Proyecto de prácticas:
Recuperador de Noticias

Naiara Pérez Leizaola
Kléber Zapata Zambrano
Noelia Saugar Villar
Rafael Estellés Tatay
Mario Díaz Acosta

1.Introducción

Este trabajo de hacer un compilador ha sido, en retrospectiva, complicado. En comparación con las prácticas previamente realizadas en clase, en nuestra opinión, el grado de dificultad y de tiempo empleado ha crecido considerablemente.

A continuación vamos a explicar el proyecto final de prácticas de la asignatura de Sistemas de Almacenamiento de Datos: El recuperador de noticias.

2. Funcionalidades básicas

2.1. Indexador

El método index_file es el encargado de indexar las noticias al ejecutar el documento SAR_Indexer.py. Primero hemos creado unas variables generales en el programa fuera del método index_file, en el init__, docid para guardar el valor del identificador de los documentos donde se encuentran las noticias y newid para guardar el valor del identificador de las noticias, ambos inicializados a 0. De vuelta al método index_file lo primero que hacemos tras abrir el archivo que se le pasa es añadir al hash de documentos en su correspondiente docid el nombre del archivo. Después lo que hace el método es recorrer la lista de noticias donde guarda en el hash de noticias(news) el docid y la posición de la noticia en la lista, mediante el método tokenize consigue los tokens(palabras) del contenido de la noticia y después recorre estos tokens. Al recorrerlos, comprueba si el token se encuentra ya en el index, en cuyo caso comprueba si la noticia se encuentra ya en el index del token en cuestión; en cuyo caso suma uno al valor, es decir, el número de veces que está el token en la noticia. Si la noticia no se encuentra en el index del token actual, la añadimos, poniendo a 1 el valor(el número de veces que está el token en la noticia). Si el token no se encuentra indexado, lo añadimos, y le agregamos la noticia y que esta tiene una vez el token en ella. Al terminar el recorrido de

tokens sumamos uno a la posición de la noticia y al newid ya que pasamos a la siguiente noticia. Al finalizar el recorrido de la lista de noticias sumamos uno al docid ya que la siguiente vez que llamemos al método el documento siguiente tendrá otro id.

2.2 Recuperador de noticias

El método solve_query es el encargado de procesar las consultas y además es necesario actualizarlo con el resto de implementaciones ya que modifican su comportamiento. En primer lugar, se procesa la consulta para que los paréntesis y los dos puntos se queden separados al realizar el split. Tras ello, pasamos a realizar las modificaciones necesarias a la consulta para que realice las implementaciones extra. Su funcionamiento básico finalmente consiste en encontrar un AND o un OR en la query y utilizar el método correspondiente para obtener la posting list conveniente. Sin embargo, por cuestión ya que encontramos otros errores que nos retrasaron dejamos los métodos sin testear y por eso aparecen comentados.

En un principio, íbamos a detectar previamente cada NOT como se muestra en la imagen y sustituirlo en la lista por la posting list reversa de la palabra que le sigue, quitando con pop esa posición, pero se acabó optando por una implementación más burda con un if ante cada and/or ya que implementamos un método más eficiente para recorrer las listas con not and y not or, método que a última hora ha sido borrado porque proporcionaba errores con las funcionalidades extra. Para hacerlo más eficiente intentamos implementar una versión del NOT_AND y NOT_OR que se encuentran en el código comentado.

```
if 'NOT' in q:
            number_of_nots = q.count('NOT')
            while number_of_nots > 0:
                position_of_not = q.index('NOT')
                if isinstance(q[position_of_not + 1], str):
                    q[position_of_not] = self.reverse_posting(
                        self.get_posting(query.pop(position_of_not + 1).lower()))
                else:
                    q[position_of_not] = self.reverse_posting(q.pop(position_of_not + 1))
                number_of_nots -= 1
```

A la hora de mostrar los resultados (**solve_and_show**), se obtienen los índices con **solve_query**. A continuación, se recorren las noticias recuperadas,

obteniendo su peso con Jaccard (su relevancia para la consulta) si la función ranking está activada.

Si snippet está activado, muestra los fragmentos de la noticia donde aparecen las palabras de la consulta.

Jaccard: Limpia el query y tokeniza tanto el query como la noticia. La división entre la longitud de la intersección y de la unión de ambos conjuntos aporta el score documento-query que buscamos

snippet: Se tokenizan tanto la query como la noticia. Recorriendo las palabras de la query, obtienes parte de la frase donde aparece cada una (en caso de que aparezcan). Dicha frase se consigue calculando la posición de la palabra en la noticia y obteniendo un array donde dicha palabra está en el centro (la frase donde aparece)

3.Funcionalidades ampliadas

3.1. Stemming

Primero se declara un vector multifield que, si se realiza la ampliación de multicampo, vale ['title', 'date', 'keywords', 'article', 'summary']. En el caso obligatorio, solo vale ['article'].

```
for field in multifield:
    # Se aplica stemming a cada token del self.index[fielc
    # En este caso solo se guarda la noticia, no la posici
    for token in self.index[field].keys():
        token_s = self.stemmer.stem(token)
        if token_s not in self.sindex[field]:
            self.sindex[field][token_s] = [token]
        else:
            if token not in self.sindex[field][token_s]:
                self.sindex[field][token_s] += [token]
```

En cada campo, se sacan los tokens y se les aplica stemming con self.stemmer. Después añadimos cada token al que le hemos hecho stemming en su correspondiente campo en el índice invertido para stems ‘sindex’.

Para el get_stemming, simplemente se recolectan todos los tokens en el stem correspondiente (sacado previamente con stem = self.stemmer.stem(term)) en un vector res.

```
res = []
if (stem in self.sindex[field]):
    for token in self.sindex[field][stem]:
        res = self.or_posting(res, list(self.index[field][token].keys()))
return res
```

3.2 Multifield

La funcionalidad multifield consiste en la creación de nuevos índices para indexar otros campos de la noticia como el título o la fecha (antes solo se indexaba el artículo).En primer lugar, se crea un segundo nivel de hashing en el diccionario **self.index** de tal forma que, por ejemplo, self.index[‘date’] es el índice invertido del field ‘date’.

Métodos como **solve_query** deben modificarse. En este caso, se buscarán los ‘:’ de la query (field : term) para poder obtener el posting correcto.

También se debe modificar el **snippet,** sustituyendo ‘:’ por una palabra inventada (en nuestro caso ‘AAABBBCCC’), para poder extraer el field y término que necesitamos buscar.

En general, Multifield fue relativamente sencillo de implementar, teniendo en cuenta que es un ejercicio opcional.

3.3. Búsquedas posicionales

Para implementar las búsquedas posicionales primero hemos tenido que editar el método index_file, donde hemos creado una variable pos_tokens inicializada a 0 en cada noticia, que va creciendo de uno en uno por cada token observado. Para implementarlo en el index, cuando vamos a añadir un nuevo token, una nueva noticia o editar el número de veces que aparece un token en una noticia, en lugar de utilizar un número que vaya sumando de uno en uno hemos utilizado una lista que vaya guardando las posiciones de pos_tokens.

Para poder hacer las búsquedas de manera que se pueda hacer una query de varios tokens seguidos hemos implementado el método get_positionals que recibe una lista de términos. Primero vemos que el primer término se encuentre indexado y en ese caso recorremos las posting list de este. De cada posting list guardamos el id de la noticia y la lista de posiciones del término en la noticia,

la cual nos recorremos. Con cada posición vamos comprobando si el resto de términos se encuentran seguidos al término anterior utilizando el booleano corrido para ver si comparten la noticia, y si la posición del término actual es la siguiente a la del anterior. Si al terminar de recorrer todos los términos este booleano sigue siendo verdadero añadimos la noticia a la lista de noticias que vamos a devolver.

En el solve query para poder buscar términos seguidos comprobamos mediante un while si hay términos seguidos y los guardamos en una lista con la que llamamos al método get_positionals en lugar de al get_posting para resolver la query correctamente.

3.4 Ranking

Para su implementación, se utilizó el método del cálculo del peso Jaccard de las noticias. Para poder ser implementada, primero se tuvo que quitar de la consulta los términos como AND, NOT u OR, que no se encuentran presentes para ser puntuados y se inicializan los vectores que almacenarán el resultado.

```
query = query.replace('AND', '')
query = query.replace('OR', '')
query = query.replace('NOT ', 'NOT')
query = query.replace(':', ' ')
res = []
resul = []
```

Luego, a partir del resultado de la consulta proporcionado, se carga cada noticia con la orden json.load. Aquí, podemos calcular el resultado del peso de cada noticia usando self.jaccard(aux, query). Como esto se realiza dentro de un bucle, se crea un vector de tuplas donde cada noticia se guarda con su valor. La ventaja de realizarlo de esta forma es que ese vector se puede ordenar en base al peso.

```
res.sort(key=lambda tup: tup[1], reverse=True)
for i in res:
    resul.append(i[0])
return resul
```

Por último, se guardan sólo los índices de las noticias (la posición 0) con ayuda de otro vector. Este es el vector que se devuelve.

3.5 Permuterm

El índice permuterm se ha decidido implementar utilizando un diccionario, el cual su profundidad dependerá de si está activo o no el

multifield. El algoritmo es muy simple, este recorre términos para cada *field* realiza su permutación y la almacena en el índice de permuterms (*ptindex*).

Por otro lado, el método el método *get_permuterm()* se utiliza para resolver las query, diferenciando si el comodín utilizado es un '*' o '?' y utilizando el índice de permuterm. La principal diferencia en ambos casos es que con el símbolo, '?' la palabra devuelta debe tener la misma longitud que la wildquery.

Para finalizar, hemos añadido las modificaciones pertinentes en el código para su correcto funcionamiento.

3.6 Paréntesis

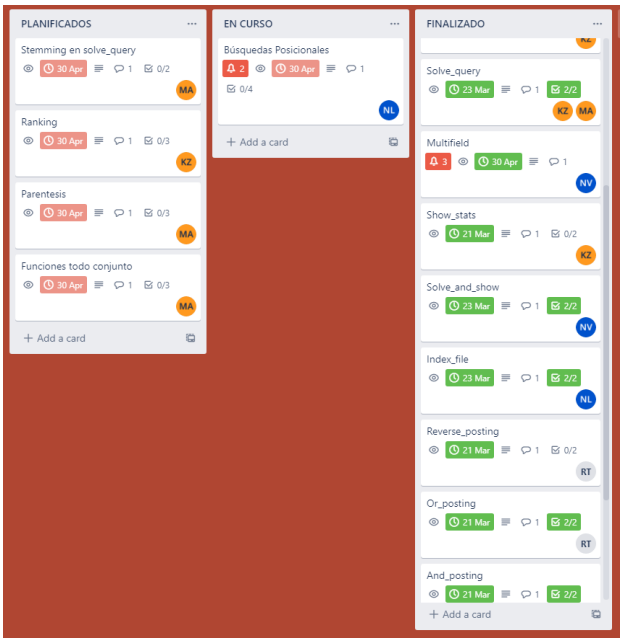
Nuestro método resuelve recursivamente los paréntesis, llamando a solve_query dentro de este, con la query contenida dentro del paréntesis, y eliminando de la query inicial todas las posiciones que había dentro para continuar.

3.7 Funcionamiento conjunto

No hay problemas para ejecutar nuestras funcionalidades en conjunto. El permuter y el stemming se hacen en la función get_posts, la cual no llamamos al usar búsquedas posicionales.

4. Métodos de organización

Para la organización del proyecto hemos utilizado la herramienta de Trello, la cual nos ha permitido que el líder del grupo pudiese controlar el trabajo que cada integrante iba realizando a lo largo del proyecto. Además, la herramienta permite poner fechas de entrega para cada apartado y conocer la distribución de los métodos y así conocer qué miembro había implementado un método en concreto.



Por otro lado, se decidió utilizar un control manual de las versiones. Esta decisión ha sido la peor de todas las que tomamos ya que ha provocado un caos en las versiones, lo que ha significado pérdidas de tiempo durante el desarrollo. Intentamos utilizar una herramienta para escribir varios a la vez pero el líder no sabia compartir sin estar el en el doc

Para finalizar, la comunicación interna y el control de versiones se utilizó la app de *WhatsApp*. Además, hemos decido utilizar en algunas sesiones la extensión de *Live Share* de Visual Studio Code.

5. Contribución

El equipo en su totalidad ha intentado distribuir de manera equitativa el trabajo. Sin embargo, en todos los puntos ha sido necesaria la colaboración y apoyo de diferentes miembros para abordar un mismo problema.