EDA FINAL

Nicolás Margenat

2022 1Q

Contents

1 Complejidades

LEVENSHTEIN:

- *Temporal*: $O(n * m)$
- *Espacial*: $O(n * m)$

donde n y m son las longitdes del $str1$ y $str2$ respectivamente.

KMP:

- *Temporal*: $O(m)$
- *Espacial*: $O(m)$

donde m es la longitud del query.

BINARY SEARCH:

- *Temporal*: $O(log_2(N))$ (recursiva e iterativa)
- *Espacial*: $O(log_2(N))$ (recursiva) y $O(1)$ (iterativa)

QUICKSORT:

- *Temporal*: $O(N^2)$
- *Espacial*: $O(N)$

MERGESORT:

- *Temporal*: $O(N * log_2(N))$
- *Espacial*: $O(N)$

BFS:

- *Temporal*: $O(V + E)$
- *Espacial*: $O(V)$ (usa Cola)

DFS:

- *Temporal*: $O(V + E)$
- *Espacial*: $O(V)$ (usa Stack)

AVL:

- *Temporal*: $O(log_2(N))$ (inserción, borrado, búsqueda)
- *Espacial*: $O(N)$

BTree:

- *Temporal*: $O(N)$ (inserción, borrado, búsqueda)

RED-BLACK TREE:

- *Temporal*: $O(log_2(N))$ (inserción, borrado, búsqueda)
- *Espacial*: $O(N)$

BST:

- *Temporal*: $O(N)$ (inserción, borrado, búsqueda)
- *Espacial*: $O(N)$

2 Lucene

Lucene usa un *archivo invertido*: conjunto de términos que indican a qué documento pertenece.

término → documento

2.1 Definiciones

Definición 1. Documento Lucene

Secuencia de *campos (fields)*. Cuando se ingresa un documento en Lucene automáticamente se le asocia un ID (docid).

Definición 2. Campo/Field

Par nombre-secuencia de 1+ términos.
Un field puede ser:

1. Sólo almacenable (se guarda literal y no se procesa) ⇒ está fuera del archivo invertido, por lo que **no** participa de las búsquedas
2. Sólo indexable ⇒ participa de las búsquedas
3. Indexable y almacenable ⇒ se guarda literal y se procesa para que participe de las búsquedas

El *field predefinido* es **TextField**: maneja tipos de datos de texto, se indexa y se tokeniza.

Definición 3. Término/Term

Un término Lucene es una secuencia de bytes (int, string, etc) asociada a cierto campo. Es importante notar que *dos secuencias de bytes con igual contenido pero asociadas a 2 campos diferentes se consideran diferentes.*
Ejemplo: Dato (título materia) ≠ Dato (apellido)

2.2 Aplicaciones

Necesitamos realizar dos aplicaciones independientes con Lucene: **IndexBuilder** y **Searcher**.

2.2.1 IndexBuilder

Aplicación que se encarga de generar el índice a partir de un conjunto de documentos Lucene y lo deja almacenado en un directorio que le digamos.

Observación: Es nuestra responsabilidad mantener el índice actualizado, es decir: re-ejecutar si agregamos o modificamos documentos.

2.2.2 Searcher

Aplicación que se encarga de aceptar consultas y utiliza el índice construido para retornar los docids que "matchearon" la consulta rankeados en cierto orden.

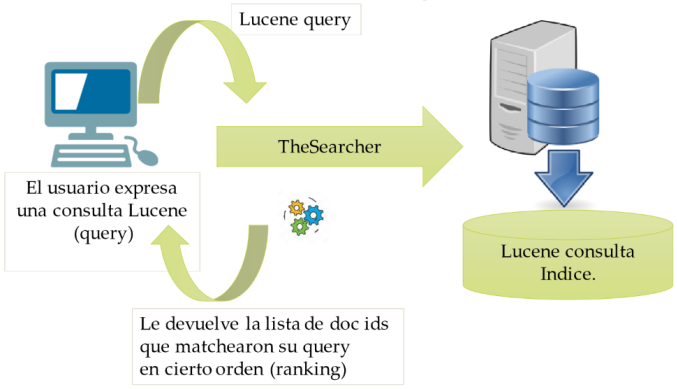


Figure 1: Proceso de Búsqueda

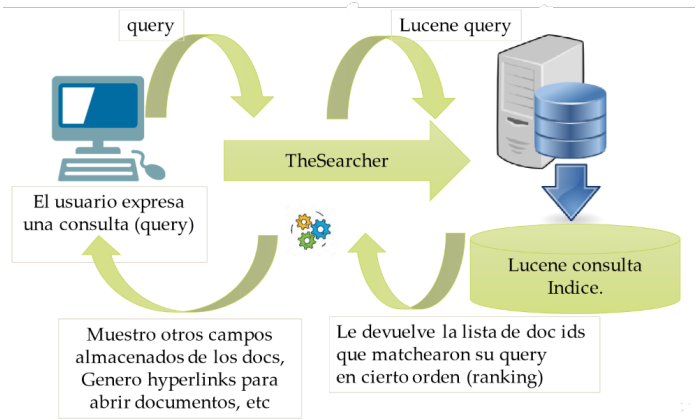


Figure 2: Proceso de Búsqueda Mejorado

2.3 Queries

2.3.1 TermQuery

Busca **un** sólo término.

2.3.2 PrefixQuery

Busca por prefijo.

2.3.3 TermRangeQuery

Busca si el término se encuentra en el intervalo especificado.

[BytesRefIzq, BytesRefDer]:
fieldName, BytesRefIzq, BytesRefDer, true, true

(BytesRefIzq, BytesRefDer):
fieldName, BytesRefIzq, BytesRefDer, false, false

[BytesRefIzq, BytesRefDer):
fieldName, BytesRefIzq, BytesRefDer, true, false

(BytesRefIzq, BytesRefDer]:
fieldName, BytesRefIzq, BytesRefDer, false, true

2.3.4 PhraseQuery

Busca secuencia.

Ejemplo de uso

```
Query query = new PhraseQuery(fieldName, word1, word2
                                    , ..., wordN);
```

2.3.5 WildcardQuery

Busca por matching de $*$ o bien ?.

Ejemplo de uso

```
Query query = new WildcardQuery(myTerm);
```

Algunos ejemplos:

```
queryStr = "g*e";    // g(0 <= letras)e
queryStr = "g?e";    // g(1 >= letras)e
queryStr = "*";      // (0 <= letras)
```

2.3.6 FuzzyQuery

Usa Damerau-Levenshtein con MaxEdit 2.

2.3.7 BooleanQuery

Es como una expresión de Lógica Proposicional.

Ejemplo de uso

Ejemplo 1:

```
String queryStr = "game AND store";
QueryParser queryParser = new QueryParser("content",
                                          new StandardAnalyzer());
```

Ejemplo 2:

```
String queryStr = "game OR store";
QueryParser queryParser = new QueryParser("content",
                                          new StandardAnalyzer());
```

2.4 Ranking

Dada una colección de N documentos $D = \{Doc_1,\ Doc_2,\ ...,\ Doc_n\}$ y una $query = term$, para aquellos documentos que matchean la consulta:

$$Score(Doc_i,\ query) = FormulaLocal(Doc_i,\ term) * FormulaGlobal(D,\ term)$$

donde

- Fórmula Local: Calcula qué tan relevante es un query respecto al documento a rankear.

$$FormulaLocal(Doc_i,\ query) = \sqrt{\frac{\#freq(term\,in\,Doc_i)}{\#term\,existentes\,en\,Doc_i}}$$

- Fórmula Global: Calcula qué tan relevante es esa query en el conjunto de documentos.

$$FormulaGlobal(DOC,\ query) = 1 + log_e\left(\frac{1 + \#docs\,en\,la\,coleccion}{1 + \#docs\,que\,contienen\,term}\right)$$

Ranking Multi-Término

Si un query consta de varios términos entonces el puntaje se calcula:

$$Score(Doc_i,\ query) = \sum_{\substack{\text{term in query} \\ \text{y no tiene NOT}}} FormulaLocal(Doc_i,\ term) * FormulaGlobal(D,\ term)$$

es decir, se hace la sumatoria de los calculos parciales de cada término del query sii no está modificado por NOT.

3 Teorema Maestro

Si una fórmula recurrente puede expresarse genéricamente así:

$$T(N) = \underbrace{a * T(\frac{N}{b})}_{\substack{\text{Invocación recursiva que} \\ \text{divide en subproblemas}}} + \underbrace{c * N^d}_{\substack{\text{Combinación de} \\ \text{soluciones parciales}}}$$

Entonces la complejidad O grande está dada por los siguientes 3 casos:

1. Si $a < b^d \Rightarrow O(N^d)$
2. Si $a = b^d \Rightarrow O(N^d * log(N))$
3. Si $a > b^d \Rightarrow O(N^{log_b(a)})$

donde

- N es el tamaño del input
- $a \in \mathbb{N}_{\geq 1}$ invocaciones recursivas por paso
- $b \in \mathbb{N}_{>1}$ mide tasa en que se reduce el tamaño del input
- $c \in \mathbb{R}_{>0}$
- $d \in \mathbb{R}_{\geq 0}$

4 Algoritmos

4.1 Red-Black Trees

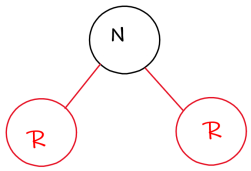
Reglas

1. Cada nodo es **ROJO** ó **NEGRO**
2. La raíz siempre es **NEGRA**
3. Las hojas siempre son **NEGRAS**
4. Las nuevas inserciones son **ROJAS**
5. Cada camino de la raíz a las hojas tiene la misma cantidad de nodos **NEGROS**
6. No puede haber nodos **ROJOS** consecutivos

Rebalanceo

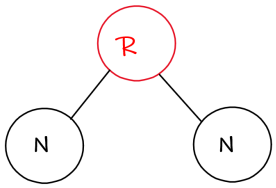
Tenemos dos casos:

1. "**BLACK** Aunt Rotate (BAR)"



Después de Rotar

2. "**RED** Aunt Colorflip (RAC)"



Después del Colorflip

4.2 Soundex

1. Utiliza una cartilla para codificar las letras:

26 Letras	Pesos fonéticos
A, E, I, O, U, Y, W, H	0 -- no se codifica
B, F, P, V	1
C, G, J, K, Q, S, X, Z	2
D, T	3
L	4
M, N	5
R	6

2. Siempre devuelve un código OUT de 4 caracteres, formado por: primero una letra y 3 dígitos (pesos fonéticos). Si hace falta se completa con 0s.

3. Algoritmo en C:

```
char IN[]= new String("...").toCharArray();
char OUT[] = {'0', '0', '0', '0'};
OUT[0]= IN[0];
int count= 1;
char current,  last= getMapping(IN[0]);
for(int i= 1; i < IN.length && count < 4; i++, last= current)
{
        char iter= IN[i];
        current= getMapping(iter);
        if (current != '0' && current != last)
                OUT[count++]= current;
}
return new String(OUT);
```

4. Algoritmo papel:

Paso 1 (opcional): Pasar a mayúsculas y dejar sólo las letras (dígitos, símbolos de puntuación, espacios, etc. se eliminan).
Paso 2: Colocar OUT[0]=IN[0].
Paso 3: Se calcula vble. **last** como el peso fonético de IN[0]
Paso 4: Para cada letra **iter** siguiente en IN y hasta completar 3 dígitos o terminar de procesar IN, hacer
3.1) calcular vble **current** con peso fonético de **iter**. Si es diferente a 0 y no coincide con **last**, appendear **current en OUT**.
3.2) **independiente del paso anterior**, tapar **last** = **current**.
Paso 5: si hace falta completar con '0's y devolver OUT.

5. **Similitud para Soundex**: Es la longitud de caracteres coincidentes entre los encodings respecto a la longitud del encoding. Por lo tanto los valores posibles son: $0, 0.25, 0.5, 0.75, 1$.

4.3 Levenshtein Distance

Definición: Es un algoritmo que calcula la mínima cantidad de operaciones necesarias para transformar un string a otro. Las operaciones validas son: *insertar, borrar o sustituir un caracter.*

1. Es una **métrica de distancia**, por lo que cumple con las siguientes propiedades:

   Simetria: Lev(str1, str2) = Lev(str2, str1)

   Desigualdad: Lev(str1, str2) + Lev(str2, str3) $\geq$ Lev(str1, str3)

2. Se lo implementa con Programación Dinámica (= tecnica que consiste en reusar valores previamente calculados para no tener que recalcularlos repetidamente)

3. Se utiliza una tabla para poner en practica la Programación Dinámica:

	♥	B	I	G		D	A	T	A
♥	0	1	2	3	4	5	6	7	8
B	1	0	1	2	3	4	5	6	7
I	2	1	0	1	2	3	4	5	6
G	3	2	1	0	1	2	3	4	5
D	4	3	2	1	1	1	2	3	4
A	5	4	3	2	2	2	1	2	3
T	6	5	4	3	3	3	2	1	2
A	7	6	5	4	4	4	3	2	1

   donde cada celda representa la distancia entre el substring de arriba y el de la izquierda.

4. Para calcular una celda, se le suma uno a la de la izquierda, uno al de arriba y para la diagonal (de arriba a la izquierda) hay dos opciones: se le suma 0 si las letras coinciden, y 1 si no coinciden.

5. Se puede normalizar el resultado de la siguiente manera:

$$LevNormalized(str1, str2) = 1 - \frac{Lev(str1, str2)}{max(str1.length(), str2.length())}$$

4.4 Q-Grams

Definición: Es un algoritmo que consiste en generar los pedazos que componen un string. La distancia entre 2 strings estará dada por la cantidad de componentes que tengan en común.

1. Canónica:

   Se toman hasta ventanitas de tamaño 3.

   Para que todas las letras participen lo mismo se le ponen $Q - 1$ metacaracteres en los extremos (donde Q es la cantidad de letras).

2. La forma de normalizarlo es la siguiente:

$$QGram(str1, str2) = \frac{\#TG(str1) + \#TG(str2) - \#TGNoShared(str1, str2)}{\#TG(str1) + \#TG(str2)}$$

   entonces 1 es maxima similitud y 0 es nula.

3. *Ejemplo*:

Para Q exactamente 2:
¿Cúal es la similitud entre salesal y vale ?

Justificar el cálculo.

Rta
Q-grams(salesal)= { #s, sa, al, le, es, sa, al, l#)
Q-grams(vale)= { #v, va, al, le, e#}.
En común 2 (ojo con los repetidos!).
Q-Gram(salesal, vale) = (8 + 5 - 9) / (8 + 5) = 0.3076

4.5 KMP

1. No vuelve a chequear un carácter que ya sabe que matcheó.
2. Utiliza la tabla de "Next" que tiene en cada posición i la longitud del *borde propio* más grande para el substring query desde 0 hasta i.
3. Para calcular la tabla "Next" se utiliza el siguiente algoritmo:

```
private static int[] nextComputation(char[] query) {
    int[] next = new int[query.length];

     int border=0;  // Length of the current border

     int rec=1;
     while(rec < query.length){
          if(query[rec]!=query[border]){
               if(border!=0)
                    border=next[border-1];
               else
                    next[rec++]=0;
          }
          else{
               border++;
               next[rec]=border;
               rec++;
          }
     }
     return next;
}
```

4. **Algoritmo KMP**

```java
public static int indexOf(char[] query, char[] target, int from) {
    int [] next = createNextArray(query);
    int targetCursor = from;
    int queryCursor = 0;
    while ( queryCursor < query.length && targetCursor < target.length ) {
        if (query[queryCursor] == target[targetCursor]) {
            queryCursor++;
            targetCursor++;
        } else {
            if (queryCursor == 0) {
                targetCursor++;
            } else {
                queryCursor = next[queryCursor - 1];
            }
        }
    }

    if (queryCursor == query.length) {
        return targetCursor - queryCursor;
    }

    return -1;
}
```

Complejidades:

- Next:

  Espacial: $O(m)$

  Temporal: $O(m)$ donde m es la longitud del query.

5 Heurísticas

5.1 Fuerza Bruta/Búsqueda Exhaustiva (con Stack o Queue)

Definición

Técnica que busca todas las posibles soluciones explorando el espacio de soluciones de forma implícita.

Se usa si un problema tiene muchas soluciones y las quiero todas.

Idea

1. Si el nodo no puede expandirse más ⇒ retornar/imprimir resultado
2. Sino por cada posibilidad para ese nodo expandir en un próximo nivel. El nodo puede:

   Resolver caso pendiente

   Explorar nuevos pendientes

   Deshacer/Quitar pendiente generado

5.2 Programación Dinámica

Definición

Técnica que permite almacenar valores que se calcularon previamente en soluciones anteriores para reusarlos, en vez de calcularlos repetidamente.

Ejemplo: Levenshtein, Dijkstra, Ackerman, Fibonacci.

5.3 Backtracking

Definición

No expande innecesariamente nodos que ya sabe (gracias a las restricciones) que no conducen a la solución.

¿Cómo evita expandir más de un nodo? Un nodo intermedio ya lleva acumulado valores. En el mejor de los escenarios completará con números bajos, es decir "1" en lo que falta.

$$\text{Si } \sum (\text{valores actuales}) + \text{ valores faltantes} * 1 > \text{umbral} \Rightarrow \text{imposible seguir}$$

Ejemplo: N-Queens.

Backtracking + Programación Dinámica

Con programación dinámica podemos agregar la suma actual como parámetro y así evitar calcular muchas veces la suma.

5.4 Divide & Conquer

Definición

Técnica que descompone un problema de tamaño N en problemas más pequeños que tengan solución. Finalmente, se debe proponer cómo construir la solución final a partir de las soluciones de los problemas menores.

Ejemplo: Mergesort, Quicksort, Búsqueda en BST.

5.5 Algoritmos Greedy/Ávidos

Definición

Técnica que busca en cada etapa un óptimo local con el objetivo de llegar al óptimo global.

Ejemplo: Algoritmo de Kruskal.