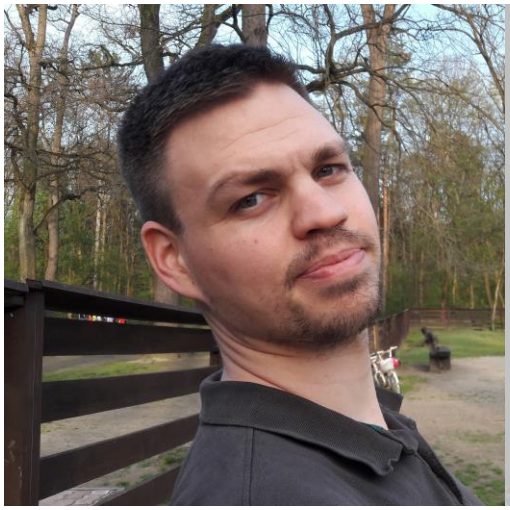


Dr. Hegedűs Péter, Dr. Ferenc Rudolf

Nagyméretű adatbázisok

Jelen tananyag a Szegedi Tudományegyetemen készült az Európai Unió támogatásával.

Projekt azonosító: EFOP-3.4.3-16-2016-00014

Szegedi Tudományegyetem
Cím: 6720 Szeged, Dugonics tér 13.
www.u-szeged.hu
www.szechenyi2020.hu

A MapReduce programozási modell

Összefoglalás

Ez a kombinált videó és olvasólecke iránymutatást ad arra vonatkozólag, hogyan telepítsd fel az Apache Hadoop rendszert Windows vagy Linux környezetre, valamint bemutatja azt, hogyan működik a MapReduce programozási modell a gyakorlatban ezen ökoszisztémán belül. Megtudhatjuk, hogyan kell egyszerű MapReduce algoritmust írni Java nyelven, valamint hogyan lehet azt lefordítani és futtatni Hadoop környezetben.
A lecke fejezetei:

- 1. fejezet: **Apache Hadoop telepítése és beüzemelése saját gépen (videó)**
- 2. fejezet: **MapReduce működése Hadoop-ban, word count feladat implementálása (olvasó)**
- 3. fejezet: **Komplexebb adatfeldolgozó MapReduce példa (olvasó)**

Téma típusa: **gyakorlati**
Olvasási idő: **40 perc**

1. fejezet
Apache Hadoop és MapReduce beállítása, IntelliJ IDEA integráció

https://www.youtube.com/watch?v=g7Qpnmi0Q-s

https://www.youtube.com/watch?v=WdIiTgYI5QI

2. fejezet
Apache Hadoop MapReduce

Ahogy korábban már láttuk, a Hadoop MapReduce egy keretrendszer, amely lehetővé teszi olyan alkalmazások készítését, amelyek hatalmas mennyiségű adatot (több terra byte-os adathalmazok) párhuzamosan dolgoznak fel nagyméretű hagyományos számítógépekből álló klasztereken (több ezer csomópont) megbízható és hibatűrő módon.

Egy MapReduce *job* általában felosztja a bemeneti adatkészletet független részekre, amelyeket a leképező (*map*) feladatok teljesen párhuzamosan dolgoznak fel. A keretrendszer rendezi a leképezések kimeneteit, majd ezután a *reduce* feladatok ezt feldolgozzák. Általában a feladat bemenete és kimenete egy fájlrendszerben kerül tárolásra. A keretrendszer gondoskodik a feladatok ütemezéséről, megfigyeléséről és a sikertelen feladatok újbóli végrehajtásáról.

Egy minimális MapReduce alkalmazáshoz meg kell adnunk a bemeneti/kimeneti adatok helyét és meg kell valósítanunk a *map* és *reduce* függvényeket a megfelelő interfészek és / vagy absztrakt osztályok implementálásával. Ezek és más job paraméterek képezik az ún. *job konfigurációt* .

A Hadoop *job kliens* ezután elküldi a feladatot (jar / végrehajtható stb.) és a konfigurációt a `ResourceManager`-nek, amely kiosztja a szoftvert / konfigurációt a worker node-oknak, elvégzi a feladatok ütemezését és megfigyelését, valamint gondoskodik az állapot- és diagnosztikai információknak a klienshez történő visszacsatolásáról.

Bár a Hadoop keretrendszert Java nyelven írták, a MapReduce alkalmazásokat nem kell feltétlenül Java-ban írni.

- A Hadoop streaming egy olyan segédprogram, amely lehetővé teszi a felhasználók számára, hogy feladatokat hozzanak létre és futtassanak bármilyen futtatható programmal (pl. Shell segédprogramokkal), mint mapper és / vagy reducer.
- A Hadoop Pipes egy SWIG kompatibilis C ++ API a MapReduce alkalmazások (nem JNI alapú) megvalósításához.

Word Count példa

Kezdjük a MapReduce programozási modellel való ismerkedést egy klasszikus feladaton keresztül. Adott egy vagy több szövegdokumentum, és a feladat az, hogy számítsuk ki az egyes szavak előfordulási gyakoriságát a teljes szöveghalmazra. Ez egy olyan feladat, amit viszonylag könnyen be tudunk illeszteni a MapReduce filozófiába. Lássuk a feladat megoldását egészben a Hadoop MapReduce keretrendszert felhasználva.

```
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;

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();
    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);
  }
}
```

A program fordítása és parancssori futtatása az alábbi módon történik (feltétel a Hadoop keretrendszer telepítése vagy docker-ben történő futtatása, lásd az első videó leckét, illetve az Apache Hadoop és HDFS gyakorlati anyagokat):

```
$ export HADOOP_CLASSPATH=$JAVA_HOME/lib/tools.jar
$ hadoop com.sun.tools.javac.Main WordCount.java
$ jar cf wc.jar WordCount*.class
```

Futtatás előtt a megfelelő input szöveg állományokat `file01` és `file02` másoljuk fel a HDFS fájlrendszerre

```
$ hadoop fs -mkdir /input
$ hadoop fs -copyFromLocal file01 /input/file01
$ hadoop fs -copyFromLocal file02 /input/file02
$ hadoop fs -ls /input
/input/file01
/input/file02

$ hadoop fs -cat /input/file01
Hello World Bye World

$ hadoop fs -cat /input/file02
Hello Hadoop Goodbye Hadoop
```

Ezután már futtathatjuk a MapReduce alkalmazásunkat e lefordított jar segítségével:

```
$ hadoop jar wc.jar WordCount /input /output

$ hadoop fs -cat /output/part-r-00000
Bye 1
Goodbye 1
Hadoop 2
Hello 2
World 2
```

A programkód lépésről lépésre

```
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);
  }
}
```

A map függvény megvalósítása a Mapper interfész implementálását jelenti, ami soronként dolgozza fel a bemenetet, aminek a típusa `Text`. Ezután tokenizálja a sort whitespace-k mentén, és minden egyes tokenhez (azaz szóhoz jelen esetben) egy kulcs-érték párt ír ki (*emit*) a következő formában: `< <szó>, 1>` Jelen példában az első map eredménye:

```
< Hello, 1>
< World, 1>
< Bye, 1>
< World, 1>
```

Míg a második map eredménye:

```
< Hello, 1>
< Hadoop, 1>
< Goodbye, 1>
< Hadoop, 1>
```

A `WordCount` példa használ `combiner` task-ot is, így minden map eredménye keresztülmegy egy lokális combiner folyamaton (ami jelen esetben ugyanaz, mint a reduce), hogy a map eredményeit lokálisan aggregáljuk, miután kulcsok szerint sorba rendeztük őket.

Ez első map eredményei a combine után így alakulnak:

```
< Bye, 1>
< Hello, 1>
< World, 2>
```

Míg a másodiké így:

```
< Goodbye, 1>
< Hadoop, 2>
< Hello, 1>
```

Ezek az adatok aztán bekerülnek a reduce függvénybe feldolgozásra:

```
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);
}
```

Figyeljük meg, hogy a reduce függvény kulcsonként kapja meg az értékek listáját, amiket a map (vagy ha használjuk a combiner) függvények állítanak elő (amennyiben combiner-t használunk, az is ily módon kapja meg a bemenetet). Jelen esetben ezeket:

```
< Bye, <1> >
< Goodbye, <1> >
< Hadoop, <2> >
< Hello, <1, 1> >
< World, <2> >
```

A reduce végeredménye tehát a következő:

```
< Bye, 1>
< Goodbye, 1>
< Hadoop, 2>
< Hello, 2>
< World, 2>
```

Amit a HDFS fájlrendszer `/output` mappába generált fájl tartalmazza. A program main metódusában számos beállítást adtunk meg:

- Ki- és bemeneti fájlok útvonalai
- Kulcs-érték típusok
- Bemeneti és kimeneti adatok formátuma

📖 3. fejezet
CSV feldolgozó MapReduce program

Készítsünk egy olyan MapReduce programot, amely kiszámítja a `daily_csv.csv` adatfájlban szereplő összes ország pénznemének USD-hez viszonyított átlagos árfolyamát a napi árfolyam adatok alapján. Az árfolyam adatfájl cím és első adatsora így néz ki:

```
Date,Country,Value
1971-01-04,Australia,0.8987
```

Az adatfájl több százezer sornyi napi árfolyam adatot tartalmaz 22 országra. Írjuk meg a fenti példa alapján azt a MapReduce alkalmazást, ami ehhez a 22 országhoz kiszámítja az összes napra vett átlagos árfolyamértéket.

A mapper megvalósítás:

```
public static class TokenizerMapper
            extends Mapper<Object, Text, Text, DoubleWritable> {

        private final static DoubleWritable currencyPrice = new
DoubleWritable();
        private final Text country = new Text();

        public void map(Object key, Text value, Context context
        ) throws IOException, InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString(), "\n");
            while (itr.hasMoreTokens()) {
                String csvLine = itr.nextToken();
                if (csvLine.startsWith("Date")) {
                    continue;
                }
                String[] csvVals = csvLine.split(",");
                country.set(csvVals[1]);
                double val = csvVals.length < 3 ? 0.0 :
Double.parseDouble(csvVals[2]);

                if (val > 0) {
                    currencyPrice.set(val);
                    context.write(country, currencyPrice);
                }
            }
        }
    }
```

- A `StringTokenizer` soronként bontja az inputot
- Egy-egy csv adatsort pedig a vesszők mentén bontjuk mezőkre
- Minden csv sor esetén, ahol van árfolyam adat, kiírunk egy kulcs-érték párt, ahol az ország neve a kulcs, a napi árfolyam pedig az érték

A reducer megvalósítás:

```
public static class AvgReducer
            extends Reducer<Text,DoubleWritable,Text,DoubleWritable> {
        private final DoubleWritable result = new DoubleWritable();

        public void reduce(Text key, Iterable<DoubleWritable> values,
                           Context context
        ) throws IOException, InterruptedException {
            double sum = 0.0;
            int count = 0;
            for (DoubleWritable val : values) {
                count++;
                sum += val.get();
            }
            result.set(sum/count);
            context.write(key, result);
        }
    }
```

- Végig megyünk az egy kulcshoz (ország) tartozó értékeken (napi árfolyam adatok) és képezzük azok átlagát
    - Úgy, hogy előbb összeadjuk az összes értéket

- Majd visszaosztjuk a napi adatok darabszámával
- Ez lesz a MapReduce program kimenete, amit a fenti módon tekinthetünk meg

✅ További feladatok

1. Készítsünk olyan MapReduce programot az árfolyam adatokat tartalmazó adat fájlhoz, ami az egyes valuták medián értékét számítja ki, nem pedig az átlagukat! ⭐
2. Készítsünk olyan MapReduce programot az árfolyam adatokat tartalmazó adat fájlhoz, ami az egyes országokhoz kiszámítja mekkora időintervallumot ölel fel a hozzájuk tartozó árfolyam adatsor! Pl. ha egy ország első bejegyzése 1997-01-01, az utolsó 1999-06-23, akkor az adott országhoz számított eredmény 2 év 5 hónap és 22 nap (tetszőleges módon kódolható az eredmény String formában). ⭐

Referenciák

[1] https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

[2] https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[3] https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[4] https://hadoop.apache.org/docs/r1.2.1/streaming.html

[5] https://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/pipes/package-summary.html

[6] https://www.datasciencecentral.com/profiles/blogs/how-to-install-and-run-hadoop-on-windows-for-beginners

[7] https://bigdataproblog.wordpress.com/2016/05/20/developing-hadoop-mapreduce-application-within-intellij-idea-on-windows-10/

[8] https://data-flair.training/blogs/hadoop-combiner-tutorial