Getting to know Apache Hadoop

Oana Denisa Balalau
Télécom ParisTech

Table of Contents

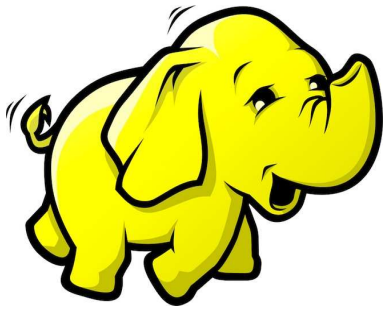
Table of Contents

Apache Hadoop

Apache Hadoop - open source software framework for distributed storage and processing of large data sets on clusters of computers.

The framework is designed to scale from a single computer to thousands of computers, using the computational power and storage of each machine.

Fun fact: Hadoop is a made-up name, given by the son of Doug Cutting (the project's creator) to a yellow stuffed elephant.



Apache Hadoop

What was the motivation behind the creation of the framework?

When dealing with "big data", each application has to solve common issues:

- storing and processing large datasets on a cluster of computers
- handling computer failures in a cluster

Solution: have an efficient library that solves these problems!

Apache Hadoop

The modules of the framework are:

- Hadoop Common: common libraries shared between the modules
- Hadoop Distributed File System: storage of very large datasets in a reliable fashion
- Hadoop YARN: framework for the application management in a cluster
- Hadoop MapReduce: programming model for processing large data sets.

Table of Contents

Hadoop Distributed File System(HDFS). Key Concepts:

Data storage: blocks. A block is a group of sectors (region of fixed size on a formatted disk). A block has the size a multiple of the sector's size and it is used to deal with bigger hard drives.

Unix like system: blocks of a few KB .

HDFS: blocks of 64/128 MB are stored on computers called DataNodes.

Hadoop Distributed File System(HDFS). Key Concepts:

Data storage: blocks. A block is a group of sectors (region of fixed size on a formatted disk). A block has the size a multiple of the sector's size and it is used to deal with bigger hard drives.

Unix like system: blocks of a few KB .

HDFS: blocks of 64/128 MB are stored on computers called DataNodes.

File system metadata : inode. An inode is a data structure that contains information about files and directories (file ownership, access mode, file type, modification and access time).

Unix like system: inode table.

HDFS : NameNode - one/several computers that store inodes.

Hadoop Distributed File System(HDFS). Key Concepts:

Data storage: blocks. A block is a group of sectors (region of fixed size on a formatted disk). A block has the size a multiple of the sector's size and it is used to deal with bigger hard drives.

Unix like system: blocks of a few KB .

HDFS: blocks of 64/128 MB are stored on computers called DataNodes.

File system metadata : inode. An inode is a data structure that contains information about files and directories (file ownership, access mode, file type, modification and access time).

Unix like system: inode table.

HDFS : NameNode - one/several computers that store inodes.

Data integrity.

Unix like system: checksum verification of metadata .

HDFS: maintaining copies of the data (replication) on several datanodes and performing checksum verification of all data.

Hadoop Distributed File System(HDFS). Goals:

- to deal with hardware failure
  - ✓ data is replicated on several machines

Hadoop Distributed File System(HDFS). Goals:

- to deal with hardware failure
  - ✓ data is replicated on several machines
- to provide a simple data access model
  - ✓ the data access model is write-once read-many-times, allowing concurrent reads of data

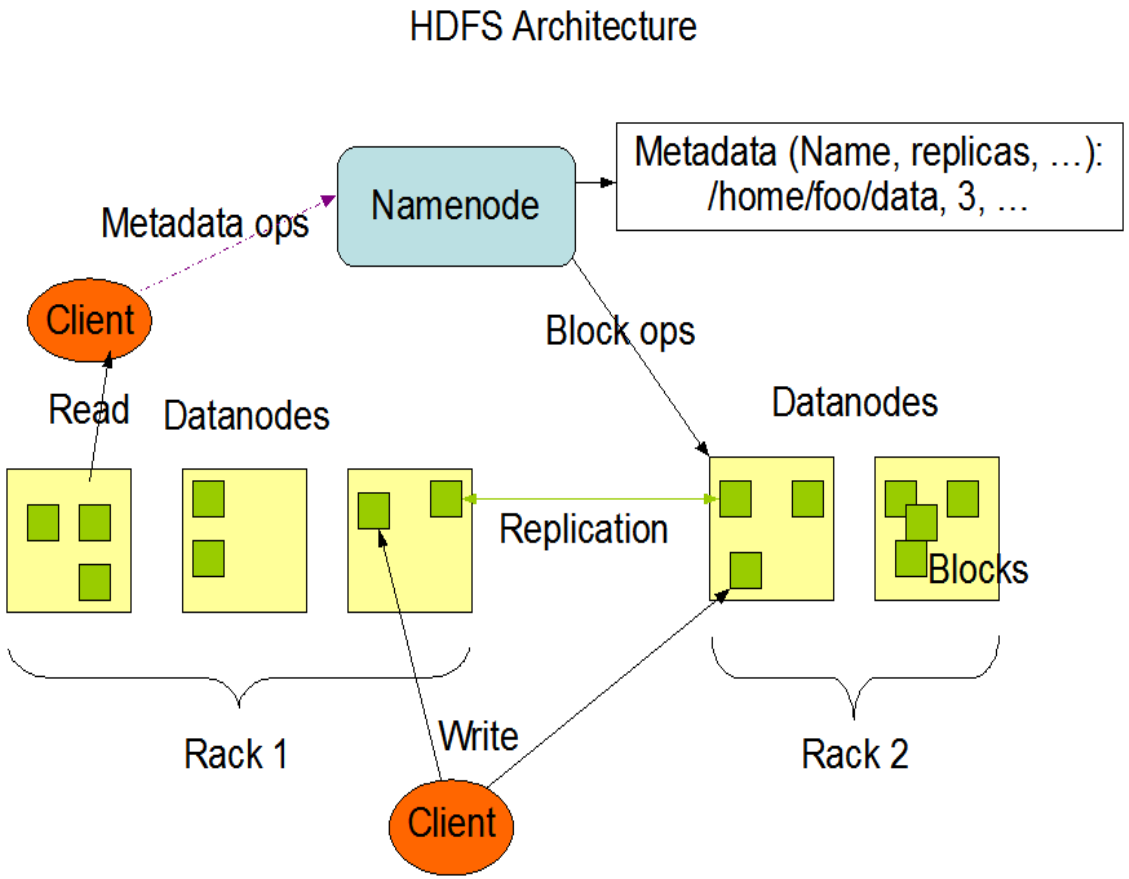
Hadoop Distributed File System(HDFS). Goals:

- to deal with hardware failure
  - ✓ data is replicated on several machines
- to provide a simple data access model
  - ✓ the data access model is write-once read-many-times, allowing concurrent reads of data
- to provide streaming data access
  - ✓ the large size of blocks makes HDFS unfit for random seeks in files (as we always read at least 64/128 MB). However big blocks allow fast sequential reads, optimizing HDFS for a fast streaming data access (i.e. low latency in reading the whole dataset)

Hadoop Distributed File System(HDFS). Goals:

- to deal with hardware failure
  - ✓ data is replicated on several machines
- to provide a simple data access model
  - ✓ the data access model is write-once read-many-times, allowing concurrent reads of data
- to provide streaming data access
  - ✓ the large size of blocks makes HDFS unfit for random seeks in files (as we always read at least 64/128 MB). However big blocks allow fast sequential reads, optimizing HDFS for a fast streaming data access (i.e. low latency in reading the whole dataset)
- to manage large data sets
  - ✓ HDFS can run on clusters of thousands of machines, proving huge storage facilities

Hadoop Distributed File System(HDFS)



Source image: `http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`

Table of Contents

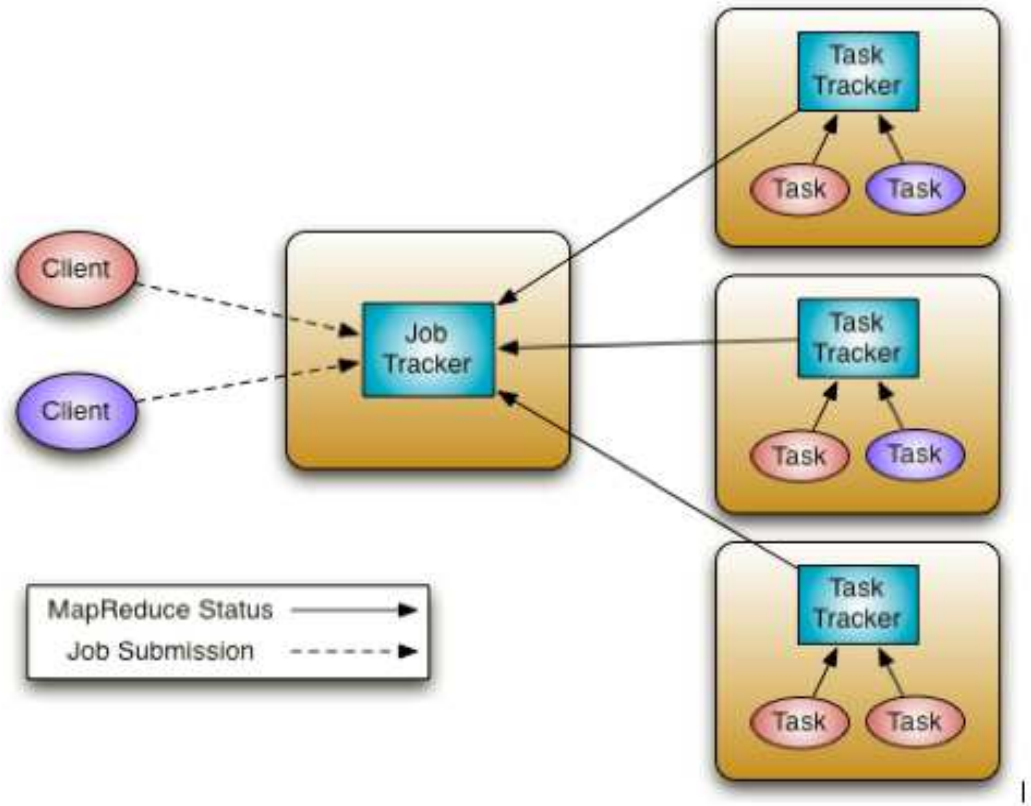
Old framework: MapReduce 1. Key Concepts

- JobTracker is the master service that sends MapReduce computation tasks to nodes in the cluster.
- TaskTrackers are slave services on nodes in the cluster that perform computation (Map, Reduce and Shuffle operations).

An application is submitted for execution:

- ✓ JobTracker queries NameNode for the location of the data needed
- ✓ JobTracker **assigns** computation to TaskTracker nodes with available computation power or near the data
- ✓ JobTracker **monitors** TaskTracker nodes during the job execution

Old framework: MapReduce 1



Source image:

`http://hortonworks.com/wp-content/uploads/2012/08/MRArch.png`

New framework: MapReduce 2 (YARN). Key Concepts

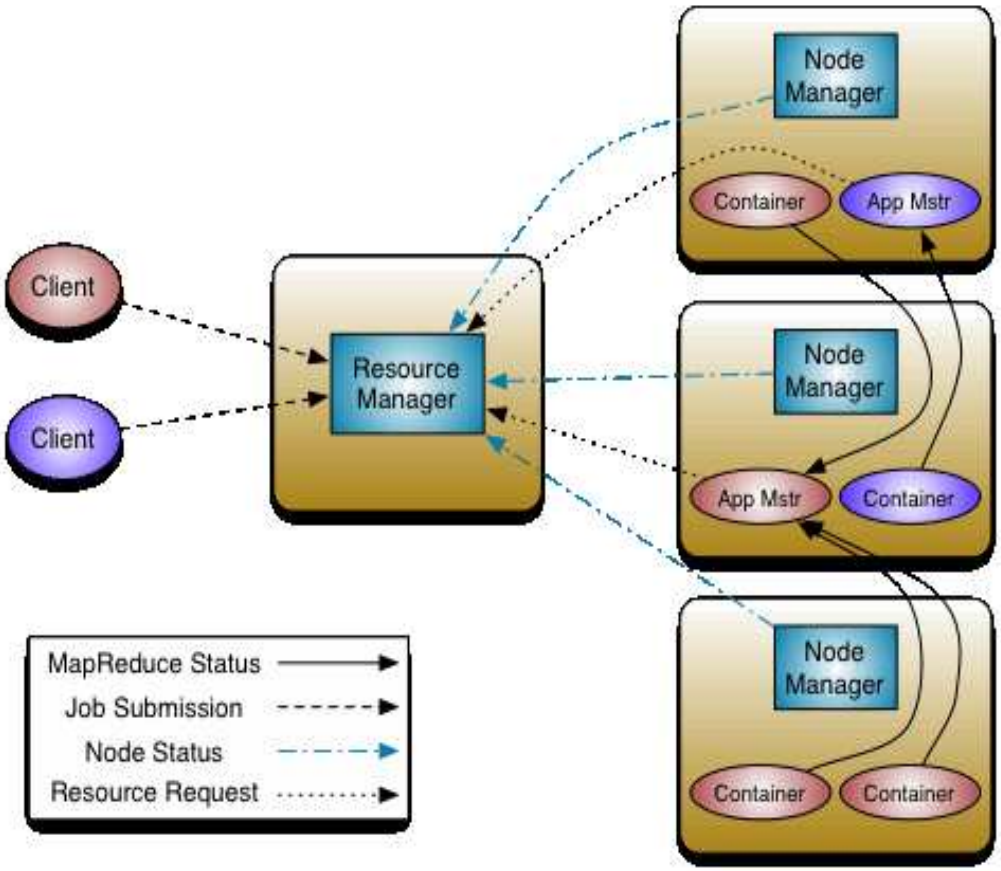
The functionalities of the JobTracker are divided between different components:

- ResourceManager: manages the resources in the cluster
- ApplicationMaster: manages the life cycle of an application

An application is submitted for execution:

- ✓ the ResourceManager will allocate a container (cpu, ram and disk) for the ApplicationMaster process to run
- ✓ the ApplicationMaster requests containers for each map/reduce task
- ✓ the ApplicationMaster starts the tasks by contacting the NodeManagers (a daemon responsible for per node resource monitoring; it reports to the Resource Manager)
- ✓ the tasks report the progress to the ApplicationMaster

New framework: MapReduce 2(YARN)



Source image:

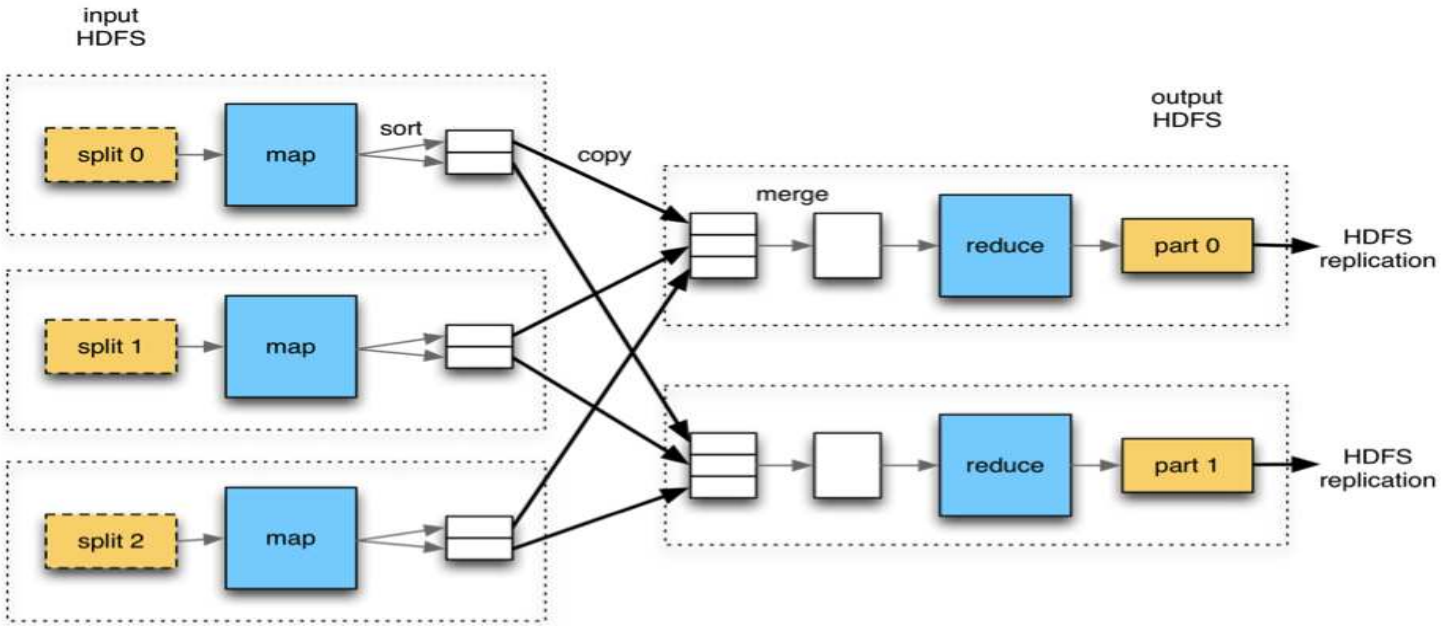
`http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/`

Table of Contents

Hadoop MapReduce

In order to perform computation on a large dataset we need a programming model that can **easily be parallelized**. This is the case for the MapReduce model.

In MapReduce the computation is split into two task: the **map task** and the **reduce task**.



Source image:
`http://blog.cloudera.com/wp-content/uploads/2014/03/ssd1.png`

Table of Contents

Counting the number of occurence of words

Problem. Suppose that we have a huge log file (or several huge log files) representing user queries on Google in the last week. We want to count the number of appearances of the words used. The files are stored on the HDFS. Example:

q1: When did the syrian civil war start?
q2: How many syrian refugees are?
...
q10000000000000: How can I become a cat?

Solution. We will write an application in Java that uses the Apache Hadoop framework.

The first notion that we need to get familiar is the one of a **job**. A MapReduce **job** is an application that processes data via map and reduce tasks.

WordCount.java

Lines 1–29 / 59

```
1  import java.io.IOException;
2  import java.util.StringTokenizer;
3  import org.apache.hadoop.conf.Configuration;
4  import org.apache.hadoop.fs.Path;
5  import org.apache.hadoop.io.IntWritable;
6  import org.apache.hadoop.io.Text;
7  import org.apache.hadoop.mapreduce.Job;
8  import org.apache.hadoop.mapreduce.Mapper;
9  import org.apache.hadoop.mapreduce.Reducer;
10 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
11 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
12
13 public class WordCount {
14   public static class TokenizerMapper
15   extends Mapper<Object, Text, Text, IntWritable>{
16
17     private final static IntWritable one = new IntWritable(1);
18     private Text word = new Text();
19
20     public void map(Object key, Text value, Context context
21         ) throws IOException, InterruptedException {
22       StringTokenizer itr = new StringTokenizer(value.toString());
23       while (itr.hasMoreTokens()) {
24         word.set(itr.nextToken());
25         context.write(word, one);
26       }
27     }
28   }
```

WordCount.java

Lines 30–58 / 59

```
30      public static class IntSumReducer
31      extends Reducer<Text,IntWritable,Text,IntWritable> {
32        private IntWritable result = new IntWritable();
33
34        public void reduce(Text key, Iterable<IntWritable> values,
35            Context context
36            ) throws IOException, InterruptedException {
37          int sum = 0;
38          for (IntWritable val : values) {
39            sum += val.get();
40          }
41          result.set(sum);
42          context.write(key, result);
43        }
44      }
45      public static void main(String[] args) throws Exception {
46        Configuration conf = new Configuration();
47        Job job = Job.getInstance(conf, "word count");
48        job.setMapperClass(TokenizerMapper.class);
49        job.setCombinerClass(IntSumReducer.class);
50        job.setReducerClass(IntSumReducer.class);
51        job.setOutputKeyClass(Text.class);
52        job.setOutputValueClass(IntWritable.class);
53        FileInputFormat.addInputPath(job, new Path(args[0]));
54        FileOutputFormat.setOutputPath(job, new Path(args[1]));
55        job.setInputFormatClass(TextInputFormat.class);
56        job.setOutputFormatClass(TextOutputFormat.class);
57        job.waitForCompletion(true);
58      }
```

The Job Class. Steps to initialize a job:

- provide a configuration of the cluster:

```
46        Configuration conf = new Configuration();
```

- call the constructor with the configuration object and a name for the job

```
47        Job job = Job.getInstance(conf, "word count");
```

- provide an implementation for the Map Class

```
48        job.setMapperClass(TokenizerMapper.class);
```

- provide an implementation for the Combiner Class

```
49        job.setCombinerClass(IntSumReducer.class);
```

- provide an implementation for the Reduce Class

```
50        job.setReducerClass(IntSumReducer.class);
```

The Job Class. Steps to initialize a job:

- specify the type of the output key/value

```
51        job.setOutputKeyClass(Text.class);
52        job.setOutputValueClass(IntWritable.class);
```

- give the location of the input/output of the application

```
53        FileInputFormat.addInputPath(job, new Path(args[0]));
54        FileOutputFormat.setOutputPath(job, new Path(args[1]));
```

- specify how the input/output will be formatted

```
55        job.setInputFormatClass(TextInputFormat.class);
56        job.setOutputFormatClass(TextOutputFormat.class);
```

- the last step: start the job and wait for its completion!

```
57        job.waitForCompletion(true);
```

The Configuration Class.

Setting the environment for a job.

Using the Configuration Class we can configure the components of Hadoop, through pairs of (property, value). For example:

- configurations for NameNode: property dfs.namenode.name.dir (path on the local filesystem where the NameNode stores its information), property dfs.blocksize (size of a block on the HDFS)

  Example:

  conf.setInt('dfs.blocksize', 268435456) - setting the block size to 256 MB

- configurations for the MapReduce applications: property mapreduce.map.java.opt (larger heap size for the jvm of the maps)

A user can add new properties that are needed for a job, before the job is submitted. During the execution of the job, properties are read only. Default values for the properties are stored in XML files on the namenode and datanodes.

Input format of a job

The input of a job is processed by the map function. All classes used to format the input must implement the interface **InputFormat**.

TextInputFormat splits a block of data into lines and sends to the map function one line at the time. Because a map function works with pairs of (key, value), these will be:

- key: byte offset of the position where the line starts
- value: the text of the line.

KeyValueTextInputFormat splits a block of data into lines. It then performs an additional split of each line (the second split looks for a given separator):

- key: the text before a separator (comma, tab, space)
- value: the text after the separator

Output format of a job

The output of a job is written to the file system by the reduce function. All classes used to format the output must implement the interface **OutputFormat**.

One important functionality provided is checking that the **output does not exist** (remember the data access mode is write-once read-many-times).

TextOutputFormat takes an output pair (key,value), converts the key and value to strings and writes them on one line: string(key) separator string(value) endl

Data types optimized for network communication

We often want to send objects over the network or to write them on the disk. For consistency reasons, programmers use the same libraries to serialize and deserialize objects.

Serialization is the process of transforming objects into a sequence of bytes.
Deserialization is the process of transforming a sequence of bytes into objects.

In Apache Hadoop a new standard for serialization/deserialization was introduced: **Writable**. The new format is designed to be more compact, to improve random access and sorting of the records in the stream.

$map : (K1, V1) \rightarrow list(K2, V2)$
$reduce : (K2, list(V2)) \rightarrow list(K3, V3)$
K1-K3, V1-V3 are replaced with Writable types
(LongWritable, DoubleWritable, NullWritable..)

Mapper Class

The framework provides a default Mapper Class.

```
1  public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
2  /** Called once for each key/value pair in the input split. Most applications
3     * should override this, but the default is the identity function. */
4    protected void map(KEYIN key, VALUEIN value,
5                       Context context) throws IOException, InterruptedException {
6      context.write((KEYOUT) key, (VALUEOUT) value);
7    }
8  }
```

Any class that will be used as a Mapper class has to be a subclass of the default class. Most applications need to override the map function.

```
14    public static class TokenizerMapper
15    extends Mapper<Object, Text, Text, IntWritable>{
16
17      private final static IntWritable one = new IntWritable(1);
18      private Text word = new Text();
19
20      public void map(Object key, Text value, Context context
21          ) throws IOException, InterruptedException {
22        StringTokenizer itr = new StringTokenizer(value.toString());
23        while (itr.hasMoreTokens()) {
24          word.set(itr.nextToken());
25          context.write(word, one);
26        }
27      }
28    }
```

Reducer Class

The framework provides a default Reducer Class.

```
1   public class Reducer<KEYIN,VALUEIN,KEYOUT,VALUEOUT> {
2    /** This method is called once for each key. Most applications will define
3      * their reduce class by overriding this method. The default implementation
4      * is an identity function.*/
5     protected void reduce(KEYIN key, Iterable<VALUEIN> values, Context context
6                           ) throws IOException, InterruptedException {
7       for(VALUEIN value: values) {
8         context.write((KEYOUT) key, (VALUEOUT) value);
9       }
10    }
11  }
```

Any class that will be used as a Reducer class has to be a subclass of the default class. Most applications need to override the reduce function.

```
30    public static class IntSumReducer
31    extends Reducer<Text,IntWritable,Text,IntWritable> {
32      private IntWritable result = new IntWritable();
33
34      public void reduce(Text key, Iterable<IntWritable> values,
35          Context context
36          ) throws IOException, InterruptedException {
37        int sum = 0;
38        for (IntWritable val : values) {
39          sum += val.get();
40        }
41        result.set(sum);
42        context.write(key, result);
43      }
```

Context Class

A Context Object provides a view of the job in the Mapper/Reducer. Some important functionalities:

- write objects in a map/reduce function

  context.write(key, new IntWritable(sum));

- access the Configuration object

  Configuration conf = context.getConfiguration();

  int nNodes = conf.getInt('numberNodes', 0);

- but also other functionalities: context.getFileTimestamps(), context.getInputFormatClass(), context.getJobID(), context.getNumReduceTasks() ...

Counter Class. Gathering statistics about the job.

During the execution of a job, there is no direct communication between map and reduce tasks. However, these tasks keep a constant communication with the ApplicationMaster in order to report progress. The communication is done through objects of the class Counter.

Built-in Counters
MAP_INPUT_RECORDS
MAP_OUTPUT_RECORDS
MAP_OUTPUT_RECORDS
REDUCE_OUTPUT_RECORDS
...
User-Defined Java Counters

Each time the desired event occurs (a record is read/written or others), the counter is incremented (locally). The aggregation of the information is performed by the ApplicationMaster.

Counter Class. Gathering statistics about the job.

The default value of a Counter is 0. Incrementing a counter in a Map/Reduce function :

```
1   public void map(Object key, Text value, Context context)
2    throws IOException, InterruptedException {
3   ...
4   Counter c = context.getCounter(myCounters.NUMNODES);
5   c.increment(1);
6   }
```

Retrieving the value of a counter of the end of a job:

```
1   job.waitForCompletion(true);
2   Counters counters = job.getCounters();
3   Counter c = counters.findCounter(myCounters.NUMNODES);
```

For more insight about Apache Hadoop

T Hadoop: The Definitive Guide, Tom White

&&

hands-on approach: coding :)