MAP REDUCE HADOOP
(STREAMING AND CHAINING JOBS)

Master Program in Computer Science
University of Calabria

Prof. F. Ricca

Streaming in Hadoop

- One can use Java to write Hadoop programs
- Hadoop supports other languages
  - Via a generic API called Streaming
  - Useful for writing simple, short MapReduce scripts
  - Any scripting/programming language using non-Java libraries

- Hadoop Streaming
  - Interacts with programs using the Unix streaming paradigm
  - Inputs come in through STDIN
  - Outputs go to STDOUT
  - *Data has to be text based and each line is considered a record*

Streaming Data Flow

- Similar to the following Unix command line:

```
cat [input_file] | [mapper] | sort
                          | [reducer] >[output_file]
```

- Only need to invoke the Streaming jar

```
bin/hadoop jar [hadoop-streaming-X.Y.Z.jar]
      -input   [input_file]
      -output  [output_files]
      -file    [script to share]
      -mapper  'mappercommand'
      -reducer 'reducercommand'
      …
```

Word count per split

```
bin/hadoop jar hadoop-streaming-X.Y.Z.jar
      -input    input.txt
      -output   count
      -mapper   'wc -l'
      -numReduceTasks 0
```

Working principles

- Works with any executable script
  - Reads a line-oriented data stream from STDIN
  - Outputs to STDOUT with Hadoop Streaming
- Each mapper
  - Sees the entire stream of data
  - Takes on the responsibility of breaking the stream into (line-oriented) record
  - In Java the framework itself breaks input data
- Streaming works on key/value pairs just like in Java
  - The tab character to separate the key from the value in a record
  - When there's no tab character, the entire record is considered the key and the value is empty text

Streaming with key/value pairs

1. The mapper
   - Reads a split through STDIN
   - Extracts each line as a record
     (can interpret each input record as a key/value pair or a line of text)
2. The framework ensures that
   - Each line of your mapper's output is a key/ value pair
   - Key/value pairs with the same key will end up at the same reducer
3. The reducer
   - key/value pairs are sorted by key
   - is responsible for performing the grouping,
   - reads one line at a time from STDIN and keeps track of the keys
4. The reducer output (STDOUT) is written to a file directly

Implement Max via script

```
./bin/hadoop jar hadoop-streaming-X.Y.Z.jar
      -input imput.txt
      -output out.txt
      -mapper 'cut -f1,3'
      -reducer 'maxGroupAttr.py'

# Equivalent to

cat imput.txt | cut -f1,3
      | sort | ./maxGroupAttr.py
```

maxGroupAttr.py

```
#!/usr/bin/python
import sys

lastKey=None
max   = 0
for line in sys.stdin:
        (key, val) = line.strip().split()
        if lastKey and lastKey!=key:
                print lastKey + "\t" + max
                max=0
                lastKey=None
        lastKey=key
        if (val > max):
                max = val
print lastKey + "\t" + max
```

diversamente da java, qui non sono raggruppate per chiave, quindi quando la chiave cambia si inizializza dinuovo il conto per il massimo locale

Streaming with the Aggregate package

- The library package called Aggregate
  - Simplifies obtaining aggregate statistics of a data set
  - Can simplify the writing of Java statistics collectors
    - Especially when used with Streaming

- Each line of the mapper's output looks like

```
function:key\tvalue
```

- The Aggregate reducer applies the function to the set of values for each key

Standard aggregation functions

Value aggregator	Description
`DoubleValueSum`	Sums up a sequence of double values.
`LongValueMax`	Finds the maximum of a sequence of long values.
`LongValueMin`	Finds the minimum of a sequence of long values.
`LongValueSum`	Sums up a sequence of long values.
`StringValueMax`	Finds the lexicographical maximum of a sequence of string values.
`StringValueMin`	Finds the lexicographical minimum of a sequence of string values.
`UniqValueCount`	Finds the number of unique values (for each key).
`ValueHistogram`	Finds the count, minimum, median, maximum, average, and standard deviation of each value. (See text for further explanation.)

Generic aggregator example

```
./bin/hadoop jar hadoop-streaming-X.Y.Z.jar
  -input  input.txt
  -output out.txt
  -mapper 'funtionMapper.py LongValueSum 0'
  -reducer aggregate
```

Lo 0 indica la colonna che viene influenzata al conto

```
./bin/hadoop jar hadoop-streaming-X.Y.Z.jar
  -input input.txt
  -output out.txt
  -mapper 'funtionMapper.py DoubleValueSum 0 2'
  -reducer aggregate
```

In questo caso vengono considerate le colonne 0 e 2, dove 0 è la chiave e 2 è il value

aggregate è la funzione in base al tipo di valore

functionMapper.py

```
#!/usr/bin/python
import sys

function = (sys.argv[1])
keyIdx = int(sys.argv[2])

if len(sys.argv) > 3:
        valIdx = int(sys.argv[3])
else:
        valIdx = -1


for line in sys.stdin:
        fields = line.strip().split()
        if fields[keyIdx] and not fields[keyIdx].startswith("#"):
                if valIdx>0:
                        print function + ":" +fields[keyIdx] + "\t" + fields[valIdx]
                else:
                        print function + ":" +fields[keyIdx] + "\t" + "1"
(END)
```

```
Usage: $HADOOP_HOME/bin/hadoop jar hadoop-streaming.jar [options]
Options:
  -input          <path> DFS input file(s) for the Map step.
  -output         <path> DFS output directory for the Reduce step.
  -mapper         <cmd|JavaClassName> Optional. Command to be run as mapper.
  -combiner       <cmd|JavaClassName> Optional. Command to be run as combiner.
  -reducer        <cmd|JavaClassName> Optional. Command to be run as reducer.
  -file           <file> Optional. File/dir to be shipped in the Job jar file.
                  Deprecated. Use generic option "-files" instead.
  -inputformat    <TextInputFormat(default)|SequenceFileAsTextInputFormat|JavaClassName>
                  Optional. The input format class.
  -outputformat   <TextOutputFormat(default)|JavaClassName>
                  Optional. The output format class.
  -partitioner    <JavaClassName>  Optional. The partitioner class.
  -numReduceTasks <num> Optional. Number of reduce tasks.
  -inputreader    <spec> Optional. Input recordreader spec.
  -cmdenv         <n>=<v> Optional. Pass env.var to streaming commands.
  -mapdebug       <cmd> Optional. To run this script when a map task fails.
  -reducedebug    <cmd> Optional. To run this script when a reduce task fails.
  -io             <identifier> Optional. Format to use for input to and output
                  from mapper/reducer commands
  -lazyOutput     Optional. Lazily create Output.
  -background     Optional. Submit the job and don't wait till it completes.
  -verbose        Optional. Print verbose output.
  -info           Optional. Print detailed usage.
  -help           Optional. Print help message.
```

Chaining MapReduce jobs (1)

- Many complex tasks need to be broken down
  - Each accomplished by an individual MapReduce job
- Chaining MapReduce jobs in a sequence
  - Like:

```
mapreduce-1 | mapreduce-2 | ...
```

- Just schedule every next job after the other in the main

```
Job.waitForCompletion(true) ? 0 : 1
```

  - To wait until preceding job completes!!

Chaining MapReduce jobs (2)

- Complex chains using the ChainMapper class
  - Like:

```
[map]+ |      reduce   [| map]*
```

  - pre-processing | standard computation | post-processing

alcune volte si fanno anche più operazioni di map prima di un reducing

```
…
ChainMapper.addMapper(job,map1.class,key.class,val.class)
ChainMapper.addMapper(job,map2.class,key.class,val.class)
ChainMapper.setReducer(job,red.class,key.class,val.class)
ChainMapper.addMapper(job,map3.class,key.class,val.class)
…
Job.waitForCompletion(true) ? 0 : 1
```

Chaining MapReduce jobs (3)

- Enforce specific dependencies
  - Specify like a chain of dependencies
  - Using JobControl and ControlledJob classes

…

```
ControlledJob cJ1 = new ControlledJob(job1)
ControlledJob cJ2 = new ControlledJob(job2)
cJ1.addDependingJob(cJ2)
```

cj1 deve finire per runnare cj2

```
JobControl jC = new JobControl("A chain")
jC.addJob(cJ1)
jC.addJob(cJ1)
jC.run()
```

LET'S START

(open, configure and run examples)

Required Software

- Eclipse
  - www.eclipse.org
- Hadoop
  - Download it from http://hadoop.apache.org/
  - Obtain everything it via Maven (https://mvnrepository.com/artifact/org.apache.hadoop)
- Small examples do not need a cluster
  - Running Hadoop in Standalone mode
- Terminal app
- Few python scripts
- Few data files