

Assignmnet 2

André Pedrosa [85098], João Abílio [84732]

Recuperação de informação

Departamento de Eletrónica, Telecomunicações e Informática

Universidade de Aveiro

13 de outubro de 2019

1 Introdução

Este relatório apresenta uma explicação do trabalho desenvolvido para o segundo assignment da disciplina "Recuperação de Informação", explicando as decisões tomadas e o funcionamento da solução.

Esta segunda entrega tem como objetivo fazer incrementos à entrega anterior de maneira a aplicar o método SPIMI durante o método de indexação, aplicar pesos *td-idf* aos termos usando o esquema de indexação *Inc.ltc* e adicionar ao indexar as posições dos termos nos documentos nos quais ele aparece.

No fim serão apresentadas as medidas de eficiência pedidas no enunciado do assignment para cada novo método de indexação.

Devido ao elevado número de classes criadas, o diagrama de classes vai ser dividido em vários que vão sendo apresentados ao longo do relatório. Estes diagramas foram gerados através do IDEA IntelliJ, consequentemente em anexo é disponibilizada a legenda da convenção usada.

2 Data Flow

O data flow da nossa solução, de modo geral não sofreu grandes alterações, apenas migramos o código de execução da pipeline de indexação para uma classe separada, em vez de ser definida na classe Main.

Na figura 1 está representado a sequência de execução da nossa solução, onde as setas azuis significam que a classe origem executa um método da classe destino e as setas vermelhas significam que a classe origem cria a classe destino. Deixando de parte as classes que têm uma seta vermelha a apontar para si, todas as classes são instanciadas na classe Main. Na figura existem

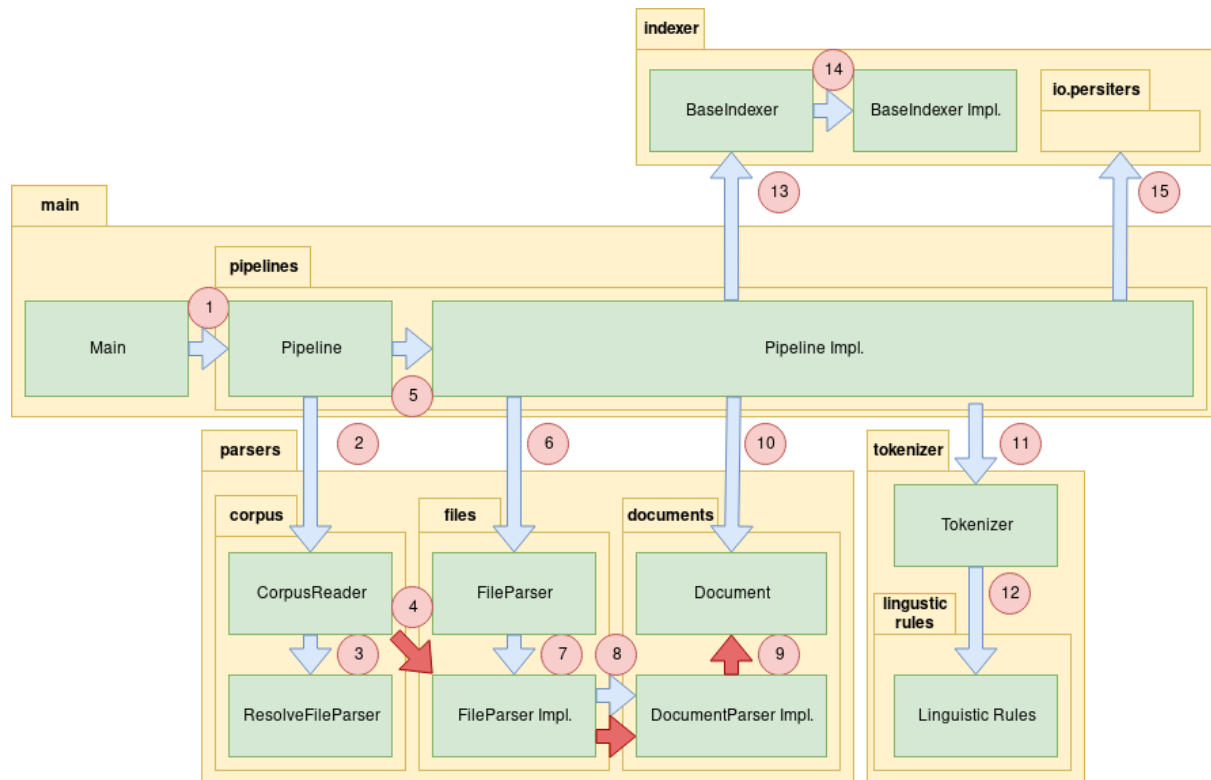


Figure 1: Diagrama de sequência da solução

várias classes em que é apresentado a classe base e a sua implementação, isto pois algumas classes base apresentam métodos *final* que chamam depois métodos abstratos que devem estar definidos em classes de descendentes (padrão Template Method).

3 Packages

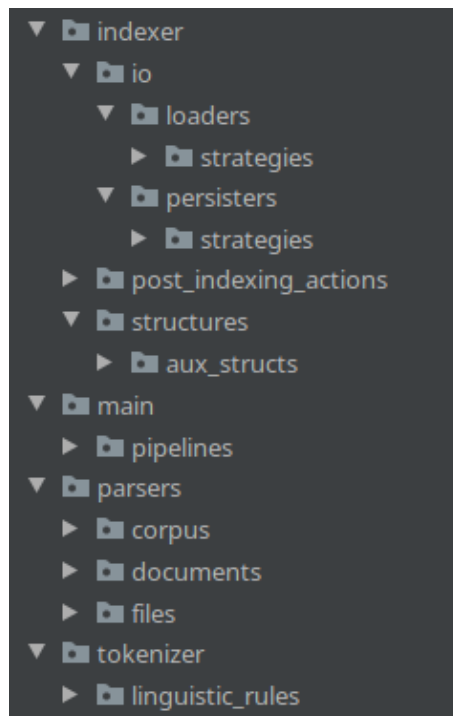


Figure 2: Árvore de packages da solução

Nesta secção vão ser apresentadas as alterações feitas a cada package relativamente à entrega anterior.

3.1 main

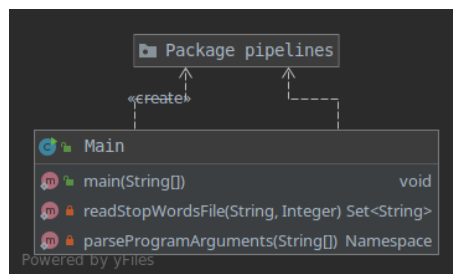


Figure 3: Diagrama de classes do package *main*

Como foi dito na secção anterior, o código de execução da pipeline de indexação que estava presente na classe *Main*, foi migrado para para uma classe do tipo *Pipeline*. Ou seja, a classe *Main* apenas tem a responsabilidade de instanciar as classes necessárias para a execução da pipeline.

3.1.1 pipelines

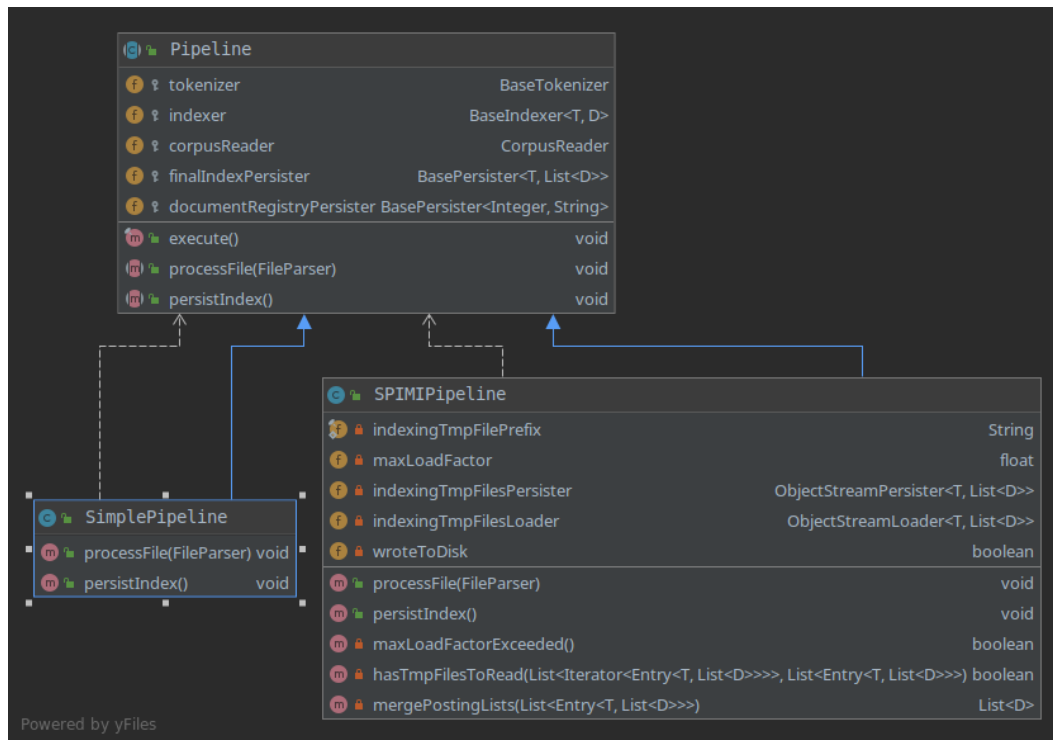


Figure 4: Diagrama de classes do package *main.pipelines*

Este package tem as classes que definem a sequência de execução do método de indexação. A classe principal (Pipeline) está encarregue de iterar sobre o corpus e passar cada FileParser à implementação de Pipeline escolhida (método processFile) e depois disso executar o método para persistir o indexer.

A classe SimplePipeline representa a pipeline que foi definida no assignment anterior, em que não tem considerações relativamente à memória no processo de indexação. A SPIMIPipeline apresenta uma implementação em que o método de indexação é feito segundo o algoritmo SPIMI, em que durante a indexação, sempre que a memória ocupada chegar a um limite (80% da memória máxima), o index atual é escrito para disco, ordenado pelos termos. De maneira a criar um index final, esta pipeline faz um merge dos vários ficheiros criados na fase anterior. Este merge é feito lendo blocos (implementado com a classes que usam métodos de buffering enquanto lêem, como por exemplo BufferedInputStream e BufferedReader) de todos os ficheiros temporários criados. Para fazer merge destes blocos, é mantida uma lista à qual é adicionada iterativamente a entry (termo e posting list) em que o termo é menor (alfabeticamente) de entre as várias entries lidas dos vários ficheiros temporários a fazer merge. Caso haja o mesmo termo em diferentes blocos é feito o merge das posting lists segundo os document ids.

Os ficheiros temporários foram escritos em binário (usando a classe ObjectOutputStream),

já que se revelou ser muito mais rápido do que aplicar a forma de persistência que o utilizador escolhe no Main, no caso deste assignment CSV. Para guardar o index final foram criados vários ficheiros, tendo cada um um número máximo de entries e para saber quais os termos presentes em cada ficheiros deste foi inserido como sufixo ao nome dos ficheiros o primeiro termo que guardam.

A classe indexer, para além de guardar o index invertido, guarda também o document registry que contém a associação entre um document id, gerado por nós cada vez que um documento é criado, e o identificador presente nos documentos, no caso deste corpus o campo PMID. Como a geração deste id é incremental e não existem vários documentos com o mesmo document id, na altura da indexação, quando a memória atingir o limite máximo, podemos já escrever esta estrutura num ficheiro final. A maneira como esta estrutura será persistida vai ser explicada mais à frente no package indexer.persisters.

3.2 parsers

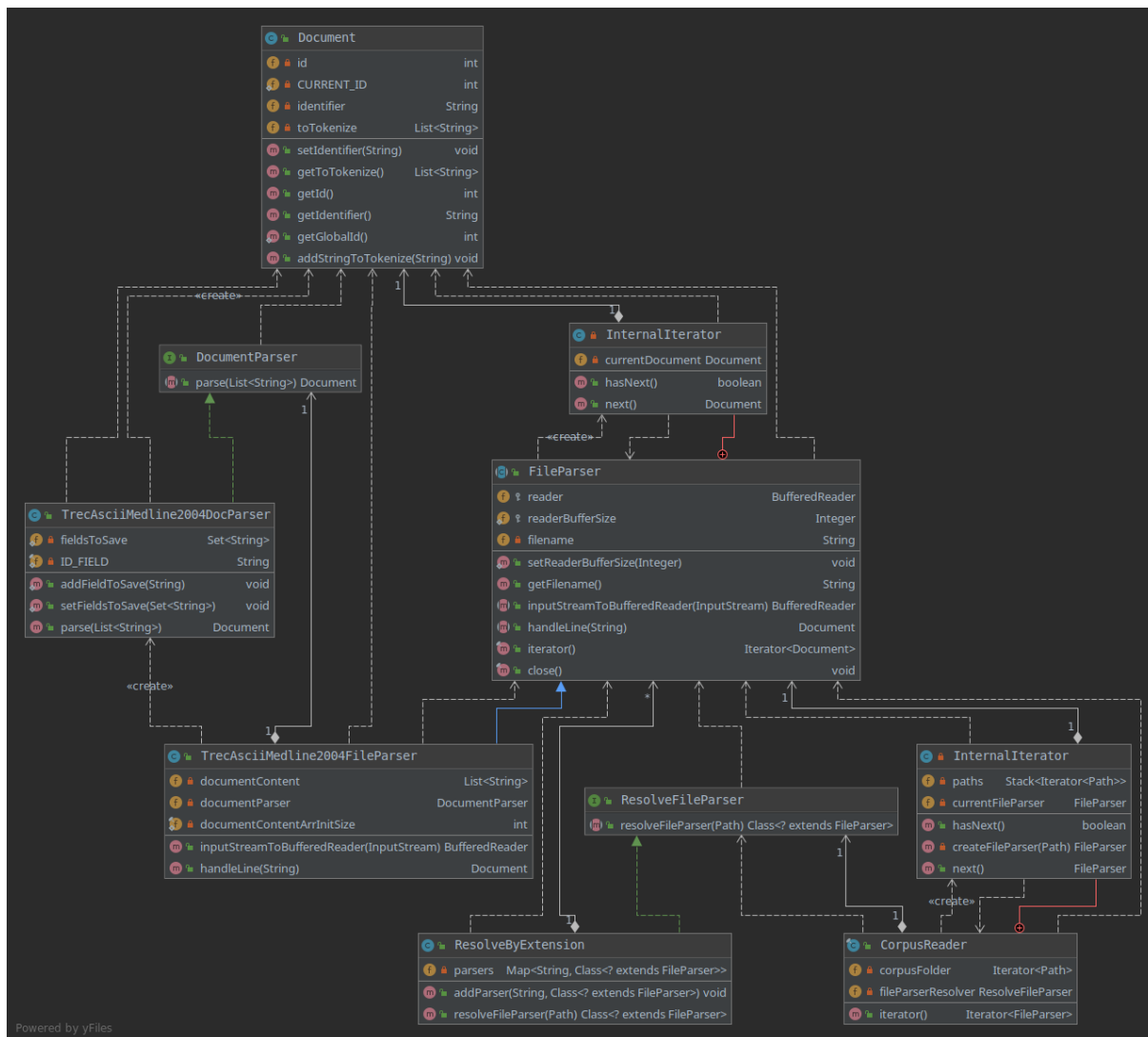


Figure 5: Diagrama de classes do package *parsers*

Este package não sofreu grandes alterações relativamente à entrega anterior, apenas foi alterado algumas mensagens de erro, os id dos documentos agora começa em 0 e algumas operações sobre strings, que não alteram o resultado final, obtidas dos ficheiros do corpus foram removidas, como `.trim()`, para tentar reduzir o número de instanciações da classe `String`.

3.3 tokenizer

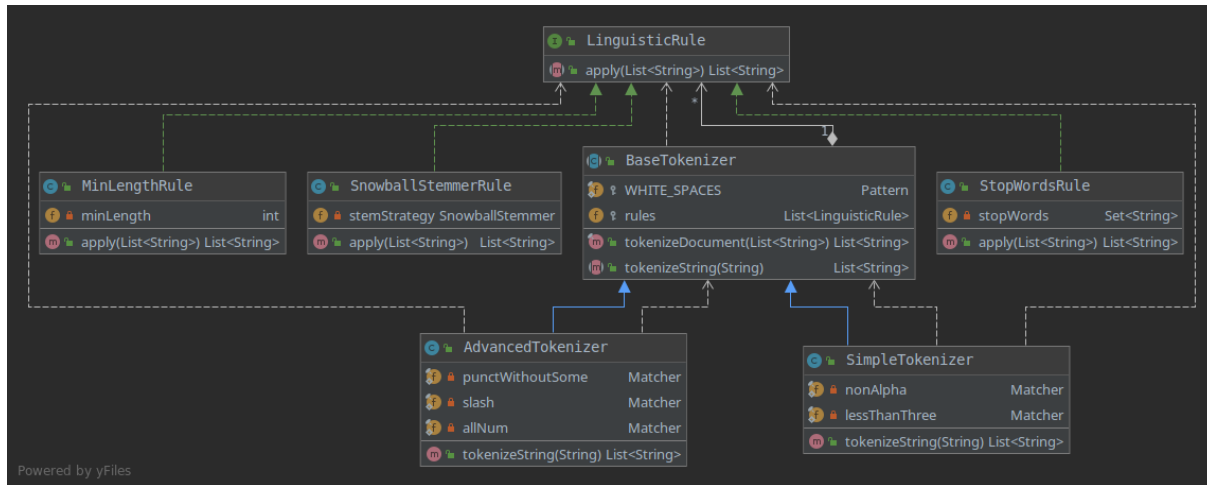


Figure 6: Diagrama de classes do package *tokenizer*

Este package também não sofreu grandes alterações relativamente à entrega anterior, apenas foi criada uma nova Linguística Rule que remove os termos que têm menos de um certo número de caracteres.

3.4 indexer

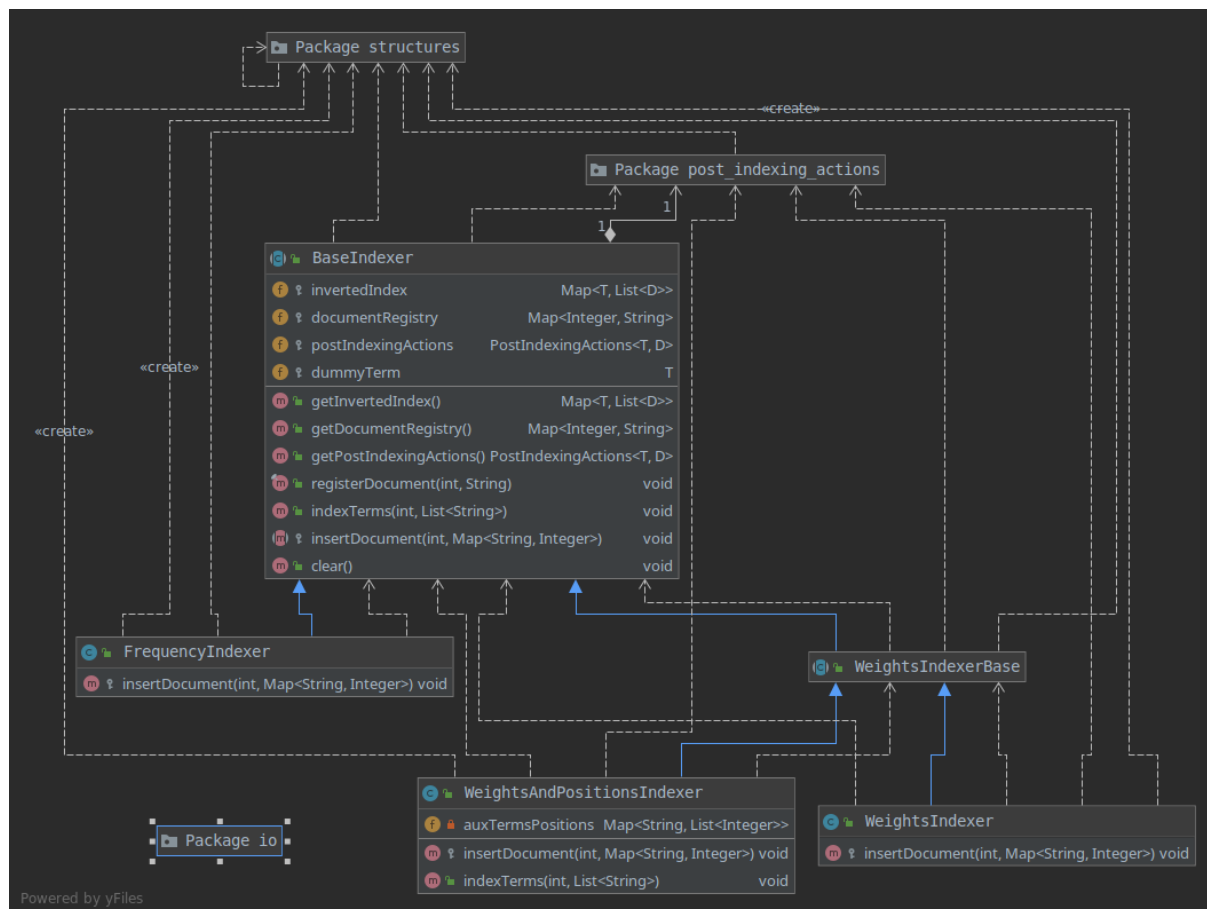


Figure 7: Diagrama de classes do package *indexer*

Este package foi o que sofreu mais alterações relativamente à entrega anterior. Foram criados novas classes indexer (árvore de classes *WeightsIndexerBase*) que guardam pesos tanto para o termos como para os documentos presentes nas posting lists. Foi criado também um novo package, *post_indexing_actions*, que contém classes que aplicam sobre cada entry do indexer ações depois do processo de indexação e antes de serem persistidos de modo final, ou seja, no caso da pipeline que aplica o algoritmo SPIMI, estas ações apenas serão aplicadas quando uma entry estiver pronta para ser escrita para o ficheiro do index final, por outras palavras, depois das posting lists dos termos iguais presentes nos vários ficheiros temporários serem combinadas. Cada indexer tem uma classe *post_indexing_action* específica associada. Por fim, foi feito um refactor relativamente à persistência das estruturas internas do indexer.

3.4.1 indexer.structures

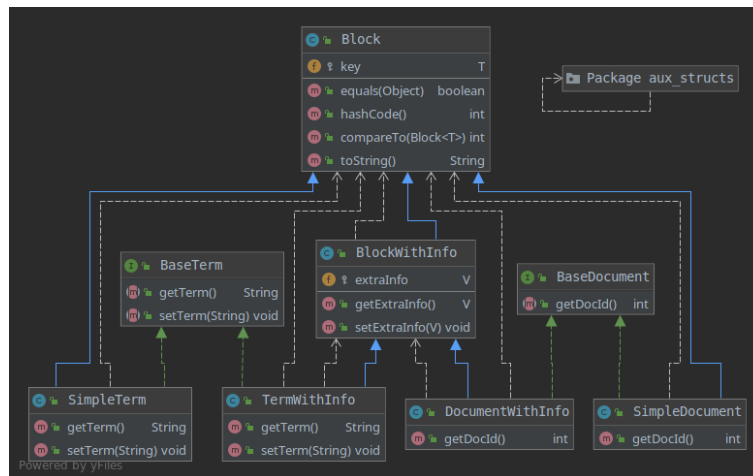


Figure 8: Diagrama de classes do package *indexer.structures*

Deste subpackage foram removidos as classes específicas de termos ou documentos, por exemplo um documento com frequência é criado com uma classe `BlockWithInfo` que implementa a interface `BaseDocument` em que o tipo da `extraInfo` da classe `BlockWithInfo` é `Integer`. Para criar um documento com `weight` (float) teria de criar uma outra classe semelhante à anterior mas o tipo da `extraInfo` seria do tipo `Float`. Para evitar isso criou-se as quatro combinações das classes `Block` (`Block` e `BlockWithInfo`) com as classes `BaseTerm` e `BaseDocument`.

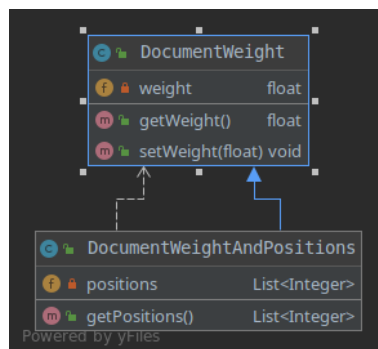


Figure 9: Diagrama de classes do package *indexer.structures.aux_structs*

O package `aux_structs` contém classes a serem usadas no campo `extraInfo` das classes descendentes da classe `BlockWithInfo`. As duas classes representadas são usadas como `extraInfo` dos documentos dos indexers que calculam `weights` para termos e documentos.

3.4.2 indexer.io.persisters

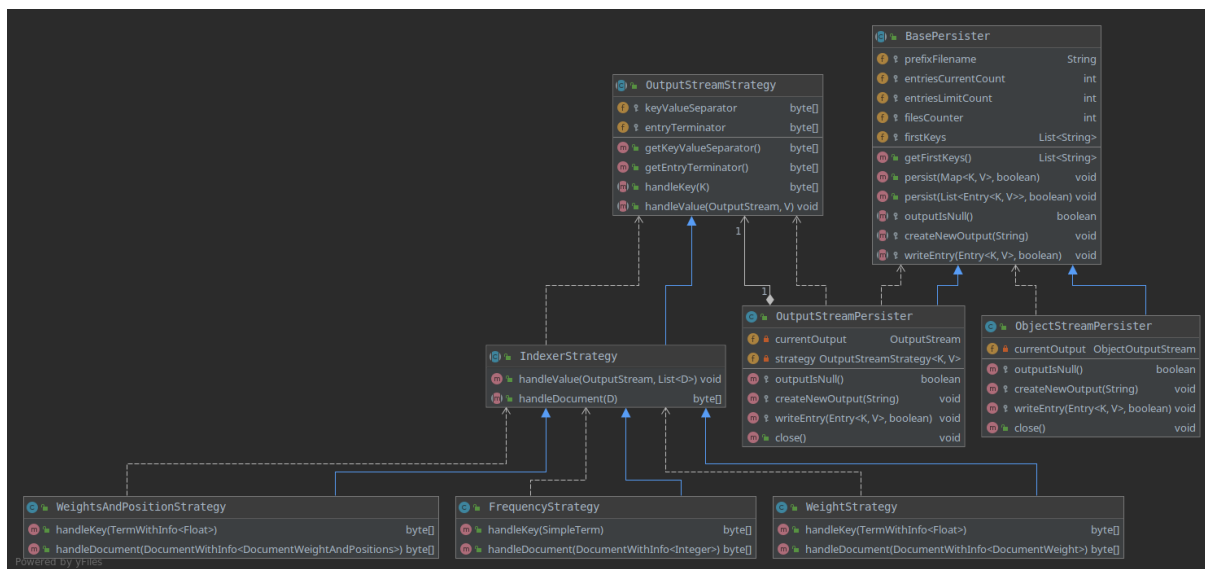


Figure 10: Diagrama de classes do package *indexer.io.persisters*

Como as estruturas internas do indexer são mapas, desenvolvemos uma nova versão dos persisters em que as classes bases (*BasePersister*) iteram sobre as as entries (key:value) ordenadas pela key, e as classes descendentes devem implementar o modo como cada entry é escrita para disco. Nos implementamos dois modos de persistência, um usando uma *ObjectOutputStream* e outro usando uma *OutputStream*. A primeira escreve as entries como objectos *Map.Entry* e o segundo escreve a representação da key em bytes, um separador, a representação do value em bytes e um terminador. Este ultimo tipo de persister implica defenir uma *OutputStreamStrategy* que define a estratégia de transformar as keys e values em bytes. No caso da estrutura inverted index do indexer estas estratégias (árvore de classes *IndexerStrategy*) transformam um termo em bytes e cada documento das posting lists também em bytes (Criam uma *String* com um determinado formato e devolvem os bytes dessas *Strings*).

Como os ficheiros escritos são sequenciais, ou seja, para aceder a uma parte algures do ficheiro é necessário ler todo o ficheiro até à posição específica. Para tentar reduzir a leitura a ficheiros a disco, cada estrutura interna do indexer vai ser escrito para vários ficheiros em que cada ficheiro tem um numero máximo de entries. Os nomes destes vários ficheiros é a concatenação do prefixo definido pelo utilizador mais um contador e a representação em string da primeira key guardada nesse ficheiro. O contador é usado pois no caso da indexação inicial do algoritmo SPIMI, vários ficheiros podem vir a ter o mesmo termo como primeiro termo.

3.4.3 indexer.io.loaders

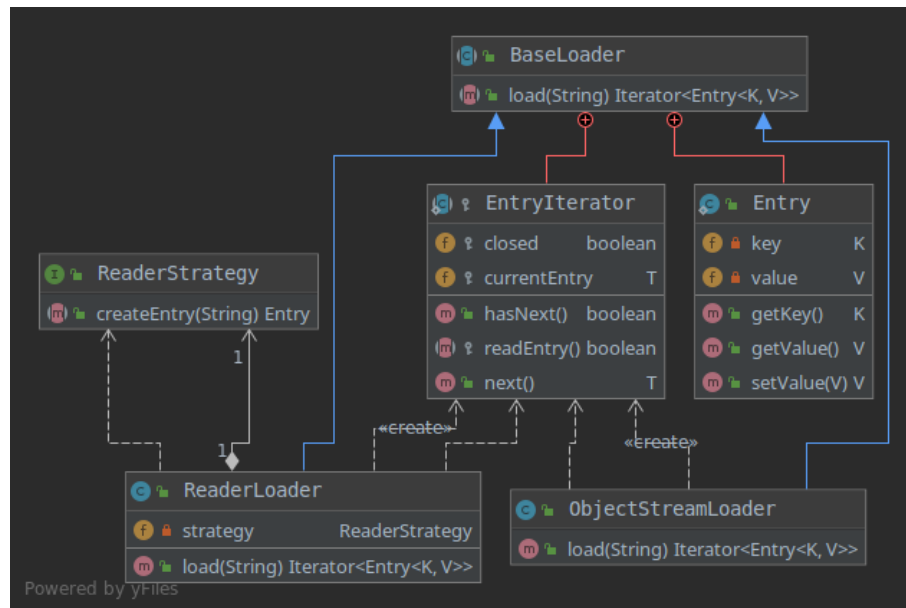


Figure 11: Diagrama de classes do package *indexer.io.loaders*

Como os persisters escrevem entries uma a uma, para criar os loaders usamos iteradores para ler dos ficheiros. A classe base, BaseLoader, define a estrutura básica do iterador o qual as classes descendentes devem completar pois a leitura de entries depende do método de escrita e da estratégia usada. Criamos duas implementações de loader, ObjectStreamLoader que lê objetos entry e o ReaderLoader que lê linha a linha. Este último tem depois uma strategy que transforma um linha num objeto Map.Entry. Como neste assignment ainda não é necessário fazer load do index final, não foram implementados strategies especificas para cada indexer.

3.4.4 indexer.post_indexing_actions

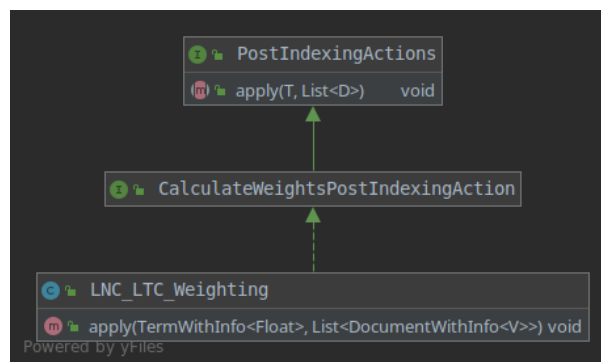


Figure 12: Diagrama de classes do package *indexer.post_indexing_actions*

Neste package encontram-se definidas ações para serem executadas sobre as várias entries da estrutura inverted index dos indexers, depois do processo de indexação e antes de a versão final ser persistida para disco. O objetivo principal é realizar cálculos que apenas podem ser feitos no fim de ter passado por todos os documentos do corpus, os quais são o cálculo do idf de cada termo (depende do número total de documentos) e a normalização dos pesos dos documentos para cada posting list (apenas pode ser calculado quando a posting list estiver completa). De maneira a conseguir calcular estes pesos, durante a indexação é calculado a frequência de cada termo para cada documento, o que depois permite calcular o idf para cada termo e o peso do termo para cada documento.





4 Resultados















Resultados usando os dois ficheiro 2004_TREC_ASCII_MEDLINE.gz, não fazendo nenhuma limitação da memória através da JVM, no início da execução do programa havia por volta de 4GB de memória disponível, escritas e leituras para o disco foram feitas sobre um HDD e o CPU i5-6300HQ CPU @ 2.30GHz.

	SPIMI com frequências	SPIMI com pesos	SPIMI com pesos e posições
Tempo de indexação (mm:ss)	5:22	6:42	10:54
Tamanho do index em disco (MB)	1150	1296	1373
Memória máxima usada (MB)*	1457	1627	1597

* segundo o gráfico presente na aba Monitor do VisualVM

5 Anexos

Item	Description
	The green arrow corresponds to the <code>implements</code> clause in a class declaration.
	The gray arrow corresponds to a call from the origin class of a method of the destination class.
	The blue arrow corresponds to the <code>extends</code> clause in a class declaration.
	This sign appears for the inner classes.

Icon	Description
	Class
	Abstract class
	Interface
	Method/function
	Interface method
	Static method
	Constant
	Field
	Property
	Final annotation
Visibility modifiers	
	Private
	Protected
	Public
	Static