MapReduce en Hadoop

Tomás Fernández Pena

Máster en Tecnologías de Análisis de Datos Masivos: Big Data

Universidade de Santiago de Compostela

Tecnologías de Computación para Datos Masivos

citius.usc.es

Índice

Índice

Java MapReduce en Hadoop

Un programa MapReduce en Java debe definir 3 clases:

1. Una clase mapper
2. Una clase reducer
3. Una clase principal, con el método `main`

Pueden crearse

- como tres clases públicas separadas
- como una sola, con el método `main`, y clases internas static para mapper y reducer

Nueva y vieja API

Los programas pueden usar la nueva API (recomendado) o la antigua

- Ambas APIs son incompatibles

Utilizaremos siempre la API nueva

- Disponible desde Hadoop 0.20.0
- Definida en el paquete `org.apache.hadoop.mapreduce`
- Algunas librerías presentes en la API vieja no han sido portadas

API vieja (desaconsejada)

- Definida en el paquete `org.apache.hadoop.mapred`

WordCount Mapper

```
public class WordCountMapper
  extends Mapper<LongWritable, Text, Text, IntWritable> {
  @Override
  public void map(LongWritable key, Text value, Context ctxt)
          throws IOException, InterruptedException {
      Matcher matcher = pat.matcher(value.toString());
      while (matcher.find()) {
          word.set(matcher.group().toLowerCase());
          ctxt.write(word, one);
      }
  }
  private Text word = new Text();
  private final static IntWritable one = new IntWritable(1);
  private Pattern pat =
          Pattern.compile("\\b[a-zA-Z\\u00C0-\\uFFFF]+\\b");
}
```

WordCount Mapper

- Extiende la clase `Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT>`
- Debe sobrescribir el método `map(KEYIN key, VALUEIN value, Mapper.Context context)`
- Usa una instancia de `Context` en la que escribe los pares clave/valor de salida
- Debe utilizar tipos de datos que implementen la interfaz `Writable` (como `LongWritable` o `Text`)
  - optimizados para serialización
- Otros métodos que se pueden sobrescribir:
  - `setup(Context context)` Llamado una vez al comienzo de la tarea
  - `cleanup(Context context)` Llamado una vez al final de la tarea

WordCount Reducer

```
public class WordCountReducer
  extends Reducer<Text, IntWritable, Text, IntWritable> {
  @Override
  public void reduce(Text key, Iterable<IntWritable> values,
     Context ctxt) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable value : values) {
          sum += value.get();
      }
      ctxt.write(key, new IntWritable(sum));
  }
}
```

WordCount Reducer

- Extiende la clase `Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT>`
- Debe sobrescribir el método `reduce(KEYIN key, Iterable<VALUEIN> values, Reducer.Context context)`
- Usa una instancia de `Context` en la que escribe los pares clave/valor de salida
- Debe utilizar tipos de datos que implementen la interfaz `Writable` (como `LongWritable` o `Text`)
  - optimizados para serialización
- Otros métodos que se pueden sobrescribir:
  - `setup(Context context)` Llamado una vez al comienzo de la tarea
  - `cleanup(Context context)` Llamado una vez al final de la tarea

WordCount Driver (I)

```
public class WordCountDriver
       extends Configured implements Tool {
  public int run(String[] arg0) throws Exception {
    if (arg0.length != 2) {
      System.err.printf("Usar: %s [ops] <entrada> <salida>\n",
              getClass().getSimpleName());
      ToolRunner.printGenericCommandUsage(System.err);
      return -1;
    }
    Configuration conf = getConf();
    Job job = Job.getInstance(conf);
    job.setJobName("Word Count");
    job.setJarByClass(getClass());
    FileInputFormat.addInputPath(job, new Path(arg0[0]));
    FileOutputFormat.setOutputPath(job, new Path(arg0[1]));
```

WordCount Driver (II)

```
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setNumReduceTasks(4);

    job.setMapperClass(WordCountMapper.class);
    job.setCombinerClass(WordCountReducer.class);
    job.setReducerClass(WordCountReducer.class);

    return (job.waitForCompletion(true) ? 0 : -1);
  }
  public static void main(String[] args) throws Exception {
   int exitCode = ToolRunner.run(new WordCountDriver(), args);
   System.exit(exitCode);
  }
}
```


WordCount Driver

- Clase `Configured`: permite acceder a la configuración del sistema
  - Método `Configuration getConf()`: devuelve la configuración actual
- Clase `Configuration`: proporciona acceso a la configuración
- Clase `Job`: permite configurar, enviar y controlar la ejecución de un trabajo
- Clase `FileInputFormat<K,V>`: describe los ficheros de entrada en un trabajo MapReduce
- Clase `FileOutputFormat<K,V>`: describe los ficheros de entrada en un trabajo MapReduce

WordCount Driver

- Interfaz `Tool`: interfaz estándar para aplicaciones MapReduce, soporta el manejo de opciones en línea de comandos
- Clase `ToolRunner`: ejecuta clases que implementan la interfaz Tool
- Clase `Path`: identifica a un fichero o directorio en el FileSystem

Número de Reducers

Tener un único reducer no suele ser una buena idea

- Más reducers: más paralelismo
- Más reducers: startup más lento, más overhead de disco y red

Número óptimo de reducers

- Según la documentación, el número ideal de reduces es 0,95 o 1,75 $\times$ (<nº de nodos> $\times$ <nº máximo de contenedores por nodo>)
- El nº de contenedores por nodo es función de las características del nodo: RAM disponible, nº de cores y nº de discos
- Una aproximación:

$$\begin{aligned} \text{Ncontenedores} = \min(&2 \times \text{Ncores}, \\ &1.8 \times \text{Ndiscos}, \\ &\text{RAMdisponible}/\text{TamañoMínimoContenedor}) \end{aligned}$$

Compilación y ejecución

- Preferiblemente, crear un jar con todas las clases y ejecutarlo con:

```
yarn jar fichero.jar [opciones]
```

- Para gestionar las aplicaciones, utilizad:
  - en general, la opción `application` del comando `yarn` (`yarn application -help` para ver las opciones)
  - para trabajos MapReduce, la opción `job` del comando `mapred` (`mapred job -help` para ver las opciones)
- Más información en
  - hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YarnCommands.html
  - hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapredCommands.html

Índice

Writables

Formato de serialización de objetos de Hadoop

- Transforma objetos en un flujo de bytes para transmisión o almacenamiento

Gran número de clases que implementan el interfaz `Writable`:

- Objetos primitivos: `BooleanWritable`, `ByteWritable`, `IntWritable`, `LongWritable`, `FloatWritable`, `DoubleWritable`, `Text` (cadenas UTF-8)
- Objetos primitivos de longitud variable: `VIntWritable`, `VLongWritable`
- Colecciones: `ArrayWritable`, `ArrayPrimitiveWritable`, `TwoDArrayWritable`, `MapWritable`, `SortedMapWritable`
- Otros: `NullWritable`, `BytesWritable`, `ObjectWritable`, `GenericWritable`,

Writables a medida

Es posible implementar writables a medida

- Implementar la interfaz `Writable`
  - Implementar los métodos `write(DataOutput out)` y `readFields(DataInput in)`
- Un objeto que se quiera usar como clave debe implementar `WritableComparable`

Ejemplo de writable

```
public class MyWritable implements Writable {
  // Incluye un entero y un long
  private IntWritable counter;
  private LongWritable timestamp;

  public void write(DataOutput out) throws IOException {
    counter.write(out);
    timestamp.write(out);
  }
  public void readFields(DataInput in) throws IOException {
    counter.readFields(in);
    timestamp.readFields(in);
  }
  public static MyWritable read(DataInput in) throws IOException {
    MyWritable w = new MyWritable();
    w.readFields(in);
    return w;
  }
}
```


Clase abstracta `FileInputFormat<K,V>`

- Define métodos para añadir uno o varios paths de entrada al trabajo `addInputPath(Job job, Path path)`,`addInputPaths(Job job, String commaSeparatedPaths)`
- Clase base para diferentes formatos de entrada concretos
- El formato a usar se especifica mediante el método `setInputFormatClass` del `Job`, por ejemplo:

```
job.setInputFormatClass(TextInputFormat.class)
```

Formatos de entrada

- `TextInputFormat`: formato por defecto para ficheros de texto
- `KeyValueTextInputFormat`: ficheros de texto con estructura clave/valor
- `NLineInputFormat`: permite que los mappers reciban un número fijo de líneas de entrada
- `SequenceFileInputFormat`: secuencias de datos binarios clave/valor
- `CombineFileInputFormat`: Permite empaquetar varios ficheros pequeños en un split de entrada de un tamaño determinado
- `FixedLengthInputFormat`: Usado para leer ficheros con registros de longitud fija

Formatos de entrada

- `TextInputFormat`:
  - Formato por defecto
  - Cada mapper recibe un trozo (*split*) del fichero y lo procesa línea a línea
  - Como clave, toma el offset en bytes en el fichero desde el comienzo de la línea (`LongWritable`)
  - Como valor, toma toda la línea
- `KeyValueTextInputFormat`:
  - Igual que el anterior, pero separa cada línea en clave/valor
  - Usa tabulado como separador por defecto
  - Para cambiar, modificar en el objeto `Configuration` la propiedad

`mapreduce.input.keyvaluelinerecordreader.key.value.separator`

Salida de ficheros

Similar a la entrada: clase abstracta `FileOutputFormat<K,V>`

- Clase base para diversos formatos de salida concretos
  - `TextOutputFormat`: formato por defecto para ficheros de texto
    - Escribe líneas de texto en formato clave/valor separados por un tabulado
    - Carácter de separación modificable: `mapreduce.output.textoutputformat.separator` API vieja `mapred.textoutputformat.separator`
  - `SequenceFileOutputFormat`: secuencias de datos binarios clave/valor
  - `MapFileOutputFormat`: Un formato de salida que escribe `MapFiles`
- El formato a usar se especifica mediante el método `setOutputFormatClass` del `Job`

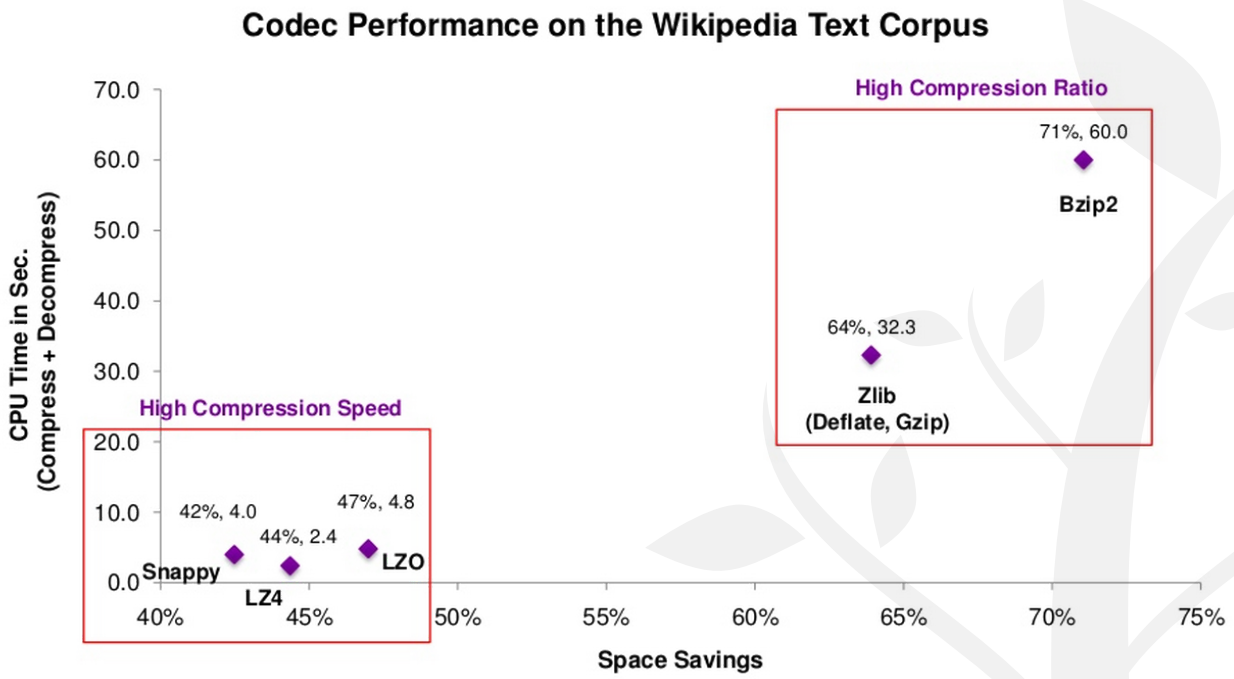
Compresión

Como vimos en el WordCount, Hadoop lee directamente ficheros comprimidos:

- Los ficheros se descomprimen automáticamente, usando la terminación del fichero para determinar que codec usar

Formato	Java/Nativo	Splittable	Codec
zlib/DEFLATE	Sí/Sí	No	`DefaultCodec`
gzip	Sí/Sí	No	`GzipCodec`
bzip2	Sí/Sí	Sí	`BZip2Codec`
LZO	No/Sí	Sí	`LzopCodec`
LZ4	No/Sí	Sí	`Lz4Codec`
Snappy	No/Sí	No	`SnappyCodec`

Compresión



Fuente: Kamat, G., Singh, S., “Compression Options in Hadoop - A Tale of Tradeoffs”, Hadoop Summit (San Jose), June 27, 2013

Compresión

Recomendaciones:

- Usar contenedores binarios como Sequence File o Avro datafile (soportan compresión y splitting)
- Usar formatos splittables
- Dividir los ficheros en trozos y comprimirlos por separado (cada trozo comprimido debería ocupar un bloque HDFS)
- No usar compresión

Ficheros Sequence

Clase `SequenceFile`

- Estructura de datos persistente para pares binarios clave/valor

Para crear un SequenceFile:

```
IntWritable key = new IntWritable(3);
Text value = new Text("tres");
Path path = new Path(uri);
writer = SequenceFile.createWriter(conf,
    SequenceFile.Writer.file(path),
    SequenceFile.Writer.keyClass(key.getClass()),
    SequenceFile.Writer.valueClass(value.getClass()));
writer.append(key, value);
IOUtils.closeStream(writer);
```

Para leer el fichero, en el programa crear un `SequenceFile.Reader` y usar `next()`

Serialización con Avro

Apache Avro es un sistema de serialización independiente del lenguaje

- Formato procesable por varios lenguajes: C, C++, C#, Java, PHP, Python y Ruby
- Puede substituir a los `Writables`
- Proporciona clases para facilitar crear programas MapReduce que usen datos Avro (usa la API vieja)

Tipos de datos en Avro

- Primitivos: null, boolean, int, long, float, double, string
- Complejos: array, map, record, enum, fixed, union

Índice

Encadenamiento de trabajos MapReduce

Es posible encadenar trabajos MapReduce

- Basta con definir varios jobs y lanzarlos unos detrás de otros
- La salida a disco de un job es leída por el siguiente

Alternativa

- Usar las clases `ChainMapper` y `ChainReducer`
- Permiten encadenar varios mappers seguidos de un reducer y cero o más mappers adicionales ([MAP+ / REDUCE MAP*])
- Reduce los accesos a disco

Para workflows complejos, se puede usar Apache Oozie o Cascading

Ordenación

Por defecto, MapReduce ordena los registros de entrada por las claves

- Uso para hacer ordenamientos de datos

Problemas:

- Las claves están entrelazadas entre los ficheros de salida
- Los valores no están ordenados

Total Sort

- Produce un conjunto de ficheros ordenados, que concatenados, formen un fichero ordenado global

Sort Secundario

- Permite que los valores aparezcan ordenados

Programación MapReduce en Hadoop Serialización y Entrada/Salida Tareas MapReduce Planificacion de tareas Alternativas a Java

Ordenación

Por defecto, MapReduce ordena los registros de entrada por las claves

- Uso para hacer ordenamientos de datos

Problemas:

- Las claves están entrelazadas entre los ficheros de salida
- Los valores no están ordenados

Total Sort

- Produce un conjunto de ficheros ordenados, que concatenados, formen un fichero ordenado global

Sort Secundario

- Permite que los valores aparezcan ordenados

CiTIUS MapReduce, TCDM 26/46

Joins de datos de diferentes fuentes

Hacer uniones de datos de diferentes fuentes es complejo en Hadoop

- Preferible usar lenguajes de alto nivel como Pig o Hive

En Hadoop hay varias soluciones:

- Si un conjunto de datos es grande y el otro pequeño, se puede replicar el pequeño en todos los nodos usando Distributed Cache
- Map-Side joins: el join se realiza antes de llegar a la función map
  - Los conjuntos de datos deben dividirse en el mismo número de particiones y estar ordenados por la clave de unión
- Reduce-side joins: el join se realiza en el reducer
  - El mapper etiqueta cada dato con su fuente y usa la clave de unión como clave de salida

Localización de dependencias

Mecanismo para copiar recursos (LocalResources) a los nodos del cluster Hadoop (DistributedCache en Hadoop v1)

- Los recursos (ficheros, archivos, librerías) se copian a todos los nodos una vez por trabajo
- Las aplicaciones pueden acceder directamente a estos recursos
- Tipos de LocalResources:
  - FILE: Ficheros normales
  - ARCHIVE: Ficheros comprimidos (jar, tar, tar.gz, zip) que descomprime el NodeManager
  - PATTERN: Híbrido de ARCHIVE y FILE (se mantiene el fichero y se descomprime parte)
- Visibilidad de los LocalResources
  - PUBLIC: accesibles por todos los usuarios
  - PRIVATE: compartidos por aplicaciones del mismo usuario en el nodo
  - APPLICATION: compartidos entre containers de la misma aplicación

Uso en línea de comandos

Si el código usa `GenericOptionParser` (o `ToolRunner`), los ficheros a copiar se indican como parámetro:

- `-files` + lista separada por comas de URIs de los ficheros
- `-archives` + ficheros comprimidos (archivos)
- `-libjars` + JARS para añadir al CLASSPATH

Ejemplo:

- `yarn jar mr.jar -files ~/fichero in out`

Uso desde la aplicación

Métodos definidos en `Job`

- `addCacheFile(URI uri)`: añade un fichero a la cache
- `addCacheArchive(URI uri)`: añade un archivo a la cache
- `addFileToClassPath(Path file)`: añade un fichero al CLASSPATH
- `addArchiveToClassPath(Path file)`: añade un archivo al CLASSPATH
- `URI[] getCacheFiles()`: obtiene los ficheros en la cache
- `URI[] getCacheArchives()`: obtiene los archivos en la cache

Contadores

Hadoop incorpora contadores para obtener datos del trabajo:

- Contadores internos: informan sobre el comportamiento de la ejecución
- Contadores definidos por el usuario (interfaz `Counter`)
  - Permiten obtener información adicional sobre las tareas
  - Definidos mediante Java enums, que permiten agrupar contadores relacionados
  - Se accede a ellos mediante el método `getCounter()` del `MapContext` o el `ReduceContext`
  - Se modifican mediante los métodos del interfaz `Counter`

WordCount2 Mapper (I)

```
public class WordCountMapper2 extends
    Mapper<LongWritable,Text,Text,IntWritable> {
  static enum CountersEnum {
    PALABRAS_INCLUIDAS
  }
  @Override
  public void setup(Context ctxt) throws IOException,
      InterruptedException {
    conf = ctxt.getConfiguration();
    caseSensitive = conf.getBoolean("wordcount.case.sensitive", true);
    if (conf.getBoolean("wordcount.skip.patterns", false)) {
      URI[] patternsURIs = Job.getInstance(conf).getCacheFiles();
      for (URI patternsURI : patternsURIs) {
        Path patternsPath = new Path(patternsURI.getPath());
        String patternsFileName = patternsPath.getName().toString();
        parseSkipFile(patternsFileName);
      }
    }
  }
```

WordCount2 Mapper (II)

```
@Override
public void map(LongWritable key, Text value, Context ctxt) throws
    IOException, InterruptedException {
  String line = (caseSensitive) ? value.toString() :
      value.toString().toLowerCase();
  for (String pattern : patternsToSkip) {
    line = line.replaceAll("\\b"+pattern+"\\b", "");
  }
  Matcher matcher = pat.matcher(line);
  while (matcher.find()) {
    String token = matcher.group();
    if (words.containsKey(token)) {
      int total = words.get(token) + 1;
      words.put(token, total);
    } else {
      words.put(token, 1);
    }
    ctxt.getCounter(CountersEnum.PALABRAS_INCLUIDAS).increment(1);
  }
}
```


WordCount2 Mapper (III)

```
@Override
public void cleanup(Context ctxt) throws
    IOException,InterruptedException {
  Iterator<Map.Entry<String, Integer>> it = words.entrySet().iterator();
  while (it.hasNext()) {
    Map.Entry<String, Integer> entry = it.next();
    String sKey = entry.getKey();
    int total = entry.getValue().intValue();
    ctxt.write(new Text(sKey), new IntWritable(total));
  }
}
```

WordCount2 Mapper (IV)

```
  private void parseSkipFile(String fileName) {
    try {
      fis = new BufferedReader(new FileReader(fileName));
      String pattern = null;
      while ((pattern = fis.readLine()) != null) {
        patternsToSkip.add(pattern);
      }
    } catch (IOException ioe) {
      System.err.println("Excepcion␣parseando␣el␣recurso␣local␣'" +
          StringUtils.stringifyException(ioe));
    }
  }
  private boolean caseSensitive;
  private Set<String> patternsToSkip = new HashSet<String>();
  private Map<String, Integer> words = new HashMap<String, Integer>();
  private Pattern pat = Pattern.compile("\\b[a-zA-Z\\u00C0-\\uFFFF]+\\b");
  private Configuration conf;
  private BufferedReader fis;
}
```


WordCount2 Driver (I)

```
public class WordCountDriver2 extends Configured implements Tool {
  public int run(String[] args) throws Exception {
    Configuration conf = getConf();
    Job job = Job.getInstance(conf);
    job.setJobName("Word␣Count␣2");
    job.setJarByClass(getClass());
    GenericOptionsParser optionParser = new GenericOptionsParser(conf,
        args);
    String[] remainingArgs = optionParser.getRemainingArgs();
    if (!(remainingArgs.length != 2 || remainingArgs.length != 4)) {
      System.err.printf("Usar:␣%s␣[ops]␣<entrada>␣<salida>␣[-skip␣
          skipPatternFile]\n", getClass().getSimpleName());
      return -2;
    }
```

WordCount2 Driver (II)

```
    List<String> otherArgs = new ArrayList<String>();
    for (int i=0; i < remainingArgs.length; ++i) {
      if ("-skip".equals(remainingArgs[i])) {
        job.addCacheFile(new Path(remainingArgs[++i]).toUri());
        job.getConfiguration().setBoolean("wordcount.skip.patterns",
            true);
      } else {
        otherArgs.add(remainingArgs[i]);
      }
    }
    FileInputFormat.addInputPath(job, new Path(otherArgs.get(0)));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs.get(1)));
```


WordCount2 Driver (III)

```
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    job.setNumReduceTasks(1);
    job.setMapperClass(WordCountMapper2.class);
    //Usa in-mapper combiner
    job.setReducerClass(WordCountReducer2.class);
    return (job.waitForCompletion(true) ? 0 : -1);
  }
  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new WordCountDriver2(), args);
    System.exit(exitCode);
  }
}
```

Ejemplo de ejecución:

```
yarn jar wordcount2.jar \
-Dwordcount.case.sensitive=true indir outdir \
-skip skipfile.txt
```

Índice

Planificación

YARN puede utilizar diferentes planificadores de tareas

- Los más usados son el *Capacity Scheduler*, y el *Fair Scheduler*

El scheduler a usar se especifica mediante la variable `yarn.resourcemanager.scheduler.class`

- Por defecto, se usa Capacity Scheduler con una única cola (denominada `default`)

Capacity Scheduler

Desarrollado por Yahoo

- Los trabajos se envían a colas, que pueden ser jerárquicas
- A cada cola se le reserva una fracción de la capacidad total
- En cada cola, los trabajos usan FIFO con prioridades
  - Permite simular un cluster MapReduce con FIFO para cada usuario u organización
- Se configura mediante el fichero `capacity-scheduler.xml`

Más información: hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html

Fair Scheduler

Desarrollado por Facebook, con el objetivo de compartir equitativamente la capacidad del cluster

- Intenta proporcionar tiempos de respuesta rápidos para trabajos cortos y QoS para trabajos en producción

Los trabajos se colocan en *pools*

- Por defecto, cada usuario tiene su propia pool
- A cada pool se le garantiza un mínimo de capacidad
  - Mínimo numero de map slots, reduce slots, y máximo número de trabajos simultáneos
- La capacidad sobrante se reparte entre los trabajos
- Es apropiativo:
  - Si un pool no recibe su capacidad mínima, el scheduler puede matar trabajos que estén usando por encima de su capacidad

Más información: hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html

Ejecución especulativa

Por defecto, la ejecución especulativa está activada

- Si una tarea se retrasa, se lanza especulativamente una copia de la misma

En un cluster muy ocupado, esto puede significar un consumo excesivo de recursos

- Se puede deshabilitar la especulación para mappers o reducers
- Más interesante para reducers
  - Un reduce duplicado implica volver a obtener las salidas de los mappers
  - Aumento del tráfico de red
- Propiedades (booleanas):
  `mapreduce.map.speculative`
  `mapreduce.reduce.speculative`

Índice

Alternativas a Java

Hadoop Streaming

- API que permite crear códigos map-reduce en otros lenguajes
- Utiliza streams Unix como interfaz entre Hadoop y el código
- Permite usar cualquier lenguaje que pueda leer de la entrada estándar y escribir en la salida estándar (Python, Ruby, etc.)
- Más información
  hadoop.apache.org/docs/stable/hadoop-streaming

Hadoop Pipes

- Interfaz C++ a Hadoop MapReduce
- Usa sockets como canal de comunicación entre el tasktracker y el proceso C++ que ejecuta el map o el reduce

Ejemplo de streaming

WordCount Mapper en Python

```
#!/usr/bin/env python3
import re, sys
pattern = '\\b[a-zA-Z\u00C0-\uFFFF]+\\b'

for line in sys.stdin:
        words = re.findall(pattern, line.lower())
        for w in words:
                print("{0}\t{1}".format(w, 1))
```

Ejemplo de streaming

WordCount Reducer en Python

```
#!/usr/bin/env python3
import sys
(last_key, total) = (None, 0)
for line in sys.stdin:
        (key, value) = line.strip().split()
        if last_key and last_key != key:
                print("{0}\t{1}".format(last_key, total))
                (last_key, total) = (key, int(value))
        else:
                (last_key, total) = (key, total+int(value))
if last_key:
        print("{0}\t{1}".format(last_key, total))
```


Ejemplo de streaming

Ejecución del código Python Nota: La opción `-file` sólo se necesita si se ejecuta en un cluster, para enviar los scripts a los nodos del cluster

```
yarn jar \
  $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
  -D mapreduce.job.reduces=2 \
  -files WordCountMapper.py,WordCountReducer.py \
  -input indir -output outdir \
  -mapper WordCountMapper.py \
  -reducer WordCountReducer.py
```

Nota: La opción `-files` sólo se necesita si se ejecuta en un cluster, para enviar los scripts a los nodos del cluster