Introduction to Map/Reduce

Kostas Solomos
Computer Science Department
University of Crete, Greece
What we will cover...

- What is MapReduce?

- How does it work?

- A simple word count example
  - (the “Hello World!” of MapReduce)
What is MapReduce?

A programming model for processing large datasets in parallel on a cluster, by dividing the work into a set of independent tasks (introduced by Google in 2005)

All we have to do is provide the implementation of two methods:
- map()
- reduce()

...but we can do much more...
How does it work?

**keys and values**
- everything is expressed as \((\text{key}, \text{value})\) pairs
  - e.g. the information that the word “hello” appears 4 times in a text, could be expressed as: ("hello", 4)

Each *map* method receives a list of \((\text{key}, \text{value})\) pairs and emits a list of \((\text{key}, \text{value})\) pairs
- the intermediate output of the program

Each *reduce* method receives, for each unique intermediate key \(k\), a list of all intermediate values that were emitted for \(k\).
Then, it emits a list of \((\text{key}, \text{value})\) pairs
- the final output of the program
### MapReduce – Input Data

<table>
<thead>
<tr>
<th>e1</th>
<th>e2</th>
<th>e3</th>
<th>e4</th>
<th>e5</th>
<th>e6</th>
<th>e7</th>
<th>e8</th>
</tr>
</thead>
</table>

5
MapReduce – Input Data Splitting
MapReduce – Mapper Input

Mapper 1

Mapper 2

Mapper 3
MapReduce – Mapper Output

Mapper 1

Mapper 2

Mapper 3
MapReduce – Shuffling & Sorting (simplified)
MapReduce – Reducing

Reducer 1

Reducer 2

Reducer 3
MapReduce – Reducing

Reducer 1

Reducer 2

Reducer 3
Example: WordCount

- **Input**: A list of (file-name, line) pairs
- **Output**: A list of (word, frequency) pairs for each unique word appearing in the input

**Idea:**

**Map:**
for each word $w$, emit a $(w, 1)$ pair

**Reduce:**
for each $(w, \text{list}(1,1,...,1))$, sum up the 1's and emit a $(w, 1+1+...+1)$ pair
Example: WordCount

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>file1.txt</td>
<td>hello, 1 world, 1</td>
</tr>
<tr>
<td>file2.txt</td>
<td>the, 1 the, 1 little, 1</td>
</tr>
<tr>
<td>file3.txt</td>
<td>big, 1 fish, 1 eat, 1 fish, 1</td>
</tr>
<tr>
<td></td>
<td>hello, 1 and, 1</td>
</tr>
<tr>
<td></td>
<td>fish, 1 chips, 1 please, 1</td>
</tr>
</tbody>
</table>
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
}
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable>

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
Combiner: a local, mini-reducer

- An optional class that works like a reducer, run locally
  - for the output of each mapper

- Goal:
  - reduce the network traffic from mappers to reducers
    - could be a bottleneck
  - reduce the workload of the reducers

WordCount Example:
We could sum up the local 1’s corresponding to the same key and emit a temporary word count to the reducer
  - fewer pairs are sent to the network
  - the reducers save some operations
WordCount with Combiner

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>file1.txt</td>
<td>hello, 1 world, 1</td>
</tr>
<tr>
<td>file2.txt</td>
<td>the, 2 little, 1</td>
</tr>
<tr>
<td>file3.txt</td>
<td>hello, fish and chips please!</td>
</tr>
</tbody>
</table>

Map → Reduce

Map

Reduce
public int run(String[] args) throws Exception {
    Job job = new Job(getConf(), "WordCount");
    job.setJarByClass(WordCount.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);
    job.setCombinerClass(Reduce.class);
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    job.setNumReduceTasks(2);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.waitForCompletion(true);
    return 0;
}
Writable types

Any key or value type has to implement the Writable interface

Examples of Objects implementing Writable:
- Text (used for Strings)
- IntWritable, LongWritable, ShortWritable, DoubleWritable, ...
- VIntWritable, VLongWritable, ... (variable-length encodings)
- ByteWritable, BooleanWritable
- ArrayWritable
- ...

Keys have to implement the WritableComparable interface, too
- if (key, value) pairs sent to a single reduce task include multiple keys, the reducer will process the keys in sorted order.

You can also define and use your own types
A simple Custom Writable

public class MyWritable implements Writable {
    private int counter;  private long timestamp;

    public void write(DataOutput out) throws IOException {
        out.writeInt(counter);
        out.writeLong(timestamp);
    }

    public void readFields(DataInput in) throws IOException {
        counter = in.readInt();
        timestamp = in.readLong();
    }

    public static MyWritable read(DataInput in) throws IOException {
        MyWritable w = new MyWritable();
        w.readFields(in);
        return w;
    }
}
Output Format

The `OutputFormat` and `RecordWriter` interfaces dictate how to write the results of a job back to the underlying permanent storage.

The default format (`TextOutputFormat`) will write `(key, value)` pairs as strings to individual lines of an output file (using the `toString()` methods of the keys and values).

The `SequenceFileOutputFormat` will keep the data in binary, so it can be later read quickly by the `SequenceFileInputFormat`.

These classes make use of the `write()` method of the specific `Writable` classes used by your MapReduce pass.
Input Formats

The `InputFormat` defines how to read data from a file into the Mapper instances.
- Hadoop comes with several implementations of `InputFormat`; e.g.
  - `TextInputFormat`:
    - The key it emits for each record is the byte offset of the line read (as a `LongWritable`), and the value is the contents of the line up to the terminating `\n` character (as a `Text` object)
  - `SequenceFileInputFormat`, for reading particular binary file formats
    - instead of reading input as text only, you read it as `(key, value)` pairs, as you have written them in a previous job

These classes make use of the `readFields()` method of the specific `Writable` classes used by your MapReduce pass.
Partitioning

Which reducer will receive the intermediate output keys and values?
- \((key, value)\) pairs with the same \(key\) end up at the same partition

The mappers partition data independently
- they never exchange information with one another
Hadoop uses an interface called \(Partitioner\) to determine to which partition a \((key, value)\) pair will go
A single partition refers to all \((key, value)\) pairs which will be sent to a single reduce task

\[ \#\text{partitions} = \#\text{reduce tasks} \]

\(\text{(each Reducer can process multiple reduce tasks)}\)
The Partitioner determines the \textit{load balancing} of the reducers
The Partitioner interface

The Partitioner interface defines the getPartition() method

- **Input:** a *key*, a *value* and the number of partitions
- **Output:** a partition id for the given *key*, *value* pair

The default Partitioner is the HashPartitioner:

```java
int getPartition(K key, V value, int numPartitions) {
    return key.hashCode() % numPartitions;
}
```

<table>
<thead>
<tr>
<th>Key</th>
<th>key.hashCode()</th>
<th>partitionId (0-2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello</td>
<td>12847</td>
<td>1</td>
</tr>
<tr>
<td>world</td>
<td>23874</td>
<td>0</td>
</tr>
<tr>
<td>map</td>
<td>82375</td>
<td>1</td>
</tr>
<tr>
<td>reduce</td>
<td>12839</td>
<td>2</td>
</tr>
</tbody>
</table>

numPartitions = 3
Fault tolerance

- The primary way that Hadoop achieves fault tolerance is through restarting tasks
  - Individual task nodes (*TaskTrackers*) are in constant communication with the head node of the system (the *JobTracker*)
  - If a TaskTracker fails to communicate with the JobTracker for a period of time (by default, 10 minutes), the JobTracker will assume that the TaskTracker in question has crashed.
  - The JobTracker knows which map and reduce tasks were assigned to each TaskTracker.
  - If the job is still in the map phase, then other TaskTrackers will be asked to re-execute all map tasks previously run by the failed TaskTracker.
  - If the job is in the reduce phase, then other TaskTrackers will re-execute all map and reduce tasks that were assigned to the failed TaskTracker.
Speculative execution

- The same input can be processed *multiple times in parallel*, to exploit differences in machine capabilities
  - by dividing the tasks across many nodes, it is possible for a few slow nodes to rate-limit the rest of the program

- When tasks complete, they announce this fact to the JobTracker
  - Whichever copy of a task finishes first becomes the definitive copy
  - If other copies were executing speculatively, Hadoop tells the TaskTrackers to abandon the tasks and discard their outputs.
  - The Reducers then receive their inputs from whichever Mapper completed successfully, first.
MRv1: JobTracker and TaskTrackers

Client

JobTracker

TaskTracker
  map
  reduce

TaskTracker
  map
  reduce

TaskTracker
  map
  reduce

Client

master

slaves
JobTracker UI

master Hadoop Map/Reduce Administration

State: RUNNING
Started: Thu Jan 22 10:58:52 EET 2016
Version: 1.2.0, r1479473
Compiled: Mon May 6 06:59:37 UTC 2013 by hortonfo
Identifier: 201501221058
SafeMode: OFF

Cluster Summary (Heap Size is 358.5 MB/889 MB)

<table>
<thead>
<tr>
<th>Running Map Tasks</th>
<th>Running Reduce Tasks</th>
<th>Total Submissions</th>
<th>Nodes</th>
<th>Occupied Map Slots</th>
<th>Occupied Reduce Slots</th>
<th>Reserved Map Slots</th>
<th>Reserved Reduce Slots</th>
<th>Map Task Capacity</th>
<th>Reduce Task Capacity</th>
<th>Avg. Tasks/Node</th>
<th>Blacklisted Nodes</th>
<th>Graylisted Nodes</th>
<th>Excluded Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>0</td>
<td>16</td>
<td>14</td>
<td>66</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>56</td>
<td>56</td>
<td>8.00</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Scheduling Information

<table>
<thead>
<tr>
<th>Queue Name</th>
<th>State</th>
<th>Scheduling Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>running</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Filter (Jobid, Priority, User, Name)  
Example: "user" will match any user field and "3208" in all fields

Running Jobs

<table>
<thead>
<tr>
<th>Jobid</th>
<th>Started</th>
<th>Priority</th>
<th>User</th>
<th>Name</th>
<th>Map % Complete</th>
<th>Map Total</th>
<th>Maps Completed</th>
<th>Reduce % Complete</th>
<th>Reduce Total</th>
<th>Reduces Completed</th>
<th>Job Scheduling Information</th>
<th>Diagnostic Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>job_201501221058_0016</td>
<td>Thu Feb 05 15:26:07 EET 2016</td>
<td>NORMAL</td>
<td>user</td>
<td>Average Edge Weight using Extended Input</td>
<td>14.09%</td>
<td>1120</td>
<td>114</td>
<td>0.00%</td>
<td>0</td>
<td>0</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
Hadoop Distributed File System (HDFS)

HDFS is a distributed file system designed to hold very large amounts of data (terabytes or even petabytes), and provide high-throughput access to this information.

Files are stored in a redundant fashion across multiple machines to ensure their durability to failure and high availability to very parallel applications.

HDFS is a block-structured file system:
- Individual files are broken into blocks of a fixed size (default 128MB).
- These blocks are stored across a cluster of one or more machines (DataNodes).
- The NameNode stores all the metadata for the file system.
HDFS nodes

NameNode:
Stores metadata only

METADATA:
/user/aaron/foo → 1, 2, 4
/user/aaron/bar → 3, 5

DataNodes: Store blocks from files
Interacting with HDFS

Unix shell-like commands (ls, mkdir, rm, …)

Important: start any HDFS command with `bin/hadoop dfs`

Examples:

*bin/hadoop dfs –ls /user/hduser/
  lists the contents of the directory /user/hduser/

*bin/hadoop dfs –copyFromLocal ~/test.txt /user/hduser/Input
  copies the local file test.txt to the HDFS directory Input
More HDFS shell commands

- `cat`, `text`, `tail`
- `mv`, `cp`, `rm`, `rmr`, `mkdir`
- `copyToLocal`, `moveToLocal`, `get`, `getmerge`
- `copyFromLocal`, `moveFromLocal`, `put`
- `count`
- `du`, `dus`
- `setrep`
NameNode WebUI

NameNode 'master:9000'

Started: Wed Feb 04 09:54:35 EET 2015
Version: 1.2.0, r1479473
Compiled: Mon May  6 06:59:37 UTC 2013 by hortonfo
Upgrades: There are no upgrades in progress.

Browse the filesystem
Namenode Logs

Cluster Summary

1123 files and directories, 2488 blocks = 3611 total. Heap Size is 125 MB / 889 MB (14%)

- Configured Capacity: 826.79 GB
- DFS Used: 114.79 GB
- Non DFS Used: 83.22 GB
- DFS Remaining: 628.78 GB
- DFS Used%: 13.88%
- DFS Remaining%: 76.05%

- Live Nodes: 14
- Dead Nodes: 0
- Decommissioning Nodes: 0
- Number of Under-Replicated Blocks: 0

NameNode Storage:

<table>
<thead>
<tr>
<th>Storage Directory</th>
<th>Type</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>/app/hadoop/tmp/dfs/name</td>
<td>IMAGE_AND_EDITS</td>
<td>Active</td>
</tr>
</tbody>
</table>

This is Apache Hadoop release 1.2.0
## Datanodes

### NameNode 'master:9000'

- **Started:** Wed Feb 04 09:54:35 EET 2015
- **Version:** 1.2.0, r1479473
- **Compiled:** Mon May  6 06:59:37 UTC 2013 by hortonio
- **Upgrades:** There are no upgrades in progress.

Browse the filesystem

Namenode Logs

Go back to DFS home

### Live Datanodes : 14

<table>
<thead>
<tr>
<th>Node</th>
<th>Last Contact</th>
<th>Admin State</th>
<th>Configured Capacity (GB)</th>
<th>Used (GB)</th>
<th>Non DFS Used (GB)</th>
<th>Remaining (GB)</th>
<th>Used (%)</th>
<th>Used (%)</th>
<th>Remaining (%)</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>slave1</td>
<td>2</td>
<td>In Service</td>
<td>59.06</td>
<td>8.28</td>
<td>5.96</td>
<td>44.82</td>
<td>14.02</td>
<td></td>
<td>75.69</td>
<td>185</td>
</tr>
<tr>
<td>slave10</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>7.88</td>
<td>5.94</td>
<td>45.23</td>
<td>13.35</td>
<td></td>
<td>76.59</td>
<td>187</td>
</tr>
<tr>
<td>slave11</td>
<td>0</td>
<td>In Service</td>
<td>59.06</td>
<td>8.51</td>
<td>5.98</td>
<td>44.57</td>
<td>14.14</td>
<td></td>
<td>75.47</td>
<td>195</td>
</tr>
<tr>
<td>slave12</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>7.07</td>
<td>5.92</td>
<td>46.06</td>
<td>11.98</td>
<td></td>
<td>78</td>
<td>167</td>
</tr>
<tr>
<td>slave14</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>6.58</td>
<td>5.94</td>
<td>46.54</td>
<td>11.14</td>
<td></td>
<td>78.8</td>
<td>155</td>
</tr>
<tr>
<td>slave15</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>7.56</td>
<td>5.94</td>
<td>45.56</td>
<td>12.8</td>
<td></td>
<td>77.15</td>
<td>189</td>
</tr>
<tr>
<td>slave2</td>
<td>0</td>
<td>In Service</td>
<td>59.06</td>
<td>7.33</td>
<td>6.55</td>
<td>45.17</td>
<td>12.41</td>
<td></td>
<td>76.49</td>
<td>167</td>
</tr>
<tr>
<td>slave3</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>9.24</td>
<td>5.9</td>
<td>43.91</td>
<td>16.65</td>
<td></td>
<td>74.36</td>
<td>204</td>
</tr>
<tr>
<td>slave4</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>8.5</td>
<td>5.91</td>
<td>44.65</td>
<td>14.4</td>
<td></td>
<td>75.6</td>
<td>188</td>
</tr>
<tr>
<td>slave5</td>
<td>2</td>
<td>In Service</td>
<td>59.06</td>
<td>9.43</td>
<td>5.99</td>
<td>43.63</td>
<td>15.97</td>
<td></td>
<td>73.88</td>
<td>209</td>
</tr>
<tr>
<td>slave6</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>7.94</td>
<td>5.89</td>
<td>45.23</td>
<td>13.44</td>
<td></td>
<td>76.59</td>
<td>185</td>
</tr>
<tr>
<td>slave7</td>
<td>2</td>
<td>In Service</td>
<td>59.06</td>
<td>8.29</td>
<td>5</td>
<td>44.77</td>
<td>14.03</td>
<td></td>
<td>75.81</td>
<td>190</td>
</tr>
<tr>
<td>slave8</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>8.6</td>
<td>5.94</td>
<td>44.52</td>
<td>14.55</td>
<td></td>
<td>75.39</td>
<td>193</td>
</tr>
<tr>
<td>slave9</td>
<td>1</td>
<td>In Service</td>
<td>59.06</td>
<td>9.54</td>
<td>5.9</td>
<td>43.61</td>
<td>16.16</td>
<td></td>
<td>73.85</td>
<td>211</td>
</tr>
</tbody>
</table>

This is Apache Hadoop release 1.2.0
Hadoop base platform brief
Resources

- Large part of this tutorial was adapted from https://developer.yahoo.com/hadoop/tutorial/index.html, under a Creative Commons Attribution 3.0 Unported License.