

Proyecto Final de Sistemas de recuperación de la información

Fragas D.¹, Santos J.¹, and Villalobos K.¹

Facultad de Matemática y Computación, Universidad de La Habana, La Habana,
Cuba

Resumen Implementación de un sistema de recuperación de la información en Django usando los modelos booleano, vectorial e indexación semántica latente.

Keywords: recuperación información · modelo booleano · modelo vectorial · indexación semántica latente

1. Introducción

El proceso de búsqueda de la información es el conjunto de tareas mediante las cuales el usuario localiza y recupera información que es pertinente para satisfacer su necesidad de información o la resolución de un problema, y por cuyo motivo ha comenzado el proceso de búsqueda. En este sentido, recuperar información significa obtener una información que alguna vez ha sido producida por alguien [1]

Un proceso de recuperación de información comienza cuando un usuario hace una consulta al sistema. Una consulta a su vez es una afirmación formal de la necesidad de una información. En la recuperación de información una consulta no identifica únicamente a un objeto dentro de la colección. De hecho varios objetos pueden ser respuesta a una consulta con diferentes grados de relevancia.

Un objeto es una identidad que está representada por información en una base de datos. En dependencia de la aplicación estos objetos pueden ser archivos de texto, imágenes, audio, mapas, videos, etc. Para recuperar efectivamente los documentos relevantes por estrategias de recuperación de información, los documentos son transformados en una representación lógica de los mismos. Cada estrategia de recuperación incorpora un modelo específico para sus propósitos de representación de los documentos.

La mayoría de los sistemas de recuperación de información computan un ranking para saber cuán bien cada objeto responde a la consulta, ordenando los objetos de acuerdo a su valor de ranking. Los objetos con mayor ranking son mostrados a los usuarios y el proceso puede tener otras iteraciones si el usuario desea refinar su consulta.

En el presente trabajo se expone una implementación de un sistema de recuperación de la información, empleando tres modelos; el modelo booleano, el modelo vectorial y la indexación por semántica latente. Aunque dicho modelo

en principio tiene caracter educativo puede ser facilmente escalado a un sistema en producción.

2. Bases teóricas de los modelos

Un modelo de recuperación de información consiste en [2] :

D : Representación de los documentos

Q : Representación de las queries

F : Un marco (framework) de modelado para D y Q y las relaciones entre ellos.

$R(q, d_i)$: Un ranking o función de similaridad que ordene los documentos con respecto a una consulta dada

2.1. Modelo booleano

El modelo booleano[2] se basa en la teoría de conjuntos y el algebra booleana, solo considera que los términos indexados estén presentes o no en el documento. Los pesos de los terminos se encuentran limitados a los valores 0 y 1. Las consultas están formadas por términos indexados unidos por los operadores *and*, *or*, *not* y $()$ para agrupar.

Formalmente el modelo booleano se divide en:

- D : Conjunto de palabras del documento (Términos de indexación).

- Q : Expresion booleana (Operadores: *and*, *or* y *not*)

- F : Álgebra booleana sobre conjuntos de términos y documentos.

- R : Un documento es relevante para la query dada si satisface la expresión booleana de acuerdo al álgebra

2.2. Modelo Vectorial

El modelo vectorial[2] representa los documentos de la colección como una serie ordenada de números reales positivos. Con ello trata de reflejar la importancia de cada término en cada documento. Son dos los factores habitualmente considerados a la hora de imponer un peso a cada término en cada documento:

- *tf* (term frequency) representa la frecuencia del término en el documento.

- *idf* (inverse document frequency) representa el inverso del número de documentos de la colección en los que aparece el término. Es un número mayor cuanto menor es el número de documentos en los que aparece un término. Es un indicador, pues, de la especificidad del término.

La representación de una consulta Q en el modelo vectorial es la misma que un documento, esto es, una serie ordenada de números reales positivos. En el caso de las consultas, los números reflejan la importancia relativa de cada término en la necesidad informativa del usuario.

El proceso de recuperación consiste en los siguientes pasos:

- Hallar la descripción de los documentos de la colección conforme a un esquema de ponderación dado, *tf.idf*, por ejemplo.

- Hallar la similitud de cada documento de la colección con la consulta conforme a una función de similitud dada, el coseno del ángulo que forman, por ejemplo.
- Mostrar al usuario los documentos ordenados en orden decreciente de su similitud con la consulta.

2.3. Indexación semántica latente

La indexación semántica latente (LSI)[3] es un método de indexación y recuperación de la información que utiliza la descomposición en valores singulares (SVD) para identificar patrones en las relaciones entre los términos contenidos en una colección de documentos no estructurados.

El modelo se sustenta en el álgebra vectorial y, más particularmente, en las operaciones con matrices. La Descomposición en Valores Singulares hace uso de la multiplicación de matrices, la traspuesta y el cálculo de valores propios.

El objetivo fundamental de LSI es encontrar una matriz A_k , aproximada a A - matriz que relaciona los pesos de los términos con los documentos -, con rango k , mucho menor que el rango de A . En esta matriz reducida se puede recuperar información que no estaba disponible directamente en A , sino que estaba latente en ella.

1) Primero debemos hallar la descomposición en valores singulares de la matriz Términos-Documentos.

O sea, encontrar T , S y D tal que $A = T_{t \times m} * S_{m \times m} * D_{d \times m}^T$ y se cumple que:

- t es la cantidad de términos indexados, d es la cantidad de documentos.
- T es la matriz de vectores propios derivada de $A \times A^T$. Representa los términos en el espacio de términos.
- S es una matriz diagonal, donde $m = \min(t, d)$ y $S_{i,i}$ es la raíz de los valores propios para $1 \leq i \leq r$, r rango de A , y cero en otro caso.
- D es la matriz de vectores propios derivada de $A^T \times A$. Representa los documentos en el espacio de documentos.

2) De la matriz S se considera solamente los k mayores valores propios, reemplazando por cero los $r - k$ menores valores propios de la diagonal. Se mantienen las columnas correspondientes en T y D^T .

3) Por último se calcula $A_k = T * S_k * D^T$. Donde k , $k < r$ es la dimensión de un espacio reducido de conceptos. El parámetro k debe ser lo suficientemente grande para evitar que se escape información relevante, pero lo suficientemente pequeño para filtrar los detalles no relevantes [4]. (Se suele usar 150, 200 o 300) [5].

Para representar una consulta en el espacio LSI hay que transformar el vector de consulta q a ese espacio

$$q_k = S_k^{-1} * T_k^T q$$

3. Modelación del sistema

En el sistema implementado usamos diferentes objetos para representar las distintas entidades del proceso de recuperación de la información así como para almacenar las mismas en la base de datos.

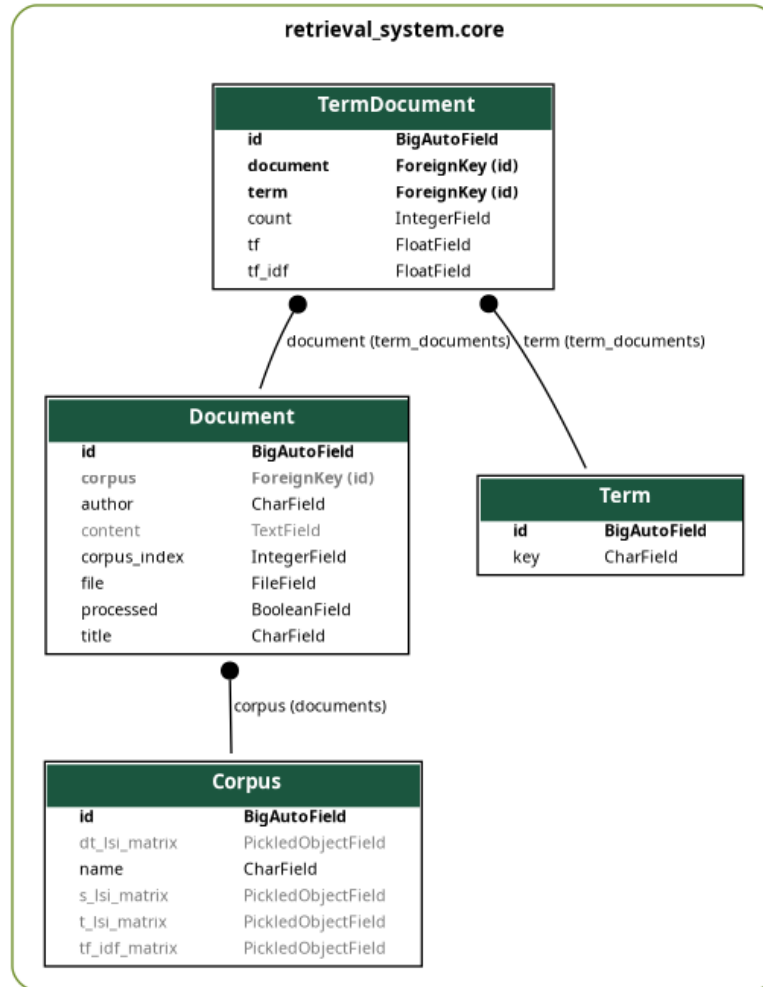


Figura 1. Relación de las entidades del sistema

Se tiene el objeto **Corpus**, que representa un conjunto de documentos, tiene un nombre para identificarlo y almacena las matrices de los pesos de los documentos y las utilizadas por el modelo LSI.

Se tienen el objeto Document para representar un documento, el mismo tiene los campos:

- title: Título del documento
- author: Autor del documento (Opcional)
- file: Copia del archivo al que refiere el documento
- corpus: Llave foránea al Corpus que pertenece el documento
- corpus_index: Índice del documento en el corpus (Se usa para evaluar el modelo)
- content: Contenido del documento, es el texto que se usa en el proceso de recuperación de la información.

Se tiene el objeto Term que representa un término. Posee el campo *key* con la representación textual del término.

Además se tiene el objeto TermDocument que representa una relación muchos a muchos entre Term y Document y tiene los campos *count*, para almacenar el número de ocurrencias del término en el documento, *tf*, para almacenar la frecuencia del término en el documento y el valor *tf-idf* para almacenar la métrica TF-IDF del término respecto al documento en el Corpus de dicho documento.

La relación de estos objetos se puede ver en la figura 1

4. Arquitectura del sistema

El proyecto está compuesto por una aplicación en Django que funciona de Backend, usa una base de datos relacional (sqlite3 en desarrollo y PostgreSQL en producción). Este backend expone una API para la realización de consultas, así como un sitio de administración que permite administrar los documentos y los corpus.

Se tiene además una aplicación cliente minimalista para la realización de consultas desarrollada en Angular.

Para correr el backend siga los siguientes pasos:

1- Instalar dependencias:

```
pip install -r requirements/local.txt
```

2- Ejecutar las migraciones para estructurar la base de datos:

```
python manage.py migrate
```

3- Montar el servidor:

```
python manage.py runserver
```

Opcionalmente puede crear un superusuario para acceder al sitio de administración en la ruta /admin.

```
python manage.py createsuperuser
```

Se tiene comandos definidos para cargar corpus de pruebas como los siguientes:

Cargar el corpus 'Cranfield'

```
python manage.py load_cranfield
```

Cargar el corpus 'Med'

```
python manage.py load_med
```

Después de cargar cada corpus deberá procesarse dicho corpus. El preprocesamiento de los corpus es una tarea relativamente costosa, pero que se ejecuta cada vez que un corpus cambie y garantiza poder hacer las consultas de forma más rápida. En la puesta en producción del sistema esta ha de ejecutarse como una tarea asíncrona al servidor que corra en otro proceso.

Para procesar un corpus de forma manual ejecute:

```
python manage.py process_corpus < corpusName >
```

donde 'corpusName' es el nombre del corpus ('cranfield', 'med') .

5. Implementación



```
1  def process_corpus(corpus: Corpus):
2      documents = corpus.documents.all()
3
4      for document in documents:
5          if not document.processed:
6              process_document(document)
7
8          log.info(f"Processed document {document.title}")
9
10     log.info("Calculating tf-idf")
11     corpus.tf_idf_matrix = calculate_tf_idf(corpus)
12
13     corpus.save()
14
15     log.info("Calculating LSI")
16     T, S, dt_lsi_matrix, documents = calculate_lsi(corpus)
17     corpus.t_lsi_matrix = T
18     corpus.s_lsi_matrix = S
19     corpus.dt_lsi_matrix = dt_lsi_matrix
20     corpus.save()
21
```

Figura 2. Código para procesar corpus

Procesar un Corpus (Fig 2) en nuestro sistema implica iterar por sus documentos y procesarlos (Fig 3).

Procesar un documento significa extraer cada término crear una instancia de Term si dicho término no se ha registrado. Además se crea una instancia de TermDocument con los valores de tf e idf correspondiente.

```
1  def process_document(document: Document):
2
3      document_tokens = tokenize_document(document.content)
4
5      max_tf = 0
6      for token in document_tokens:
7
8          term, _unused = Term.objects.get_or_create(key=token)
9
10         term_document, created = TermDocument.objects.get_or_create(
11             document=document, term=term
12         )
13
14         if created:
15             term_document.count = 1
16         else:
17             term_document.count += 1
18             term_document.tf = term_document.count / len(document_tokens)
19             term_document.save()
20             max_tf = max(max_tf, term_document.tf)
21
22     # Normalize frequencies
23     for term_document in document.term_documents.all():
24         term_document.tf = term_document.count / max_tf
25         term_document.save()
26
27     document.processed = True
28     document.save()
```

Figura 3. Código para procesar documento

```

1
2 def calculate_tf_idf_matrix(corpus):
3
4     all_terms = enumerate(
5         Term.objects.filter(term_documents__document__corpus=corpus).distinct()
6     )
7     all_terms_dict = {term: index for index, term in all_terms}
8
9     documents = corpus.documents.all()
10    all_documents = enumerate(documents)
11    all_documents_dict = {document: index for index, document in all_documents}
12
13    tf_idf_matrix = {}
14
15    for document in all_documents_dict.keys():
16        tf_idf_matrix[document] = np.zeros(len(all_terms_dict))
17        for term_document in document.term_documents.prefetch_related("term"):
18            tf_idf_matrix[document][
19                all_terms_dict[term_document.term]
20            ] = term_document.tf_idf
21        document.doc_vector = tf_idf_matrix[document]
22        document.save()
23        log.info(
24            f"CALCULATED TF-IDF VECTOR FOR DOCUMENT {document.id} in corpus {corpus.name}"
25        )
26
27    return np.stack([document.doc_vector for document in documents])
28

```

Figura 4. Calcular matrix TF-IDF

Por último se calcula la matriz TF-IDF (Fig 4) asociada al Corpus así como las matrices T, S y DT utilizadas por el modelo LSI (Fig 5).



```
1 def calculate_lsi(corpus: Corpus):
2
3     tfidf_matrix, documents = corpus.tf_idf_matrix, corpus.documents.all()
4
5     A = np.transpose(tfidf_matrix)
6
7     T, S, DT = np.linalg.svd(A, full_matrices=False)
8
9     minimum = min(np.shape(A))
10    k = int(minimum * 60 / 100)
11
12    # Truncate matrix
13    S[k:] = 0
14    S = np.diag(S)
15
16    S = np.linalg.inv(S)
17    DT = np.transpose(DT)
18
19    return T, S, DT, documents
20
```

Figura 5. Calcular matrices para LSI

6. Realización de Consultas

Para la realización de consultas se puede hacer peticiones GET al endpoint <http://127.0.0.1:8000/api/search/>

Este endpoint toma varios parametros de consulta:

- type: Modelo a usar. Posibles valores: boolean — vectorial[default] — lsi;
- corpus: Corpus sobre el cual hacer la consulta
- query: Consulta a realizar

Ejemplo:

- [http://127.0.0.1:8000/api/search/?type=boolean&corpus=cranfield&query=electron%20and%20distribution%20or%20\(%20thermodynamic%20and%20heat%20\)](http://127.0.0.1:8000/api/search/?type=boolean&corpus=cranfield&query=electron%20and%20distribution%20or%20(%20thermodynamic%20and%20heat%20))

- <http://127.0.0.1:8000/api/search/?type=vectorial&corpus=med&query=acid%20concentration%20in%20testosterone>

La aplicación cliente permite realizar estas consultas desde una interfaz visual intuitiva. Está desarrollada en Angular usando Angular Material.

Para ejecutarla moverse al directorio front-end:

1- Instalar dependencias:

```
npm install -g @angular/cli  
npm install
```

2- Montar servidor:

```
ng serve
```

La aplicación se estará sirviendo en el puerto 4200

6.1. Implementación

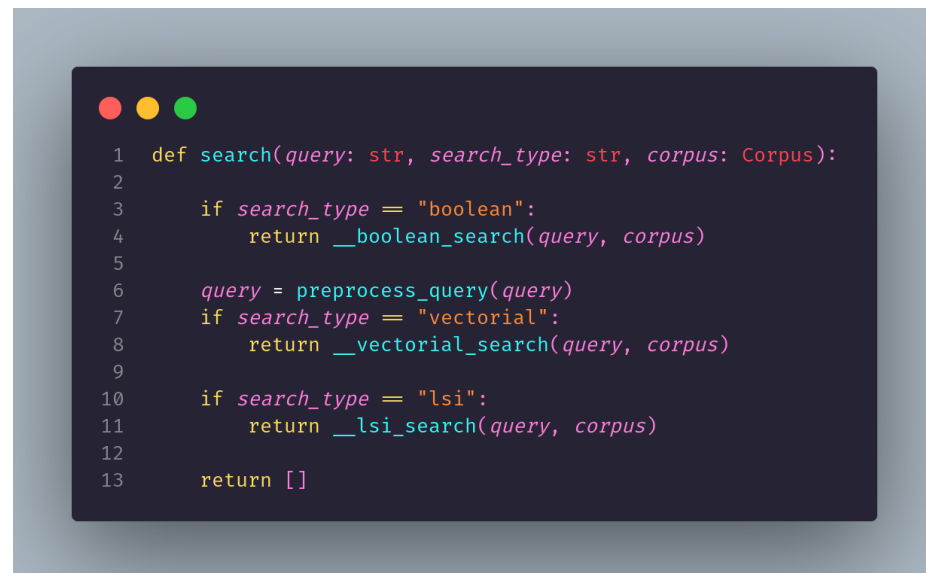


Figura 6. Realizar consultas

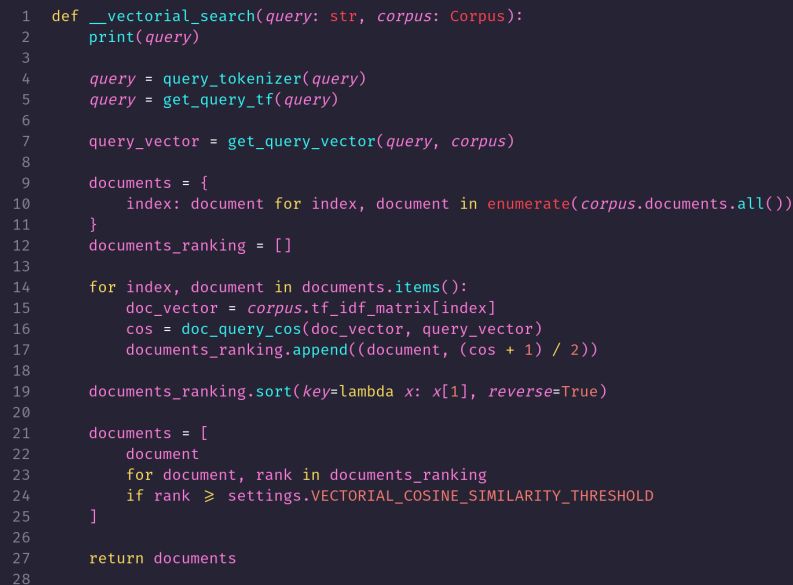
La realización de una consulta entra por la función *search* (Fig 6), la cual en dependencia del parámetro *search_type* delega la consulta a una función en dependencia del modelo.

Para las consultas basadas en el modelo booleano, estas se parsean usando la gramática:

```
E -> TX  
X -> or TX | epsilon
```

```
T -> FY
Y -> and FY | epsilon
F -> not Z | Z
Z -> (E) | b
```

y se evalúan mediante operaciones de conjuntos implementadas en los **QuerySet** de Django.



```
1 def __vectorial_search(query: str, corpus: Corpus):
2     print(query)
3
4     query = query_tokenizer(query)
5     query = get_query_tf(query)
6
7     query_vector = get_query_vector(query, corpus)
8
9     documents = {
10         index: document for index, document in enumerate(corpus.documents.all())
11     }
12     documents_ranking = []
13
14     for index, document in documents.items():
15         doc_vector = corpus.tf_idf_matrix[index]
16         cos = doc_query_cos(doc_vector, query_vector)
17         documents_ranking.append((document, (cos + 1) / 2))
18
19     documents_ranking.sort(key=lambda x: x[1], reverse=True)
20
21     documents = [
22         document
23         for document, rank in documents_ranking
24         if rank >= settings.VECTORIAL_COSINE_SIMILARITY_THRESHOLD
25     ]
26
27     return documents
28
```

Figura 7. Consultas con el modelo vectorial

Para el modelo vectorial (Fig 7) se transforma la consulta a su vector TF-IDF en el espacio de los documentos y se halla el ranking mediante el Coseno-Similitud. Se tiene un umbral configurable para determinar los documentos recuperados.

Para el modelo LSI, primeramente se transforma la consulta a su vector TF-IDF en el espacio de los documentos, luego se transforma al espacio de semántica latente y se halla el ranking por el Coseno-Similitud con el vector del documento en dicho espacio (Fig 8).

```

1  def evaluate(query, corpus: Corpus):
2
3      T, S, DT, documents = (
4          corpus.t_lsi_matrix,
5          corpus.s_lsi_matrix,
6          corpus.dt_lsi_matrix,
7          corpus.documents.all(),
8      )
9
10     T_dot_S_transpose = np.dot(T, S).T
11
12     query_vec = np.transpose(np.dot(T_dot_S_transpose, np.transpose(query)))
13
14     document_ranking = []
15
16     for doc in range(np.shape(DT)[0]):
17         doc_vector = DT[doc]
18         doc_query_cos = np.dot(query_vec, doc_vector) / (
19             norm(query_vec) * norm(doc_vector)
20         )
21         document_ranking.append((documents[doc], (doc_query_cos + 1 / 2)))
22
23     document_ranking.sort(key=lambda x: x[1], reverse=True)
24     return document_ranking
25

```

Figura 8. Consultas con LSI

7. Evaluación de los modelos

Se implementaron las medidas de precisión, recobrado y f1 para la evaluación de los modelos. Existe un comando para realizar dicha evaluación sobre un corpus.

```
python manage.py evaluate_model corpus model query rel measure
```

Donde:

- corpus: nombre del corpus que se utiliza para la evaluación
- model: modelo a evaluar ¡'boolean' — 'vectorial' — 'lsi' ¡
- query: dirección del fichero con las consultas
- rel: dirección del fichero con los documentos relevantes por consultas
- measure: medida a usar para la evaluación ¡'precision' — 'recall' — 'f1' ¡

Los ficheros query y rel se esperan en formato json con la siguiente estructura:

- query:

```
{
  "1": {
    "id": "1",
    "text": "what similarity laws must be obeyed when constructing ..."
  },
  "2": {
    "id": "2",
    "text": "what are the structural and ..."
  },
  ....
}
```

Donde cada elemento del diccionario raíz es una consulta siendo el campo *text* la consulta en sí.

- rel:

```
{
  "1": {
    "13": {
      "ground_truth": "1"
    },
    "14": {
      "ground_truth": "1"
    },
    "15": {
      "ground_truth": "1"
    },
    ...
  },
}
```

Donde cada elemento del diccionario raíz tiene como llave la consulta y es un diccionario cuyas llaves representan los documentos relevantes dentro del corpus.

7.1. Resultados

Se evaluaron los corpus *Med* y *Cranfield* con las métricas precisión, recobrado y f1.

	Precisión	Recobrado	F1
Cranfield	0.262	0.277	0.219
Med	0.605	0.2	0.253

Cuadro 1. Evaluación del modelo vectorial

	Precisión	Recobrado	F1
Cranfield	0.136	0.435	0.190
Med	0.506	0.529	0.493

Cuadro 2. Evaluación del modelo LSI

8. Ventajas y desventajas del sistema

Nuestro sistema tiene como ventaja el poder aprovechar los cálculos de procesamiento almacenados en una base de datos para la realización más efectiva de las consultas. Usar una arquitectura Cliente-Servidor permite su despliegue Web y ser usado desde cualquier dispositivo. Además sirve para administrar los documentos de un sistema.

Otra ventaja importante es la genericidad de los documentos y la posible extensión mediante plugins, ejemplo un crawler podría subir los documentos visitados al sistema o se podrían subir de forma manual.

Cabe destacar que el sistema presenta una gran desventaja, el procesamiento de los documentos, al necesitar de lecturas y escrituras en una base de datos puede ser enormemente costoso, por lo que se plantea que en entornos de producción dichas tareas corran de forma asíncronas y solo cuando sean totalmente necesarias. Por ejemplo si se tiene un crawler añadiendo documentos constantemente procesar el conjunto de documentos después de subir cierta cantidad y no cuando se añada uno a uno.

Referencias

1. Abadal E, Codina LL. Bases de Datos Documentales: Características, funciones y método. Madrid: Síntesis; 2005. p. 29-92.
2. Fleitas C., Sánchez M.: C2: MODELOS DE RECUPERACIÓN DE INFORMACIÓN: BOOLEANO Y VECTORIAL. 3er Año Licenciatura en Ciencia de la Computación Departamento De Programación Facultad De Matemática Y Computación Universidad De La Habana, 2022
3. Pazo, A., González, N.: Indexación Semántica Latente. Un Modelo de Recuperación De Información. 2022
4. Baeza-Yates R. y Ribeiro-Neto B.: Modern Information Retrieval
5. Manning C. D., Raghavan P. y Schütze H., An Introduction to Information Retrieval, Cambridge University Press, 2009.