Software para el análisis de datos

Paralelismo y Sistemas Distribuidos

Tipos de software

- Modelos de programación y *runtimes*
- Almacenamiento de datos
- Procesado de datos

Requerimientos

- Se prioriza la escalabilidad horizontal y la tolerancia a fallos sobre el alto rendimiento
- Tipos de arquitectura
  - Cliente-Servidor (master-slave)
    - modelo *Thin-client*: el cliente sólo hace de interfaz con el usuario
    - modelo *Fat client:* el servidor sólo se encarga de la gestión de datos y el cliente se encarga de implementar la lógica de la aplicación
  - Peer-to-Peer: arquitectura decentralizada en la que cualquier nodo puede recibir peticiones e implementar el servicio
  - (…)

Modelos de programación

- Pensados para ejecutar las mismas funciones sobre una gran cantidad de datos → Computación guiada por los datos
  - Crear partición de datos
  - Ejecutar de manera concurrente la misma función sobre cada partición
- Ejemplos
  - Apache Hadoop
  - Spark

Modelos de programación: MapReduce

- Propuesto por Google e inspirado en los lenguajes de programación funcionales
- Pensado para aplicaciones masivamente paralelas: mismos cálculos sobre grupos independientes de datos
- El procesado de datos se divide en dos partes
  - Map:
    - Su entrada es una porción de datos identificados por una clave
    - Su resultado es un resultado (parcial) identificado por una clave que puede ser distinta a la de entrada
  - Reduce:
    - Es opcional y ejecuta la agregación de los resultados parciales (generados por la función de *map*)
    - Su entrada es una lista de valores generados por el map que tienen la misma clave

1.5

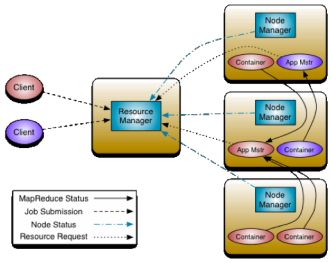
Frameworks para MapReduce

- El framework que implenta el modelo debe ofrecer
  - Interfaz para el particionado de datos
  - Gestión de ejecución y de paralelismo
  - Integración con almacenaje de datos
  - Están pensados para ejecutarse en datacenter/clouds de grandes dimensiones
  - Ofrecen escalabilidad y tolerancia a fallos
- Ejemplo: Apache Hadoop

1.6

Apache Hadoop: Arquitectura

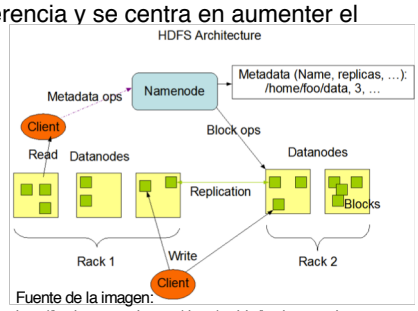
- Implementa un entorno de ejecución MapReduce junto con un sistema de ficheros distribuido (HDFS → Hadoop Distributed File System)
- Arquitectura del entorno MapReduce
  - Basada en Apache YARM (Yet Another Resource Manager)
  - Componentes
    - 1 ResourceManager (master): planificación de tareas y reinicia el container del master de la aplicación si falla
    - 1 NodeManager per node (slave): responsable de crear los *containers* necesarios para ejecutar la aplicación
    - 1 MRAppMaster por aplicacion: se ejecuta en un *container* y comprueba el estado de las tareas, reiniciando las que hayan fallado



Fuente de la imagen: https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/YARN.html

Apache Hadoop: HDFS

- Objetivos
  - Cada nodo *Comodity* hardware
    - Pensado para clusters con gran cantidad de nodos
  - Tolerancia a fallos
  - Grandes conjuntos de datos
  - Tipo de uso: se escribe una sola vez y se leen muchas
    - Simplifica la gestión de coherencia y se centra en aumenter el throughput
- Arquitectura
  - Namenode
    - Master
    - Implementa el espacio de nombre y la asignación de bloques a Datanodes
  - Datanodes
    - Slave
    - Implementa los accesos a bloques



Fuente de la imagen: http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html#Introduction

Apache Hadoop: HDFS

- Fichero divididos en bloques del mismo tamaño (excepto el último)
- Los bloques están distribuidos entre los Data nodes
- Replicación para dar toleracia a fallos
  - Política de asignación de réplicas es muy relevante
- Se intenta favorecer la localidad en el acceso a datos
  - Normalemente los DataNodes corren en los nodos de procesado
- Los ficheros sólo se pueden escribir una vez, y sólo se permite un escritor al mismo tiempo
- Tamaño de bloque y número de réplicas configurable a nivel de fichero
- Tolerancia a fallos del server
  - Multiples Namenodes (uno activo y el otro en *stand by*
  - NameNode activo puede mantener varias copias de los metadatos
  - Sistema de ficheros comparitdo entre los NameNodes para acelerar el proceso de recuperación

1.9

Apache Hadoop: MapReduce

- Pasos de una aplicación MapReduce
  - Dividir los datos de entrada en porciones independientes
    - El entorno proporciona clases que implementan el particionado (InputFormat) pero cada usuario puede implementar la suya propia
  - Procesar cada una en paralelo (*map task*)
  - El entorno ordena los resultados parciales y se los envía a las tareas de *reduce* como datos de entrada
- Los datos de entrada y de salida están organizados en parejas <key,value>
  - conjunto de pares <key,value>
  - las clases de key y de value deben implementar el interfaz *Writable* (para ser serializables) y la clase de key también debe implementar *WritableComparable* para la fase de ordenación
- Flujo de datos:

  <k1,v1>→ **map**→<k2,v2>→**combine**→ <k2,v2>→**reduce**→<k3,v3>
- Mínima implementación: función de map y/o de reduce

1.10

Apache Hadoop: API

- map
  - Recibe como entrada un conjunto de pares <key,value> y produce como salida otro conjunto de pares <key,value>, que pueden ser de distintio tipo y/o valores
  - Se lanza un mapper para cada partición de los datos de entrada (*InputSplit*)
- combiner
  - Función opcional para agrupar valores de salida con la misma key de los mappers locales (para minimizar mensajes a los reducers). El input es la salida de los mappers, y el output los pares <key, lista_valores>
- reducer
  - La salida del map (o de los combiner) son ordenados y particionados para mandárselos a los reducers
  - El número de reducers no depende de los datos de entrada y es un parámetro configurable (se puede hacer que sean 0 si no se necesita fase de reduce)
  - El programador puede decidir cómo hacer las particiones para cada reducer si implementa su propio *partitioner*

1.11

Ejemplo de código: WordCount (I)

```
public class WordCount {

  /* here definition of TokenizerMapper.class and IntSumReducer

      public static void main(String[] args) throws Exception {
          Configuration conf = new Configuration();
          Job job = Job.getInstance(conf, "word count");
          job.setJarByClass(WordCount.class);
          job.setMapperClass(TokenizerMapper.class);
          job.setCombinerClass(IntSumReducer.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);
      }

}
```

1.12

Ejemplo de código: WordCount (II)

```
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);
    }
}
}
```

1.13

Ejemplo de código: WordCount (III)

```
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);
}
}
```

1.14