Práctica Hadoop

Procesamiento de Datos a Gran Escala

Antonio Coín Castro
Luis Antonio Ortega Andrés

18 de octubre de 2020

Puesta a punto del entorno Hadoop

Una vez que hemos instalado Hadoop correctamente en el sistema, y hemos editado los archivos de configuración necesarios para activar el modo *pseudo-distributed* con HDFS, iniciamos este último con la siguiente orden (partiendo del directorio de instalación de Hadoop):

```
sbin/start-dfs.sh
```

También creamos un directorio en el sistema de archivos HDFS donde vamos a alojar nuestros conjuntos de datos:

```
hdfs dfs -mkdir /user/bigdata/p1
```

No debemos olvidarnos de copiar nuestros archivos a HDFS con la orden `-copyFromLocal`:

```
hdfs dfs -copyFromLocal <ruta_local> /user/bigdata/p1
```

Compilación y ejecución

Utilizamos el script `compile.sh` para compilar nuestro programa y obtener un `.jar` listo para ser ejecutado. Añadimos una pequeña modificación al script original, permitiendo que el ejecutable generado tenga un nombre a nuestra elección. Para ello debemos considerar un argumento más (`$2`) y realizar el siguiente cambio en la última línea del script:

```
jar -cvf $2.jar -C ${file} .
```

El fichero completo quedaría entonces como sigue:

```
#!/bin/bash
file=$1
name=$2
HADOOP_CLASSPATH=$(hadoop classpath)
rm -rf ${file}
mkdir -p ${file}
javac -classpath $HADOOP_CLASSPATH -d ${file} ${file}.java
jar -cvf ${name}.jar -C ${file} .
```

Finalmente, podemos ejecutar nuestro programa en el entorno pseudo-distribuido con la siguiente orden:

```
bin/hadoop <name>.jar <class_name> /user/bigdata/p1/<input_file> \
  <output_dir>
```

1. Programa WordCount

Nos planteamos primero el problema de construir un programa para contar el número de apariciones de cada una de las palabras de un texto, que en este caso será un fragmento del Quijote (archivo `Quijote.txt`).

La estrategia a seguir dentro del paradigma de programación MapReduce será la siguiente. En primer lugar, dividimos el texto en unidades de un cierto tamaño, y pasamos cada uno de estos trozos a los Mappers. En la fase distribuida de Map, extraemos las palabras (que en principio suponemos separadas por espacios en blanco, tabulaciones o retornos de carro) y enviamos a los Reducers parejas `(palabra, 1)`, indicando que esa palabra en concreto aparece una vez. Después, en la fase de Shuffle se juntan las parejas cuya clave es igual (en este caso, la clave es la propia palabra), y se agrupan sus segundos elementos en una lista. Finalmente, en la fase Reduce se reciben parejas `(palabra, (1, 1, 1, ...))`, y lo que hacemos es sumar todos los elementos de la lista en el segundo elemento de la pareja, para obtener el número total de apariciones. Finalmente devolvemos parejas de palabras junto a su conteo.

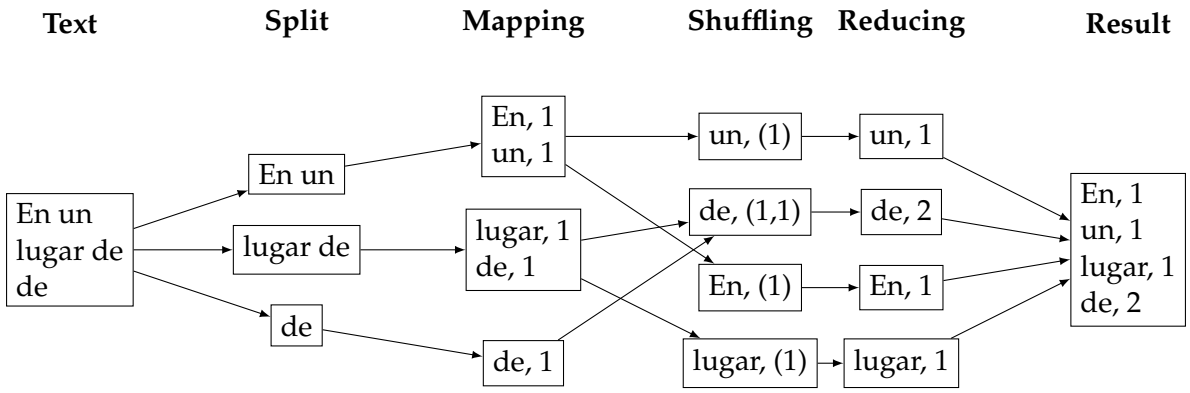

Figura 1: Esquema del programa WordCount.

En una sintaxis más funcional, podríamos escribir el programa como

```
words_by_line = text.flatMap { line => line.split(" ") }
result = word_by_line.map { w => (w, 1) }
                     .reduceByKey { (v1, v2) => v1 + v2 }
```

Además de esto, modificamos el programa para que no tenga en cuenta mayúsculas y minúsculas, ni signos de puntuación. Veamos con detalle los principales elementos del programa en Java.

Mapper

Definimos una clase para nuestro Mapper que extiende la interfaz homónima de Hadoop, proporcionando el formato de entrada y salida de nuestras parejas. En concreto, recibimos parejas `(Object, Text)` (donde ignoramos la clave) y proporcionamos como salida parejas `(Text, IntWritable)`. Las clases `Text` e `IntWritable` son clases de Hadoop que representan a los tipos de datos de Java `String` e `Int`, respectivamente [1]. En el método `map` tokenizamos la entrada con `StringTokenizer` (dividir en palabras), y para cada palabra guardamos en la salida la propia palabra junto con la constante 1.

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

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context)
      throws IOException, InterruptedException {
      StringTokenizer itr =
        new StringTokenizer(value.toString(), " \n\t\r\f");
      while (itr.hasMoreTokens()) {
          word.set(itr.nextToken());
          context.write(word, one);
      }
    }
  }
```

Para conseguir que el programa no tenga en cuenta los signos de puntuación ni las mayúsculas y minúsculas, debemos realizar dos cambios distintos:

- Para evitar los signos de puntuación, los añadimos al conjunto de delimitadores de palabras. Para ello configuramos el parámetro de delimitadores del constructor de la clase `StringTokenizer`.
- Para no distinguir mayúsculas y minúsculas, pasamos cada uno de los caracteres de cada palabra a minúscula utilizando la función `toLowerCase()`.

Así, la línea que modificamos queda como sigue:

```
StringTokenizer itr = new StringTokenizer(
  value.toString().toLowerCase(), " \n\t\r\f.,;:-¡¿?!()\"\'");
```

Reducer

En el Reducer recibimos parejas de la forma `(Text, Iterable<IntWritable>)` con las palabras y una lista con '1's (uno por cada aparición contabilizada), y devolvemos parejas

[1] Estas clases optimizan y reducen el *overhead* a la hora de serializar y deserializar objetos.

`(Text, IntWritable)` con las palabras y el número total de apariciones. Para calcular este último número simplemente sumamos los elementos de la lista asociada a cada palabra.

```
public static class IntSumReducer extends
  Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
      Context context) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

En este caso no hay que realizar ninguna modificación para incluir los cambios que queríamos.

Main

Desde la función principal solo tenemos que establecer la configuración apropiada del entorno de trabajo, especificando el nombre de las clases que hemos creado y que implementan las fases de Map y Reduce. También es necesario especificar el formato final de salida de las parejas `(clave, valor)`, que como ya dijimos será `(Text, IntWritable)`. También añadimos las rutas del fichero de entrada y del directorio de salida, e invocamos el trabajo (`Job`) que hemos creado.

```
public static void main(String[] args) throws Exception {
    Job job = new Job(conf, "wordcount");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
```

1.1. Conclusiones

Los resultados de ejecución de nuestro programa sin modificar se encuentran en el archivo `wc_orig`, mientras que la salida del programa modificado está en el archivo `wc_modif`. Podemos ver como en el primero aparecen repetidas varias palabras, ya sea por capitalización de la primera letra cuando comienzan oraciones, o porque se contabilizan los signos de puntuación. Sin embargo, en el segundo archivo esto ya no ocurre.

En la solución obtenida al ejecutar directamente los ejemplos de `hadoop-map-reduce` (archivo `wc_example`) podemos ver como no se utilizan separadores correctos, diferenciando por ejemplo las palabras `que` y `(que`. Ante esto, los resultados obtenidos mediante la versión modificada resultan ser más precisos. Concretamente, en la versión de ejemplo el número total de palabras distintas que se contabilizan es de 10533, mientras que en la versión modificada este número desciende hasta 7492.

La versión de ejemplo y nuestra versión no modificada coinciden en que ambas detectan como palabras diferentes algunas que deberían ser consideradas la misma. Además, podemos ver que el número total de palabras contabilizadas es el mismo en ambas versiones.

1.2. Cuestiones planteadas

Pregunta 1. *¿Dónde se crea hdfs? ¿Cómo se puede elegir su localización?*

La localización del sistema de archivos HDFS la determina la variable `dfs.datanode.data.dir`, cuyo valor por defecto es `file://${hadoop.tmp.dir}/dfs/data` según la documentación de Hadoop. Este valor se puede cambiar en el archivo `hdfs-site.xml`:

```
<property>
  <name>dfs.datanode.data.dir</name>
  <value>file://${hadoop.tmp.dir}/dfs/data</value>
</property>
```

Además, la variable `${hadoop.tmp.dir}` se puede consultar y modificar en `core-site.xml`,y tiene como valor por defecto `/tmp/hadoop-${user.name}`.

Pregunta 2. *Si estás utilizando hdfs, ¿cómo puedes volver a ejecutar WordCount como si fuese single.node?*

Para volver a configurar Hadoop para funcionar como *single.node* debemos revertir los cambios que hicimos en los archivos de configuración para habilitar HDFS. Es decir, eliminar las secciones `fs.defaultFS` del archivo `core-site.xml` y `dfs.replication` de `hdfs-site.xml`.

Otra opción para mantener activo HDFS pero ejecutar nuestro programa con archivos de entrada locales es especificar en la ruta de los archivos que pasamos al programa si son locales o están en HDFS. Para ello podemos añadir a nuestras rutas el prefijo `file://` para archivos

locales y el prefijo `hdfs://` para archivos en HDFS (debemos proporcionar rutas absolutas). Así, para ejecutar el programa WordCount con archivos de entrada y salida en el directorio local, usaríamos la orden:

```
bin/hadoop jar WordCount.jar uam.WordCount \
  file:///opt/hadoop/p1/Quijote.txt file:///opt/hadoop/p1/out
```

Pregunta 3. *En el fragmento del Quijote, ¿cuales son las 10 palabras más utilizadas? ¿Cuántas veces aparecen el artículo "el" y la palabra "dijo"?*

Para resumir la información del archivo de salida utilizamos código `Python` (archivo `utils.py`). En primer lugar, creamos un `dataframe` de `Pandas` con la salida facilitada por Hadoop:

```
df = pd.read_csv("wc_modif", delimiter="\t", header=None)
```

Para obtener las 10 palabras más frecuentes ordenamos el dataframe:

```
df.sort_values(by=1, ascending=False, inplace=True)
print(df.head(10))
```

Las 10 palabras mas utilizadas son: `que` (3055), `de` (2816), `y` (2585), `a` (1428), `la` (1423), `el` (1232), `en` (1155), `no` (916), `se` (753) y `los` (696).

Para buscar el número de ocurrencias de una palabra, buscamos su fila correspondiente:

```
print(df.loc[df[0] == "el"])
print(df.loc[df[0] == "dijo"])
```

La palabra `el` aparece un total de 1232 veces y la palabra `dijo` aparece 272 veces.

2. Programa PlayerAge

En esta sección planteamos resolver un problema distinto al de contar palabras. Utilizaremos el *dataset* `players.csv` proporcionado, donde tenemos información de algunos jugadores de la Premier League, sus edades y el equipo al que pertenecen. Nuestro objetivo será calcular para cada equipo una serie de estadísticos sobre la edad. En concreto, calcularemos de forma distribuida la media de edad, la edad mínima y la edad máxima de los jugadores de cada equipo.

La idea para el cálculo distribuido será similar a la de `WordCount`. En el archivo `players.csv` cada línea representa un jugador, donde en la primera columna encontramos su nombre, en la segunda el equipo al que pertenece y en la tercera su edad, separados por comas. Lo que haremos será distribuir las líneas en Mappers, donde devolveremos para cada jugador una pareja `(equipo, edad)` (ahora la clave es el equipo). Posteriormente, en el Reducer calcularemos la media, el mínimo y el máximo con los métodos usuales (sumas y comparaciones), y devolveremos una pareja `(equipo, (media, minimo, maximo))`.

Reutilizaremos la mayor parte del código Java desarrollado en la sección anterior, por lo que solo comentaremos aquí las partes en las que hacemos cambios. El código completo se puede consultar en el archivo `PlayerAge.java`. Para subir los archivos, compilar y ejecutar podemos seguir los mismos pasos que con `WordCount`, es decir:

```
$ bin/hdfs dfs -copyFromLocal players.csv /user/bigdata/p1
$ ./compile.sh PlayerAge PlayerAge
$ bin/hadoop jar PlayerAge.jar uam.PlayerAge \
  /user/bigdata/p1/players.csv /user/bigdata/p1/out_player
```

Mapper

En la clase Mapper solo debemos cambiar con respecto al programa anterior que ahora en vez de devolver siempre la constante 1, devolvemos la edad del jugador en cuestión. Para ello declaramos una variable de clase que será el *wrapper* de este número, y le daremos valor cuando extraigamos la edad del jugador, previamente convertida a entero.

```
private Text team = new Text();
private IntWritable age = new IntWritable();
```

En este caso no necesitamos tokenizar toda la entrada, ya que solo nos interesan las tres primeras columnas. Utilizamos el método `split` de la clase `String` que nos permite accesos aleatorios a las palabras, y escribimos en el `Context` el nombre del equipo y la edad del jugador.

```
public void map(Object key, Text value, Context context)
  throws IOException, InterruptedException {
    String[] split = value.toString().split(",");
    team.set(split[1]);
    age.set(Integer.parseInt(split[2]));
    context.write(team, age);
}
```

Reducer

En el Reducer recibimos ahora parejas `(equipo, (edad1, edad2, ...))`, donde el segundo elemento es una lista con las edades de todos los jugadores del equipo en cuestión. Definimos primero las variables que contendrán la media, el mínimo y el máximo:

```
private IntWritable meanAge = new IntWritable();
private IntWritable minAge = new IntWritable();
private IntWritable maxAge = new IntWritable();
```

En la función `reduce` simplemente calculamos la media, el mínimo y el máximo de la lista que recibimos, y lo guardamos en la salida que hemos preparado.

```
public void reduce(Text key, Iterable<IntWritable> values, Context context)
  throws IOException, InterruptedException {
    int ageSum = 0;
    int count = 0;
```

```
    int _minAge = 200;
    int _maxAge = 0;

    for (IntWritable val : values) {
        int value = val.get();
        ageSum += value;
        count++;
        if (value > _maxAge){
            _maxAge = value;
        }
        if (value < _minAge){
            _minAge = value;
        }
    }

    meanAge.set(ageSum/count);
    minAge.set(_minAge);
    maxAge.set(_maxAge);
}
```

Finalmente, para devolver parejas en el formato deseado, creamos primero una lista de `IntWritable` que contenga nuestros tres valores calculados, y luego las escribimos en el `Context`.

```
IntWritable result[] = {meanAge, minAge, maxAge};
context.write(key, Arrays.asList(result));
```

Nótese que para poder hacer esto hemos tenido que modificar la cabecera de la clase para especificar el formato de salida:

```
public static class MeanReducer
  extends Reducer<Text,IntWritable,Text,Iterable<IntWritable> > {...}
```

Main

En la función principal tan solo debemos cambiar el nombre de las clases que actúan ahora como Mapper y Reducer.

2.1. Conclusiones

Tras ejecutar nuestro programa, obtenemos un archivo de salida (`player_age`) en el que en cada línea tenemos el nombre de un equipo, y los tres valores que nos interesaban sobre él: la media de edad, la edad del jugador más jóven y la edad del jugador más viejo.

```
Arsenal         [26, 21, 35]
Bournemouth         [26, 20, 37]
```

```
Brighton+and+Hove         [28, 23, 36]
Burnley         [27, 24, 35]
Chelsea         [27, 21, 35]
Crystal+Palace         [28, 21, 38]
Everton         [26, 19, 36]
Huddersfield         [26, 20, 35]
Leicester+City         [27, 20, 33]
Liverpool         [24, 17, 31]
Manchester+City         [27, 20, 34]
Manchester+United         [25, 19, 35]
Newcastle+United         [26, 21, 34]
Southampton         [24, 20, 32]
Stoke+City         [28, 18, 36]
Swansea         [27, 19, 34]
Tottenham         [25, 21, 33]
Watford         [28, 21, 36]
West+Brom         [28, 18, 37]
West+Ham         [27, 21, 33]
```

Como último comentario, cabe destacar que hemos aprovechado en el Reducer para calcular a la vez los tres estadísticos que queríamos, pero para ello ha sido necesario cambiar el tipo de retorno en la cabecera de nuestra clase `MeanReducer` cuando extendemos la clase `Reducer`.