

Lecture 06

Map Reduce Java

Zoran Djordjević

@Zoran B. Djordjević

1

Reference

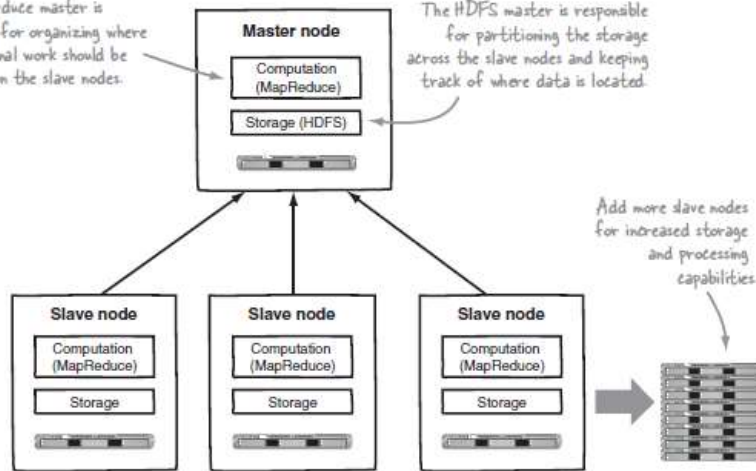
- This set of slides is based on *Hadoop in Practice* by Alex Holmes, Manning 2012
- Hadoop-0.20-MapReduce example classes (MRv1).

@Zoran B. Djordjević

2

High Level Hadoop Architecture

The MapReduce master is responsible for organizing where computational work should be scheduled on the slave nodes.



@Zoran B. Djordjević

3

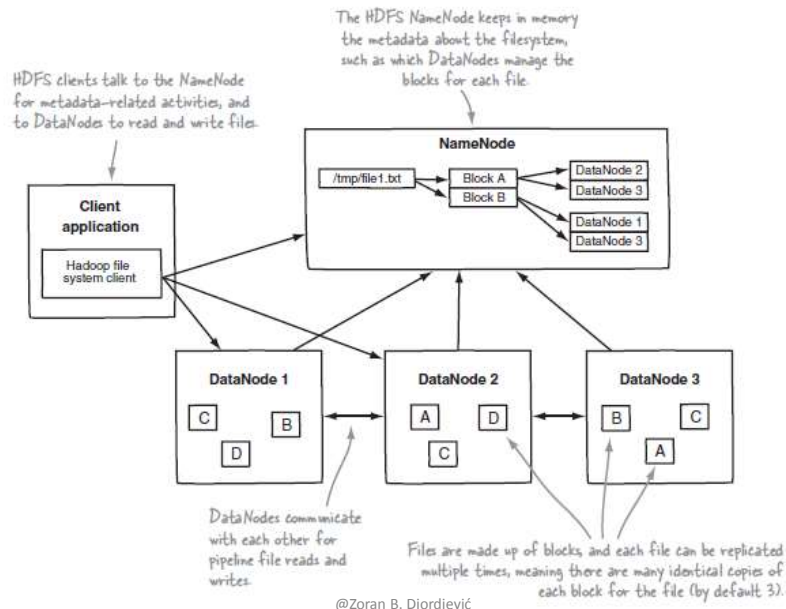
HDFS Hadoop Distributed File System

- HDFS is the storage component of Hadoop. It's a distributed file system that's modeled after the Google File System (GFS).
- HDFS is optimized for high throughput and works best when reading and writing large files (gigabytes and larger). HDFS prefers one file over many files.
- To support this throughput HDFS leverages unusually large (for a file system) block sizes (64 MB, 128 MB) and data locality optimizations to reduce network input/output (I/O). On a regular OS, the block size is 8-32KB.
- Scalability and availability are also key traits of HDFS, achieved in part due to data replication and fault tolerance.
- HDFS replicates files for a configured number of times, is tolerant of both software and hardware failure, and automatically re-replicates data blocks on nodes that have failed.
- The following figure shows a logical representation of HDFS

@Zoran B. Djordjević

4

HDFS Architecture



5

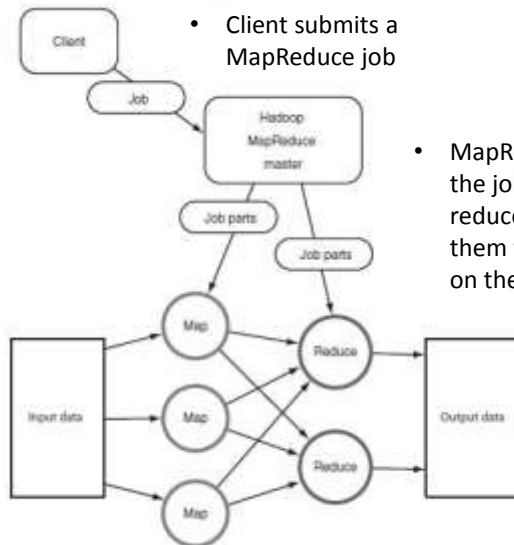
MapReduce

- MapReduce is a batch-based, distributed computing framework modeled after Google's paper on MapReduce (1).
- MapReduce allows you to parallelize work over a large amount of raw data, such as combining web logs with relational data from an OLTP database to model how users interact with your website. This type of work, which could take days or longer using conventional serial programming techniques, can be reduced down to minutes using MapReduce on a large Hadoop cluster.
- MapReduce model simplifies parallel processing by abstracting away the complexities involved in working with distributed systems, such as computational parallelization, work distribution, and dealing with unreliable hardware and software.
- MapReduce allows the programmer to focus on addressing business issues, rather than getting tangled up in distributed system configuration and stability issues.
- MapReduce decomposes work submitted by a client into small parallelized map and reduce workers.
- The map and reduce constructs used in MapReduce are borrowed from those found in the Lisp functional programming language, and use a shared-nothing model to remove any parallel execution interdependencies

@Zoran B. Djordjević

6

Architecture of MapReduce Application



- Client submits a MapReduce job

- MapReduce decomposes the job into map and reduce tasks, and schedules them for remote execution on the slave nodes.

The role of the programmer is to define **map** and **reduce** functions, where the **map** function outputs key/value tuples, which are processed by **reduce** functions to produce the final output.

@Zoran B. Djordjević

7

What determines the Number of Map Tasks

- The number of maps tasks is driven by the number of DFS blocks in the input files.
- The right level of parallelism for maps seems to be around 10-100 maps/node, although this can go up to 300 or so for very cpu-light map tasks. Task setup takes awhile, so it is best if the maps take at least a minute to execute.
- One can control the number of Map task by modifying `JobConf`'s `conf.setNumMapTasks(int num)`. This could increase the number of map tasks, but will not set the number below that which Hadoop determines via splitting the input data.
- The `mapred.map.tasks` parameter is just a hint to the `InputFormat` for the number of maps. The default `InputFormat` behavior is to split the total number of bytes into the right number of fragments. However, in the default case the DFS block size of the input files is treated as an upper bound for input splits. A lower bound on the split size can be set via `mapred.min.split.size`.
- Thus, if you expect 10GB of input data and have 128MB DFS blocks, you'll end up with 82 maps, unless your `mapred.map.tasks` is even larger. Ultimately the `InputFormat` determines the number of maps.
- Number of tasks can radically change the performance of Hadoop. Increasing the number of tasks increases the framework overhead, but increases load balancing and lowers the cost of failures.
- At one extreme is the 1 map/1 reduce case where nothing is distributed. The other extreme is to have 1,000,000 maps/ 1,000,000 reduces where the framework runs out of resources for the overhead.

@Zoran B. Djordjević

8

Interface `InputFormat<K, V>`

- `InputFormat` describes the input-specification for a Map-Reduce job.
- The Map-Reduce framework relies on the `InputFormat` of the job to:
- Validate the input-specification of the job.
- Split-up the input file(s) into logical `InputSplits`, each of which is then assigned to an individual `Mapper`.
- Provide the `RecordReader` implementation to be used to glean input records from the logical `InputSplit` for processing by the `Mapper`.
- The default behavior of file-based `InputFormats`, typically sub-classes of `FileInputFormat`, is to split the input into *logical* `InputSplits` based on the total size, in bytes, of the input files. However, the `FileSystem` blocksize of the input files is treated as an upper bound for input splits. A lower bound on the split size can be set via `mapreduce.input.fileinputformat.split.minsize`.
- Clearly, logical splits based on input-size is insufficient for many applications since record boundaries are to be respected. In such cases, the application has to also implement a `RecordReader` on whom lies the responsibility to respect record-boundaries and present a record-oriented view of the logical `InputSplit` to the individual task.

@Zoran B. Djordjević

9

Number of Reduce Tasks

- The right number of reduces seems to be 0.95 or $1.75 * (\text{nodes} * \text{mapred.tasktracker.tasks.maximum})$.
- At 0.95 all of the reduces can launch immediately and start transferring map outputs as the maps finish.
- At 1.75 the faster nodes will finish their first round of reduces and launch a second round of reduces doing a much better job of load balancing.
- Currently the number of reduces is limited to roughly 1000 by the buffer size for the output files (`io.buffer.size * 2 * numReduces << heapSize`). This will be fixed at some point, but until it is it provides a pretty firm upper bound.
- The number of reduces also controls the number of output files in the output directory, but usually that is not important because the next map/reduce step will split them into even smaller splits for the maps.
- The number of reduce tasks can also be increased in the same way as the map tasks, via `JobConf`'s `conf.setNumReduceTasks(int num)`.

@Zoran B. Djordjević

10

map Fuction

- The **map** function takes as input a key/value pair, which represents a logical record from the input data source.
- In the case of a file, this could be a line, where line number is the key and line itself the value, or if the input source is a database table, the key could be the primary key of the row and the value the row itself.

```
map(key1, value1) → list(key2, value2)
```

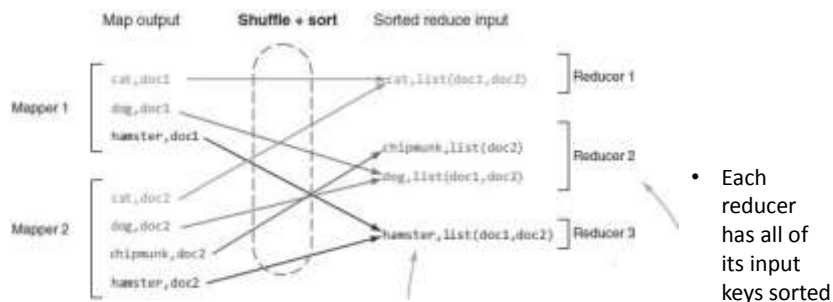
- The map function produces zero or more output key/value pairs for that one input pair.
- For example, if the map function is a filtering function, it may only produce output if a certain condition is met. Map function could be a demultiplexing operation, with a single input key/value yielding multiple key/value outputs.
- Usually, a map functions produces a smaller number of key-value pairs than it consumes. There is nothing in the framework that prevents a map function to produce a list with more elements than the one consumed.

@Zoran B. Djordjević

11

Shuffle and Sort Phase

- The shuffle and sort phases are responsible for two primary activities: determining the reducer that should receive the map output key/value pair (called partitioning); and ensuring that, for a given reducer, all its input keys are sorted.



- Map outputs for the same key go to the same reducer, and are then sorted and combined together to form a single input record for the reducer.
- A lot of the power of MapReduce is in what occurs in between the map output and the reduce input, i.e. in the shuffle and sort phases

@Zoran B. Djordjević

12

Partition Function

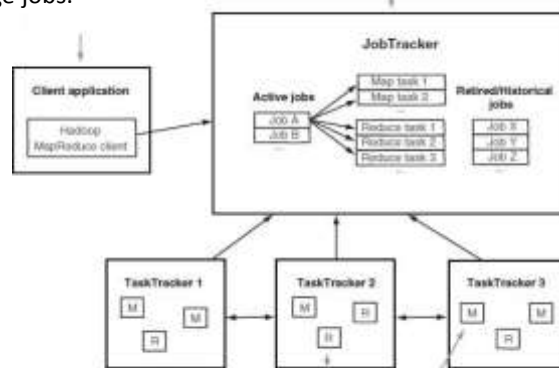
- Each *Map* function output is allocated to a particular *reducer* by the application's *partition* function for [sharding](#) purposes. The *partition* function is given the key and the number of reducers and returns the index of the desired *reducer*.
- A typical default is to [hash](#) the key and use the hash value [modulo](#) the number of *reducers*. It is important to pick a partition function that gives an approximately uniform distribution of data per shard for [load-balancing](#) purposes, otherwise the MapReduce operation can be held up waiting for slow reducers (reducers assigned more than their share of data) to finish.
- Between the map and reduce stages, the data is *shuffled* (parallel-sorted / exchanged between nodes) in order to move the data from the map node that produced it to the shard in which it will be reduced. The shuffle can sometimes take longer than the computation time depending on network bandwidth, CPU speeds, data produced and time taken by map and reduce computations.

@Zoran B. Djordjević

13

MapReduce Control Architecture for MRv1

- MapReduce clients talk to the JobTracker to launch and manage jobs.
- The JobTracker coordinates activities across the slave TaskTracker processes. It accepts MapReduce job requests from clients and schedules map and reduce tasks on TaskTrackers to perform the work.



Map and reduce child processes.

The *TaskTracker* is a daemon process that spawns child processes to perform the actual map or reduce work. Map tasks typically read their input from HDFS, and write their output to the local disk. Reduce tasks read the map outputs over the network and write their outputs back to HDFS.

@Zoran B. Djordjević

14

Working with files

- Before we can run Hadoop programs on data stored in HDFS, we'll need to put the data into HDFS first. Let's assume we've already formatted and started a HDFS file system. We are working with a pseudo-distributed configuration as a playground.
- Let's create a directory and put a file in it.
- HDFS has a default working directory of `/user/$USER`, where `$USER` is login user name. The directory is not automatically created for us.
- We create the directory with the `mkdir` command. For the purpose of illustration, we use username `joe`. On CDH4, when creating new user directory, you need to `sudo` your commands as user `hdfs`

```
# sudo -u hdfs hadoop fs -mkdir /user/joe
# sudo -u hdfs fs -chown joe /user/joe
# sudo -u hdfs hadoop fs -ls /user
drwxr-xr-x - joe supergroup 0 2013-03-15 /user/joe
```

@Zoran B. Djordjević

15

Working with files

- Hadoop's `mkdir` command automatically creates parent directories if they don't already exist, similar to the Unix `mkdir` command with the `-p` option. So the preceding command will create the `/user` directory too.
- Let's check on the directories with the `ls` command.

```
hadoop fs -ls /
```

- You'll see the `/user` directory at the root `/` directory.

```
drwxr-xr-x - joe supergroup 0 2009-01-14 10:23 /user
```

- If you want to see all the subdirectories, in a way similar to Unix's `ls` with the `-R` option, you can use Hadoop's `ls -R` command.

```
hadoop fs -ls -R /
```

- You'll see all the files and directories recursively.

```
drwxrwxrwt - hdfs supergroup 0 2013-03-09 10:01 /tmp
drwxr-xr-x - hdfs supergroup 0 2013-03-15 07:56 /user
drwxr-xr-x - joe supergroup 0 2013-03-15 07:56 /user/joe
drwxr-xr-x - cloudera supergroup 0 2013-03-14 13:53
/user/cloudera
```

@Zoran B. Djordjević

16

Copying a file to new HDFS directory

- We are ready to add files to HDFS.
- We should first become user `joe`

```
[cloudera@localhost ~]$ su -- joe
Password: xxxxxxxxxx
[joe@localhost cloudera]$
```
- In the shared folder of my VM I have a file called `all-bible`.

```
$ hadoop fs -put /mnt/hgfs/sharedfolder/all-bible /user/joe
$ hadoop fs -ls
Found 1 items
-rw-r--r--  1 joe supergroup 5258688 2013-03-15 08:31 all-bible
```
- The number 1 in the above listing tells us how many times is a particular file replicated. Since we have a single machine, 1 is appropriate.
- The replication factor is 3 by default, but could be set to any number.

@Zoran B. Djordjević

17

Fetching and examining files from HDFS

- The Hadoop command `get` does the exact reverse of `put`. It copies files from HDFS to the local file system.
- To retrieve file `all-bible` from HDFS and copy it into the current local working directory, we run the command

```
hadoop fs -get all-bible .
```
- A way to examine the data is to display data. For small files, Hadoop `cat` command is convenient.

```
hadoop fs -cat all-bible
```
- We can use any Hadoop file command with Unix pipes to forward its output for further processing by another Unix commands. For example, if the file is huge (as typical Hadoop files are) and you're interested in a quick check of its content, you can pipe the output of Hadoop's `cat` into a Unix `head`.

```
hadoop fs -cat all-bible | head
```
- Hadoop natively supports `tail` command for looking at the last kilobyte of a file.

```
hadoop fs -tail all-bible
```

@Zoran B. Djordjević

18

Deleting files and directories

- Hadoop command for removing files is `rm`.

```
hadoop fs -rm example.txt
```

- To delete files and directories recursively use

```
hadoop fs -rm -R directory/*
```

- To delete empty directories use

```
hadoop fs -rmdir directory
```

@Zoran B. Djordjević

19

Let us run an example, WordCount ☹

- Hadoop CDH4.6 tar balls contain source code, including examples

<http://www.cloudera.com/content/support/en/documentation/CDH-tarballs/CDH-tarballs-latest.html> . Let us download Apache Hadoop tarball:

```
hadoop-src-2.0.0-cdh4.6.0.tar.gz
```

- One could get that file from <http://hadoop.apache.org> , as well
- On Window's side you could use 7-zip to open that file and turn it first into a tar archive, and then into a directory `hadoop-2.0.0-cdh4.6.0`
- You can copy the file to VM's `sharedfolder` and un-tar it on Linux side.
- In that case, on VM command prompt, type

```
$ cd /mnt/hgfs/sharedfolder
```

```
$ tar -zxvf hadoop-src-2.0.0-cdh4.6.0.tar.gz
```

- `-z` uncompresses the archive with `gzip` command.

- You will get directory `src`. Under that directory and the directory under `hadoop-mapreduce-project` you can find examples for MapReduce jobs. We could navigate to

```
src\hadoop-mapreduce-project\hadoop-mapreduce-examples\src\main
```

- and fetch World famous `WordCount.java` program.

@Zoran B. Djordjević

20

WordCount.java

```
package org.apache.hadoop.examples;
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;
import org.apache.hadoop.util.GenericOptionsParser;
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);    }
            }
    }
}
```

@Zoran B. Djordjević

21

WordCount.java

```
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);
    }
}
```

@Zoran B. Djordjević

22

WordCount.java

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new
        GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length != 2) {
        System.err.println("Usage: wordcount <in> <out>");
        System.exit(2);
    }
    Job job = new Job(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(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

@Zoran B. Djordjević

23

Compile and Run

- In the home directory of user cloudera I created a directory `wc_classes` and copied `WorCount.java` there

```
$cp /mnt/hfgs/sharedfolder/./examples/WordCount.java ~/wc_classes
```

- Compiling this class turned into a small issue. Literature tells you to add `hadoop-core.jar` and `hadoop-client.jar` to the classpath and do something like:

```
$javac -classpath "/usr/lib/hadoop-core.jar:/usr/lib/hadoop-client.jar" -d . WordCount.java
```

- In order to find those jar files I did the following

```
$su      # became a root user
$cd /    # went to the root of the directory tree
$find . -name hadoop-core.jar -print
```

- and found

```
./usr/lib/hadoop-0.20-mapreduce/hadoop-core.jar
```

- By the way, I ask `find` to start looking right here (`.`) for a file with name `hadoop-core.jar` and once it finds it, to print file location.
- I was less lucky with `hadoop-client.jar`. I did not find it.

@Zoran B. Djordjević

24

Hadoop classpath command

- We are almost sure that Hadoop installation has all the jar-s we need and that Hadoop should know about them.
- Hadoop command `classpath` serves that purpose. It reveals the essential jars. If you type

```
$hadoop classpathhadoop
```

- You get them all.

```
/etc/hadoop/conf:/usr/lib/hadoop/lib/*:/usr/lib/hadoop/./
*:/usr/lib/hadoop-hdfs/./:/usr/lib/hadoop-
hdfs/lib/*:/usr/lib/hadoop-hdfs/.//*:/usr/lib/hadoop-
yarn/.//*:/usr/lib/hadoop-0.20-
mapreduce/./:/usr/lib/hadoop-0.20-
mapreduce/lib/*:/usr/lib/hadoop-0.20-mapreduce/.//*
```

- You might not need them all, but that is another much smaller problem.
- An important feature of Linux (Unix) is that you can invoke a command within another command by placing the former between reverse ticks, like ``hadoop classpath``

@Zoran B. Djordjević

25

Compiling WordCount.java

- In the directory `wc_classes`, as user `cloudera`, we type:

```
$ javac -classpath `hadoop classpath` -d . WordCount.java
```

- `-d .` tells `javac` to start building package directories starting here (`.`).

```
$ ls
```

```
org WordCount.java
```

```
$ cd org/apache/hadoop/examples
```

```
$ pwd
```

```
/home/cloudera/wc_classes/org/apache/hadoop/examples
```

```
$ ls
```

```
WordCount.class WordCount$IntSumReducer.class
```

```
WordCount$TokenizerMapper.class
```

- It appears that command `hadoop classpath` fed the list of hadoop jars to the `-classpath` option of `javac` command and the compilation ran smoothly.
- We ended up with three class files because class file `WordCount.java` had two inner classes besides the main `WordCount` class.

@Zoran B. Djordjević

26

Jaring WordCount MapReduce program

- Before trying to run our compiled class, we need to `jar` it. We type

```
$ jar -cvf wordcount2.jar org/*
```

```
added manifest
adding: org/apache/(in = 0) (out= 0) (stored 0%)
adding: org/apache/hadoop/(in = 0) (out= 0) (stored 0%)
adding: org/apache/hadoop/examples/(in = 0) (out= 0) (stored 0%)
adding:
org/apache/hadoop/examples/WordCount$TokenizerMapper.class (in =
1790) (out= 765) (deflated 57%)
adding: org/apache/hadoop/examples/WordCount.class (in = 1911)
(out= 996) (deflated 47%)
adding:
org/apache/hadoop/examples/WordCount$IntSumReducer.class (in =
1789) (out= 747) (deflated 58%)
```

```
$ ls
```

```
org wordcount2.jar WordCount.java
```

@Zoran B. Djordjević

27

Preparing HDFS input and output directories

- MapReduce jobs read their inputs from and deliver their outputs to HDFS files in HDFS directories `input` and `output`.
- Switch to user `joe` and, if you have not done that already, create HDFS directory `input` and copy a file, `all-bible`, to that HDFS directory

```
$ hadoop fs -mkdir input
$ hadoop fs -copyFromLocal /mnt/.../all-bible input
$ hadoop fs -ls input
rw-r--r--  1 cloudera supergroup    5258688 2013-03-11
14:12 input/all-bible
```

- You should also make sure that the `output` directory is not there, since Hadoop will give you an error otherwise.

```
$ hadoop fs -rm -R output
```

@Zoran B. Djordjević

28

Running WordCount MapReduce program

- On the command prompt, on the single line, we type `hadoop jar`:
\$ `hadoop jar wordcount2.jar org.apache.hadoop.examples.WordCount input output`

```
13/03/15 10:33:41 WARN mapred.JobClient: Use GenericOptionsParser for parsing the
arguments. Applications should implement Tool for the same.
13/03/15 10:33:42 INFO input.FileInputFormat: Total input paths to process : 1
13/03/15 10:33:44 INFO mapred.JobClient: Running job: job_201303141804_0001
13/03/15 10:33:45 INFO mapred.JobClient: map 0% reduce 0%
14/03/07 07:36:59 INFO mapred.JobClient: map 44% reduce 0%
14/03/07 07:37:02 INFO mapred.JobClient: map 100% reduce 0%
. . .
14/03/07 20:21:21 INFO mapred.JobClient: Job Counters
14/03/07 20:21:21 INFO mapred.JobClient:   Launched map tasks=1
14/03/07 20:21:21 INFO mapred.JobClient:   Launched reduce tasks=1
14/03/07 20:21:21 INFO mapred.JobClient:   Data-local map tasks=1
. . . . .
13/03/15 10:34:08 INFO mapred.JobClient:   Reduce input records=60756
13/03/15 10:34:08 INFO mapred.JobClient:   Reduce output records=60756
13/03/15 10:34:08 INFO mapred.JobClient:   Spilled Records=203990
13/03/15 10:34:08 INFO mapred.JobClient:   CPU time spent (ms)=8970
13/03/15 10:34:08 INFO mapred.JobClient:   Physical memory (bytes)
snapshot=205197312
13/03/15 10:34:08 INFO mapred.JobClient:   Virtual memory (bytes)
snapshot=772063232
13/03/15 10:34:08 INFO mapred.JobClient:   Total committed heap usage
(bytes)=132190208
```

@Zoran B. Djordjević

29

Examine the Results

```
$ hadoop fs -ls output
Found 3 items
-rw-r--r-- 1 cloudera supergroup 0 2013-03-15 10:34
output/_SUCCESS
drwxr-xr-x - cloudera supergroup 0 2013-03-15 10:33 output/_logs
-rw-r--r-- 1 cloudera supergroup 717079 2013-03-15 10:34
output/part-r-00000
$ hadoop fs -cat output/part-r-00000 | tail -20
youth. 15
youth: 7
youth; 8
youth? 2
youthful 1
youths 1
youths,1
zeal 13
zeal, 3
zealous8
zealously 2
{ 12
```

@Zoran B. Djordjević

30

Organization of WordCount.java, map()

- Java class WordCount contains two inner classes.
- MAP routine is in the inner class TokenizerMapper extending Mapper class.
- Types listed with Mapper class <Object, Text, Text, IntWritable> are the key/value types for your inputs and outputs.

```
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()); # Casts token as Text
            context.write(word, one);
        }
    }
}
```

@Zoran B. Djordjević

31

Organization of WordCount, reduce()

- REDUCE routine is implemented by the inner class IntSumReducer that extends class Reducer.
- Types next to Reduce are the input/output types
- reduce() is called once per unique output key of Mapper (word) and is fed a list of values of word counts, i.e. values of the Mapper.
- reduce() iterates over all supplied counts (list values) and sums them.
- Finally, reduce() writes the result to the Context.

```
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) { # Iterate over all values (1-s)
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

@Zoran B. Djordjević

32

Organization of WordCount, driver code, main()

- The last step is to write the driver code that will set all the necessary properties to configure and run MapReduce job.
- We need to let the framework know what classes should be used for the map and reduce functions, and also where our input and output is located.
- By default MapReduce assumes you're working with text; if you were working with more complex text structures, or altogether different data storage technologies, you would need to tell MapReduce how it should read and write from these data sources and sinks.

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new
        GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length != 2) {
        System.err.println("Usage: wordcount <in> <out>");
        System.exit(2);
    }
    . . . .
}
```

- Class Configuration is the container for job configs. Its content is available to both mapper and reducer.
- GenericOptionsParser

@Zoran B. Djordjević

33

WordCount.java, driver code, main()

```
Job job = new Job(conf, "word count");
```

- The setJarByClass method of class Job determines the JAR that contains the class that is passed-in, which is copied by Hadoop into the cluster and subsequently set in the Task's classpath so that your MapReduce classes are available to the Task.

```
job.setJarByClass(WordCount.class);
```

- Method setMapperClass identifies the Map class

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

- We did not write Combiner or Reducer, but could use a standard ones

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

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

```
job.setOutputKeyClass(Text.class);
```

```
job.setOutputValueClass(IntWritable.class);
```

- FileInputFormat and FileOutputformat are standard Hadoop classes describing input and output text files.

```
FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
```

```
FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
```

```
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

```
}
}
```

@Zoran B. Djordjević

34

org.apache.hadoop.util.GenericOptionsParser

- GenericOptionsParser is a utility to parse command line arguments generic to the Hadoop framework.
- GenericOptionsParser recognizes several standard command line arguments, enabling applications to easily specify a namenode, a jobtracker, additional configuration resources etc.
- The supported generic options are:
 - conf <configuration file> specify a configuration file
 - D <property=value> use value for given property
 - fs <local|namenode:port> specify a namenode
 - jt <local|jobtracker:port> specify a job tracker
 - files <comma separated list of files> specify comma separated files to be copied to the map reduce cluster
 - libjars <comma separated list of jars> specify comma separated jar files to include in the classpath.
- Examples:

```
$ bin/hadoop dfs -fs darwin:8020 -ls /data list /data directory
in dfs with namenode darwin:8020
$ bin/hadoop dfs -D fs.default.name=darwin:8020 -ls /data list
/data directory in dfs with namenode darwin:8020
$ bin/hadoop dfs -conf hadoop-site.xml -ls /data list /data
directory in dfs with conf specified in hadoop-site.xml
```

@Zoran B. Djordjević

35

org.apache.hadoop.mapreduce.Job

Public class Job extends

org.apache.hadoop.mapreduce.task.JobContext

- Class Job is submitter's view of the Job.
- It allows the user to configure the job, submit it, control its execution, and query the state.
- The set methods only work until the job is submitted, afterwards they will throw an IllegalStateException.
- Normally the user creates the application, describes various facets of the job via [Job](#) and then submits the job and monitor its progress.

@Zoran B. Djordjević

36

You could write MapReduce programs in local Eclipse

- Go to <http://www.eclipse.org/downloads/>
- Select your operating system and download.
- For example, I downloaded eclipse-jee-juno-SR2-win64.zip
- You just unzip the file on your C: drive and you are ready to use it.
- If you are using CDH4.6 and have JDK1.7_51 installed on your VM, please install the same JDK on you local machine.
- With Java you can have several installations. You just need to change JAVA_HOME to point to the one you want to use currently.
- Similarly, your current Java needs to be present in you PATH variable as %JAVA_HOME%\bin; .
- In C:\eclipse, you will see eclipse.exe. You can make a shortcut or not. Just click on the executable and Eclipse will open.
- Of course, you can run Eclipse on your CentOS instance as well.
- You do what ever you find convenient.

@Zoran B. Djordjević

37

Your First Glimps of Eclipse

- If you are new to Java, and have tons of time, you should read the Overview, What's New section and review all Tutorials and Samples.
- They are really good.



@Zoran B. Djordjević

38

Compiling code in Eclipse

- You need to create project.
- Go to **File > New > Java Project**
- Give project a name, i.e. **MapReduce**.
- Click **Next** and **Finish**
- New project will show up in **Package Explorer**.
- If asked to accept **Java Perspective**, do accept.
- **Perspective** is an arrangement of tools on your Eclipse adjusted to the nature of your current project.



@Zoran B. Djordjević

39

Create new Java package

- Right click on **src** under your project name and select **New > Package**.
- Since we want to compile class **WordCount** in package **org.apache.hadoop.examples**, use that as the package name.
- Next open your **sharedfolder** directory, this time from you PC side, highlight class **WordCount.java** in your **hadoop-mapreduce/./examples** folder and simply drag it to the newly created package.
- Eclipse will tell you right way that you have many errors (48). The issue is that we have not supplied Eclipse with the **classpath**, the way we supplied it to **javac** on the Linux command prompt.
- Hadoop's **classpath** command will not work on you PC/Mac.
- You have to set what Eclipse calls the **Build Path**. You also have to have necessary jars on your PC.

@Zoran B. Djordjević

40

Build Path

- Right click on you project (MapReduce) > Build Path > Configure Build Path> Libraries.
- In the widget that opens, select Add External JARs.
- If you have not already expanded Hadoop tar ball
hadoop-2.0.0-cdh4.6.0.tar.gz
- in the directory /mnt/hgfs/sharedfolder, type on the Linux side
\$tar -zxvf hadoop-2.0.0-cdh4.6.0.tar.gz
- or the same command from your Cygwin prompt or use WinZIP or 7-Zip from the Windows side. The last two appear to work as well. In the expanded directory, in the folder
hadoop-2.0.0-cdh4.6.0\share\hadoop\mapreduce1
- there are several jars. I added hadoop-core-2.0.0-mr1-cdh4.6.0.jar to the Build Path. My error count went down to 2. Build was complaining about missing org.apache.hadoop.conf.Configuration class. I added
hadoop-2.0.0-cdh4.6.0\share\hadoop\common\hadoop-common-2.0.0-cdh4.6.0.jar
- Build was complaining about missing : org.apache.commons.cli.Options. That is an Apache Commons class, not a Hadoop class. Go to
http://commons.apache.org/proper/commons-cli/download_cli.cgi
- And download: commons-cli-1.2-bin.ta.gz.
- Expand and add commons-cli-1.2.jar to you Build Path. U r done. Almost.

@Zoran B. Djordjević

41

Export your Project

- We need to package compiled class WordCount as a jar.
- Right click on your project. Select Export > Java > JAR file
- Select src folder, your package and then leave selected only WordCount.java object.
- Specify where you want your jar saved and how you want it named. I named mine wordcount3.jar to distinguish it from the one I named wordcount2.jar, earlier. Click Finish.
- There are a few other things you can select but you do not have to worry about them now.
- You have generated new wordcount3.jar file.
- You can copy that file to your sharedfolder and transfer it to the Linux box and run it with the help of hadoop jar command the same way as you ran wordcount2.jar which we generated on the Linux side.

@Zoran B. Djordjević

42

Export jar Widget



@Zoran B. Djordjević

43

Hadoop Data Types

- The MapReduce framework uses keys and values. Though we often talk about certain keys and values as integers, strings, and so on, they are not exactly standard Java classes, such as Integer, String, and so forth.
- This is because the MapReduce framework has a certain defined way of serializing the key/value pairs in order to move them across the cluster's network, and only classes that support this kind of serialization can function as keys or values in the framework.
- More specifically, classes that implement the `Writable` interface can be values, and classes that implement the `WritableComparable<T>` interface can be either keys or values.
- Note that the `WritableComparable<T>` interface is a combination of the `Writable` and `java.lang.Comparable<T>` interfaces.
- We need the comparability requirement for keys because they will be sorted at the reduce stage, whereas values are simply passed through.
- Hadoop comes with a number of predefined classes that implement `WritableComparable`, including wrapper classes for all the basic data types.

@Zoran B. Djordjević

44

Frequently used Types

Class	Description
<code>BooleanWritable</code>	Wrapper for a standard Boolean variable
<code>ByteWritable</code>	Wrapper for a single byte
<code>DoubleWritable</code>	Wrapper for a Double
<code>FloatWritable</code>	Wrapper for a Float
<code>IntWritable</code>	Wrapper for a Integer
<code>LongWritable</code>	Wrapper for a Long
<code>Text</code>	Wrapper to store text using the UTF8 format
<code>NullWritable</code>	Placeholder when the key or value is not needed

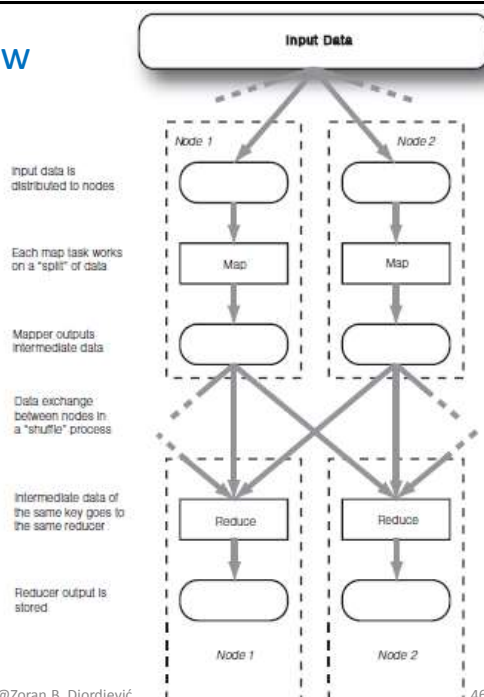
- Keys and values can take on types beyond the basic ones which Hadoop natively supports.
- You can create your own custom type as long as it implements the `Writable` (or `WritableComparable<T>`) interface.

@Zoran B. Djordjević

45

MapReduce Data Flow

- The general MapReduce data flow. Note that after distributing input data to different nodes, the only time nodes communicate with each other is at the "shuffle" step.
- This restriction on communication greatly helps scalability.



@Zoran B. Djordjević

46

Mapper

- Mapper maps input key/value pairs to a set of intermediate key/value pairs.
- Maps are the individual tasks which transform input records into a intermediate records. The transformed intermediate records need not be of the same type as the input records. A given input pair may map to zero or many output pairs.
- To serve as the mapper, a class extends the Mapper class. Mapper class includes two methods that effectively act as the constructor and destructor for the class:

```
void setup(org.apache.hadoop.mapreduce.Mapper.Context context)
```

- is called once at the beginning of the task. In this method you can extract the parameters set either by the configuration XML files or in the main class of your application. Call this function before any data processing begins.

@Zoran B. Djordjević

47

Mapper

- `void cleanup(org.apache.hadoop.mapreduce.Mapper.Context context)` is called once at the end of the task as the last action before the map task terminates, this function should wrap up any loose ends—database connections, open files, and so on.
- `void map(KEYIN key, VALUEIN value, org.apache.hadoop.mapreduce.Mapper.Context context)` is called once for each key/value pair in the input split. is responsible for the data processing step. It utilizes Java generics of the form `Mapper<K1, V1, K2, V2>` where the key classes and value classes implement the `WritableComparable` and `Writable` interfaces, respectively. Its single method is to process an individual (key/value) pair:
- `void run(org.apache.hadoop.mapreduce.Mapper.Context context)` Expert users can override this method for more complete control over the execution of the Mapper.

@Zoran B. Djordjević

48

Supplied Mappers

- Hadoop provides a few useful mapper implementations.
- You can use them as mappers in you application if they provide the functionality you need.

Class	Description
<code>IdentityMapper<K, V></code>	Implements <code>Mapper<K,V,K,V></code> and maps inputs directly to outputs
<code>InverseMapper<K, V></code>	Implements <code>Mapper<K,V,V,K></code> and reverses the key/value pair
<code>RegexMapper<K></code>	Implements <code>Mapper<K,Text,Text,LongWritable></code> and generates a (match, 1) pair for every regular expression match
<code>TokenCountMapper<K></code>	Implements <code>Mapper<K,Text,Text,LongWritable></code> and generates a (token, 1) pair when the input value is tokenized

@Zoran B. Djordjević

49

Reducers

- Reduces a set of intermediate values which share a key to a smaller set of values.
- Reducer implementations can access the `Configuration` for the job via the `JobContext.getConfiguration()` method.
- Reducer has 3 primary phases:

Shuffle

- The Reducer copies the sorted output from each Mapper using HTTP across the network.

Sort

- The framework merge sorts Reducer inputs by keys (since different Mappers may have output the same key).
- The shuffle and sort phases occur simultaneously i.e. while outputs are being fetched they are merged.

Reduce

- In this phase the `reduce(Object, Iterable, Context)` method is called for each <key, (collection of values)> in the sorted inputs.

@Zoran B. Djordjević

50

Reducer method summary

- `protected void cleanup(org.apache.hadoop.mapreduce.Reducer.Context context)` **Called once at the end of the task.**
- `protected void reduce(KEYIN key, Iterable<VALUEIN> values, org.apache.hadoop.mapreduce.Reducer.Context context)` **This method is called once for each key.**
- `void run(org.apache.hadoop.mapreduce.Reducer.Context context)` **Advanced application writers can use the run() method to control how the reduce task works.**
- `protected void setup(org.apache.hadoop.mapreduce.Reducer.Context context)` **Called once at the start of the task.**

@Zoran B. Djordjević

51

Combiners

- In many situations with MapReduce applications, we may wish to perform a “local reduce” before we distribute the mapper results. In the `WordCounter` example, if the job processes a document containing the word “the” 574 times, it’s much more efficient to store and shuffle the pair (“the”, 574) once instead of the pair (“the”, 1) 574 times.
- Hadoop provides several ready made combiners. It also provides several ready made mappers and reducers.
- On the following slide we see an implementation of the same `WordCount` example using only provided tools.

@Zoran B. Djordjević

52

Use of Predefine Mapper and Reducer

- For our simple example, we actually have to write only the driver for this MapReduce program because we could use Hadoop's predefined `TokenCountMapper` class and `LongSumReducer` class.

```
public class WordCount2 {
    public static void main(String[] args) {
        JobClient client = new JobClient();
        JobConf conf = new JobConf(WordCount2.class);
        FileInputFormat.addInputPath(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(LongWritable.class);
        conf.setMapperClass(TokenCountMapper.class);
        conf.setCombinerClass(LongSumReducer.class);
        conf.setReducerClass(LongSumReducer.class);
        client.setConf(conf);

        try {
            JobClient.runJob(conf);
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

@Zoran B. Djordjević

53

Reading and Writing

- Input data usually resides in large files, typically tens or hundreds of gigabytes or even more.
- One of the fundamental principles of MapReduce's processing power is splitting of the input data into *chunks*. You process these chunks in parallel using multiple machines.
- In Hadoop terminology these chunks are called ***input splits***.
- The size of each split should be small enough for a more granular parallelization. (If all the input data is in one split, then there is no parallelization.)
- On the other hand, each split shouldn't be so small that the overhead of starting and stopping the processing of a split becomes a large fraction of execution time.
- The principle of dividing input data (which often can be one single massive file) into splits for parallel processing explains some of the design decisions behind Hadoop's generic FileSystem as well as HDFS in particular.

@Zoran B. Djordjević

54

FSDataInputStream

- Hadoop's File System provides the class `FSDataInputStream` for file reading rather than using Java's `java.io.DataInputStream`.
- `FSDataInputStream` extends `DataInputStream` with random read access, a feature that MapReduce requires because a machine may be assigned to process a split that sits right in the middle of an input file. Without random access, it would be extremely inefficient to have to read the file from the beginning until you reach the location of the split.
- HDFS is designed for storing data that MapReduce will split and process in parallel. HDFS stores files in blocks spread over multiple machines. Roughly speaking, each file block is a split.
- As different machines will likely have different blocks, parallelization is automatic if each split/ block is processed by the machine that it's residing at. Furthermore, as HDFS replicates blocks in multiple nodes for reliability, MapReduce can choose any of the nodes that have a copy of a split/block.

@Zoran B. Djordjević

55

InputFormat

- The way an input file is split up and read by Hadoop is defined by one of the implementations of the `InputFormat` interface . `TextInputFormat` is the default Input-Format implementation, and it's the data format we've been implicitly using up to now.
- It's often useful for input data that has no definite key value, when you want to get the content one line at a time. The key returned by `TextInputFormat` is the byte offset of each line, and we have yet to see any program that uses that key for its data processing.

@Zoran B. Djordjević

56

Popular InputFormat classes

InputFormat	Description
TextInputFormat	Each line in the text files is a record. Key is the byte offset of the line, and value is the content of the line. key: LongWritable value: Text
KeyValueTextInputFormat	Each line in the text files is a record. Key is the byte offset of the line, and value is the content of the line. key: LongWritable value: Text Each line in the text files is a record. The first separator character divides each line. Everything before the separator is the key, and everything after is the value. The separator is set by the <code>key.value.separator.in.input</code> line property, and the default is the tab (<code>\t</code>) character. key: Text value: Text
SequenceFileInputFormat <K, V>	An InputFormat for reading in sequence files. Key and value are user defined. Sequence file is a Hadoop-specific compressed binary file format. It's optimized for passing data between the output of one MapReduce job to the input of some other MapReduce job. key: K (user defined) value: V (user defined)
NLineInputFormat	Same as TextInputFormat, but each split is guaranteed to have exactly N lines. The <code>mapred.line.input.format.linespermap</code> property, which defaults to one, sets N. key: LongWritable value: Text

@Zoran B. Djordjević

57

OutputFormat

- MapReduce outputs data into files using the `OutputFormat` class, which is analogous to the `InputFormat` class. The output has no splits, as each reducer writes its output only to its own file.
- The output files reside in a common directory and are typically named `part-nnnnn`, where *nnnnn* is the partition ID of the reducer.
- `RecordWriter` objects format the output and `RecordReader`s parse the format of the input.
- Hadoop provides several standard implementations of `OutputFormat`. Almost all the ones we deal with inherit from the `FileOutputFormat` abstract class;
- `InputFormat` classes inherit from `FileInputFormat`.
- You specify the `OutputFormat` by calling `setOutputFormat()` of the `JobConf` object that holds the configuration of your MapReduce job.

@Zoran B. Djordjević

58

Main OutputFormat classes

OutputFormat	Description
TextOutputFormat<K, V>	Writes each record as a line of text. Keys and values are written as strings and separated by a tab (\t) character, which can be changed in the mapred.textoutputformat.separator property.
SequenceFileOutputFormat<K, V>	Writes the key/value pairs in Hadoop's proprietary sequence file format. Works in conjunction with SequenceFileInputFormat.
NullOutputFormat<K, V>	Outputs nothing.

@Zoran B. Djordjević

59

References

- 1) MapReduce: Simplified Data Processing on Large Clusters, by Jeffrey Dean and Sanjay Ghemawat, 2004
<http://static.googleusercontent.com/media/research.google.com/en/us/archive/mapreduce-osdi04.pdf>
- 2) Hadoop in Practice, by Alex Holmes, Manning, 2012

@Zoran B. Djordjević

60