Búsqueda y Minería de Información

P2: Motores de Búsqueda e Indexación

Javier Alejandro Lougedo Lorente javier.lougedo@estudiante.uam.es
Alba Ramos Pedroviejo alba.ramosp@estudiante.uam.es

Búsqueda y Minería de Información Group 2461
Profesor: Alejandro Bellogín Kouki
Grado de Ingeniería Informática - Universidad Autónoma de Madrid
5 de abril de 2021, Madrid, España

Resumen

En la presente práctica implementaremos distintos motores de búsqueda y la indexación de documentos por distintas vías, como son las listas de *postings*, diccionarios y otros modelos almacenables en *RAM* y en memoria, así como la implementación de *PageRank*. Esto se hace con la ayuda de las librerías *Whoosh*, *BeautifulSoup4*, *ZipFile* y de alguna librerías *Python*, cuya instalación y uso están comentados en el fichero *readme_lib.txt*.

Índice

1. Ejercicio 1: Un modelo vectorial eficiente

En este ejercicio vamos a mejorar drásticamente la implementación que ya realizamos previamente, dejándola lenta al lado de las nuevas, aplicando los algoritmos que hemos aprendido en teoría, utilizando listas de postings en lugar de un índice forward.

1.1. Método Orientado a Términos - TermBasedVSMSearcher

En primer lugar, vamos a implementar el modelo vectorial coseno por el método orientado a términos.

Para ello, implementamos la clase de la siguiente forma:

Código Python 1: TermBasedVSMSearcher

```python
class TermBasedVSMSearcher(Searcher):
    def __init__(self, index, parser=BasicParser()):
        super().__init__(index, parser)

    def search(self, query, cutoff):
        query_terms = self.parser.parse(query)
        ranking = SearchRanking(cutoff)
        posting_values = {}
        scores = []

        for term in query_terms:
            for doc_id, _ in self.index.postings(term):
                if doc_id not in posting_values.keys():
                    posting_values[doc_id] = []
                posting_values[doc_id].append(term)

        for doc_id in posting_values.keys():
            score = self.score(doc_id, posting_values[doc_id])
            if score:
                ranking.push(self.index.doc_path(doc_id), score)

        return ranking

    def score(self, doc_id, query_terms):
        product = 0
        for term in query_terms:
            product += tf(self.index.term_freq(term, doc_id)) * \
                idf(self.index.doc_freq(term), self.index.ndocs())

        module = self.index.doc_module(doc_id)
        return (product / module) if module else 0
```

Implementa, ya que en este caso lo vimos más cómodo de esta manera, dos funciones, como de costumbre, una para calcular el *score* y otra para calcular la puntuación y el *ranking* en sí de la colección, tal y como se nos enseña en las clases de teoría. No tuvimos problemas en la implementación y nos resultó sencilla y rápida, aunque intuimos por los tiempos obtenidos que hay alguna ligera mejora posible todavía.

1.2. Método Orientado a Documentos - DocBasedVSMSearcher

En este primer ejercicio opcional se pide llevar a cabo la implementación del método orientado a documentos, con el heap de postings que se implementa en el siguiente apartado.

Código Python 2: DocBasedVSMSearcher

```
class DocBasedVSMSearcher(Searcher):
    def __init__(self, index, parser=BasicParser()):
        super().__init__(index, parser)

    def search(self, query, cutoff):
        ndocs = self.index.ndocs()
        qterms = self.parser.parse(query)
        indices = [0] * len(qterms)  # Cosecuentials
        postings = []
        acc = [0] * ndocs  # Accumulators
        ranking = SearchRanking(cutoff)

        for q in qterms:
            postings += [[p for p in self.index.postings(q)]]

        heap = []
        heapify(heap)
        for q in range(len(qterms)):
            heappush(heap, (postings[q][indices[q]], q))
            indices[q] += 1  # iterate cosecuentially

        currDoc = nsmallest(1, heap)[0][0][0]
        try:
            while(1):
                element = heappop(heap)
                docid = element[0][0]
                freq = element[0][1]
                q = element[1]
                score = tf(freq) * idf(self.index.doc_freq(qterms[q]), ndocs)

                if docid != currDoc:
                    acc[currDoc] /= self.index.doc_module(currDoc)
                    # insert only if not less than top if full
                    if len(ranking.ranking) < ranking.cutoff:
                        ranking.push(self.index.doc_path(
                            currDoc), acc[currDoc])
                    elif score > nsmallest(1, ranking.ranking)[0][0]:
                        ranking.push(self.index.doc_path(
                            currDoc), acc[currDoc])

                    currDoc = docid
                acc[currDoc] += score

                if indices[q] < len(postings[q]):
                    heappush(heap, (postings[q][indices[q]], q))
                    indices[q] += 1

        except Exception:  # El heap se ha vaciado
            acc[currDoc] /= self.index.doc_module(currDoc)
            if len(ranking.ranking) < ranking.cutoff:
                ranking.push(self.index.doc_path(currDoc), acc[currDoc])
            elif score > nsmallest(1, ranking.ranking)[0][0]:
                ranking.push(self.index.doc_path(currDoc), acc[currDoc])

        return ranking
```

En este caso, decidimos no implementar una función de *score* y realizarlo de esta manera porque nos resultaba más sencillo visualizarlo y no vimos la necesidad. En cuando a decisiones de diseño e implementación, no tuvimos muchos problemas tras asistir a las lecciones de teoría y haber realizado la práctica anterior.
Hemos seguido así la teoría lo máximo posible, creando una lista de acumuladores de *score* coseno para cada documento, y una lista de índices para cada término de la consulta, lo que nos permite realizar una iteración cosecuencial sobre sus listas de *postings*:

1. Obtenemos las listas y creamos un *heap* de tantos nodos como términos en la consulta, insertando el primer elemento de cada *qterm*.

2. Tras eso, mientras queden elementos, extraemos la cima (el *heap* se recolocará solo) y calculamos *tf-idf* (ya que la lista de *postings* viene con frecuencias).

3. Si el elemento pertenece al mismo *doc_id* que el que se extrajo la última vez, se acumula su *tf-idf*, si no, se calcula el *score* coseno del *doc_id* anterior, dividiendo su acumulado entre su módulo, y se inserta al *ranking*, acumulando además el *tf-idf* del elemento extraído.

4. Si quedan elementos en la lista de *postings* de ese *qterm*, se insertará el siguiente.

5. Sabremos que el *heap* se vacía porque *heappop* lanzará una excepción, en cuyo caso calculamos el *score* coseno del último elemento y se inserta al *ranking*.

Todas las inserciones se realizan comprobando si el *heap* de *ranking* se ha llenado de *cutoff*, en cuyo caso solo se inserta el elemento si tiene un *score* mayor al de la cima.

1.3. Heap de Ránking - SearchRanking

Se pide reimplementar la clase *SearchRanking* para utilizar un *heap* de *ranking*, de la que nos podremos aprovechar en ambos métodos implementados en este primer apartado. Este será un *minHeap* de tuplas *(score, id)*, implementado con ayuda de la librería ***heapq***, que parte de una lista y la transforma en un *minHeap* mediante *heapify*. Nos permite insertar nodos con *heappush*, donde insertaremos tuplas como las anteriormente descritas. Para el *score*, emplearemos un método *iter* que devolverá un iterador, formado por una lista de tuplas que deben estar ordenadas en función de su *score*, en orden decreciente (cabe recordar en este punto la teoría, donde creamos un *minHeap* y luego el *ranking* era el reverso de este). Para ordenarlas, usamos la función *nlargest*, que devolverá los N elementos más grandes del *heap* en forma de una lista de tuplas. Este N es, como cabe suponer, el *cutoff*, que resulta útil para emplear este *heap* en otros modelos, y no solo en el orientado a documentos. Finalmente, el *main* espera las tuplas como *(doc_id, score)*, así que se invierten en el propio iterador.

La implementación realizada es la que se encuentra en la página a continuación.

Código Python 3: SearchRanking

```python
class SearchRanking:
    def __init__(self, cutoff):
        self.ranking = []
        heapify(self.ranking)
        self.cutoff = cutoff

    def push(self, docid, score):
        heappush(self.ranking, (score, docid))

    def __iter__(self):
        return iter([tuple(reversed(e)) for e in nlargest(
            min(len(self.ranking), self.cutoff),
            self.ranking)])
```

2. Ejercicio 2: Índice en RAM

En este ejercicio llevaremos a cabo la implementación de un índice propio que tenga las mismas funciones que la que ya definimos en la práctica anterior, basada en ***Whoosh***. Para implementarlo en RAM, se nos recomienda el uso de la librería ***pickle***, a fin de guardarlo en disco y leerlo serializado. Se divide este ejercicio en dos apartados: la implementación del índice en sí y de su *builder*.

2.1. Estructura de índice - RAMIndex

Para la implementación del índice en RAM hemos utilizado un diccionario igual que los que hemos visto en la teoría: las claves son los términos, y el valor de estas será la lista de *postings* (en formato *(doc_id, frecuencia)* de dicho término. De ese modo, para obtener la lista de *postings* de un determinado término, basta con indexar a través de dicho término en el diccionario y ya tendremos, de manera rápida y sencilla, su lista. Obtener el vocabulario es además tan fácil como obtener las claves del diccionario, con lo cual muchos procesos se simplifican bastante, y la implementación resulta sencilla, como veremos a continuación.
De esta forma, para implementar el índice en RAM, creamos un Índice base con dos variables nuevas, que serán un diccionario con los términos y sus *postings* de la forma y con la estructura ya mencionada. Redefinimos las dos funciones de *postings* y *all_terms* del índice, y añadimos los apartados adicionales de los ficheros a las funciones de *open* y *save*, que llamarán adicionalmente a las del objeto base. Así, obtenemos un índice funcional de forma rápida y sencilla, pues las implementaciones de *postings* y *all_terms* son rápidas y directas, a través de este diccionario, y las de *open* y *save* bastante lógicas e inmediatas, tomando como referencia las base y la ayuda de ***pickle***. No es necesario implementar nada más en este punto.
La implementación se encuentra así en la siguiente página.

Código Python 4: RAMIndex

```python
class RAMIndex(Index):
    def __init__(self, dir_path=None):
        self.terms_dict = {}  # Indice en si: {term: postings}
        self.postings_path = os.path.join(dir_path, Config.POSTINGS_FILE)
        super().__init__(dir_path)

    def postings(self, term):
        return self.terms_dict[term]

    def all_terms(self):
        return self.terms_dict.keys()

    def save(self, dir_path):
        with open(os.path.join(dir_path, Config.PATHS_FILE), 'wb') as f:
            pickle.dump(self.docmap, f)
        with open(os.path.join(dir_path, Config.DICTIONARY_FILE), 'wb') as f:
            self.terms_dict = collections.OrderedDict(
                sorted(self.terms_dict.items()))  # Orden alfabetico
            pickle.dump(self.terms_dict, f)
        super().save(dir_path)

    def open(self, dir_path):
        super().open(dir_path)

        try:
            with open(os.path.join(dir_path, Config.PATHS_FILE), 'rb') as f:
                self.docmap = pickle.load(f)
            with open(os.path.join(dir_path, Config.DICTIONARY_FILE), 'rb') as f:
                self.terms_dict = pickle.load(f)
        except OSError:
            # the file may not exist the first time
            pass
```

2.2. Construcción del índice - RAMIndexBuilder

Para construirlo, modificamos la función de *commit*, donde se guardan las estructuras del índice tal y como se indica más adelante, así como la de *index_document*. Aquí, cada vez que llega un nuevo documento, almacenamos su *path* y contamos los documentos que llevamos, a fin de ir asignando un *doc_id* actualizado a cada documento. Se crea un diccionario auxiliar de término-frecuencia donde vamos almacenando la información que tendrá el índice, y posteriormente *parseamos* el documento, vía *BasicParser* (o cualquier otro indicado), y recorremos cada término realizando el conteo en el diccionario y actualizando su información. Tras procesar el documento, recorremos los términos de este diccionario y se va añadiendo al índice la lista de *postings* de dicho documento, pues ya conocemos la frecuencia.
En cuanto a la función del *commit*, su implementación resulta sencilla, pues viendo lo que hacía el código fuente de ***Whoosh*** dedujimos que bastaba con hacer un *dump* del índice en sí para posteriormente cargarlo, así como llamar al *save* del índice.
El resto de funciones se heredan del *Builder* base, pues son iguales y no necesitan ningún tipo de cambio o actualización para funcionar correctamente.

Código Python 5: RAMIndexBuilder

```python
class RAMIndexBuilder(Builder):
    def __init__(self, dir_path, parser=BasicParser()):
        super().__init__(dir_path, parser)
        self.path = dir_path
        self.index = RAMIndex(dir_path)

    def commit(self):
        self.index.save(self.path)
        with open(self.path + Config.INDEX_FILE, "wb") as f:
            pickle.dump(self.index, f)

    def index_document(self, path, text):
        new_doc_id = self.index.ndocs()
        self.index.add_doc(path)
        text = BasicParser.parse(text)
        doc_term_freq = {}  # Diccionario term:freq auxiliar en este documento
        for pos, term in enumerate(text):
            if self.index.terms_dict.get(term) is None:
                doc_term_freq[term] = 0
                self.index.terms_dict[term] = []
            elif doc_term_freq.get(term) is None:
                doc_term_freq[term] = 0
            doc_term_freq[term] += 1
        for term in doc_term_freq.keys():
            self.index.terms_dict[term].append(
                [new_doc_id, doc_term_freq[term]])
```

3. Ejercicio 3: Índice en disco

En este segundo ejercicio opcional, se nos pide llevar a cabo la implementación del índice en **disco**. Para ello, se nos ocurrió incluso que heredase de **RAMIndex**, pero decidimos evitar esto para tener una implementación algo más clara y flexible, llegado el caso, heredando directamente de **Index** en su lugar.

3.1. Estructura de índice - DiskIndex

En primer lugar vamos a hablar de la estructura del índice. En esta ocasión, lo creamos con un diccionario de términos y *postings*, otro de términos y sus *offsets* para cuando sea necesario ir a memoria, otro con sus *paths*, como el de **RAMIndex**, y un conteo de los documentos, que facilita las funciones de *ndocs*. En cuanto a la función de *postings*, en esta ocasión es mucho más compleja. Lo que hacemos es abrir el fichero en el que tenemos todo almacenado de la manera más eficiente a la par que sencilla y clara que hemos encontrado, se va a la posición que indique el offset para dicho término y lee dicha fila del documento, cargando los *postings*. En cuanto a la función de *doc_freq*, a fin de obtener resultados más eficientes, la reimplementamos también, pues en esta ocasión en lugar de cargar los *postings* y medir su longitud, lo que hace es leer un número que hemos introducido al guardar los *postings* en la función *save*, que es la longitud de todos ellos. De esta forma, esta función pasa a ser mucho más eficiente. En cuanto al *save* y al *open*, se aprovechan en gran medida de ***pickle***,

guardando el fichero en un formato cómodo para la función de *postings*, que hemos intentado que adicionalmente, sea legible.

Código Python 6: DiskIndex

```python
class DiskIndex(Index):
    def __init__(self, dir_path=None):
        self.postings_dict = {}  # term:[postings]
        self.terms_dict = {}  # {term: offsetPostings}
        self.postings_path = os.path.join(dir_path, Config.POSTINGS_FILE)
        self.count = 0
        super().__init__(dir_path)

    def postings(self, term):
        res_postings = []
        with open(self.postings_path, 'r') as f:
            f.seek(self.terms_dict[term])
            doc_amount, raw_postings = f.readline().strip().split("=")
            raw_postings = raw_postings.split(" ")
            for i in range(0, int(doc_amount) * 2, 2):
                res_postings.append(
                    [int(raw_postings[i]), int(raw_postings[i+1])])
        return res_postings

    def all_terms(self):
        return self.terms_dict.keys()

    def doc_freq(self, term):
        with open(self.postings_path, 'r') as f:
            f.seek(self.terms_dict[term])
            result = f.readline().split("=")[0]
        return int(result)

    def save(self, dir_path):
        with open(os.path.join(dir_path, Config.PATHS_FILE), 'wb') as f:
            pickle.dump(self.docmap, f)
        with open(os.path.join(dir_path, Config.POSTINGS_FILE), 'w') as f:
            for term, postings in self.postings_dict.items():
                self.terms_dict[term] = f.tell()
                n_postings = len(postings)
                f.write(str(n_postings) + "=")
                for p in postings:
                    f.write(str(p[0]) + " ")
                    f.write(str(p[1]) + " ")
                f.write("\n")
        with open(os.path.join(dir_path, Config.DICTIONARY_FILE), 'wb') as f:
            self.terms_dict = collections.OrderedDict(
                sorted(self.terms_dict.items()))  # Orden alfabetico
            pickle.dump(self.terms_dict, f)
        super().save(dir_path)  # Guarda modulos

    def open(self, dir_path):
        super().open(dir_path)  # Carga modulos
        self.postings_dict = {}
        try:
            with open(os.path.join(dir_path, Config.PATHS_FILE), 'rb') as f:
                self.docmap = pickle.load(f)
            with open(os.path.join(dir_path, Config.DICTIONARY_FILE), 'rb') as f:
                self.terms_dict = pickle.load(f)
        except OSError:
            # the file may not exist the first time
            pass
```

Con esta estructura, obtener los *postings* de un término implica leer el fichero de *postings* y desplazarse al offset indicado. Una vez ahí, leer la línea y recorrer los *docid-freq* que hemos almacenado. Por otro lado, obtener la cantidad de documentos donde aparece un término es tan sencillo como mirar el fichero de *postings*, desplazarse al offset de dicho término y leer el primer número. Obtener el vocabulario se haría igual que **RAMIndex**. El resto de funcionalidad es igual que en **Index**.
Adicionalmente, cuando nos encontramos con un problema en cuanto a la eficiencia que no fuimos capaces de ubicar inicialmente, implementamos una variante, *CachedDiskIndex*, donde guardamos una pequeña caché en RAM para evitar ir muchísimas veces a disco a por el mismo término cuando estemos realizando búsquedas. Esto optimiza bastante los tiempos, como pudimos observar en su momento, pese a tener una implementación ineficiente del guardado y lectura del fichero de *postings*. Posteriormente, mejoramos estas funciones y manera de guardarlo obteniendo una mayor eficiencia y haciendo que no fuese necesaria esta pequeña caché, que sencillamente lo que hacía era almacenar en un diccionario aquellos *postings* que había leído recientemente. Esto se podría implementar, con algún tipo de *heap* o similar, que en cuanto pasase de un determinado número de elementos en la caché, o de una determinada cantidad de memoria, eliminase el elemento por el que se consultó hace más tiempo, para cargar uno más consultado de memoria.

3.2. Construcción del índice - DiskIndexBuilder

En este caso, es exactamente igual que la de **RAMIndex**, pues el tratamiento es el mismo, excepto por la línea 5, donde se crea un **DiskIndex** en lugar de un **RAMIndex**.

Código Python 7: DiskIndexBuilder

```
1  class DiskIndexBuilder(Builder):
2      def __init__(self, dir_path, parser=BasicParser()):
3          super().__init__(dir_path, parser)
4          self.path = dir_path
5          self.index = DiskIndex(dir_path)
6
7      def commit(self):
8          self.index.save(self.path)  # Guardamos las estructuras del indice
9          with open(self.path + Config.INDEX_FILE, "wb") as f:
10             pickle.dump(self.index, f)  # Guardamos el indice en si
11
12     def index_document(self, path, text):
13         new_doc_id = self.index.ndocs()
14         self.index.add_doc(path)
15         text = BasicParser.parse(text)
16         doc_term_freq = {}  # Diccionario term:freq en este documento
17         for pos, term in enumerate(text):
18             if self.index.postings_dict.get(term) is None:
19                 doc_term_freq[term] = 0
20                 self.index.postings_dict[term] = []
21             elif doc_term_freq.get(term) is None:
22                 doc_term_freq[term] = 0
23             doc_term_freq[term] += 1
24         for term in doc_term_freq.keys():
25             self.index.postings_dict[term].append(
26                 [new_doc_id, doc_term_freq[term]])
```

4. Ejercicio 4: Motor de búsqueda proximal

Este ejercicio nos costó especialmente, a la hora de implementar el algoritmo del *score* de manera adecuada y eficiente. Finalmente, gracias a un pequeño estudio de dos de los *papers* que encontramos en Internet, logramos implementarlo con éxito. En él se pide, básicamente, implementar un motor de búsqueda proximal.

4.1. Implementación - ProximitySearcher

Como ya hemos comentado, para implementarlo nos ayudamos de un par de *papers* de Internet, así como las diapositivas en teoría. Es un algoritmo que en papel y en los ejercicios teóricos resulta sencillo y rápido de aplicar, pero que a la hora de programar y traducir a código nos dio problemas. Sin embargo, finalmente conseguimos una implementación exitosa, como se podrá observar más abajo, en la cual no hay comentarios explicativos de código (como si los hay en el código fuente en sí mismo, a fin de reducir el tamaño de la figura).
De esta forma, lo que hacemos es, en primer lugar, es *parsear* la *query*, crear un *ranking* de *cutoff* elementos y observar cuales son los documentos válidos con todas las consultas, a fin de calcular su *score*. Esto lo hacemos cogiendo los documentos que contienen el primer elemento de la consulta y aplicando la intersección con los que contienen el segundo elemento, la intersección del resultado con los que contienen el tercero, etc. Así, acabamos con un *set* de *doc_ids* con los documentos que contienen todos y cada uno de los términos, independientemente del orden de aparición.
Tras esto, lo que haremos será obtener y limpiar toda la información vía *positional postings*. Lo que hacemos es, básicamente, llamar a *positional postings*, eliminando todos y cada uno de los documentos que no figuren en el *set* de documentos válidos.
Cuando ya lo tenemos listo, para cada uno de estos documentos **higienizados**, llamamos a la función de *score*, que obtendrá B y A de cada grupo de elementos y calculará el *score*. Esta es la parte que más dificultades tuvimos en implementar, ya que no terminábamos de entender muy bien este algoritmo. Su pseudocódigo es el siguiente:

```
score ← 0
for j=1 to |q| do p[j] ← 0
b ← max_j posList[j][p[j]]
while b < ∞ do
    i ← 0
    for j=1 to |q| do
        // Avanzar todas las listas hasta posición b
        while posList[j][p[j]+1] ≤ b do p[j]++
        // Nos queremos quedar en i con la posición más baja de las |q| listas
        if posList[j][p[j]] < posList[i][p[i]] then i ← j
    a ← posList[i][p[i]]
    score ← score + 1/(b - a - |q| + 2)
    b ← posList[i][++p[i]]
return score
```

Figura 1: Pseudocódigo de la función evaluadora

Cabe mencionar que nuestra implementación, gracias a estas higienizaciones y limpiezas, resulta extremadamente eficiente, como se puede observar comparando tiempos de ejecución.

Código Python 8: ProximitySearcher

```python
class ProximitySearcher(Searcher):
    def __init__(self, index, parser=BasicParser()):
        super().__init__(index, parser)

    def search(self, query, cutoff):
        query_terms = self.parser.parse(query)
        ranking = SearchRanking(cutoff)

        suitable_doc_ids = set()
        for doc_id, _ in self.index.postings(query_terms[0]):
            suitable_doc_ids.add(doc_id)

        if len(query_terms) > 1:
            for term in query_terms[1:]:
                current_doc_ids = set()
                for doc_id, _ in self.index.postings(term):
                    current_doc_ids.add(doc_id)
                suitable_doc_ids &= current_doc_ids

        cleaned_positions = {}
        for term in query_terms:
            term_clean_positions = {}
            term_positions = self.index.positional_postings(term)
            for doc_positions in term_positions:
                if doc_positions[0] in suitable_doc_ids:
                    term_clean_positions[doc_positions[0]] = doc_positions[1]
            cleaned_positions[term] = term_clean_positions

        for doc_id in suitable_doc_ids:
            ranking.push(self.index.doc_path(doc_id),
                         self.score(cleaned_positions, doc_id, query_terms))

        return ranking

    def score(self, cleaned_positions, doc_id, query_terms):
        posList = []
        for term in query_terms:
            posList.append(cleaned_positions[term][doc_id] + [math.inf])
        q_len = len(query_terms)
        if q_len == 1:
            return float(len(posList[0]) - 1)
        a = - 1
        score = 0
        p = [0] * q_len
        b = max([t_list[0] for t_list in posList])

        while b != math.inf:
            i = 0
            for j in range(q_len):
                while posList[j][p[j]+1] <= b:
                    p[j] += 1
                if posList[j][p[j]] < posList[i][p[i]]:
                    i = j
            a = posList[i][p[i]]
            score += 1 / (b - a - q_len + 2)
            b = posList[i][p[i]+1]

        return score
```

5. Ejercicio 5: Índice posicional

En este cuarto y último ejercicio voluntario, se nos pide implementar una variable adicional que extienda los índices con la inclusión de soporte para posiciones en las listas de *postings*. En nuestro caso, este heredará de la implementación en *RAM*, ya que para los datos que manejamos es más rápido y tiene unas estructuras similares.

5.1. Estructura de índice - PositionalIndex

Conocer la lista de *postings* como hasta ahora implica, para un término, iterar sobre las claves de su diccionario (*docids*) y ver la longitud de cada lista de posiciones (freq). La lista de *postings* posicionales simplemente implica, para un término, iterar sobre su diccionario y obtener la lista de posiciones de cada *docid*. Lo demás sería igual que en **RAMIndex**, que es la superclase.
También probamos, y podemos, hacer que herede de DiskIndex. Sin embargo, nos parecía ligeramente más lento y poco ágil en este sentido. El resultado final de relaciones de herencias entre los distintos índices se puede observar más adelante en el diagrama de clases.

Código Python 9: PositionalIndex

```python
class PositionalIndex(RAMIndex):
    def __init__(self, dir_path=None):
        self.terms_dict = {}
        super().__init__(dir_path)

    def positional_postings(self, term):
        dict = self.terms_dict[term]
        res = []
        for doc, pos in dict.items():
            res += (doc, pos) # [ [ doc1, [posiciones] ], [ doc2, [posiciones] ], ...]

        return res

    def postings(self, term):
        dict = self.terms_dict[term]
        res = []
        for doc, pos in dict.items():
            res.append((doc, len(pos)))
        return res
```

5.2. Construcción del índice - PositionalIndexBuilder

Para construir el índice, no hemos de almacenar frecuencias esta vez, sino posiciones, con lo que vamos a tener un índice que sea un diccionario término-diccionario, donde este segundo diccionario será uno con la forma *docid*-posiciones. Así, el diccionario auxiliar que teníamos antes con la forma término-frecuencia, se convierte en uno de la forma término-posiciones, con lo que podremos conocer las posiciones de manera sencilla, vía la función *enumerate* al iterar.

Código Python 10: PositionalIndexBuilder

```python
class PositionalIndexBuilder(RAMIndexBuilder):
    def __init__(self, dir_path, parser=BasicParser()):
        super().__init__(dir_path, parser)
        self.index = PositionalIndex(dir_path)

    # Vamos creando el indice en RAM
    def index_document(self, path, text):
        new_doc_id = self.index.ndocs()
        self.index.add_doc(path)
        text = BasicParser.parse(text)
        doc_term_pos = {} # Diccionario term: [pos1, pos2...]
        for pos, term in enumerate(text):
            if self.index.terms_dict.get(term) is None:
                doc_term_pos[term] = []
                self.index.terms_dict[term] = {}
            elif doc_term_pos.get(term) is None:
                doc_term_pos[term] = []
            doc_term_pos[term].append(pos)

        for term in doc_term_pos.keys():
            self.index.terms_dict[term][new_doc_id] = doc_term_pos[term]
```

6. Ejercicio 6: PageRank

En el último ejercicio de esta práctica, se nos pide implementar el algoritmo *PageRank* en la clase **PageRankDocScorer**, que devolverá un *ranking* de manera similar a como lo hace *Searcher*.
Cuando creamos este *scorer*, leemos el documento de grafos y vamos almacenando todos los nodos del grafo (sin repetición), así como un diccionario con los nodos a los que apunta cada nodo (origen-destinos) y otro con los nodos que apuntan a cada nodo (destino-orígenes). Los sumideros se detectan porque tendrán la lista de destinos vacía, y los podemos guardar en una lista aparte para acelerar el proceso de *ranking* posterior.

6.1. Implementación - PagerankDocScorer

Para implementar *PageRank* se ha seguido el algoritmo visto en clase. Inicialmente se les da a todos los nodos un *PageRank* de 1, y luego para cada nodo se calcula su *PageRank* con respecto a los valores de los nodos que le apuntan (más los sumideros) en la iteración anterior. El valor de la iteración anterior se almacena en un diccionario *PageRankPrev* una vez hechos los cálculos de esa iteración. Esto se realiza para tantas iteraciones como nos especifiquen. Finalmente, los valores se ordenan de mayor a menor y se devuelven como tuplas (nodo, *PageRank*) hasta alcanzar el *cutoff*.

Código Python 11: PagerankDocScorer

```python
class PagerankDocScorer():
    def __init__(self, graphfile, r, n_iter):
        self.n_iter = n_iter
        with open(graphfile, "r) as f:
            lines = f.read().splitlines()
        self.nodes = set()  # Elementos no ordenados y sin repetición
        self.outs = {}  # Clave: nodo, valor: lsita de a quienes apunta
        self.to_from_nodes = {}  # Clave: nodo, valor: lista de quienes le apunatn
        for line in lines:
            ele = line.split('\t')
            self.nodes.add(ele[0])
            self.nodes.add(ele[1])
            try:
                self.outs[ele[0]].append(ele[1])
            except:
                self.outs[ele[0]] = [ele[1]]
            try:
                self.to_from_nodes[ele[1]].append(ele[0])
            except:
                self.to_from_nodes[ele[1]] = [ele[0]]
            try:
                if self.outs[ele[1]]:
                    pass
            except:
                self.outs[ele[1]] = []
            try:
                if self.to_from_nodes[ele[0]]:
                    pass
            except:
                self.to_from_nodes[ele[0]] = []
        self.sumideros = []
        for node in self.outs.keys():
            if len(self.outs[node]) == 0:
                self.sumideros.append(node)
        self.n = len(self.nodes)
        self.r = r

    def rank(self, cutoff):
        p1 = self.r / self.n
        p2 = 1 - self.r
        res = []
        pageRankPrev = {}
        pageRankAct = {}
        for n in self.nodes:
            pageRankPrev[n] = 1
        for epoc in range(self.n_iter):
            for n in self.nodes:
                sum = 0
                for origin in self.to_from_nodes[n]:
                    sum += pageRankPrev[origin] / len(self.outs[origin])
                for i in self.sumideros:
                    sum += pageRankPrev[i] / self.n
                pageRankAct[n] = p1 + p2 * sum
            for n in pageRankAct.keys():
                pageRankPrev[n] = pageRankAct[n]
        ordered = dict(sorted(pageRankAct.items(), key=lambda item: item[1], reverse=True))
        cont = 0
        for k, v in ordered.items():
            res.append((k, v))
            cont += 1
            if cont == cutoff:
                break
        return res
```

7. Diagrama de Clases

Hemos realizado, tal y como se nos pedía en la entrega, un diagrama de clases **sencillo** en el que observar y entender las relaciones jerárquicas y herencias de las distintas clases. Hemos tratado de reducir al máximo la información mostrada para quedarnos con los detalles más importantes, así como simplificar las relaciones a gran escala para no sobrecomplicar el diagrama. El diagrama es este:

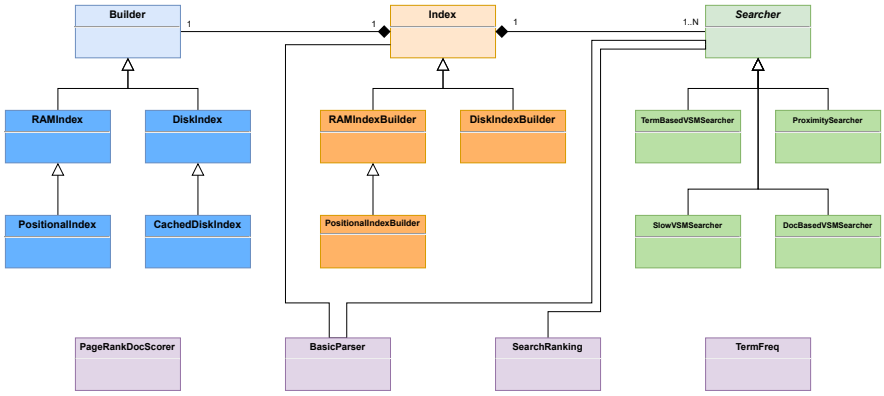

Figura 2: Diagrama de clases de los buscadores implementados.

En él, podemos observar los siguientes detalles relevantes: los *Builders* han de tener un Índice al que dan lugar (cada uno con el suyo propio, p.e. *RAMIndex* con *RAMIndexBuilder*), y lo mismo ocurre con los *Searchers* (que además es abstracta y deberá ser implementada). Los *indices* y *searchers* tienen *parsers*, en este caso solo BasicParser, y los *searchers* hacen uso del *ranking*, en este caso de nuevo solo uno. El resto de relaciones son claras.

8. Estudio y análisis del rendimiento

Para este estudio, se ha realizado la ejecución en un ordenador *Mac Mini* con *chipset* M1, con las siguientes características (más información aquí):

Hardware Overview:

Model Name:	Mac mini
Model Identifier:	Macmini9,1
Chip:	Apple M1
Total Number of Cores:	8 (4 performance and 4 efficiency)
Memory:	16 GB
System Firmware Version:	6723.61.3
Serial Number (system):	C07F30VEQ6NY
Hardware UUID:	69582B71-A8D4-5869-B740-6ED666CC3DC6
Provisioning UDID:	00008103-000C684E21EA001E
Activation Lock Status:	Disabled

Figura 3: Información del entorno de ejecución

Los resultados obtenidos han sido los siguientes:

RAMIndex	Construcción del Índice			Carga del Índice	
Colección	Tiempo de Indexado	Consumo máx. RAM	Espacio en Disco	Tiempo de carga	Consumo máx. RAM
Toy1	*0.001466989517211914 seconds*	*89-92 %*	*1312 space*	*0.0001490116119384765 6 seconds*	*89-92 %*
Toy2	*0.0008068084716796875 seconds*	*89-92 %*	*1199 space*	*0.0001399517059326172 seconds*	*89-92 %*
Urls	*1.9586701393127441 seconds*	*89-92 %*	*139734 space*	*0.028786897659301758 seconds*	*89-92 %*
1K	*32.3735918998718 26 seconds*	*89-92 %*	*7264736 space*	*1.4944581985473633 seconds*	*89-92 %*
10K	*216.32021570205688 seconds*	*89-92 %*	*51154094 space*	*13.291764974594116 seconds*	*89-92 %*

Figura 4: Datos obtenidos de RAMIndex

DiskIndex	Construcción del Índice			Carga del Índice	
Colección	Tiempo de Indexado	Consumo máx. RAM	Espacio en Disco	Tiempo de carga	Consumo máx. RAM
Toy1	*0.0039370059967041016 seconds*	*89-92 %*	*1203 space*	*0.000102996826171875 seconds*	*89-92 %*
Toy2	*0.004070281982421875 seconds*	*89-92 %*	*1102 space*	*0.000104904174804687 5 seconds*	*89-92 %*
Urls	*2.15397310256958 seconds*	*89-92 %*	*125906 space*	*0.0013191699981689453 seconds*	*89-92 %*
1K	*37.9258201122283 94 seconds*	*89-92 %*	*5978775 space*	*0.036860942840576 17 seconds*	*89-92 %*
10K	*241.544331073761 seconds*	*89-92 %*	*42537198 space*	*0.155346870422363 28 seconds*	*89-92 %*

Figura 5: Datos obtenidos de DiskIndex

PositionalIndex	Construcción del Índice			Carga del Índice	
Colección	Tiempo de Indexado	Consumo máx. RAM	Espacio en Disco	Tiempo de carga	Consumo máx. RAM
Toy1	*0.0008111000061035156 seconds*	*89-92 %*	*1331 space*	*0.000109910964965820 31 seconds*	*89-92 %*
Toy2	*0.0012600421905517578 seconds*	*89-92 %*	*1210 space*	*0.000113010406494140 62 seconds*	*89-92 %*
Urls	*1.817927360534668 seconds*	*89-92 %*	*209804 space*	*0.004964113235473633 seconds*	*89-92 %*
1K	*33.027628898620605 seconds*	*89-92 %*	*11889007 space*	*1.0943729877471924 seconds*	*89-92 %*
10K	*220.53490924835205 seconds*	*89-92 %*	*79017238 space*	*16.21553111076355 seconds*	*89-92 %*

Figura 6: Datos obtenidos de PositionalIndex

Comentando los mismos, el primer detalle a tener en cuenta, y es uno bastante relevante y que por suerte tenemos localizado, es el uso de la RAM. No hay muchas herramientas para tomar dicha medida de forma fiable, más allá de crear un proceso *sniffer* que estudie y mida cuanta RAM va gastando a cada momento. Por ello, optando por una solución más sencilla, decidimos intentar medir la RAM nosotros en todo momento, lo cual dio lugar a problemas, como los que observamos en la toma de datos, debido a que en los ordenador con *Mac OS*, concretamente este *Mac M1*, se hace un uso generalmente muy dinámico de la RAM en todos los programas, e independientemente de que modelo ejecutásemos, obteníamos siempre los mismos resultados. Es por ello que, a modo de por lo menos confirmar nuestras sospechas, estudiamos desde el Monitor de Sistema los valores que iba tomando el proceso *Python* en función de el modelo empleado. Nuestras sospechas fueron confirmadas, obteniendo valores muy altos para *PositionalIndex*, seguido de cerca por *RAMIndex* y muy de lejos por *DiskIndex*, que gastaba menores cantidades.

En cuanto a los tiempos, donde hemos podido llevar a cabo una mucho mejor toma de datos, observamos los siguientes detalles: en primer lugar, como es de esperar, *DiskIndex* es mucho más lento en construcción, aunque no excesivamente, tan solo un 15 % más lento que *RAMIndex*. *PositionalIndex*, para nuestra sorpresa, resulta ser muy próximo a *RAMIndex*, con diferencias poco notables en construcción. En cuanto a tiempos de carga, sin embargo, es donde llega el momento de arrasar de *DiskIndex*. Al tener que cargar una cantidad minúscula de datos, en comparación con los otros dos, podemos observar como es mucho más rápido (del orden de casi 100 veces más rápido) que los otros dos. Cabe mencionar, sin embargo, que estos tiempos pueden resultar engañosos, pues luego en búsqueda *DiskIndex* tardará algo más que estos dos, aunque si implementamos una caché intermedia como la que mencionamos anteriormente, obtendremos unos muy buenos resultados.
Por último, en lo que respecta al espacio en disco, nos sorprendimos muchísimo con los resultados observados, puesto que eran menos en *DiskIndex* y *RAMIndex*. Utilizamos para esta medida la funcionalidad que se facilitaba en el programa *main*, con lo que intuimos que funcionaría adecuadamente. De esta forma, como nos resultaba fácil de suponer, *PositionalIndex* ocupa más que *RAMIndex* (es lógico, al almacenar más información). Sin embargo, *DiskIndex* no era, para nuestra sorpresa, el que más información almacenaba, sino todo lo contrario. Creemos que esto se puede deber a que, al almacenar la información y volver a cargarla de nuevo, usando *pickle* en lugar de almacenar los *postings* de manera algo más eficiente, es posible que demos lugar a datos de un mayor volumen. Hemos comprobado estas mediciones investigando manualmente en la carpeta *index* generada por el programa y confirmando que, efectivamente, ocupa menos la de *DiskIndex*.
De esta forma, *DiskIndex*, que inicialmente no nos convencía porque creíamos que sería mucho más lento, ha demostrado ofrecer muy grandes y llamativas ventajas en comparación con los otros dos.

9. Conclusiones

Esta práctica nos ha permitido ahondar mucho más en los distintos motores de búsqueda que hemos ido implementando, a fin de crear distintos buscadores, modelos e índices, que nos han permitido entender plenamente su funcionamiento y analizarlo de cerca. Concretamente, nos ha abierto los ojos en dos detalles muy concretos, y es que *DiskIndex* de primeras no parecía especialmente bueno, de hecho, nos generaba nuestras dudas, pero con la Caché intermedia planteada nos puede permitir un muy buen funcionamiento, y mejor organizado todavía más aún; y por otro lado, que la búsqueda proximal es excesivamente compleja en algún aspecto, pero nos permite destacar, como hemos visto en los *score*, documentos que pueden ser muy aptos para nuestra búsqueda, permitiéndonos trabajar esa proximidad de los datos.