Universidad Politécnica de Valencia

ETSINF

Buscador Web

Asignatura:

Sistemas de Almacenamiento y Recuperación de Información

Autores:

Victor Merino Caballé

Javier Branchadell Allepuz

Francisco Javier Jiménez Ríos

Profesor:
Vicent Ahuir Esteve

Valencia, 20 de mayo de 2024

Índice

1 Equipo de Trabajo

1.1. Enumeración de los miembros del grupo

N°	Nombre
1	Victor Merino Caballé
2	Javier Branchadell Allepuz
3	Francisco Javier Jiménez Ríos

1.2. Descripción de la contribución de cada miembro al proyecto

Nombre	Contribución
Victor Merino Caballé	(33.3 %)
Javier Branchadell Allepuz	(33.3 %)
Francisco Javier Jiménez Ríos	(33.3 %)

2 Enumeración de las funcionalidades extra implementadas y descripción de su implementación

2.1. Archivo `SAR_lib.py`

- **index_dir**: Este método recorre de manera recursiva el directorio con los artículos en formato JSON y los envía al método `index_file` para su indexación. Para su implementación, mas allá de lo que ya se nos proporcionaba solo nos hemos asegurado de que se hago la indexacion con steaming y con permuterm.self.make.permuterm y self.use.steaming eran las varibales booleanas proporcionadas que nos permitian saber si estas opciones estaban activadas o no.

- **index_file**: Procesa cada archivo JSON, tokeniza su contenido y añade los términos al índice invertido. Para su implementación, leemos cada línea del archivo JSON que contiene un artículo. Usamos un tokenizer para descomponer el texto en palabras individuales, eliminando caracteres no alfanuméricos y normalizando las palabras a minúsculas. Cada término se inserta en el índice invertido, donde la clave es el término y el valor es una lista de identificadores de documentos donde aparece ese término.

- **make_stemming**: Implementa la funcionalidad de stemming, que reduce las palabras a su raíz morfológica. Utilizamos el algoritmo de Porter Stemmer adaptado para el español (proporcionado). Durante la indexación, cada palabra es transformada a su forma base (stem) y se almacena en un índice de stems. Esto permite que las búsquedas no dependan de la forma exacta de las palabras, devolviendo resultados relevantes incluso cuando los términos de búsqueda sufren alguna variacion. Usamos para ello el diccionario proporcionado stem.

- **make_permuterm**: Genera el índice permuterm, que facilita la búsqueda con comodines (`*` y `?`). Para esto, rotamos cada palabra de manera sistemática y las almacenamos en un índice especial. Por ejemplo, la palabra "casa" se almacena como "casa$", "asa$c", "sa$ca", y "a$cas". Este índice permite realizar búsquedas con comodines de manera eficiente: una búsqueda por "ca*a" puede ser transformada internamente a una búsqueda por "adollarca" en el índice permuterm. Implementamos este método para asegurar que las búsquedas con comodines sean rápidas y precisas, evitando la necesidad de realizar búsquedas exhaustivas sobre todos los términos posibles. Igual que en el caso del metodo anteripr tambien usamos el metodo específico para ello que es ptindex.

- **show_stats**: Muestra estadísticas del proceso de indexación, incluyendo el número de archivos y artículos indexados, así como la cantidad de tokens, stems y permuterms generados. Esta funcionalidad se implementó utilizando contadores que se actualizan durante el proceso de indexación, nos ha sido util para comprobar que se hacian los

procesos de indexacion de forma oportuna. Como en otros métodos tambien nos hemos apoyado en las variables self.multifield y self.use.steaming, para saber si dichas funcionalidadesde estaban activadas.

- **solve_query**: Este método resuelve consultas complejas permitiendo el uso de conectivas AND, OR y NOT, así como paréntesis para modificar la precedencia. La implementación de este método sigue varios pasos:

  Primero, la consulta se convierte en una lista de tokens mediante la función `split()`, separando cada término y operador:

  ```
  tokens = query.split()
  ```

  Luego, se convierte la lista de tokens de notación infija a notación postfija. Empleamos una pila para manejar los operadores y asegurar que se respeten las precedencias y asociaciones de los operadores ya que de otros formas nos surgian conflictos en algunas ocasiones. Se itera sobre cada token y vamos decidiendo en funcion del mismo:

  ```
  output = []
  stack = []
  for token in tokens:
      if token in ('AND', 'OR', 'NOT'):
          while (stack and stack[-1] != '(' and
                 precedence(stack[-1]) >= precedence(token)):
              output.append(stack.pop())
          stack.append(token)
      elif token == '(':
          stack.append(token)
      elif token == ')':
          while stack and stack[-1] != '(':
              output.append(stack.pop())
          stack.pop()  # Remove '(' from stack
      else:
          output.append(token)
  while stack:
      output.append(stack.pop())
  ```

  Una vez obtenida la expresión en notación postfija, se evalúa utilizando una pila, donde se realizan las operaciones lógicas (metodos explicados abajo) de acuerdo con los operadores encontrados y volvemos a guardar ese resultado para aplicar la siguiente operación sobre el mismo cuando proceda:

```
stack = []
for token in output:
    if token in ('AND', 'OR', 'NOT'):
        if token == 'AND':
            right = stack.pop()
            left = stack.pop()
            stack.append(and_posting(left, right))
        elif token == 'OR':
            right = stack.pop()
            left = stack.pop()
            stack.append(or_posting(left, right))
        elif token == 'NOT':
            operand = stack.pop()
            stack.append(not_posting(operand))
    else:
        stack.append(get_posting(token))
return stack[0]
```

- **precedence**: Este método determina la precedencia de los operadores lógicos en las consultas. Asigna un valor de precedencia a cada operador. Este metodo que hemos decidido implementar de esta forma porque vimos que usar numeros y crear un metodo era mas limpio que utilizar while y hacerlo dentro del propio codigo. Pues este es probablemente el metodo mas elaborado y de la otra forma cuando teniamos un fallo nos era mas complejo de detectar. Pues es más fácil siempre depurar con codigo mas limpio y legible:

```
def precedence(op):
    if op == 'NOT':
        return 3
    elif op == 'AND':
        return 2
    elif op == 'OR':
        return 1
    else:
        return 0
```

La precedencia asignada es:

- AND y NOT tienen la mayor precedencia, asegurando que estas operaciones se evalúen primero.

- OR tiene una precedencia menor.
- Los paréntesis se utilizan para forzar la precedencia deseada en las consultas es decir que todo lo que está dentro se evaluara antes.

- **evaluate_postfix**: Este método evalúa expresiones en notación postfija, también conocida como notación polaca inversa, segun hemos podido leer en diversas fuentes que consultamos, usaban esta como solución factible para este problema, pues cualquier otra que probabamos era código excesivamente enrevesado y poco eficiente. En un inicio, nuestra forma de gestión de cola era similar y lo único que habían eran más condicionales dejando muchísmas lineas de codigo. El método sigue varios pasos:

```
def evaluate_postfix(postfix_tokens):
    stack = []
    for token in postfix_tokens:
        if token in ('AND', 'OR', 'NOT'):
            if token == 'AND':
                right = stack.pop()
                left = stack.pop()
                result = and_posting(left, right)
            elif token == 'OR':
                right = stack.pop()
                left = stack.pop()
                result = or_posting(left, right)
            elif token == 'NOT':
                operand = stack.pop()
                result = not_posting(operand)
            stack.append(result)
        else:
            stack.append(get_posting(token))
    return stack[0]
```

Se inicializa una pila vacía y se itera sobre cada token en la expresión postfija:

- Si el token es un operando (término de búsqueda), se empuja en la pila.
- Si el token es un operador (AND, OR, NOT), se extraen los operandos necesarios de la pila, se aplica el operador y el resultado se vuelve a empujar en la pila.
- Al final de la iteración, la pila contendrá el resultado final de la consulta.

Este método garantiza que las consultas se evalúen correctamente, respetando la precedencia de los operadores y aplicando las operaciones lógicas necesarias.

- **get_posting**: Obtiene la lista de postings asociada a un término. Es un método muy basico pero vital ya que es el que nos pemite recuperar las lista de artids. Lo hacemos muy básico recuperando del diccionario que durante la indexación habíamos almacenado.

- **get_stemming**: Obtiene la lista de postings asociada al stem de un término, permitiendo búsquedas más generales y flexibles. Esta funcionalidad se integra con el índice de stems creado por `make_stemming`. Es un metodo bastante sencillo ya que lo unico que hace es buscar en el diccionario del indice de stemaing y luego una vez tenemos la palabra asociada en el stem. Una vez recuperado se busca dicho palabra en el indice general para sacar el posting list.

- **get_permuterm**: Obtiene la lista de postings utilizando el índice permuterm, permitiendo realizar búsquedas con comodines de manera eficiente. La implementación busca las rotaciones de la palabra en el índice permuterm y combina los resultados. La dorma en la que lo hacemso es comproabando el tipo de elemento que viene faltando en la palabra si el asterisco o si es el interrogante. Esto hace variar mucho la programación porque de una forma el numero de letras que faltan es mayor a 1 y con el interrogante solo puede faltar una letra. Una vez dividido miramos que el permuterm de esa palabra empiece y acabe por la división que hemos hecho. una vez recuperada la palabra la buscamos y podemos devolver su posting list.

- **reverse_posting**: Invierte una lista de postings, útil para implementar la operación lógica NOT. Esta función crea una nueva lista de postings que incluye todos los documentos que no están en la lista original. Implementación rápida haciendo una resta de las claves que estan en nuestra posting list proporcionado y se lo restamos todas. Esta diferencia nos deja las que no estan en nuestra lista. SOlo nos queda devolver esto.

- **and_posting**: Realiza la intersección (AND) de dos listas de postings, devolviendo documentos que contienen ambos términos. Utilizamos un algoritmo de fusión lineal para combinar las listas de manera eficiente. Usamos la implementacion vista en teoria asegurandonos que las listas que pasamos están en orden.

- **or_posting**: Realiza la unión (OR) de dos listas de postings, devolviendo documentos que contienen al menos uno de los términos. Este método también utiliza un algoritmo de fusión para combinar las listas de postings. Este lo hemos implementado igual que el de antes.

- **solve_and_show**: El método `solve_and_show` resuelve una consulta y muestra los resultados, incluyendo un fragmento (snippet) del contenido si está habilitado. Primero, obtiene y cuenta los resultados. Si el número de resultados excede un límite y la opción para mostrar todos no está activada, muestra solo los primeros resultados. Para cada resultado, obtiene el archivo JSON correspondiente y extrae el artículo. Si `self.show_snippet` está habilitado, se analiza el contenido del artículo, se tokeniza, y se buscan términos relevantes, generando un snippet que muestra 5 palabras antes

y después del término. Este snippet se muestra junto con la información del artículo, proporcionando una vista previa concisa y relevante. Además, el método tiene en cuenta el tipo de sistema operativo para manejar correctamente los directorios y rutas de archivos, asegurando compatibilidad tanto en sistemas Unix como en Windows.

Manejo de snippets:

```
def solve_and_show(self, query: str):
    [...]
        if self.show_snippet:
            print(f"Snippet: ")
            file = jsonFile[docid]
            if sys.platform == 'darwin' or sys.platform == 'linux':
                file = file.replace('\\', '/')
            for i, line in enumerate(open(file)):
                if i == int(art_id):
                    j = self.parse_article(line)
            terms = []
            for i in self.terms:
                term, field = i
                terms.append(term)
            body = self.tokenize(j['all'])
            visited = []
            for i, token in enumerate(body):
                if token in terms and token not in visited:
                    if i < 5:
                        snip_list = body[:i + 5]
                        snip = ' '.join(snip_list)
                        print(f"{snip}... ", end='')
                    elif i > len(body) - 5:
                        snip_list = body[i - 5:]
                        snip = ' '.join(snip_list)
                        print(f"...{snip} ", end='')
                    else:
                        snip_list = body[i - 5:i + 5]
                        snip = ' '.join(snip_list)
                        print(f"...{snip}... ", end='')
                    for word in snip_list:
                        if word in terms:
                            visited.append(word)
            print("\n")
    [...]
```

2.2. Archivo `SAR_Crawler_lib.py`

- **parse_wikipedia_textual_content**: Realiza el análisis del contenido textual de una entrada de Wikipedia y devuelve un diccionario con la estructura deseada (url, title, summary, sections). Para implementar esta función, utilizamos expresiones regulares que identifican y extraen el título, resumen, secciones y subsecciones del artículo. El texto bruto se divide usando patrones específicos para los títulos y secciones, asegurando que cada parte del contenido se almacene correctamente en el diccionario. Lo unico que tuvimos que hacer es entender bien como funcionaban los patrones que ya se nos porporcionaban en el codigo, e ir gestionando con los mismo de forma correcta la entrada. Este diccionario nos será util posteriormente para estructurar el contenido de manera coherente. De no estar correctamente implementado nos fallaria la indexación, ya que depende completamente de lo que aquí generemos.

- **start_crawling**: Realiza el crawling utilizando una lista de URLs iniciales y controla el proceso de captura de los artículos, gestionando la cola de prioridad de URLs y el almacenamiento de documentos. EL inicio del codigo se nos proporcinaba, por ello solo faltaba la lógica principal del crawling. Primero, inicializamos un conjunto para las URLs visitadas y una cola de prioridad para las URLs pendientes, donde cada entrada incluye la profundidad del crawling. Mientras la cola no esté vacía y el límite de documentos no se haya alcanzado, se extrae una URL de la cola, se verifica si ha sido visitada y si la profundidad es adecuada. Luego, se descarga el contenido de la página y se parsea utilizando `parse_wikipedia_textual_content`. Si el artículo es válido, se añade a la lista de documentos capturados. Las nuevas URLs extraídas del artículo se añaden a la cola de prioridad si no han sido visitadas. Finalmente, los documentos se guardan en lotes definidos por el tamaño de batch, y se sigue este proceso hasta cumplir las condiciones de parada definidas.

3 Descripción y justificación de las decisiones de implementación realizadas

3.1. Permuterm

Como ya hemos explicado en las descripciones de los métodos del punto 2, el permuterm hace uso de los métodos propuestos make_permuterm y get_permuterm. El primero de ellos se implementa tomando cada palabra del índice y generando cada una de sus permutaciones posibles, según lo que aprendimos en las clases de teoría. A partir de ahí, se guarda de forma adecuada en el índice que nos proporcionaban también con dicho fin. La recuperación sí que nos supuso un poco más de desafío. Nuestro primer objetivo era separar la palabra convenientemente para poder realizar la búsqueda, como también vimos en las clases de teoría.

Luego consideramos dos posibles opciones. La primera era que la palabra fuera incompleta pero que el comodín fuera '*'; la segunda era que el comodín fuera '?'. La diferencia radica en que en el primer caso el número de letras faltantes es variable, mientras que en el segundo solo falta 1 letra. Esto lo solucionamos guardando qué tipo de comodín se usaba y, a partir de ahí, en el primer caso, guardar todas las palabras recuperadas y, en el segundo, solo las que tuvieran la misma longitud inicial (ya que en la inicial faltaba una letra pero era sustituida por el interrogante).

3.2. Steamming

El apartado de Stemming no nos ha supuesto muchos problemas. Con el método que realizaba la stematización, llevamos a cabo la stematización de cada término y, al igual que en el método Permuterm o en la indexación general, guardamos la stematización del término junto con el término de forma conveniente en el diccionario que también se nos proporcionaba para ello.

La recuperación tampoco fue demasiado compleja, ya que al estar todo bien indexado solo necesitábamos buscar en el índice la stematización y, una vez recuperado el término asociado, buscar su posting list.

3.3. Multifield

Para la realización del multifield hemos hecho lo siguiente: Si la opcion multifield esta activada a la hora de indexar indexamos teniendo en cuenta todods posibles. SI esta no esta activada indexamos teneiendo en cuenta que solo hay un field y que este es all. Tomamos esta decision para que luego el diccionario tuviese siempre la misma forma y el primer parametro siempre fuese field, de lo contrario en el caso de multifield no estar activo el primer campo no hubiera hecho falta y hubiera sido mas compleja la implementación de otros metodos. A la hora de hacer una busqueda, esto no supone un problema, porque buscamos si la query se ha hecho usando multifield o no. SI se ha hecho usandolo, nos quedamos con el parametro field y lo utilizamos en la recuperacion de la posting list asociada a dicho termino (o pareja de términos) si no lo hay para generalizar, lñe asignamos nostros el field all. Con el objetivo de seguir la lógica antes descrita.

3.4. Paréntesis

Esta funcionalidad lo que realiza es dividir la consulta según las prioridades de los paréntesis. Durante el recorrido de cada token en la consulta, verificamos si es un operador lógico (AND, OR, NOT). En caso afirmativo, procesamos su posición en la cola de salida considerando su precedencia respecto a los operadores presentes en la pila. Si el token es un paréntesis de apertura, lo añadimos directamente a la pila de operadores. Si es un paréntesis de cierre, desplazamos los operadores de la pila a la cola de salida hasta encontrar un paréntesis de apertura correspondiente, eliminándolo posteriormente de la pila. Cuando el token es un término de búsqueda, lo procesamos para identificar tanto el campo como el término en sí, y lo añadimos a la cola de salida. Este enfoque asegura que los paréntesis se gestionen correctamente, respetando la precedencia de los operadores y agrupando los términos de búsqueda según corresponda.

3.5. Snippet

A la hora de realizar el snippet, tenemos en cuenta que queremos recuperar las zonas en las que aparecen las palabras buscadas. Sin embargo, dado que una palabra puede aparecer muchas veces a lo largo de un documento, hemos tomado la decisión de mostrar solo una vez cada una de las palabras buscadas en la consulta. Las zonas que aparecerán serán 5 palabras a la izquierda y 5 palabras a la derecha, según nuestra opinión lo necesario para que esté contextualizado. Si la consulta es muy compleja, es decir, más de 4 términos en la consulta, no mostraremos snippet.

3.6. Posicional

Finalmente, la búsqueda posicional no la hemos implementado. Como ya le comentamos a nuestro profesor de prácticas, hemos tomado esta decisión debido a que hicimos el resto de opciones sin tener en cuenta a esta. Cuando decidimos ponernos a implementarla, la transformación de nuestro índice invertido a un índice invertido posicional suponía realizar muchas modificaciones al resto de métodos para garantizar el correcto funcionamiento del programa. Valoramos la opción de generar un diccionario aparte solo con el fin de tener el índice posicional. No obstante, esto luego supuso, a la hora de recuperar, realizar diversos cambios que ya nos suponían de nuevo una cantidad muy grande de horas de pruebas. Quizás tendríamos que haber realizado el trabajo desde un inicio teniendo en cuenta que queríamos hacer todas las funcionalidades extra, ya que este era nuestro objetivo, pero al ir implementándolas paso a paso, la carga de trabajo supuso que desistieramos de implementar esta funcionalidad.

4 Descripción del método de coordinación utilizado por los miembros del grupo en la realización del proyecto

Para la realización del proyecto, se utilizó una metodología de trabajo en equipo con reuniones semanales para coordinar las tareas y revisar el progreso. Cada miembro del grupo se encargó de una parte específica del proyecto:

Nombre	**Contribución**
Victor Merino Caballé	Implementación del crawler y parsing de contenido
Javier Branchadell Allepuz	Desarrollo del indexador y tokenización, implementación de funcionalidades adicionales (stemming, permuterm)
Francisco Javier Jiménez Ríos	Pruebas, documentación, mostrar los resultados correctamente, snippets y encargado del controlador de versiones

Se utilizaron herramientas de control de versiones (Git) para integrar el trabajo de todos los miembros y asegurar que todos tuvieran acceso a la versión más reciente del código.

Contribución:

Método	**Realización**
index_dir	Victor Merino Caballé
index_file	Javier Jiménez
make_stemming	Javier Branchadell
make_permuterm	Javier Branchadell
show_stats	Javier Jiménez
solve_query	Javier Branchadell, Javier Jiménez
get_posting	Javier Jiménez
precedence	Javier Branchadell, Javier Jiménez
evaluate_postfix	Javier Branchadell, Javier Jiménez
get_stemming	Javier Branchadell
get_permuterm	Javier Branchadell
reverse_posting	Victor Merino Caballé
and_posting	Victor Merino Caballé
or_posting	Victor Merino Caballé
solve_and_show	Javier Jiménez
parse_wikipedia_textual_content	Victor Merino Caballé
start_crawling	Victor Merino Caballé

Cuadro 1: Métodos y sus respectivas realizaciones

5 Opiniones subjetivas y dificultades afrontadas

A lo largo de la memoria ya hemos ido incluyendo las diferentes dificualtades y desafíos que nos hemos ido encontrando. Por ello, solo nos queda hacer una breve conslusión.

En general, el proyecto fue una experiencia enriquecedora que nos permitió aplicar los conocimientos adquiridos en clase a un problema real. Aprendimos mucho sobre la importancia de la planificación y la división de tareas, así como la importacia de una buena comunicación y trabajo en equipo. Con ello hemos podido garantizar que pese a la división de tareas, la puesta en común del trabajo realizado por cada uno de los integrantes de este grupo nos ha sido muy sencilla.