- Memòria del PFC -
Big data: Ús de mostreig per al càlcul de solucions adaptades a restriccions en el temps d'execució

David Castro Martínez

23/06/2014

Contents

1 Introducció

En l'era de la informació cada vegada emmagatzemem més dades, registrem tot i a tothom, tant com sigui possible. Volem saber les dades bursàtils de tot el planeta i registrem totes les transaccions econòmiques. Volem saber cóm es comporta la gent, on viatgen, com viatgen, qué diuen i on ho diuen. Volem saber què compren, què els hi agrada, on decideixen usar el seu temps i com ho fan. Però en l'era de la informació qui té èxit no és qui té moltes dades sinó qui sap usar-les millor.

Tenim eines que ens permeten estudiar aquestes dades, podem analitzar-les, buscar patrons, fer recomptes... per intentar extreure'n conclusions que ens siguin útils. Però simplement perquè el volum d'informació és massiu fa d'aquest un procés costós en temps, energia i recursos. Aquest projecte presenta i desenvolupa una nova eina que permet obtenir resultats útils a un cost molt menor.

La clau que ens permet treballar amb volums de dades tan grans és el paral·lelisme. Per resoldre una consulta sobre un conjunt gran de dades el que fem es dividir-les en parts i computar la consulta en cadascuna d'aquestes parts de manera independent i paral·lela, llavors recopilem i combinem tots aquests resultats parcials per obtenir la resposta total. Així podem donar resposta a preguntes que trigaríem molt més a respondre si intentéssim analitzar totes les dades de cop. Tot i així el cost es significatiu, es necessita un temps considerable i un sistema de computadors que pugui treballar en paral·lel de manera coordinada.

La situació que ens plantegem llavors és que segons com sigui la natura de l'estudi que volem fer pot ser que els resultats obtinguts de processar només una fracció de les dades totals ja ens siguin útils. Per exemple, si les dades consisteixen en un conjunt d'entrades on cada entrada es la elecció que ha fet cada persona d'entre varies opcions, una consulta que ens pot interessar és saber quina proporció de persones ha triat una certa opció A. Si processem totes les dades obtindríem la resposta, però si en processem només una fracció d'aquestes el que obtindrem és una resposta parcial que tot i no ser la real és suficientment bona com a per a fer-se una idea de la proporció real. És a dir, ens plantegem que el resultat de processar només una mostra de les dades totals ens pot donar resultats útils a una fracció del cost que suposaria processar totes les dades.

Aquest projecte implementa una modificació en la tecnologia que s'usa actualment per resoldre aquest tipus de consultes de manera que es pugui activar un mecanisme de mostreig per obtenir resultats parcials a una fracció del cost total. Aquesta tecnologia s'anomena Hadoop.

2 Hadoop

2.1 La Big Data

La Big Data és el terme general que es fa servir per referir-se a la situació que Hadoop vol gestionar; es refereix a l'escenari que es dona quan s'ha d'emmagatzemar i treballar amb volums de dades tan grans que les bases de dades relacionals que normalment es fan servir no són capaces de gestionar.

El problema principal no és tant l'emmagatzemament en si mateix com en

poder després recuperar les dades i processar-les en un temps raonable. Els sistemes que implementen la gestió de la Big Data, com Hadoop, es basen en aprofitar el paral·lelisme. La idea fonamental que es vol aplicar es que si una maquina triga un cert temps en recuperar i processar les dades, cent maquines haurien de trigar cent vegades menys. És a dir, el sistema ha de ser escalable horitzontalment, a l'afegir més maquines el temps de execució ha de veure's reduït proporcionalment. Això permet una aproximació molt senzilla al problema original: si el volum de dades amb el que cal treballar es tan gran que el temps d'execució no és acceptable, només cal afegir maquines en paral·lel fins que poguem processar les dades en un temps raonable. Aquí és on entren el clústers de computació. Els centres de Big Data consisteixen en una xarxa de maquines que no sols emmagatzemen les dades sinó que també les processen quan cal treballar sobre aquestes.

Hi ha una altra diferència fonamental de la Big Data amb les bases de dades tradicionals, a banda del volum de dades, i és que la Big Data són dades no estructurades. Les bases de dades relacionals, les més usades, mantenen dades que tenen referències entre si, les entrades coneixen la estructura del sistema i creen relacions entre elles. Aquestes bases de dades són capaces de recuperar ítems particulars ràpidament, ja que fan ús de les relacions entre les dades per navegar i filtrar quines entrades han de recuperar. Per contra, la Big Data no és estructurada, és a dir les dades no tenen relacions entre elles, es tracta simplement d'un conjunt molt gran d'entrades que contenen alguna informació particular que existeix autònomament, que no depèn del valor d'altres entrades.

La idea de la Big Data és mantenir un munt de dades molt simples que seran útils quan es recorrin consecutivament, entrada per entrada, consultant els seus valors per extreure'n alguna conclusió, com per exemple algun tipus de recompte. Així doncs, mentre que les bases de dades relacionals estan dissenyades per a recuperar items arbitraris molt ràpidament, els sistemes amb Big Data estan dissenyats per a fer un accés consecutiu d'un gran volum de les dades.

Per exemple, un base de dades relacional típica podria contenir usuaris, i aquests usuaris tindrien un conjunt d'informació i ítems associats. Una operació útil en aquesta base de dades podria ser recuperar un usuari en particular per poder obtenir tota la informació relacionada amb aquest quan ens és necessària. Un exemple de Big Data podria ser el registre de temperatures a diferents llocs al llarg del temps. Cada entrada indicaria simplement el lloc, el moment i quina és la lectura de la temperatura, aquestes entrades no requereixen de cap altra informació, són dades completes per si mateixes. Una operació útil en aquest model podria ser voler saber la mitjana de temperatura en una certa localitat al llarg d'un període de temps. Llavors el sistema recorreria totes les dades de principi a fi i aniria recopilant les temperatures de les entrades que siguin del lloc i moment demanats.

De fet per aprofitar el paral·lelisme, els problemes que es poden solucionar amb la Big Data han de tenir la propietat de poder resoldre's en un subconjunt de les dades, sense importar en quin context es troben, de manera que es pugui definir la solució del problema total com una recopilació de les solucions del problema en cada una de les parts de les dades.

Ara ja sabem quina és la natura de la Big Data i quin tipus d'operacions en volem fer, però cóm s'executen exactament aquestes operacions amb un sistema de màquines paral·leles? La resposta el dona un model anomenat MapReduce.

2.2 El model MapReduce

El MapReduce és el paradigma de processament de la Big Data en un clúster. Consisteix en dues fases: la fase de map i la fase de reduce. La fase de map recorre, filtra i classifica totes les dades d'entrada, la fase de reduce llavors pren aquestes dades classificades i les combina, les "redueix", per obtenir-ne resultats.

Més en particular, quan es vol executar una tasca MapReduce en un clúster, aquest és el procès que segueixen els seus nodes:

- El node principal pren la entrada de dades i la divideix en parts, llavors reparteix aquestes parts a la resta de nodes.
- Cada node de processament rep una part de la entrada i executa la fase de map, és a dir, filtra quines dades de quines entrades són útils. Llavors a aquestes entrades de dades escollides se les assigna una clau, de manera que el resultat de tot plegat és un conjunt de parelles <clau, dades>.
- Llavors aquestes parelles s'envien als nodes que executen les operacions de reduce(), agrupant-les per clau. Els nodes reduce executen la funció reduce() per extreure'n la informació necessaria.
- Per últim el sistema recull les sortides dels reduces i les combina per donar la reposta final.

Les dades es troben originalment repartides pels mateixos nodes que les processaran, quan el node principal fa la divisió de les dades per a fer els maps té en compte la localitat de les dades. És a dir, intenta assignar cada part de les dades al mateix node on es troba. Això evita el cost extra d'haver de moure-les des de el node d'emmagatzemament fins al node de processament assignat.

2.3 Apache Hadoop

Apache Hadoop és un software obert per a la gestió de Big Data escrit en Java. És a dir, s'encarrega tant de emmagatzemar les dades com d'implementar el model MapReduce en el clúster que estigui gestionant. A continuació es detallen aquests dos aspectes

2.3.1 HDFS

El "Hadoop distributed file system" o HDFS és el sistema de fitxers que fa servir Hadoop per guardar les dades. Aquest és un sistema distribuït, és a dir el sistema de fitxers no es troba en una sola maquina sino que abasta el conjunt de nodes del clúster. De fet, un sol fitxer es pot trobar dividit en varies maquines, a més l'HDFS crea replicacions de les dades per millorar la seguretat, velocitat en recuperar les dades i augmentar la localitat del processament.

D'entre tots els nodes destaca el NameNode. El NameNode s'encarrega de mantenir la metadata del sistema, és a dir és la maquina que registra quins fitxers es troben en el sistema i com està repartit i replicat entre tota la resta de nodes.

Tal i com s'ha exposat en l'apartat anterior, el sistema HDFS està dissenyat per optmitzar les operacions de lectura de grans volums de dades seqüencialment.

La lectura aleatòria de dades particulars és, comparativament, una operació costosa. A més, és un sistema dissenyat per al processament de consultes sobre les dades estàtiques i no per al manteniment de dades dinàmiques en el temps. És a dir, s'espera que les dades s'escriguin una vegada i es llegeixin moltes, cada vegada que es vulgui fer un estudi. La reescriptura de les dades no és una operació afavorida en el dissseny del sistema i comporta costos extres.

Un bloc és la unitat mínima de memòria que un sistema de fitxers pot escriure o llegir. La mida d'un bloc d'un sistema de fitxers domèstic és de 4 KiloBytes, comparativament la mida per defecte d'un bloc de HDFS és de 64 MegaBytes. Cada vegada que un sistema de fitxers ha de llegir o escriure un bloc ha de pagar un cost en temps; tenir blocs tan grans en l'HDFS permet que la lectura de grans volums de dades és realitzi accedint a un nombre més petit de blocs, reduint així el temps necessari per a completar la operació.

Així doncs per treballar a l'HDFS cal connectar-se amb el NameNode i indicar-li les operacions que es volen realitzar. Aquest, llavors, traduirà les instruccions en operacions particulars de lectura o escriptura de blocs repartits per la resta de nodes del sistema.

2.3.2 El JobTracker i els TaskTrackers

A Hadoop, un "job" és la unitat de treball que es vol realitzar. Consisteix d'unes dades d'entrada, un programa MapReduce i uns fitxers de configuració del sistema. El programa MapReduce és bàsicament una implementació de les funcions `map()` i `reduce()`, és a dir, està indicant què vol fer amb les dades d'entrada. L'execució d'un job està dividida en "tasks" i n'hi han de dos tipus "map tasks" i "reduce tasks", que corresponen a les subdivisions indicades prèviament.

El JobTracker és un node del sistema encarregat de gestionar i organitzar els diferents Jobs que es volen executar en el clúster. Un TaskTracker és un node treballador del sistema que executa les funcions de `map()` i/o de `reduce()`. Els TaskTrackers envien regularment senyals d'estat al JobTracker que controla que el job es completi satisfactoriament, per exemple si un TaskTracker fallés el JobTracker reassignaria les tasques d'aquest TaskTracker a un altre node per a completar l'execució del job.

Així doncs anem a reescriure el procés MapReduce explicat previament en termes de Hadoop:

- El JobTracker pren la entrada i la divideix en "splits", crea una map task per cada split i les reparteix als TaskTrackers
- Cada TaskTracker executa la funció `map()` donada pel programa MapReduce introduit. Cada entrada de l'split a processar s'anomena "record", la funció de `map()` llavors crea les parelles `<clau, record>` necessàries.
- Llavors aquestes parelles s'envien als TaskTrackers que executen les operacions de `reduce()`, agrupant-les per clau.
- Per últim, el sistema recull les sortides dels reduces i les combina per generar la repsosta final que s'escriu en un fitxer en el mateix HDFS.

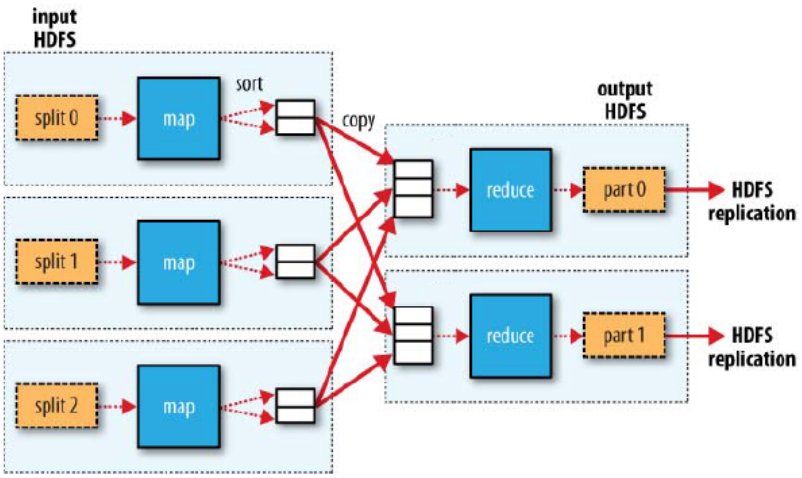

Figure 1: Esquema map-reduce

Un split no és el mateix que un bloc. Un bloc és la unitat estructural amb la que l'HDFS treballa. Un split es defineix tan sols com un subconjunt de les dades d'entrada. Però com que els splits han de ser llegits i enviats al seu TaskTracker la opció més natural i eficient és fer que els splits equivalguin a blocs del sistema.

Tot i així es possible definir qualsevol mida i criteri per als splits. La interfície `InputFormat` proporciona les dues funcions que s'han d'implementar per a dividir i llegir la entrada:

- `InputSplit[] getSplits(JobConf job, int numSplits)`
  Divideix la entrada en la llista de splits per al sistema

- `RecordReader<K,V> getRecordReader(InputSplit split, JobConf job, Reporter reporter)`
  Obté el RecordReader que s'usarà per recórrer els records d'un l'split.

La classe abstracta `FileInputFormat` proporciona la implementació estàndar de `getSplits`. I existeixen diverses subclasses de `FileInputFormat` que implementen el record reader segons varis criteris: cada línia és un record, un grup de línies és un record...

2.4 Ús pràctic de hadoop

Per tocar una mica de peus a terra, a continuació descriure, en termes generals, com és i com s'usa Hadoop a la pràctica. Un cop descarregada, la carpeta de Hadoop té el següent aspecte:

```
$ ls -l hadoop-0.20.203.0/
bin
build
build.xml
```

```
c++
CHANGES.txt
conf
contrib
docs
hadoop-ant-0.20.203.0.jar
hadoop-core-0.20.203.0.jar
hadoop-examples-0.20.203.0.jar
hadoop-test-0.20.203.0.jar
hadoop-tools-0.20.203.0.jar
ivy
ivy.xml
lib
librecordio
LICENSE.txt
NOTICE.txt
README.txt
src
webapps
```

Els elements que ens importen són:

- La carpeta `bin` conté els binaris de la aplicació.
- La carpeta `src` conté el codi font, mentre que la de `build` conté els fitxers compilats d'aquest.
- La carpeta `conf` conté fitxers de configuració del sistema.

La resta són llibreries que necesita el codi, fitxers necessaris per a la compilació i uns quants fitxers `.jar` amb exemples i utilitats preparades.

Per a usar Hadoop el primer que cal és tenir-lo funcionant en un sistema HDFS. Per a aconseguir això cal tenir la configuració adecuada en elsfitxers de `conf` que d'entre d'altres coses indiquen sobre quina xarxa es vol desplegar Hadoop i l'HDFS. Si és la primera vegada caldrà també formatejar els sistema de fitxers per a crear un HDFS nou. Llavors només cal arrencar tots els threads de Hadoop mitjançant `bin/start-all.sh`, en essència s'initialitzen 4 threads:

- El `JobTracker` que és el punt d'entrada i qui programa els jobs a executar al sistema.
- El `TaskTracker` que gestiona els nodes treballadors.
- El `NameNode` que manté la metadata del sistema de fitxers.
- El `DataNode` que es qui en realitat executa les operacions en el sistema de fitxers, consultant al `NameNode`. Hi sol haver més d'un `DataNode`.

Un cop el sistema està ben configurat i en marxa ja es pot treballar amb Hadoop en aquest clúster. Per fer-ho cal crear un programa MapReduce que implementi les funcions `map()` i `reduce()`, i un `main()` per a construir i llençar el Job. Llavors només cal cridar aquest programa amb el binari de Hadoop i passar-li un fitxer (HDFS) d'entrada.

```
bin/hadoop jar elMeuPrograma.jar $FITXERS_INPUT $DESTI_OUTPUT
```

Per a posar la entrada a l'HDFS o recuperar la sortida, aquest mateix binari és el que serveix per fer crides al sistema de fitxers. Per exemple, per copiar fitxers cap a l'HDFS podem usar:

```
bin/hadoop dfs -copyFromLocal fitxer_local $DESTI_HDFS
```

Podem trobar crides de manera similar per a totes les commandes essencials d'un sistema de fitxers.

Amb això ja tenim una idea més clara de com es intervenen els conceptes teorics darrere del paradgime del MapReduce i l'HDFS a la practica.

3 Objectius del projecte

L'objectiu del projecte es implementar i estudiar la viabilitat d'usar sampleig com a mecanisme d'obtenció de resultats parcials. El sistema de sampleig consisteix en escollir un subconjunt dels splits d'entrada i només processar aquests. Això implica la modificació del codi font de Hadoop, la seva configuració i potencialment el codi dels jocs de proves. A més, aquest mecanisme ha de poder ser activat i desactivat sense haver de recompilar Hadoop, és a dir, el sampleig ha de ser una opció disponible mitjançant variables en els fitxer de configuració de Hadoop i/o pel codi particular del job a executar.

El clúster Arvei de la FIB servirà d'entorn de treball i proves, permetent així l'execució del sampling en un sistema Hadoop distribuït real.

4 Implementació

4.1 Modificacions a Hadoop

Tal i com hem vist, la classe responsable de crear els splits ha de implementar la interfície `InputFormat`, que ja està parcialment implementada en la classe `FileInputFormat`. El codi que trobem aquí per a la creació dels splits és el següent:

FileInputFormat.java

```
public List<InputSplit> getSplits(JobContext job
                                  ) throws IOException {
  long minSize = Math.max(getFormatMinSplitSize(),
      getMinSplitSize(job));
  long maxSize = getMaxSplitSize(job);

  // generate splits
  List<InputSplit> splits = new ArrayList<InputSplit>();
  List<FileStatus>files = listStatus(job);
  for (FileStatus file: files) {
    Path path = file.getPath();
    FileSystem fs = path.getFileSystem(job.getConfiguration());
    long length = file.getLen();
```

```
    BlockLocation[] blkLocations = fs.getFileBlockLocations(file, 0,
        length);
    if ((length != 0) && isSplitable(job, path)) {
      long blockSize = file.getBlockSize();
      long splitSize = computeSplitSize(blockSize, minSize, maxSize);

      long bytesRemaining = length;
      while (((double) bytesRemaining)/splitSize > SPLIT_SLOP) {
        int blkIndex = getBlockIndex(blkLocations,
            length-bytesRemaining);
        splits.add(new FileSplit(path, length-bytesRemaining, splitSize,
                                 blkLocations[blkIndex].getHosts()));
        bytesRemaining -= splitSize;
      }

      if (bytesRemaining != 0) {
        splits.add(new FileSplit(path, length-bytesRemaining,
            bytesRemaining,
                 blkLocations[blkLocations.length-1].getHosts()));
      }
    } else if (length != 0) {
      splits.add(new FileSplit(path, 0, length,
          blkLocations[0].getHosts()));
    } else {
      //Create empty hosts array for zero length files
      splits.add(new FileSplit(path, 0, length, new String[0]));
    }
  }

  // Save the number of input files in the job-conf
  job.getConfiguration().setLong(NUM_INPUT_FILES, files.size());

  LOG.debug("Total # of splits: " + splits.size());
  return splits;
}
```

Aquest mètode retorna la llista de splits amb els que es crearan els tasktrackers. Cal notar que un `InputSplit` és un objecte de metadata, és a dir, no conté les dades d'entrada és tan sols un objecte que indica on es troben les dades de l'split que representa: a quin fitxer i a quin offset dins del fitxer.

Els splits creats per aquest mètode tenen de mida

```
splitSize = computeSplitSize(blockSize, minSize, maxSize)
```

que el que fa es prendre el resultat de:

```
splitSize = max(minimumSize, min(maximumSize, blockSize))
```

però, per defecte, tenim que `minimumSize < blockSize < maximumSize` així doncs, el tamany d'un split és exactament el tamany d'un block: 64 MegaBytes.

Cal notar també que per cada arxiu el mètode comprova si es divisible (`isSplitable(...)`), és a dir, és possible prendre tot un fitxer com un split

si per algun motiu no es volgués subdividir la entrada. Això també posa de relleu, però, que un split no pot abarcar més enllà d'un fitxer.

Per tant, per implementar el sistema de sampling, només cal modificar aquest codi per a que es vagi saltant regions de cada fitxer d'entrada segons ens convingui. Però no el modificarem aquí el codi ja que aquesta és una classe abstracta i, per tant, la subclasse que s'usi podria sobreescriure aquesta mètode el que anul·laria les nostres modificacions.

Així doncs, cal veure com es determina quina subclasse que de `InputFormat` és la que Hadoop ha d'usar. Només hi han dues opcions o bé ha d'estar en el codi del programa MapReduce o bé es un parametre de configuració.

En aquest cas trobem que la classe Job conté el mètode que busquem:

```
public void setInputFormatClass(Class<? extends InputFormat> cls)
                        throws IllegalStateException
```

En resum, només cal fer servir una classe que extengui `FileInputFormat` i modificar `getSplits(...)` com convingui. El que segueix és un exemple d'un sampling que pren només els splits parells.

Sampling splits parells

```
import java.io.IOException;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.InputSplit;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reporter;

public class SamplingFileInputFormat extends FileInputFormat
{
    @Override
    public InputSplit[] getSplits(JobConf job, int numSplits)
            throws IOException
    {
        InputSplit[] splits = super.getSplits(job, numSplits);
        InputSplit[] newSplits = new InputSplit[(splits.length/2];
        for(int i = 0; i < splits.length; ++i)
        {
            if (i%2 == 0) newSplits[i/2] = splits[i];
        }
        return newSplits;
    }

    @Override
    public RecordReader getRecordReader(InputSplit split, JobConf job,
            Reporter reporter) throws IOException
    {
        //retornar el RecordReader que convingui
    }

}
```

Aquest és un sampling del 50%, el fet de que es prenguin els splits alternadament, en comptes de prendre tota la primera meitat i deixar-se la segona, té una certa importància, ja que potser les dades tenen algun ordre inherent i prendre la primera meitat podria esbiaixar el resultat parcial obtingut. Per exemple, continuant amb el cas de les temperatures, si estiguessin ordenades cronològicament llavors prenent la primera meitat estaríem tenint en compte només temperatures antigues, mentre que prenent els splits parells tenim informació més variada.

Per activar el sampling, només caldria afegir la següent línia al codi que volem executar:

```
job.setInputFormatClass(SamplingFileInputFormat.class);
```

Voldríem, ara, que el sampleig no estigués "hardcodejat" sinó que es pogués activar i desactivar fàcilment des de la configuració, sense haver de recompilar Hadoop.

A la carpeta `conf` podem trobar el fitxer `core-site.xml` que conté propietats del sistema i on podem afegir les nostres pròpies variables. Per exemple, podem definir una variable "sampling.factor" on podem indicar quina proporció de sampling volem usar, i podem definir que si val 0, o si la variable no està assignada, no faci sampling.

core-site.xml

```
<configuration>
        ...
        <property>
                <name>sampling.factor</name>
                <value>2</value>
        </property>
</configuration>
```

Llavors, només cal modificar el codi anterior per a que tingui en compte el valor d'aquesta variable. Per consultar la variable cal recuperar l'objecte `Configuration` del `JobContext` (que es passa per paràmetre) i fer:

```
Configuration conf = jobContext.getConfiguration();
int samplingFactor = conf.getInt(sampling.factor, 0); //0 s el valor
    que agafa si la variable no est definida
```

Anem a veure un exemple complet amb aquestes modificacions incloses. Per fer-ho agafarem l'exemple canònic de Hadoop: wordcount. Wordcount compta cuantes vegades apareix cada paraula de l'entrada. Així doncs el seu programa MapReduce, que implementa `map()`, `reduce()` i un main() que el configura i el llença, és el següent:

WordCount.java

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
```

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

public class WordCount {

  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);
      }
    }
  }

  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);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf,
        args).getRemainingArgs();
    if (otherArgs.length != 2) {
      System.err.println("Usage: wordcount <in> <out>");
      System.exit(2);
    }
    Job job = new Job(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(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Anem a crear una altra classe `SamplingFileInputFormat.java` amb la mateixa idea fonamental però amb dues diferències importants:

- Aquesta vegada extendrem la classe `TextInputFormat`, que a la seva vegada hereda de `FileInputFormat`. L'única diferència és que `TextInputFormat` implementa el mètode `getRecordReader(...)` que ens havíem deixat per implementar abans. `TextInputFormat` considera cada línia com un record.
- Llegirem la variable `sampling.factor` de la configuració. El que farem és que si la variable no està definida o val 0, no usarem sampling (100% del fitxer). Per contra, si tenim un factor definit, aquest indicarà quants splits agafem de cada grup de 10. Així, per exemple, un `sampling.factor=5` representa un mostreig del 50%.

Sampleig n/10 splits

```
 private static class SamplingFileInputFormat extends TextInputFormat
{
    @Override
    public List<InputSplit> getSplits(JobContext job) throws IOException
%     {
        List<InputSplit> splits = super.getSplits(job);
        int factor = job.getConfiguration().getInt("sampling.factor", 0);
        if (factor == 0) return splits;
        List<InputSplit> sampleSplits = new ArrayList<InputSplit>();
        int count = 0;
        for (int i = 0; i < splits.size(); ++i)
        {
          if (i%10 == 0) count = 0;
          if (count < factor)
          {
            sampleSplits.add(splits.get(i));
            count++;
          }
        }
        return sampleSplits;
    }
}
```

Un petit apunt: aquesta vegada `getSplits(...)` retorna una `List` de splits i no un array de splits. Això és degut a que estem usant una altra versió de la

classe `InputFormat` que té aquest petit canvi en el prototip de `getSplits(...)`, però són funcionalment equivalents. Però això no té cap trascendencia.

I ara hem de indicar-li al Job que usi aquesta classe:

```
job.setInputFormatClass(SamplingFileInputFormat.class)
```

Amb tot queda:

WordCount.java amb sampling

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;

import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.JobContext;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import java.util.List;
import java.util.ArrayList;

public class WordCount {

  //codi de map() sense canvis
  //codi de reduce() sense canvis

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf,
        args).getRemainingArgs();
    if (otherArgs.length != 2) {
      System.err.println("Usage: wordcount <in> <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setInputFormatClass(SamplingFileInputFormat.class); //usem la
        classe on hem implementat el sampling
    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(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
```

```
  }

  private static class SamplingFileInputFormat extends TextInputFormat
  {
      @Override
      public List<InputSplit> getSplits(JobContext job) throws
          IOException
       {
          List<InputSplit> splits = super.getSplits(job);
          int factor = job.getConfiguration().getInt("sampling.factor",
              0);
          if (factor == 0) return splits;
          List<InputSplit> sampleSplits = new ArrayList<InputSplit>();
          int count = 0;
          for (int i = 0; i < splits.size(); ++i)
          {
            if (i%10 == 0) count = 0;
            if (count < factor)
            {
              sampleSplits.add(splits.get(i));
              count++;
            }
          }
          return sampleSplits;
      }
  }
}
```

Així per a qualsevol job de hadoop que vulguem executar només cal activar l'ús de la classe que hem definit en el main i canviar el fitxer `core-site.xml` segons el sampleig que vulguem.

4.2 Integració al clúster

En aquest apartat exposaré de quina manera es traslladen aquestes modificacions locals a un clúster d'execució distribuïda.

El clúster Arvei funciona amb un sistema de cues gestionades amb el software "Sun Grid Engine". Aquest sistema permet que els usuaris enviïn feines al sistema, llavors aquest reparteix els recursos i assigna els temps d'execució a les diferents peticions a mida que pot anar servint-les. Per executar al clúster cal moure a un disc comú (tots els nodes hi tenen accès) tot el software, configuració i fitxers d'entrada que es vulguin usar en la petició. S'ha de definir un fitxer `job.sh` que contingui les tasques a fer i enviar-lo a la cua d'execució del sistema mitjançant la comanda `qsub`. En particular, per executar un job de Hadoop cal cridar:

```
qsub -pe hadoop $NUM_NODES -l hadoop_master=1 job.sh
```

Al fitxer `job.sh` cal definir unes variables de configuració que indiquen on és Hadoop i amb quina configuració s'ha d'executar. Amb això, el primer que farà el sistema és inicialitzar els processos de Hadoop als nodes tal i com hem vist a l'apartat 2.4.

capçalera de job.sh

```
 # Nom del job
#$ -N hadoopJob1
# Indicamos el shell a usar:
#$ -S /bin/bash
# Variables de configuraci per a indicar on es troba Hadoop.
#$ -v JAVA_HOME=/Soft/java/jdk1.6.0_30,
     HADOOP_HOME=/scratch/nas/1/$USER/hadoop-0.20.203.0,
     HADOOP_CONF=/scratch/nas/1/$USER/conf_hadoop/conf


export CONF=/scratch/nas/1/$USER/conf_hadoop/conf


### Executar la feina a fer ###
```

En el nostre cas, volem executar un job de Hadoop sobre uns fitxers d'entrada, però Hadoop només pot llegir o escriure en el sistema HDFS. És per això, que el procés que hem de fer és el següent:

- Copiar les dades d'entrada des del disc comú al sistema HDFS
- Executar el job de hadoop amb aquesta entrada
- Recuperar la sortida, copiant-la des del HDFS al disc comú

Un simple script que executa això és:

```
### Definim uns fitxers de treball dins de l'HDFS:
INPUT=$JOB_NAME"_"$JOB_ID"_IP"
OUTPUT=$JOB_NAME"_"$JOB_ID"_OP"

### 1) Copiem l'entrada al sistema HDFS
${HADOOP_HOME}/bin/hadoop --config $CONF dfs -copyFromLocal
    "/scratch/nas/1/$USER/input" $INPUT


### 2) Executem un job de Hadoop amb entrada $INPUT i sortida a
    $OUTPUT/hadoop-output
${HADOOP_HOME}/bin/hadoop --config $CONF jar
    $HADOOP_HOME/build/hadoop-examples-0.20.203.1-SNAPSHOT.jar
    wordcount $INPUT $OUTPUT/hadoop-output


### 3) Copiem la sortida al disc com:
OUTPUT_DIR=$USER_HOME/$OUTPUT
mkdir $OUTPUT_DIR
${HADOOP_HOME}/bin/hadoop --config $CONF dfs -copyToLocal
    $OUTPUT/hadoop-output $OUTPUT_DIR
```

Un cop es faci "submit" del fitxer `job.sh` a la cua del sistema és possible monitoritzar el seu estat mitjançant la comanda `qstat`. També es possible cancelar-lo amb la comanda `qdel`. A continuació hi ha l'script que executa una de les proves que discutiré en el següent apartat. Fa essencialment el mateix

però a més prèn mesures de temps, redirigeix la sortida estandar i d'error de la tasca de hadoop i executa la feina de Hadoop més d'una vegada:

big job.sh

```
# Cambiamos el nombre del job
#$ -N bigHadoop50
# Indicamos el shell a usar:
#$ -S /bin/bash
# Indicamos las versiones a usar de hadoop (imprescindible):
#$ -v JAVA_HOME=/Soft/java/jdk1.6.0_30,
      HADOOP_HOME=/scratch/nas/1/dcastro/hadoop-0.20.203.0,
      HADOOP_CONF=/scratch/nas/1/dcastro/conf_hadoop/conf5


export CONF=/scratch/nas/1/$USER/conf_hadoop/conf5

USER_HOME=/scratch/nas/1/$USER
RESULTS=$USER_HOME/"results_"$JOB_NAME"_"$JOB_ID
mkdir $RESULTS

### Definimos unos directorios de trabajo dentro del HDFS:
INPUT=$JOB_NAME"_"$JOB_ID"_IP"
OUTPUT=$JOB_NAME"_"$JOB_ID"_OP"

T_INPUT0=`date +"%s"`
### Copiamos los libros al sistema de ficheros HDFS de hadoop:
${HADOOP_HOME}/bin/hadoop --config $CONF dfs -copyFromLocal
    "/scratch/nas/1/$USER/librosBig" $INPUT
${HADOOP_HOME}/bin/hadoop --config $CONF dfs -copyFromLocal
    "/scratch/nas/1/$USER/librosBig/big_book.txt" $INPUT/big_book2.txt
T_INPUT1=`date +"%s"`
TOTAL_TIME_INPUT=`expr $T_INPUT1 - $T_INPUT0`
echo $TOTAL_TIME_INPUT > $RESULTS/inputtime.txt

${HADOOP_HOME}/bin/hadoop --config $CONF dfs -ls $INPUT >$RESULTS/ls.txt

ACUMULATED_TIME=0

for i in {1..5}; do
T0=`date +"%s"`

### Contamos las palabras, usando el ejemplo que viene con hadoop:
${HADOOP_HOME}/bin/hadoop --config $CONF jar
    $HADOOP_HOME/build/hadoop-examples-0.20.203.1-SNAPSHOT.jar
    wordcount $INPUT $OUTPUT"/libros-output_"$i >>$RESULTS/stdout.txt
    2>>$RESULTS/stderr.txt

echo "***" >>$RESULTS/stdout.txt
echo "***" >>$RESULTS/stderr.txt

T1=`date +"%s"`

TOTAL_TIME=`expr $T1 - $T0`
ACUMULATED_TIME=`expr $ACUMULATED_TIME + $TOTAL_TIME`
```

```
echo $TOTAL_TIME > $RESULTS"/time_"$i.txt
done

echo $ACUMULATED_TIME > $RESULTS/acumulated_time.txt

### Copiamos los datos del disco de hadoop HDFS a nuestra cuenta en el
    NAS:
OUTPUT_DIR=$USER_HOME/$OUTPUT
mkdir $OUTPUT_DIR
${HADOOP_HOME}/bin/hadoop --config $CONF dfs -copyToLocal $OUTPUT
    $OUTPUT_DIR
```

5 Resultats

A continuació discutiré els resultats d'algunes proves realitzades a Arvei amb wordcount i diverses configuracions de sampleig.

Les proves consistixen en execucions del wordcount modificat a diversos percentatges de sampleig. L'entrada és de 45 GigaBytes, aproximadament, i usarem 8 nodes del cluster per a processar la consulta. Mesurarem el temps d'execució.

La sortida d'una execució consisteix en una llista de parelles `<paraula, nombre de repeticions>`. Per exemple, les primeres línies de l'execució sense sampleig (100% del fitxer) són:

Output wordcount 100%

```
 !Mal   20802
"Al     20804
"Cuando 41608
"Cuidados       20806
"De     41608
"Defects,"      20798
"Desnudo        20798
"Dijo   20798
"Dime   20806
"Don    20798
```

Com que l'entrada és de fet un llibre concatenat amb si mateix moltes vegades, si observem la sortida de l'execució amb 50% de sampleig veiem que els valors són, aproximadament, la meitat que l'execució sense sampleig:

Output wordcount 50%

```
 !Mal   10476
"Al     10476
"Cuando 20952
"Cuidados       10476
"De     20952
"Defects,"      10470
"Desnudo        10470
"Dijo   10470
"Dime   10476
```

Anem a veure quant triguen 5 execucions de cada cas:

	100% fitxer	50% fitxer
Temps d'execució	1676	863
(segons)	1661	835
	1674	847
	1651	832
	1649	834
Mitjana	1662,2	842,2

Veiem que de mitjana triga 1662 segons (27.7 minuts) en processar el 100% del fitxer mentre que amb un sampling del 50% el temps cau a la meitat amb 842 segons.

Més en detall, els 45 GB d'entrada suposen 684 splits, i, per tant, cada node de mitjana està processant 85.5 splits. Pel cas amb sampling els splits es veuen reduits a 344. El motiu pel que no és la meitat exacta és perque la entrada en realitat està dividida en 3 fitxers que sumen un total de 45 GB, i, com hem vist, els splits es generen fitxer a fitxer, per tant, si un fitxer no té un nombre de blocs múltiple de 10 el sampling no és exacte. Mai s'agafaran fraccions de blocs per arribar al % demanat, això seria contraproduent ja que és més rapid treballar amb splits que són blocs.

Cal notar que el cost de copiar els fitxers d'entrada a l'HDFS té un cost considerable, de mitjana 2000 segons. No hi ha cap diferència entre els diferents casos ja que aquest cost està determinat per dos agents: el volum de dades (45 GB) i el nombre de nodes (8). Això sí, es pot observar que és un valor molt variable, ha près valors des de 1326 fins a 2991 segons, depenen de la càrrega del sistema.

Repetint aquestes proves, podem obtenir la següent gràfica representant la mitjana de temps de execució per samplings del 100%, 50%, 40%, 30%, 20%, 10%.

La execució de les proves amb diferent nombre de nodes no sembla canviar el patró de decreixement. En particular 5 execucions amb 4 nodes:

	100% fitxer	50% fitxer	20% fitxer
Temp d'execució	2680	1357	524
	2640	1294	390
	2660	1300	388
	2598	1304	441
	2607	1287	412
Mitjana	2637	1308,4	431

5.1 Conclusions

El resultats mostren que el sampleig està funcionant adequadament. Els splits del fitxer estan sent inclosos i exclosos del processament de dades depenent del valor de la variable de configuració amb la que s'executi.

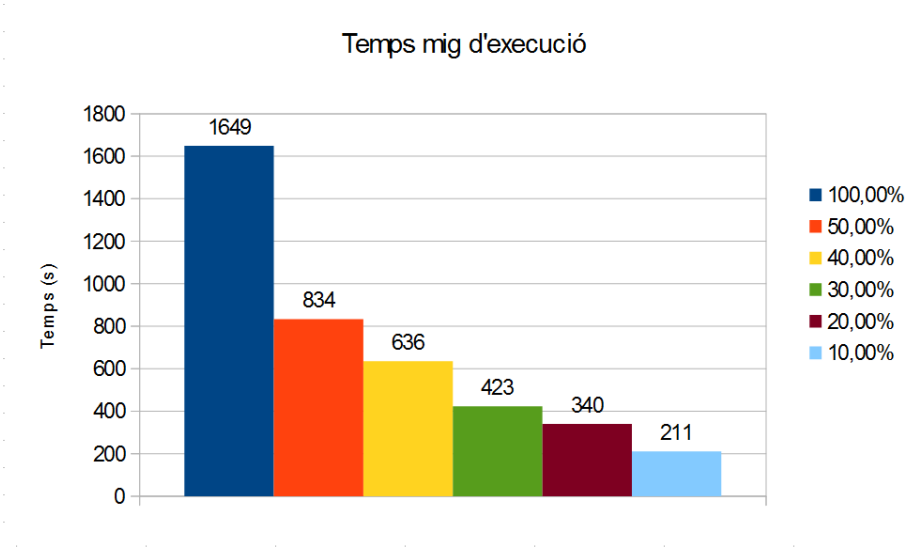

Figure 2: Temps d'execució

La velocitat a la que decreix el temps d'execució a mida que es va fent samplings més petits és molt bona. És pràcticament una relació directa 1:1 entre el temps d'execució i el factor de sampling. Es podria haver donat el cas que el sistema tingués uns certs costos inherents que no escalessin tan bé amb el tamany de les dades.

No és així, doncs els resultats indiquen que el sistema escala molt bé. Un fitxer amb una mida del N% ($0 < N < 100$) respecte d'un altre s'executarà en un N% del temps. Que aquest decreixement en el tamany de l'entrada estigui causat pel sistema de mostreig, en comptes de ser simplement fitxers més petits, no sembla tenir cap repercusió negativa.

Crec que aquest és un resultat molt interessant perque permet jugar amb una nova dimensió a l'hora enfrontar-se a un tasca de Big Data. Si el temps d'execució esperat és prohibitiu o si el sistema té molta càrrega de treball, abans l'única solució sería afegir més hardware per incrementar el paral·lelisme. Ara, a més, podríem fer servir el mecanisme de sampleig per reduir el temps d'execució, i que a diferència d'afegir hardware, redueix el cost computacional total necesssari per processar el job.

6 Planificació

6.1 Pla previst

A continuació especifico la planificació original del projecte, des de l'aprenentatge de Hadoop i de l'entorn de treball fins a les proves finals.

6.1.1 Desenvolupar el sampling bàsic

El primer objectiu important del projecte és la modificació del codi font de Hadoop per tal de que executi el sampling que volem. És a dir, que en comptes de treballar sobre tots els blocs (divisions) que composen les dades es prengui un de cada N blocs, s'executin els càlculs sobre aquests i s'obtinguin resultats només en base a aquest subconjunt de les dades.

6.1.2 Integració a Arvei

En aquest punt, quan el nucli de la tecnologia que vull desenvolupar ja està prou madur i permet fer execucions de prova, és el moment adient per a traslladar-lo al cluster i veure com cal configurar l'entorn de treball per fer tests i mesures empíriques. Aconseguir integrar a Arevi principalment significa aconseguir tres coses:

- Primer portar el codi, compilar-lo i aconseguir llençar una simple execució de prova en el sistema.
- Aconseguir un major control sobre les dades de entrada que es volen usar. En particular veure com cal treballar en el cliuster per tal de executar Hadoop sobre un volum gran de dades. Això es un objectiu rellevant perquè la creació de les dades en si mateixes es una tasca que porta temps i consumeix una cuantitat de recusos significativa.
- Aconseguir informació sobre com es comporta el cluster en les execucions: temps de execució, quins blocs s'han usat, quants nodes del cluster s'han fet servir, com s'ha dividit la feina...

6.1.3 Estudi i millora de la localitat

Veient com es comporta el sistema al clúster és possible estudiar com està treballant. En particular es pot veure si el sistema està aconseguint un alt grau d'execució local. És a dir, cada node del clúster executa càlculs sobre diferents blocs, si es dona el cas que els blocs de dades amb els que ha de treballar estan emmagatzemats en el mateix node llavors no cal perdre temps en portar les dades d'un altre lloc i la execució és més ràpida.

6.1.4 Mesura del comportament del sistema i documentació

Per últim, cal fer proves i mesures que mostrin com es comporta el sistema amb diverses configuracions, sampleigs i tamany de les dades d'entrada i veure, així, com de efectiu és el mecanisme de sampling i si hi ha alguna situació en particular on es especialment efectiu o inefectiu.

La programació de les tasques es pot veure en el següent diagrama de Gantt.

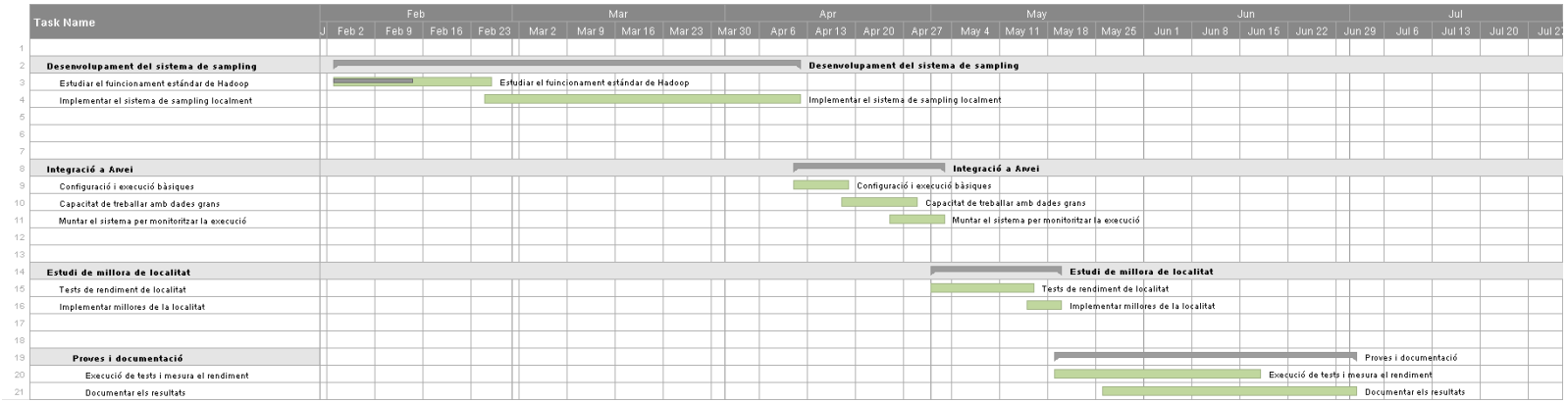

Figure 3: Planificació prevista

6.2 Execució del pla

Principalment hi ha hagut una desviació respecte del pla previst a l'informe previ.

Una potencial millora considerada per al sistema de sampleig era canviar el criteri de selecció dels blocs escollits per tal de millorar la localitat de la execució. És a dir, intentar triar blocs que siguin locals als nodes que estan lliures a mida que va avançant la execució, en comptes de blocs arbitraris. Això potencialment podria fer que alguns casos poguem escollir un bloc que s'executarà de manera local per sobre d'altres que causarien un overhead en la xarxa en haver de transportar el bloc des del node d'emmagatzemament fins al node de processament. Val a dir, que aquesta seria una millora sobre el que ja intenta fer Hadoop per si mateix, que sempre intenta maximitzar la localitat de la execució.

Però aquesta idea no és coherent amb la implementació particular de Hadoop. La determinació dels splits es fa en la preparació del job, no durant la execució, durant aquesta, Hadoop ja assigna amb preferència als nodes disponibles els blocs locals. Així doncs, els processos de determinar els splits i l'assignació d'aquests es fa en moments molt diferents de la execució. Intentar fer un sistema que combini ambdós requereix reconsiderar i reconstruir la manera com Hadoop aproxima el problema de repartició de nodes, aquesta és una tasca complexa i que no té garanties de que resulti en la creació d'un sistema més eficient. A més sempre cal considerar que el sampleig es una opció, i que cal preservar el funcionament original en cas de que no estigui activat.

És per això que fer modificacions per intentar maximitzar encara més la localitat d'una execució amb Hadoop queda fora de l'abast d'aquest projecte. Per tant, les tasques definides a l'informe previ que assignaven un període de temps en estudiar i implementar aquests canvis han quedat simplificades en un temps de estudi que no crea a subtasques de implementació ni testeig.

Així doncs el diagrama de Gantt quedaría de la següent manera:

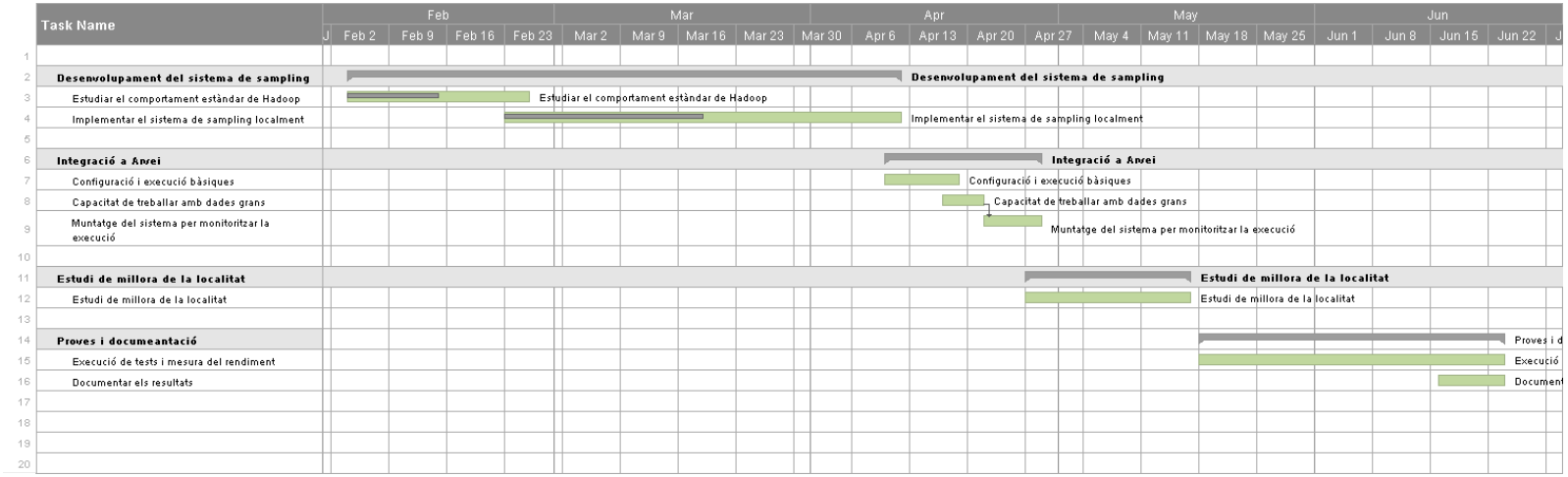

Figure 4: Execució del pla

6.3 Recompte i cost

Tasca	# hores
Estudiar el comportament estàndar de Hadoop	280
Implementar el sistema de sampling localment	175
Configuració i execució a Arvei	70
Capacitat de treballar amb dades grans	35
Muntatge del sistema per monitoritzar la execució	35
Estudi de millora de la localitat	105
Execució de tests i mesura del rendiment	175
Documentar els resultats	35
Total:	700

El cost monetari del projecte està determinat pel nombre d'hores, que a 40 €per hora fa un total de 28000 €. A més caldria considerar si el temps d'ús del clúster té un cost associat, en cas de que per exemple fos d'un servei de lloguer per hores. En aquest cas han sigut necessaris l'ús de 8 nodes del clúster durant 100h (no consecutives).

7 Impacte socioeconomic

Hadoop, i l'anàlisi de la big data en general, té un cost monetari i energètic associat. Per tant, qualsevol àmbit on es pugui aplicar aquest projecte inclourà aquests costos inherents. Però precisament si s'adoptés el mecanisme de sampling el canvi essencial estaria en la reducció d'aquests costos. Per una banda es minimitzaríen els costos energètics, reduint així l'impacte mediambiental d'aquesta activitat, i per l'altra disminuiria el cost monetari de l'execució.

Aquesta és una eina que poden fer servir les empreses i organitzacions que treballin amb la big data quan els hi convingui. Per exemple, gràcies a la possibilitat de poder obtenir resultats preliminars amb facilitat, és possible anar intentant diferents plantejaments per a resoldre el problema. A més, la reducció del temps de execució no tan sols impacta positivament en els costos del projecte sinó que també redueix la càrrega del sistema, permetent així que s'executin altres feines que haurien de ser posposades altrament, el que millora l'eficiència general de l'empresa.

En resum l'aplicació, quan sigui adient, del mecanisme de mostreig pot tenir un impacte positiu en els costos econòmics, computacionals i en la productivitat de la societat.