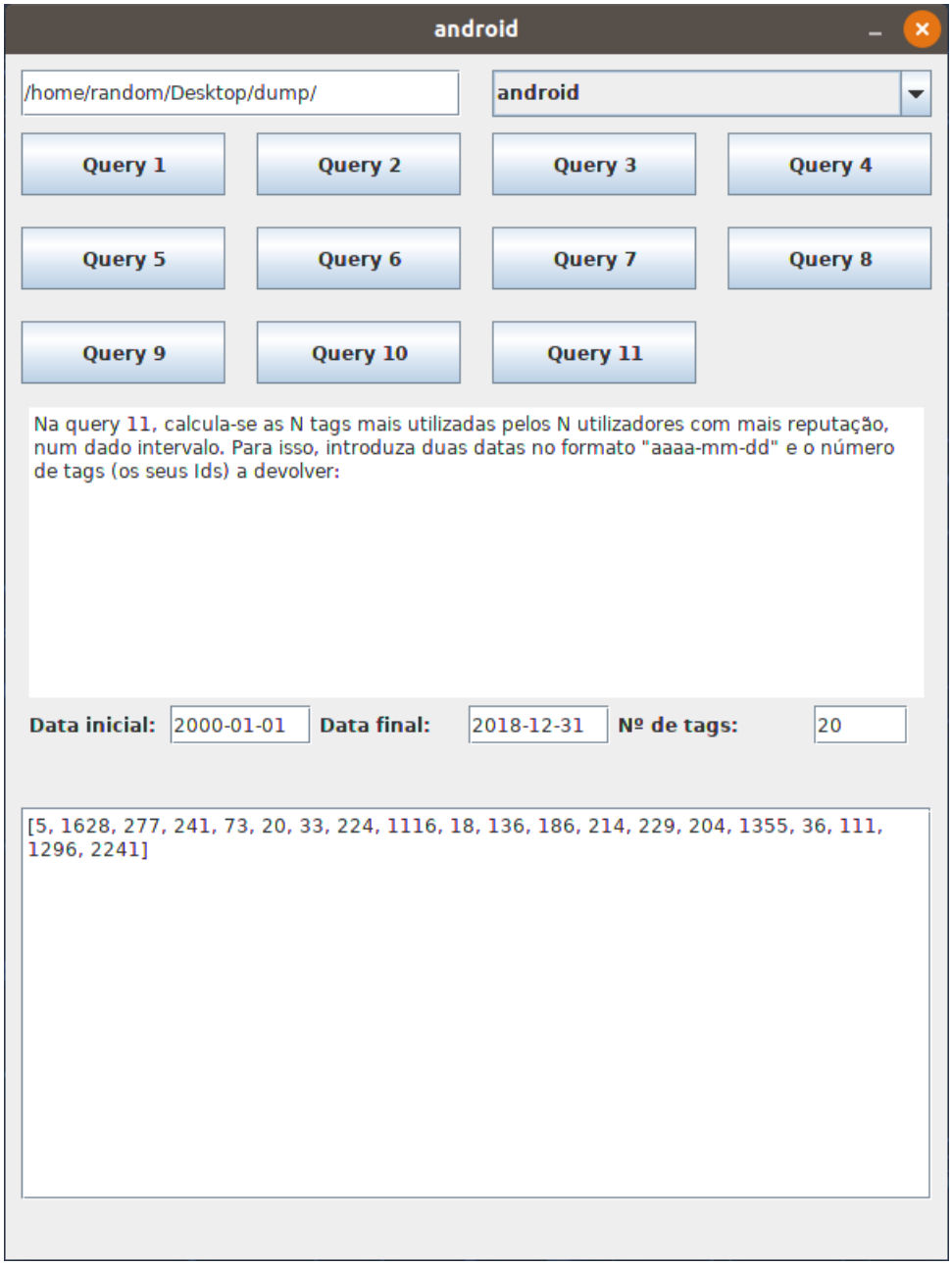
1. É obtido um `ArrayList` com (no máximo) N `Ids` de *users*, ordenados pela sua reputação através da estrutura auxiliar já mencionada, criada para o efeito.
2. A lista obtida é percorrida, os respetivos `users` procurados no devido `LinkedHashMap` e para cada um:
   (a) Obtêm-se os *posts* que são perguntas e foram efetuados no período de tempo especificado.
   (b) As *tags* destes posts são registadas, ou o seu número de ocorrências incrementao, num `HashMap` criado para este efeito.
3. Preenchida a tabela, todas as suas entradas são inseridas num `TreeSet` onde ficam ordenadas pelo número de ocorrências.
4. Retornam-se os `Ids` nas chaves das (no máximo) N primeiras entradas do referido `TreeSet`.

# 5 Interface Gráfica e modelo MVC

Tomámos também a liberdade de implementar uma interface gráfica (GUI) recorrendo à ferramenta *Swing* e ao modelo *Model View Controller*, *MVC*.

Esta interface consiste numa possível tradução e síntese visual daquilo em que consiste este projeto: a escolha e *loading* de uma *dump* de dados, a introdução de *inputs* para uma determinada *query*, e a resposta por parte do programa.

A implementação desta funcionalidade foi simples, mais uma vez devido à existência de imensas ferramentas de suporte ao *Java* e também devido ao facto de a parte obrigatória do projeto estar pronta para ser utilizada como *Model* do MVC simplesmente recorrendo à interface `TADCommunity`.




Exemplo de estado da interface gráfica

# 6 Estratégias para melhorar a Eficiência

Tendo em conta o grande volume de dados que nos propusemos a processar neste projeto, surge a necessidade de adotarmos estratégias que melhorem a eficiência das operações que levamos a cabo.

Uma decisão que tomámos de modo a melhorar a eficiência foi qual implementação da interface `Map` utilizar. Utilizámos `LinkedHashMap` porque, uma vez que é implementada utilizando listas ligadas, nunca tem espaços vazios que precisem de ser atravessados numa travessia (efetuada em qualquer procura por um valor, se recordarmos o funcionamento de uma tabela de *hash*).

A única pequena desvantagem dos `LinkedHashMap` em relação aos `HashMap` é na criação e inserção de valores, no entanto, foi um compromisso que estivemos dispostos a fazer, dado que estas estruturas são criadas durante o *load*.

Outra forma de tornar o nosso programa mais eficiente foi, durante o `load`, construir estruturas auxiliares que, embora armazenassem informação que pudesse ser obtida a partir das árvores de *posts* e *users*, e levassem a um ligeiramente maior gasto de memória, eram úteis a muitas *queries* e poupavam imensos cálculos e travessias.

Uma prática que procurámos ter foi a utilização de arrays sempre que possível. Um bom exemplo da nossa "aproximação"aos *arrays* é o facto de utilizarmos `ArrayList` nas estruturas que nos devolvem os N `users` com mais reputação ou `posts` efetuados.

# 7 Conclusão

Tal como na primeira fase pudemos concluir, há sempre um compromisso performance vs. segurança, a organização da nossas estruturas toma um papel central nos fatores que influenciam a eficiência das *queries*.

Verificámos o esperado, que era ser muito mais simples implementar esta solução em *Java* do que em C, dado o leque de ferramentas (a maior parte nativa da linguagem) com que pudemos contar para resolver qualquer problema.

Resta refletir sobre as duas fases do projeto e afirmar que muito dificilmente, se nos propusessem novamente uma tarefa semelhante, escolheríamos C para implementar a solução. No fundo, apenas uma ínfima parte dos projetos requerem virtudes que apenas linguagens como o C podem oferecer. Num mundo cada vez mais orientado aos objetos e exigente por por produtividade, é natural que vigore a utilização de linguagens com um maior nível de abstração.