BMI Practica 2

Leah Hadeed & Lorenzo Vela

1) Para esta práctica, hemos optado por hacer los siguientes ejercicios:

- 1.1 método orientado a términos
    - Para este ejercicio, codificamos la función de search del TermBasedVSMSearcher. Recorremos la lista de los términos del query y usamos un diccionario para guardar los acumuladores de cada termino. Para luego ordenarlos, usamos un heap para sacar los primeros N términos hasta el cutoff dado.
- 1.2 método orientado a documentos
    - For this excercise we've followed the logic presented in the slides. We've created a vector(self.allPostings) that contains tuples of 3 elements: [0] is the docid of the document, [1] is the frequency of the term in that document and [2] is the index of the query's term whom I'm referring to. For example, if the tuple is referring to the postings of the third term of the query, it will have tup[2]=3. We have to store this information because the tree will always have to contain one element for each query term(so in the tree for all the nodes, nodes[2] will be distinct, from 1 to |query|).
      After ordinating allPostings basing on the docid (from smallest to biggest), we start pushing for every term of the query, the first tuple that appears in the array: this is to guarantee that we are pushing the documents from smallest to biggest docid for every term.
      As while we push elements(based on the first value of the tuple, the docid), we'll remove them from the allPostings array, helped by the support array(created in order to not update the allPostings array while we're scrolling it).
      The pop of the first element is not inside the "while" construct because in this special case we can't obviously compare its docid with the previous exctracted one.
      The variable 'temp' is nothing more than the accumulator that we've seen in class: while the docid remains the same (sOutTup[0] = fOutTup[0]) we just sum there all the frequencies for the same docid.
      In order to calculate the final score we'll divide the accumulator "term" by the module of the document.
      The final structure is the dictionary "score" that has as key the docid and as value the score of the document: for every element in this structure we will call the push on the ranking's heap described below.
- 1.3 Heap de ranking
    - In this excercise we are using the library heapq to handle the ranking heap. The '__init__ function takes as input the cutoff(this number will be the tree's size). Moreover the 'push' function takes as parameter score and docid. In this case the order matters: indeed the score will be the key-value for the functions heapq.heappush and heapq.heappop.
      We firstly push elements until the cutoff dimension is reached, and then we push and pop all the others.
      The logic behind our code is mainly encapsulated in th scrolling of the heap: in order to do that we should before pop everything out, append all the nodes in the "nodes" array, and then, after updating it, push all the array back in the tree. The heapq library ensures us that the tree will always be ordered.
- 2 Índice en RAM

- Para este ejercicio, codificamos el Builder y el RAMIndex. Para el Builder, guardamos la lista de postings en un diccionario Python con clave termino y valor tupla docid y frecuencia. Usamos la librería de pickle para guardar los datos en fichero (POSTINGS_FILE). Luego guardamos los nombres de los documentos en un fichero aparte (INDEX_FILE). Cuando creamos el RAMIndex, leemos de los ficheros y metemos los documentos en el docmap y calculamos los módulos de los scores.

- 6 PageRank
  - Para este ejercicio, seguimos el algoritmo de clase de PageRank. Creamos dos diccionarios de conexiones (in y out), y los rellenamos leyendo del fichero de grafos dado con clave nodo de entrada/salida y valor lista de conexiones de entrada/salida. Tomamos la unión de claves de los dos diccionarios para encontrar los sumideros. Para calcular el valor P' de los nodos, calculamos la suma del valor de las conexiones de salida con el siguiente calculo:

```
        for k in keys:
            self.p[k] = 1/N
        #print(self.outConnections)
        sinks = self.inConnections.keys() - self.outConnections.keys()
        # Begin iterations
        for n in range(n_iter):
            for k in keys:
                self.p_p[k] = (1-r) / N
            for i in self.outConnections:
                for j in self.outConnections[i]:
                    self.p_p[j] += (r * self.p[i] / len(self.outConnections[i]))
+ (r * self.p[i] * len(sinks) / N)
            for k in keys:
                self.p[k] = self.p_p[k]
```

2) El siguiente diagrama muestra las clases implementadas para la práctica.

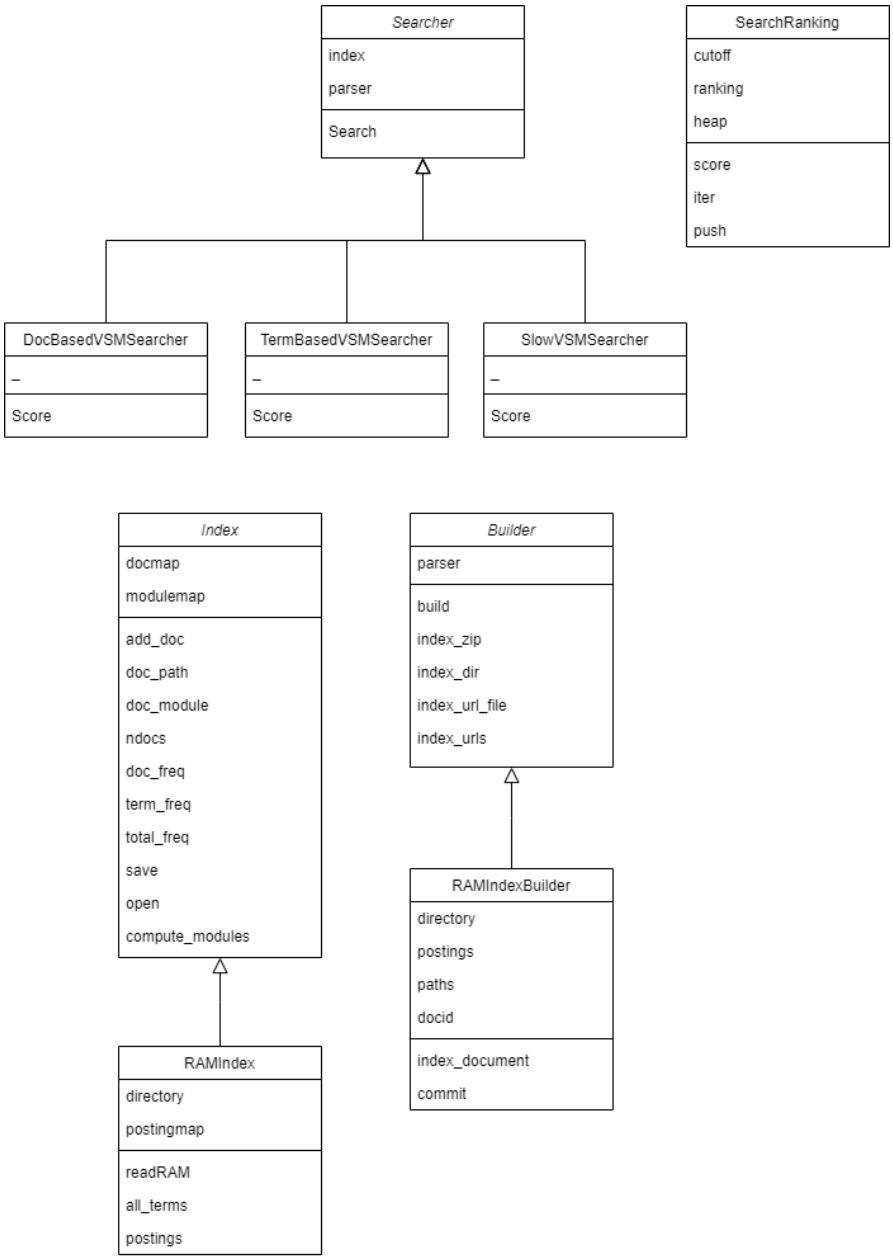

Figure 1: Diagrama de Clases

Los searchers heredan de la clase abstracta Searcher y en la función score, calcula el ranking utilizando la clase SearchRanking. Como tenemos implementado además el heap de ranking en la clase searchranking, ordenamos los documentos utilizando esa misma clase.

Para los índices, como solo creamos el RAMIndex y su Builder, estas clases heredan directamente de Index y Builder con algunas funciones sobreescritas para reflejar la nueva funcionalidad del índice. El RAMBuilder crea el índice y lo escribe en un fichero, mientras que el RAMIndex lee del fichero y rellena los campos de este, como el docmap y el modulomap.

3) A continuación mostramos una tabla de rendimientos:

	Construccion del Indice			Carga del indice	
	Tiempo de indexado	Consumo max RAM	Espacio en disco	Tiempo de carga	Consumo max RAM
Toy1	0.000597477		1705	0.000375509	
Toy2	0.000514507		1598	0.000333786	
1K	48.27505898		13943669	2.07007885	
10K	338.3613372		95847901	15.33669233	
urls	2.071116686		347171	0.131609917	