Large-scale distributed computing systems

Hadoop Lab

January 2017

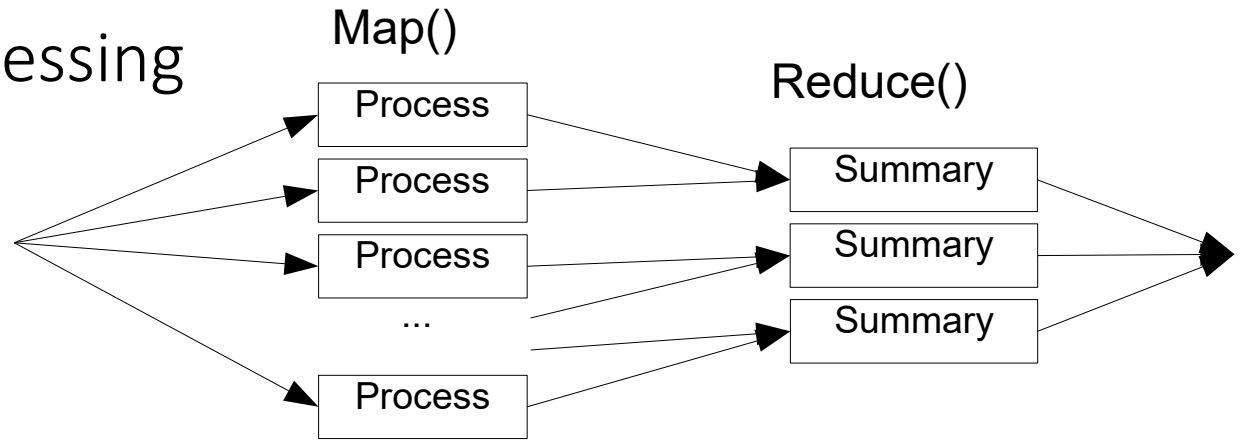
Jean-Pierre Lozi, Johan Montagnat
CNRS, I3S, SPARKS
http://www.i3s.unice.fr/~johan/

MapReduce objectives

- Process large amounts of data in a distributed system
  - Invented to index Web pages for Google search engine
- Targeted to achieve High Throughput Computing
  - One (map) function ran over many pieces of data, independently
  - Scan all data once during each run
  - Different to database access (to small pieces of data, frequently)
- Take advantage of locality
  - Data is big: run code as close as possible to data
    - Usually directly on the data node...
  - Ensures completion
    - Manages data chunks replicates (ensures the map function is ran once over each chunk of data)
    - Re-run on failures

MapReduce parallelization schema

- Two-step processing



- Map step: function executed over all data chunks
  - Massive data parallelism
  - Produce partial result on each computing node
- Reduce step: partial results are aggregated
  - Shuffling phase to transport all results to a single node
  - Reduce function to summarizes all partial results and deliver final result
  - Possibly several reduce functions ran in parallel for efficiceny
    - ...possibly followed by a new Map-Reduce phase

MapReduce process in practice

- Central use of (key, value) structures
  - All data chunks are values, each of them indexed by a key
  - (key, value) pairs are distributed on data nodes
  - Key and Value may be of any type
- Map() function
  - Transforms (key, value) inputs into (key, value) outputs
  - Input/Output key and value types can be anything
- Data shuffling
  - Aggregates values from results with same output key
- Reduce() function
  - Processes aggregated pairs and produces… (key, value) pairs
- MapReduce development consists in finding how to model a problem to fit this generic structure

Why Hadoop Implementation?

- Hadoop is a robust and strongly supported Map-Reduce implementation
- Started after 2004 publications on Google File System and MapReduce framework
- 4-years of effort to reach web-scale
  - With funding from Yahoo! and ever larger community
  - Finally maintained by the Apache foundation
- Hadoop in practice
  - Focus on Map() and Reduce() function development only
  - Hadoop handles:
    - Data distribution and replication, keys load balancing, data shuffling
    - Map() and Reduce() functions execution, fault tolerance...

HADOOP IN PRACTICE

- **Let's start with an example: we have files that contain meteorological data**
- **These files contain records, each record is one line, containing:**
  - The code of a weather station on five digits
  - The year when the temperature was recorded
  - The average temperature for that year times ten on four digits (we'll suppose they're all positive to simplify things, multiplication to avoid floats). Only one data point per year here so it's not really Big Data, but this is just a toy example, we could have a lot more records, one per hour for instance.
  - Many more fields such as the wind speed, humidity, etc.
- **An example of a record:** `123451950016393627 43...`

HADOOP IN PRACTICE

- The input data will look like this:

```
12345195001639362743
12123195001341892769
12111195001311271987
12094194902231212122
12093194901651209182
...
```

- The data can be stored in many files, one per weather station, one per year… Etc.
- **We will use Hadoop to calculate the maximum average temperature for each year**

HADOOP IN PRACTICE

```
12345195001639362743
12123195001341892769
12111195001311271987
12094194902231212122
12093194901651209182
...
```

- **What will the input of the Map function be?**
  - Each line produces a (key, value) pair
  - We can ignore the key (usually the character offset), the value is the contents of the line:

```
(0, 12345195001639362743)
(20, 12123195001341892769)
(40, 12111195001311271987)
(60, 12094194902231212122)
(80, 12093194901651209182)
...
```

HADOOP IN PRACTICE

```
123451950016393627 43
121231950013418927 69
121111950013112719 87
120941949022312121 22
120931949016512091 82
...
```

- **What will the Map function do?**
  - It will discard the key, parse the values, and return (key, value) pairs where the key is the year and the value is the average temperature. The output will be:

```
(1950, 0163)
(1950, 0134)
(1950, 0131)
(1949, 0223)
(1949, 0165)
...
```

  - So basically our Map function will be a Java function that takes two parameters, the key (a number) and the value (a string), it will parse the string using the standard API, and produce the (key, value) pair as the output…

HADOOP IN PRACTICE

```
1234519500163936274
1212319500134189276
1211119500131127198
1209419490223121212
1209319490165120918
...
```

- **What will the Shuffle phase do?**
  - As we have seen earlier, it will concatenate the values for each key.

    Plus, keys are sorted:

    ```
    (1949, [0223, 0165])
    (1950, [0163, 0134, 0131])
    ...
    ```
  - We do not have to implement this phase: it is done automatically.

HADOOP IN PRACTICE

```
12345195001639362743
12123195001341892769
12111195001311271987
12094194902231212122
12093194901651209182
...
```

- **What will the Reduce phase do?**
  - It will just calculate the maximum of each list. The input was:

```
(1949, [0223, 0165])
(1950, [0163, 0134, 0131])
...
```

  - The output will be:

```
(1949, 0223)
(1950, 0163)
...
```

  - **And that is all. We have the result we want!** All we have to do is to implement two very simple functions in Java: Map and Reduce.

HADOOP IN PRACTICE

```
12345195001639362743
12123195001341892769
12111195001311271987
12094194902231212122
12093194901651209182
...
```

- **One last thing before we start writing the code…**
- **Hadoop uses its own serialization because** Java serialization is known to be inefficient
- **Result: a special set of data types**
  - All implement the "Writable" interface
  - Most common types shown here…
    …more specialized types exist (SortedMapWritable, ObjectWritable…)

Name	Description	JDK equivalent
IntWritable	32-bit integers	Integer
LongWritable	64-bit integers	Long
DoubleWritable	Floating-point numbers	Double
Text	Strings	String

HADOOP IN PRACTICE

```
12345195001639362743
12123195001341892769
12111195001311271987
12094194902231212122
12093194901651209182
...
```

- **First thing to do when you write a MapReduce program :** find the input types of the Map and Reduce functions.
- Map takes pairs like (0, 12345195001639362743), and produces pairs like (1950, 0163). Input types = (LongWritable, Text), output types = (Text, IntWritable), for instance. (We could use an IntWritable for the year too, but we never use its numerical properties.)
- **Consequently, the Map class will extend:**

```
Mapper<LongWritable, Text, Text, IntWritable>
```

- **And contain this function:**

```
public void map(LongWritable key, Text value, Context context)
```

HADOOP IN PRACTICE

```
1234519500163936274
1212319500134189276
1211119500131127198
1209419490223121212
1209319490165120918
...
```

- **First thing to do when you write a MapReduce program :** find the input types of the Map and Reduce functions.
- Reduce takes pairs like (1949, [0223, 0165]), and produces pairs like (1949, 0165). Input types = (Text, IntWritable), output types = (Text, IntWritable), for instance.

- **Consequently, the Reduce class will extend:**

```
Reducer<Text, IntWritable, Text, IntWritable>
```

- **And contain this function:**

```
reduce(Text key, Iterable<IntWritable> values,
Context context)
```

HADOOP IN PRACTICE

```
123451950016393627 43
121231950013418927 69
121111950013112719 87
120941949022312121 22
120931949016512091 82
...
```

```
class MaxTemperature {
    static class MaxTemperatureMapper
            extends Mapper<LongWritable, Text value, Context context> {

        public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {
            String line = value.toString();
            String year = line.substring(5, 9);
            int temp = Integer.parseInt(line.subString(9, 13));
            context.write(new Text(year), new IntWritable(temp));
        }
    }
    ...
    // Now write the reducer.
    ...
    // And create the main() function that creates the job and launches it.
}
```