Asigantura:

Procesamiento de datos a gran Escala

Práctica 1-binParte 1: Programación básica en Java con Hadoop

1. Instalación de Hadoop

Vamos a utilizar una máquina virtual en la que instalaremos la distribución de Hadoop, para ello hay que seguir los paso indicados en el tutorial instalación de Hadoop.

Una vez realizado el tutorial de instalación:

0. Arrancamos la MV (usuario: bigdata, password: bigdata).

1. Abrimos una terminal del sistema.

2. Nos movemos a la carpeta donde se encuentra la distribución de Hadoop (/opt/hadoop)

3. Arrancamos los servicios asociados a HDFS Hadoop.

> sbin/start-dfs.sh
> bin/hdfs dfs -ls /

Nota: si no se a introducido datos en el HDFS aparecerá vacio

Introducir datos con

> bin/hdfs dfs -mkdir /user

> bin/hdfs dfs -mkdir /user/bigdata

> bin/hdfs dfs -mkdir /user/bigdata/prueba

> bin/hdfs dfs -put etc/hadoop/*.xml /user/bigdata/prueba

4. PodemosDeberías poder acceder a la web del NameNode en http://localhost:50070

5. Podemos chequear que los servicios en java se han lanzado como procesos.

> top

HDFS (Hadoop Distributed File System), permite organizar los datos en directorios y ficheros. Proporciona una interfaz de línea de comandos denominada shell FS, que permite al usuario interactuar con los datos de HDFS, que son accesibles a los programas MapReduce de Hadoop.

Hay 2 métodos para interactuar con HDSF:

a) Usar la línea de comandos e invocar la *shell* usando el formato: >**hdfs dfs <args>**

```
>hdfs dfs –ls /

>hdfs dfs –mkdir myTestDir
```

b) También se puede manipular HDFS usando la consola Web

en http://localhost:50070

Importante recordar:

- Rutas relativas (por defecto) y absolutas: /user/bigdata.
- HDFS **no** es un sistema de fichero POSIX: **no** podemos hacer todo lo que haríamos sobre un FS normal. Sí podemos hacer ciertas cosas aplicando *pipes* a la salida del comando sobre HDFS, pero no se ejecutan de forma distribuida.
- Los ficheros en HDFS se almacenan de forma distribuida: partición en bloques, replicación, …
- Los comandos **hadoop fs** y **hdfs dfs** son equivalentes (los encontraréis de forma indistinta en la documentación), aunque se está migrando a la utilización del segundo.

Ejercicio 1.1: ¿Qué ficheros ha modificado para activar la configuración del HDFS? ¿Qué líneas ha sido necesario modificar?

Ejercicio 1.2: Para pasar a la ejecución de Hadoop sisn HDFS ¿ es suficiente con

con parar el servicio con stop-dfs.sh? ¿Cómo se consigue?

2. Ejecución de la aplicación de ejemplo wordcount

Hadoop proporciona programas de ejemplo que puedo ejecutar para ver una aplicación map/reduce corriendo.

1. Cambiar al directorio Linux donde esta el programa **hadoop-example.jar**

```
> cd /opt/hadoop/share/hadoop/mapreduce
```

2. Introducir en un directorio de HDFS / un fichero datos. Una forma de hacerlo es:

```
> /opt/hadoop/bin/hdfs dfs -copyFromLocal <ruta-fichero-local> <ruta-directorio-hdfs>
```

Por ejemplo algo similar a:

/opt/hadoop/bin/hdfs dfs -copyFromLocal /home/bigdata/quijote.txt /user/bigdata

3. Ejecutad el programa de contar palabras

> /opt/hadoop/bin/hadoop jar hadoop-mapreduce-examples-3.1.2.jar wordcount *<fichero/directorio de entrada>* *<directorio de salida>*

Nota: El <directorio de salida> no tiene que existir

Por ejemplo algo similar a

/opt/hadoop/bin/hadoop jar hadoop-mapreduce-examples-2.8.1.jar wordcount /user/bigdata/quijote.txt /user/bigdata/salidaq

4. Visualizar los ficheros creados en el directorio de salida

```
> /opt/hadopp/hdfs dfs –ls <directorio de salida>
```

Por ejemplo algo similar a

/opt/hadoop/bin/hdfs dfs -ls /user/bigdata/salidaq

5. Ver el contenido del fichero part-r-00000 que tiene el resultado

```
>/opt/hadoop/hdfs dfs –cat /<directorio de salida>/part-r-00000
```

Por ejemplo algo similar a

/opt/hadoop/bin/hdfs dfs -cat /user/bigdata/salidaq/part-r-00000

3. Ejercicio: programación de aplicaciones Hadoop con Java

3.1 Esqueleto del programa Java

Descargar de moodle el material para la práctica:

a. Esqueleto de programa Java (*WordCount.java*)
b. Fichero de ejemplo (*quijote.txt*)

La estructura del programa Java que hemos copiado al clúster es la siguiente:

```
1  package uam;
2  import java.io.IOException;
3  import java.util.*;
4
5  import org.apache.hadoop.conf.*;
6  import org.apache.hadoop.mapreduce.*;
7  import org.apache.hadoop.io.*;
8  import org.apache.hadoop.fs.Path;
9  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
10 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
11
12 public class WordCount {
13
14   public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>{
15
16   }
17
18   public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable> {
19
20   }
21
22   public static void main(String[] args) throws Exception {
23
24   }
25 }
```

- Definición del paquete donde se sitúan nuestras clases (*uam*).
- Importación de los paquetes que se utilizarán en la aplicación.
- Creación de la clase *WordCount*, que contiene:
  - La definición de una clase *TokenizerMapper*
  - La definición de una clase *IntSumReducer*
  - La definición de un método *main*, que será el que se ejecutará al invocar nuestro programa Java final.

3.2 Completar la clase *mapper*

El *mapper* procesa cada una de las líneas de datos por separado, y genera pares <clave, valor> para que luego sean procesados en la fase de reducción. Como vamos a manipularla como texto, debemos convertir la línea a texto.

En nuestro ejemplo vamos a *tokenizar* las palabras, es decir, procesar cada una de las palabras de la línea de texto leída, para luego procesarlas de forma iterativa mediante un bucle *while*.

Nota: Esto puede hacer que tengamos que incluir nuevas librerías.

En nuestro ejemplo, la clase *TokenizerMapper* que creamos deberá extender **Mapper<Objext, Text, Text, IntWritable>**. En dicha clase debemos:

- Definir un método público llamado **map**.
- El código debería ser similar al siguiente:

```
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());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}
```

Nótese que nuestra clase *TokenizerMapper* extiende la clase *Mapper* de la librería de Hadoop para Java, y tiene los modificadores *<Object, Text, Text, IntWritable>*. Estos modificadores nos indican el tipo (clase) de los pares <clave, valor> que entran y salen del método. Por ejemplo, en nuestro caso el par <clave, valor> de entrada son de clase *Object* y *Text* respectivamente, y se generan pares con un *Text* como clave y un *IntWritable* como valor.

Nota: creamos un objeto de tipo *IntWritable* que represente el número 1 para utilizarlo como el **valor** que nuestro mapper le dará a cada una de las claves de salida que genere.

Recordemos que queríamos una versión de WordCount que fuera *case insensitive* y a la que no le afectaran los caracteres no alfabéticos.

Para lograr este objetivo, utilizaremos la documentación de los

paquetes de Java para buscar algún método que nos permita realizar lo que queremos:

http://docs.oracle.com/javase/7/docs/api/

No hay que olvidarse de salvar el trabajo al terminar (y durante) la edición del código de la clase.

3.3 Completar la clase *reducer*

En la fase de reduce, se procesan se recibe una lista de valores asociados a a una misma clave. La tarea de la clase Reducer es la de procesar todos estos valores y generar (normalmente) un único valor resumen de salida.

En nuestro ejemplo, la clase *IntSumReducer* que creamos deberá extender **Reducer<Text, IntWritable, Text, IntWritable>**. En dicha clase debemos:

- Definir un método público llamado **reduce**.
- El código debería ser similar al siguiente:

```
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);
  }
}
```

3.4 Completar el método *main*

En la clase principal WordCount, debemos completar el método *main* para determinar el comportamiento del programa una vez lo invoquemos.

```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();

  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);
}
```

Dentro del método *main*:

- Creamos el trabajo Hadoop (*job*)
- Configuramos las clases asociadas al trabajo, a las etapas map y reduce, y los datos de salida.
- Configuramos (en este caso ficheros) entrada y salida.
- Esperamos a la finalización del trabajo.

3.5 Compilar la aplicación

Escriba script de compilación (*compilar.bash*) similar al de la figura que utilizaremos para generar una aplicación hadoop a partir de nuestro código Java.

```
#!/bin/bash

file=$1

HADOOP_CLASSPATH=$(hadoop classpath)
echo $HADOOP_CLASSPATH

rm -rf ${file}
mkdir -p ${file}

javac -classpath $HADOOP_CLASSPATH -d ${file} ${file}.java
jar -cvf ${file}.jar -C ${file} .
```

Este script debe:

- Acceder al *classpath* de la distribución Hadoop de nuestro clúster. El *classpath* contiene las rutas donde se encuentran los códigos de nuestra versión de Hadoop, y que son utilizados a la hora de generar el fichero ".jar" final que podremos ejecutar.
- Compila las clases que aparecen en nuestro código java, generando fichero ".class" asociado a cada una de ellas.
- Genera un fichero ".jar" final que aglutina todo nuestro código.
- Recibe como argumento el nombre del fichero que contiene nuestro código (quitando la extensión ".java")

Para ejecutarlo:

```
> ./compilar.bash WordCount
```

Tras este paso, se habrá generado un nuevo fichero "WordCount.jar" que contiene la aplicación Hadoop que podremos invocar.

Mejore el script para que el fichero .jar tenga otro nombre que se indique como segunda entrada al ejecutar el script.

3.6 Ejecutar la aplicación

A la hora de invocar la aplicación que hemos creado, debemos ejecutar el siguiente comando:

```
> Hadoop jar WordCount.jar uam.WordCount <fichero de entrada> <directorio de salida>
```

Donde:

- *WordCount.jar* es el fichero que hemos generado en la fase de compilación.
- *uam.WordCount* hace referencia a la jerarquía que hemos creado (*uam* es el nombre del paquete en el que hemos definido nuestra clase principal *WordCount*). Al hacer esta llamada, se ejecuta el método *main* de la clase *WordCount*.

a. Modificar la aplicación WordCount y compara con la proporcionada en los ejemplos de hadoop map-reduce

1. Modificar el ejemplo de WordCount que hemos tomado como partida, para que no tenga en cuenta signos de puntuación, ni las mayúsculas/minúsculas volver a ejecutar la aplicación.

2. Comparar resultados de la aplicación desarrollada con la que se puede ejecutar directamente en los ejemplos hadoop map-reduce

Preguntas a responder:

3.1 ¿Dónde se crea hdfs? ¿ Cómo se puede elegir su localización?

3.2 Si estás utilizando hdfs ¿Cómo puedes volver a ejecutar WordCount como si fuese single.node?

En el fragmento del Quijote

3.3 ¿Cuál son las 10 palabras más utilizadas?

3.4 ¿Cúantas veces aparece:

- El articulo “el”
- La palabra “dijo”