

Motor de Búsqueda usando Hadoop

Franklin Canaza, Kevin Salazar

Noviembre del 2020

1 Implementación de algoritmos

1.1 Índice Invertido

Adaptado de <https://timepasstechies.com/map-reduce-inverted-index-sample/>

El algoritmo considera como entradas un conjunto de archivos ‘1.txt’, ‘2.txt’, ‘3.txt’, etc donde los números indican el ID del documento. El contenido de estos archivos es el título y el *abstract* de cada documento, los cuales son *papers* en idioma inglés.

El único archivo de salida está formado por las líneas “*computer 1.txt 7.txt 97.txt*”, por ejemplo, las cual indica que la palabra *computer* se encuentra en los archivos ‘1.txt’, ‘7.txt’ y ‘97.txt’.

La parte *map* del algoritmo tiene como entrada las líneas de los diferentes archivos, estas son tokenizadas hacia un string, de cada token se remueven los puntos, comas, paréntesis y se convierten a minúscula. Se itera sobre este string y se va escribiendo como par *clave,valor*, el token y el nombre del archivo, respectivamente.

```
public static class InvertedIndexMapper extends Mapper<Object, Text, Text, Text> {

    private Text word = new Text();
    private Text fileNameValue = new Text();

    public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {

        StringTokenizer itr = new StringTokenizer(value.toString().
            replaceAll("[,.()]", "").toLowerCase());
        String fileName = ((FileSplit) context.getInputSplit()).
            getPath().getName();
        fileNameValue.set(fileName);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, fileNameValue);
        }
    }
}
```

```

    }
}
}

```

La parte *reduce* recibe los pares (palabra, Lista de archivos donde está esa palabra), y luego de iterar sobre esa lista para eliminar duplicados retorna el par que se escribirá en el archivo final del algoritmo.

```

public static class IntSumReducer extends Reducer<Text,IntWritable,Text,
                                                IntWritable> {

    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {

        StringBuilder sb = new StringBuilder();
        boolean first = true;
        for (Text value : values) {

            if (first) {
                first = false;
            } else {
                sb.append(" ");
            }

            if (sb.lastIndexOf(value.toString()) < 0) {
                sb.append(value.toString());
            }

        }
        result.set(sb.toString());

        context.write(key, result);
    }
}

```

1.2 PageRank

Adaptado de la implementación de Daniele Pantaleone, <https://github.com/danielepantaleone/hadoop-pagerank>

El algoritmo de pageRank tiene como entrada un archivo “links.txt” cuyas líneas son un par de números que representan el ID del documento, el primero tiene un link o referencia hacia el segundo, es útil para formar el grafo pues de él se tendrán los nodos(IDs) y las aristas(links).

Como salida tiene un archivo que indica el rank de cada documento, son dos

columnas por fila, la primera indica el ID del documento y la segunda su rank. El que esté ordenado ayudará durante el proceso de consulta en el motor.

El algoritmo agrupa 3 *jobs*, los dos primeros tienen map-reduce, el último solo aplica map. Job 1, el mapper toma las líneas del archivo y por cada línea va agregando los nodos a un *set* y va generando como salida el par textual {nodoFrom, nodoTo_i}, el reducer toma el archivo de combinación con pares {nodo, lista de nodos con los que tiene un link_i} y da como resultado un par de valores que consideran el rank inicial de dicho nodo, este rank será de 1.0, el par es {nodo, rank links_i}.

Job 2, con el resultado del job anterior genera un par similar, pero con el rank actualizado({nodo, rankActualizado links_i}), esta actualización se dará a lo largo de las iteraciones que, junto el parámetro de damping, es parámetro del algoritmo. Dentro de cada iteración el mapper retorna pares de valores textuales, el primer valor es un nodo que forma parte de los links de su nodo origen, el segundo valor es la concatenación del rank del nodo origen y la cantidad de links que tiene este nodo origen, hay que tener en cuenta que el último par es diferente, su primer valor indica el nodo origen y el segundo sus nodos link, esto para poder formar con el reducer nuevamente el par de salida final del job, {nodo, rankActualizado links_i}, que será entrada en la siguiente iteración para este job.

Job 3, solo consta del mapper el cual toma la salida del job 3({nodo, rankActualizado links_i}) y forma el par {nodo, rank_j}. Esta salida es pasada al archivo de salida final.

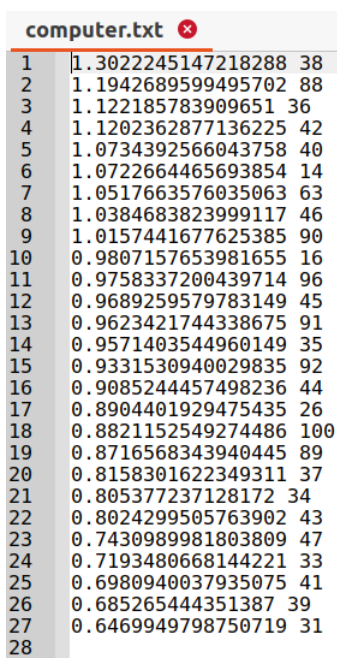
Podemos resumir este proceso con lo siguiente:

```
//job 1
// map:
//      nodoFrom nodoTo
// reduce:
//      nodo rank(1.0) link1,link2,link3
//job 2
// map:
//      link1 rankNodo cantLinksdeNodo
//      link2 rankNodo cantLinksdeNodo
//      link3 rankNodo cantLinksdeNodo
//      nodo |link1,link2,link3
// reduce:
//      nodo rankNuevo link1,link2,link3
// Job 3
// map:
//      nodo rank
```

2 Motor de Búsqueda

El motor de búsqueda requiere de un conjunto de archivos para poder hacer la consulta. Estos archivos determinan qué artículos son más relevantes para cada palabra. Es decir, por cada palabra existente en la totalidad de documentos se tiene un archivo 'computer.txt', por ejemplo para la palabra *computer*, formando por dos columnas, la primera indica el rank, y la segunda el ID del documento, está ordenado por rank para que la consulta tome las primeras líneas, según se requiera.

La generación de los archivos anteriores se hace de manera local con un script en python. Este toma como entrada el archivo resultado del invertedIndex y del pageRank, lee cada línea de este primer archivo por lo que obtiene la palabra y la lista de los ID de los documentos que la contienen, estos últimos los busca de manera binaria en el archivo de pageRank, al tener todos los ranks de sus links los ordena y los escribe en el archivo que tiene como nombre la palabra. Este archivo es de la forma siguiente:



	computer.txt
1	1.3022245147218288 38
2	1.1942689599495702 88
3	1.122185783909651 36
4	1.1202362877136225 42
5	1.0734392566043758 40
6	1.0722664465693854 14
7	1.0517663576035063 63
8	1.0384683823999117 46
9	1.0157441677625385 90
10	0.9807157653981655 16
11	0.9758337200439714 96
12	0.9689259579783149 45
13	0.9623421744338675 91
14	0.9571403544960149 35
15	0.9331530940029835 92
16	0.9085244457498236 44
17	0.8904401929475435 26
18	0.8821152549274486 100
19	0.8716568343940445 89
20	0.8158301622349311 37
21	0.805377237128172 34
22	0.8024299505763902 43
23	0.7430989981803809 47
24	0.7193480668144221 33
25	0.6980940037935075 41
26	0.685265444351387 39
27	0.6469949798750719 31
28	

Figure 1: Archivo que indica el rank de los documentos que contienen la palabra 'computer'.

La interfaz principal del motor de búsqueda se muestra en la figura 2. Para el controlador utilizamos el lenguaje de programación Python con el framework Flask.

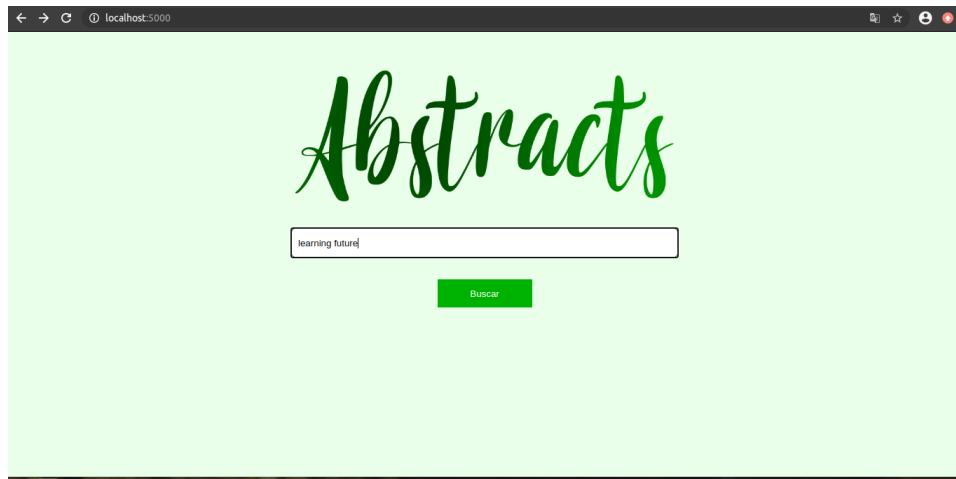


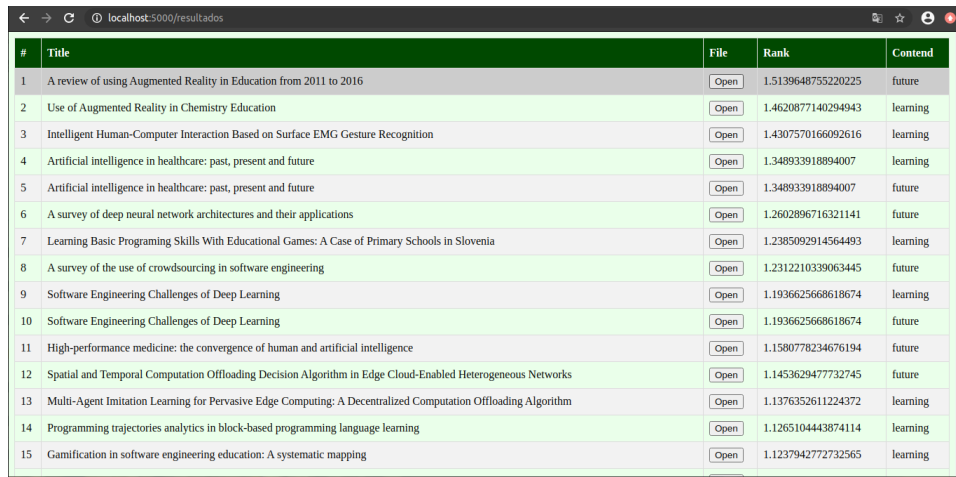
Figure 2: Pagina principal del motor de búsqueda.

Para realizar una búsqueda se pasa una variable que contiene la cadena ingresada de la interfaz al controlador mediante la función `request`, luego esta cadena se separa en palabras y se agregan a un arreglo `words`:

```
strsearch = request.form("form")
words = strsearch.split()
```

Una vez obtenidas las palabras se leen los archivos para obtener los datos necesarios, como el `rank`, nombre del archivo, y título. Finalmente se unen todos los resultados obtenidos y se ordenan para mostrarlos en la vista de resultados (figura 3), el código se muestra a continuación:

```
for word in words:
    line=f.readline()
    ranks.append(line[0:line.find(' ')])
    tempfile=line[line.find(' ')+1:len(line)-1]
    files.append(tempfile)
    srcfile="Abstracts/"+tempfile+".txt"
    f2=open(srcfile)
    title=f2.readline()
    f2.close()
    titles.append(title)
    wordtemp.append(word)
#Unimos y ordenamos
lists = sorted(zip(ranks,files,wordtemp,titles),reverse=True)
return render_template('resultados.html',lists=lists)
```



#	Title	File	Rank	Contend
1	A review of using Augmented Reality in Education from 2011 to 2016	Open	1.5139648755220225	future
2	Use of Augmented Reality in Chemistry Education	Open	1.4620877140294943	learning
3	Intelligent Human-Computer Interaction Based on Surface EMG Gesture Recognition	Open	1.4307570166092616	learning
4	Artificial intelligence in healthcare: past, present and future	Open	1.348933918894007	learning
5	Artificial intelligence in healthcare: past, present and future	Open	1.348933918894007	future
6	A survey of deep neural network architectures and their applications	Open	1.2602896716321141	future
7	Learning Basic Programing Skills With Educational Games: A Case of Primary Schools in Slovenia	Open	1.2385092914564493	learning
8	A survey of the use of crowdsourcing in software engineering	Open	1.2312210339063445	future
9	Software Engineering Challenges of Deep Learning	Open	1.1936625668618674	learning
10	Software Engineering Challenges of Deep Learning	Open	1.1936625668618674	future
11	High-performance medicine: the convergence of human and artificial intelligence	Open	1.1580778234676194	future
12	Spatial and Temporal Computation Offloading Decision Algorithm in Edge Cloud-Enabled Heterogeneous Networks	Open	1.1453629477732745	future
13	Multi-Agent Imitation Learning for Pervasive Edge Computing: A Decentralized Computation Offloading Algorithm	Open	1.1376352611224372	learning
14	Programming trajectories analytics in block-based programming language learning	Open	1.1265104443874114	learning
15	Gamification in software engineering education: A systematic mapping	Open	1.1237942772732565	learning

Figure 3: Resultados obtenidos de la búsqueda "learning future".

Como tercera vista tenemos el titulo del abstract y su descripcion en la figura 4. Obtenemos el nombre del archivo mediante la función request para abrir el documento.

```
strfile = request.form['file']
strfile = "Abstracts/"+strfile+".txt"
f=open(strfile)
titlefile=f.readline()
descriptionfile=f.readline()
f.close()
return render_template('abstract.html', title=titlefile,description=descriptionfile)
```

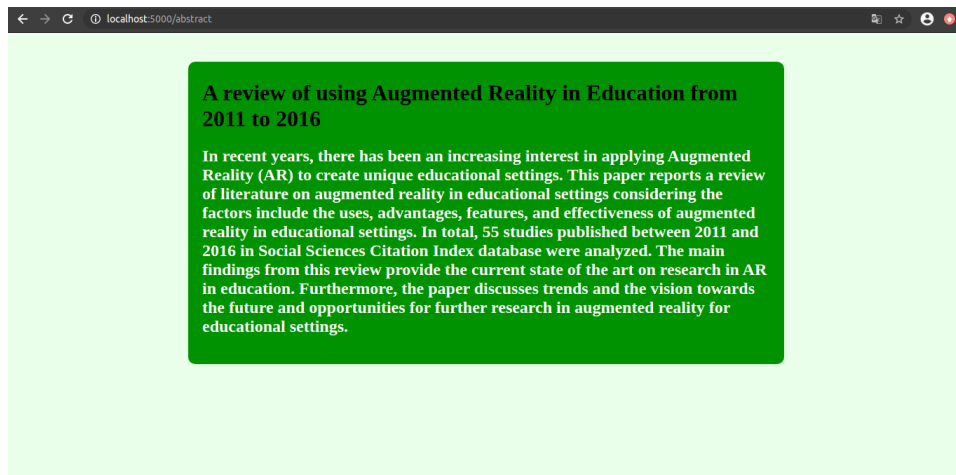


Figure 4: Vista de un archivo seleccionado.

3 Problemas y desafíos

Dado que AWS tiene su sistema de almacenamiento S3 algunas funciones de Hadoop dedicadas a tratar el *file system* no necesariamente son compatibles con S3 y HDFS a la vez. Por ejemplo, cada vez que se ejecuta una función map-reduce la carpeta output no debe existir para ello se puede usar una función que elimina esta carpeta si existe previamente, pero su aplicación con S3 no ha funcionado en las pruebas que hemos realizado.

El script de python lleva a memoria el contenido, por ejemplo, del archivo 'computer.txt', este no es tan grande como el archivo de salida de pageRank pero su tamaño podría llegar a ser considerable pues sería proporcional al tamaño de todos los documentos que contengan esa palabra. Una solución alternativa para esto podría ser recortar este archivo; es decir, no considerar la totalidad de ranks sino, tal vez, los 500 primeros. Esto reduciría en gran medida el uso de memoria cuando se quiera actualizar estos archivos pues se compararían los nuevos ranks solo con los mejores 500.

En un motor de búsqueda el manejo de excepciones y errores con una gran cantidad de datos se hace complejo debido a la gran cantidad de palabras existentes, además una búsqueda que incluye dos a mas palabras se torna un desafío porque existen diversos factores que pueden afectar en la búsqueda como por ejemplo el ingreso de una palabra inexistente en la base de datos o mal escrita.