UNIVERSIDAD COMPLUTENSE MADRID

Apache Hadoop

Enrique Martín (emartinm@ucm.es)
Sistemas de Gestión de Datos y de la Información
Master Ing. Informática
Fac. Informática

Contenidos

Bibliografía

Bibliografía

- *Hadoop: The Definitive Guide, 4th edition*. Tom White. O'Reilly (2015)
- *Hadoop in Practice, 2nd edition*. Alex Holmes. Manning (2014)

Introducción

Hadoop

- Framework de código abierto para la computación distribuida de manera escalable y confiable
- El núcleo está compuesto por 3 módulos:
  - Sistema de ficheros distribuido **HDFS** (similar al GFS de Google)
  - Hadoop **YARN**: gestor de recursos en clústeres
  - **Hadoop MapReduce**

Historia de Hadoop

- Surge en 2006 como una generalización de Apache Nutch (robot y motor de búsquedas) para aplicar MapReduce a la resolución de problemas generales, usando un sistema de ficheros distribuido
- Desde el principio, recibe el apoyo de Yahoo! (recursos y personal) para convertirse en un proyecto maduro
- En 2008, se convierte en un proyecto Apache de primer nivel
- Desde el inicio Hadoop MapReduce demuestra su potencial:
  - Abril de 2008: Yahoo! ordena 1 TB en 3,5 minutos utilizando 910 nodos
  - Noviembre de 2008: Google ordena 1 TB en 68 segundos usando su MapReduce
  - Mayo de 2009: Yahoo! Ordena 1 TB en 62 segundos usando Hadoop

Ecosistema Hadoop

- Además de HDFS, YARN y MapReduce, que forman el núcleo de Hadoop, existen otros módulos complementarios relacionados:
  - Modelos de cómputo de alto nivel (Pig, Hive, Spark)
  - Bases de datos NoSQL (HBase, Cassandra)
  - Aprendizaje Automático (Mahout)
  - Manejo de datos (Avro, Sqoop)
  - Desplegado y gestión de clústeres (Ambari)

MapReduce en Hadoop

MapReduce en Hadoop

- En Hadoop las tareas MapReduce se escriben en Java. Se debe crear un fichero JAR con todo el código necesario
- El *mapper* se define como una clase que hereda de `Mapper` y contiene el método `map()`
- El *reducer* se define como una clase que hereda de `Reducer` y contiene el método `reduce()`.
- Los métodos `map()` y `reduce()` emiten las parejas a través de un objeto de tipo `Context`: `context.write(clave, valor)`
- Todos los valores que se emitan en las parejas deben ser *serializables*. Existen tipos predefinidos como `Text` o `IntWritable` para los tipo más comunes
- A continuación veremos el código de una tarea MapReduce para contar las repeticiones de cada palabra usando MapReduce en Hadoop

Mapper en Hadoop

```
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()) {
      String word_aux = itr.nextToken();
      word.set(word_aux);
      context.write(word, one);
    }
  }
}
```

Reducer en Hadoop

```
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);
  }
}
```

Configurar la tarea MapReduce en Hadoop

```
JobConf conf = new JobConf();
Job job = Job.getInstance(conf);
job.setJarByClass(WordCount.class);
job.setMapperClass(TokenizerMapper.class);
job.setReducerClass(IntSumReducer.class);

// Declaración de tipos de salida para el mapper y reducer
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(IntWritable.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

// Archivos de entrada y directorio de salida
FileInputFormat.addInputPath(job, new Path( "../texto.txt" ));
FileOutputFormat.setOutputPath(job, new Path( "salida" ));

// Ejecuta la tarea y espera a que termine. El argumento
// booleano es para obtener información verbosa
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

Hadoop Distributed Filesystem (HDFS)

HDFS

- Componente de Hadoop para el sistema de ficheros distribuido: los ficheros están distribuidos entre diferentes máquinas o nodos de una red
- Sigue ideas similares al GFS de Google
- Diseñado para soportar **ficheros muy grandes**: cientos de GB o TB
- Proporciona buen rendimiento para aplicaciones de *escritura única-lecturas múltiples*
  - Alta velocidad para leer el fichero **completo**, no así su latencia
  - Adecuado cuando los análisis involucran casi todos los datos del fichero, o se repiten varios análisis

HDFS

- Diseñado para ejecutarse en máquinas de bajo coste (*commodity hardware*)
- Estas máquinas se apilan en *racks*, que se interconectan para forman clústeres

Fuente: `https://www.flickr.com/photos/br1dotcom/5740649659/`

HDFS

- Al tener muchas máquinas de bajo coste, lo más probable es que diariamente falle alguna
- La caída de un equipo no puede suponer la pérdida del acceso a datos almacenados en HDFS
- Por lo tanto, HDFS está diseñado para tener una **alta tolerancia a fallos utilizando replicación**

Bloques en HDFS

- Un bloque es la unidad mínima de datos que se puede transferir en un sistema de archivos o disco duro
- Tamaños de bloque normales son:
  - Discos duros: 512 bytes
  - Sistemas de ficheros: pocos KB
- En HDFS el tamaño de bloque por defecto son **64 MB / 128 MB**. *¿Por qué son tan grandes los bloques en HDFS?* **Para «amortizar» el tiempo de búsqueda**:
  - Un disco duro con una velocidad de 600 MB/s (un valor posible para un disco SATA 3.0) tardará 0.8 ms en transmitir un bloque de 500 kB. Si tarda 10ms en buscarlo, tardará más en buscarlo que en transferirlo
  - Con un bloque de 128 MB, tardaríamos 10ms + 213 = 223 ms en leerlos, es decir, el tiempo de búsqueda sería aproximadamente un 4 % del tiempo total

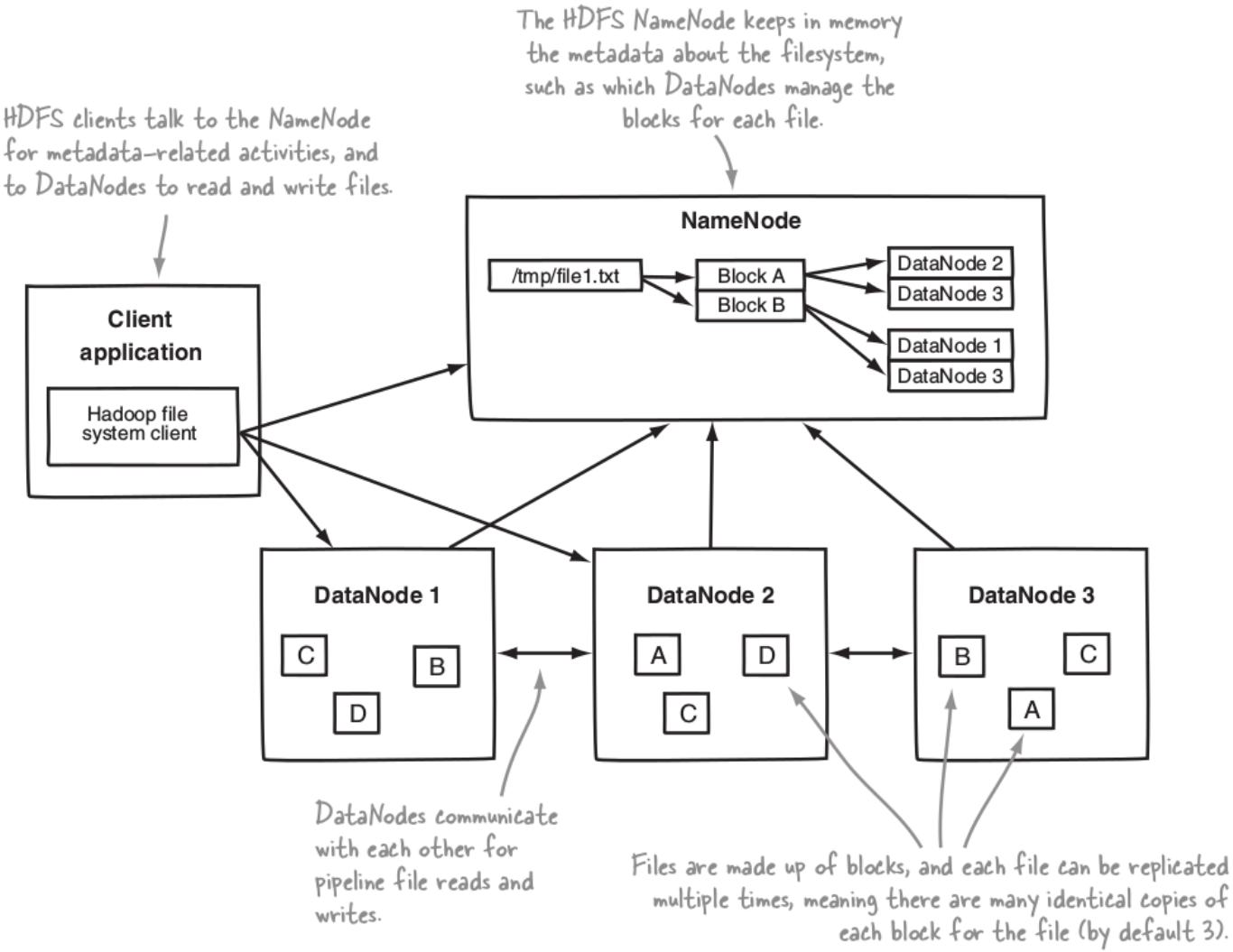
Bloques en HDFS

- La abstracción de bloques es cómoda porque:
  - Permite dividir un fichero en distintos nodos: cada nodo almacenará uno o varios bloques
  - Simplifica el almacenamiento: tamaño fijo, los metadatos (permisos, dueño, acceso) se pueden almacenar por separado
  - Facilita la replicación para conseguir tolerancia a fallos y alta disponibilidad

Organización de HDFS

- En un clúster HDFS existen dos tipos distintos de nodos con una relación principal-secundarios
- **Namenode**: equipo principal que almacena toda la información de los ficheros
  - Metadatos: permisos, accesos
  - Índice de dónde están los bloques
- **Datanodes**: equipos secundarios que almacenan los bloques de datos

Organización de HDFS



Fuente: Hadoop in Practice, Alex Holmes. Manning (2012)

Organización de HDFS

- Los *namenodes* son muy importantes, así que hay que asegurarse que toleran bien los fallos. Hay dos alternativas:
  - Escribir periódicamente el estado del sistema de ficheros, para poder recuperarlo si este falla
  - Tener *namenodes* secundarios que copian el estado, y que estarían disponibles si el principal falla
- En sistemas con una necesidad de almacenamiento muy alta, se pueden utilizar *namespace volumes*:
  - Cada *namenode* se encargará (de manera independiente) de una localización particular: `/usr`, `/share`, etc.
  - Perder un *namenode* significa perder una localización concreta
  - Los *datanodes* pueden almacenar bloques de diferentes *namenodes*

Lectura de ficheros en HDFS

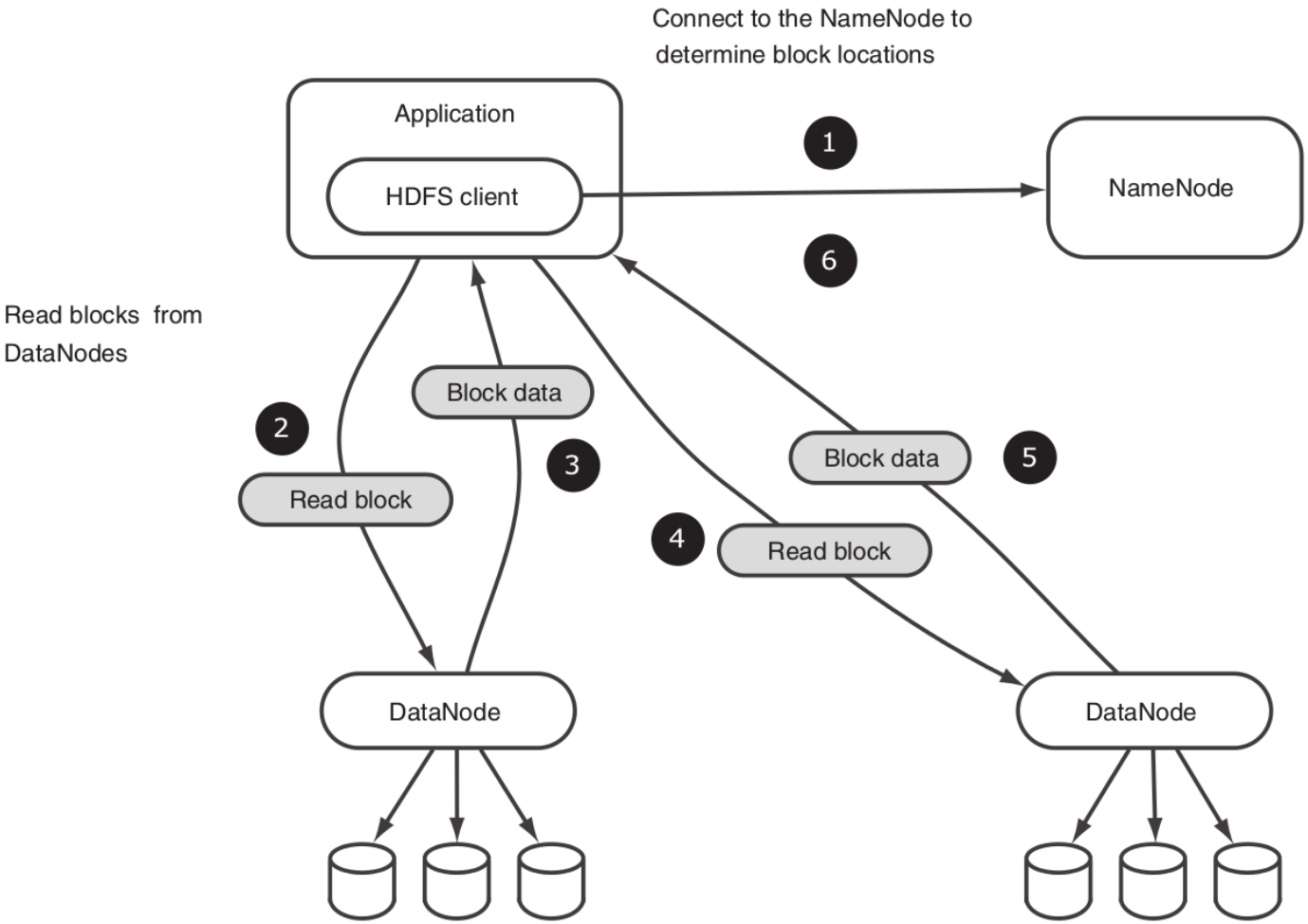


Figure C.3 HDFS interactions for reading a file

Fuente: `https://stackoverflow.com/questions/9258134/about-hadoop-hdfs-file-splitting`

Lectura de ficheros en HDFS

- Al principio, se piden los *datanodes* donde están los primeros 10 bloques del archivo. Los nodos se devuelven ordenados por **cercanía**
- En HDFS *cercanía* significa:
  1. Mismo equipo (si es un *datanode*)
  2. Equipo en el mismo *rack*
  3. Equipo en otro *rack*
- Se repite el proceso de conectar a *datanodes* cercanos para leer los bloques, pidiendo los *datanodes* de los 10 siguientes bloques según se necesitan
- Si alguna lectura/conexión falla, se prueba otro *datanode* y se recuerda para no probarlos más adelante
- Se comprueban los *checksums* de los bloques. Si hay algún error, se informa al *namenode* y se prueba otro *datanode*

Escritura de ficheros en HDFS

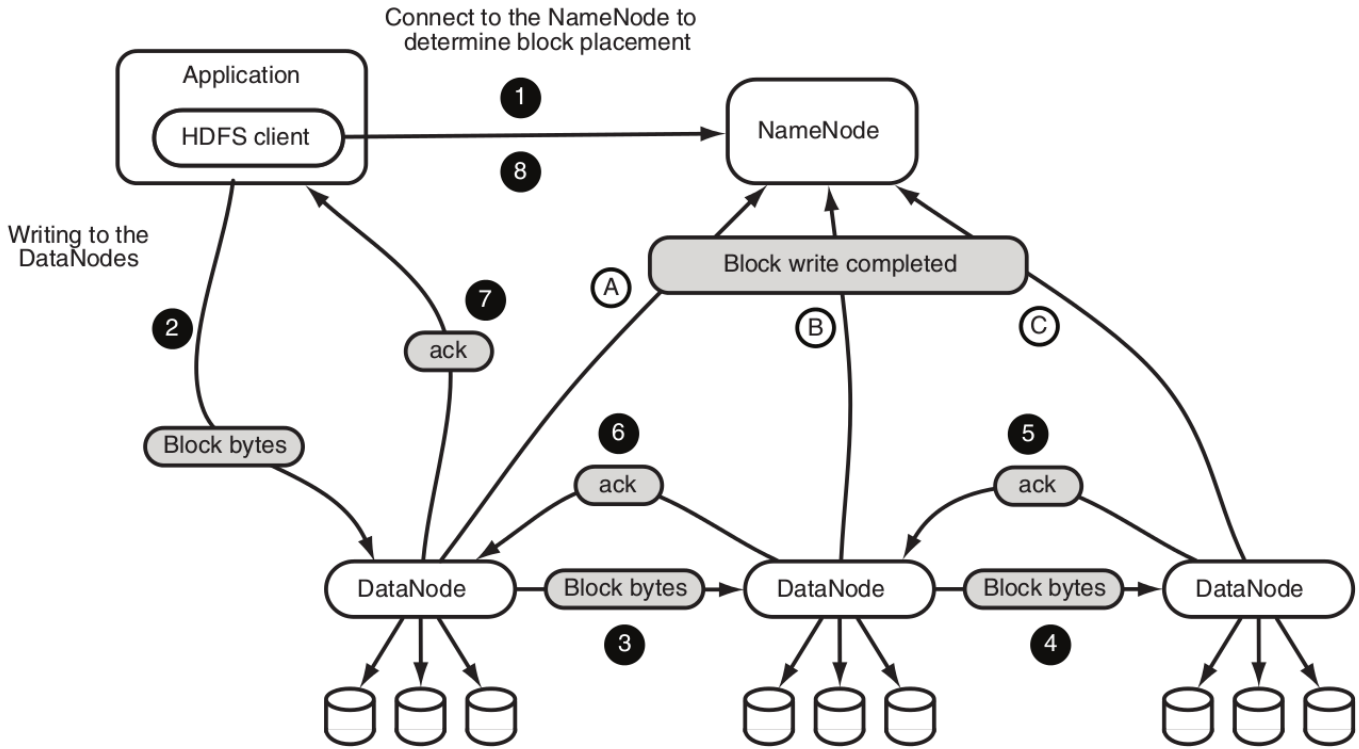


Figure C.2 **HDFS write data flow**

Fuente: `https://stackoverflow.com/questions/9258134/about-hadoop-hdfs-file-splitting`

Escritura de ficheros en HDFS

- Primero se contacta con el *namenode* para crear el fichero
- El namenode proporciona una lista de *datanodes* para el primer bloque teniendo en cuenta la cercanía. **Al menos una copia estará fuera del *rack***
- Cuando hay suficientes datos, se envían al primer datanode
- El primer *datanode* almacena los datos y los reenvía datos al siguiente, y así sucesivamente
- Cada *datanode* debe confirmar que ha escrito correctamente cada bloque. Solo cuando todos lo confirman la escritura, un bloque se da por escrito