Lecture 03

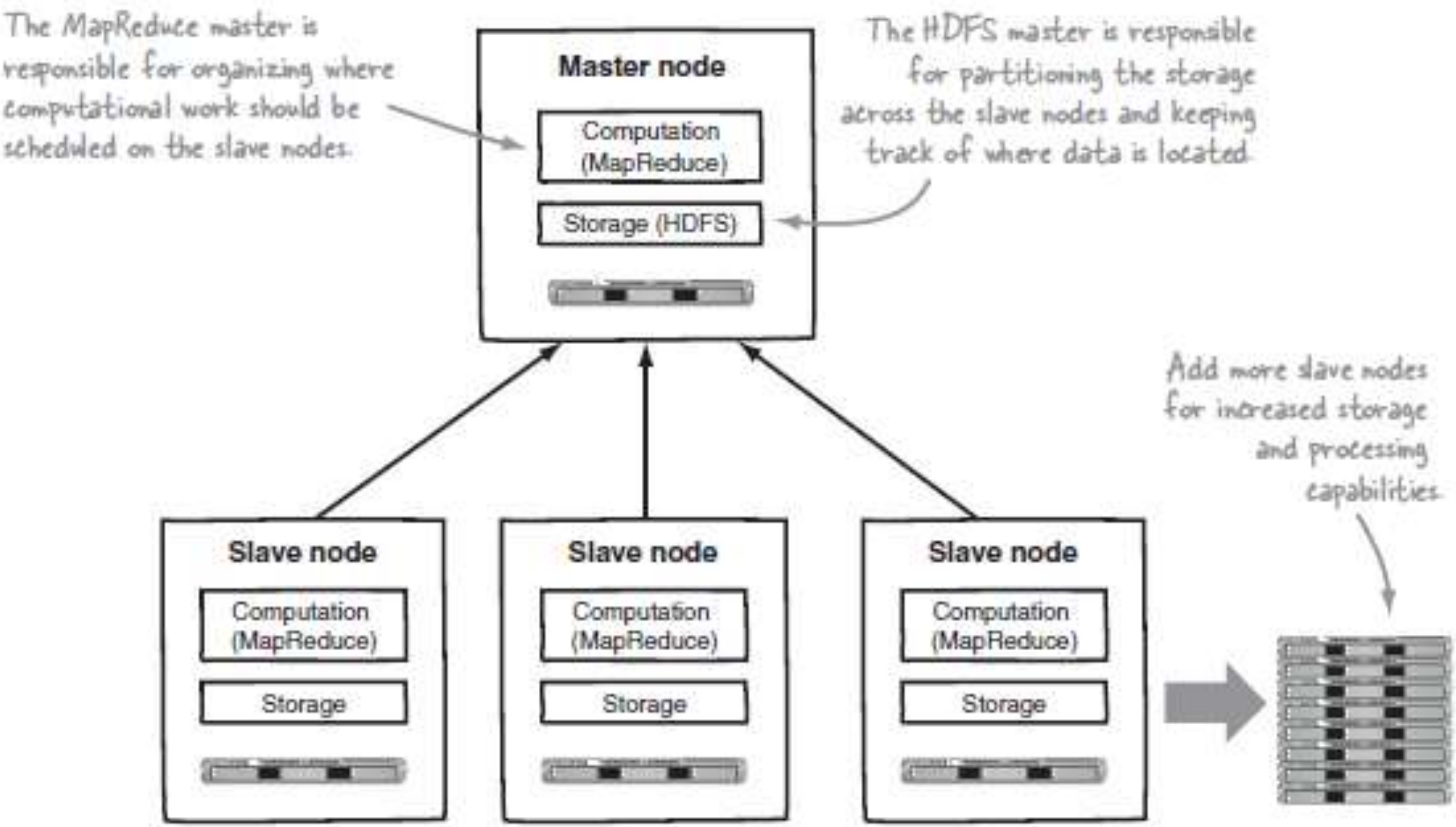
Map Reduce Java API

Zoran Djordjević

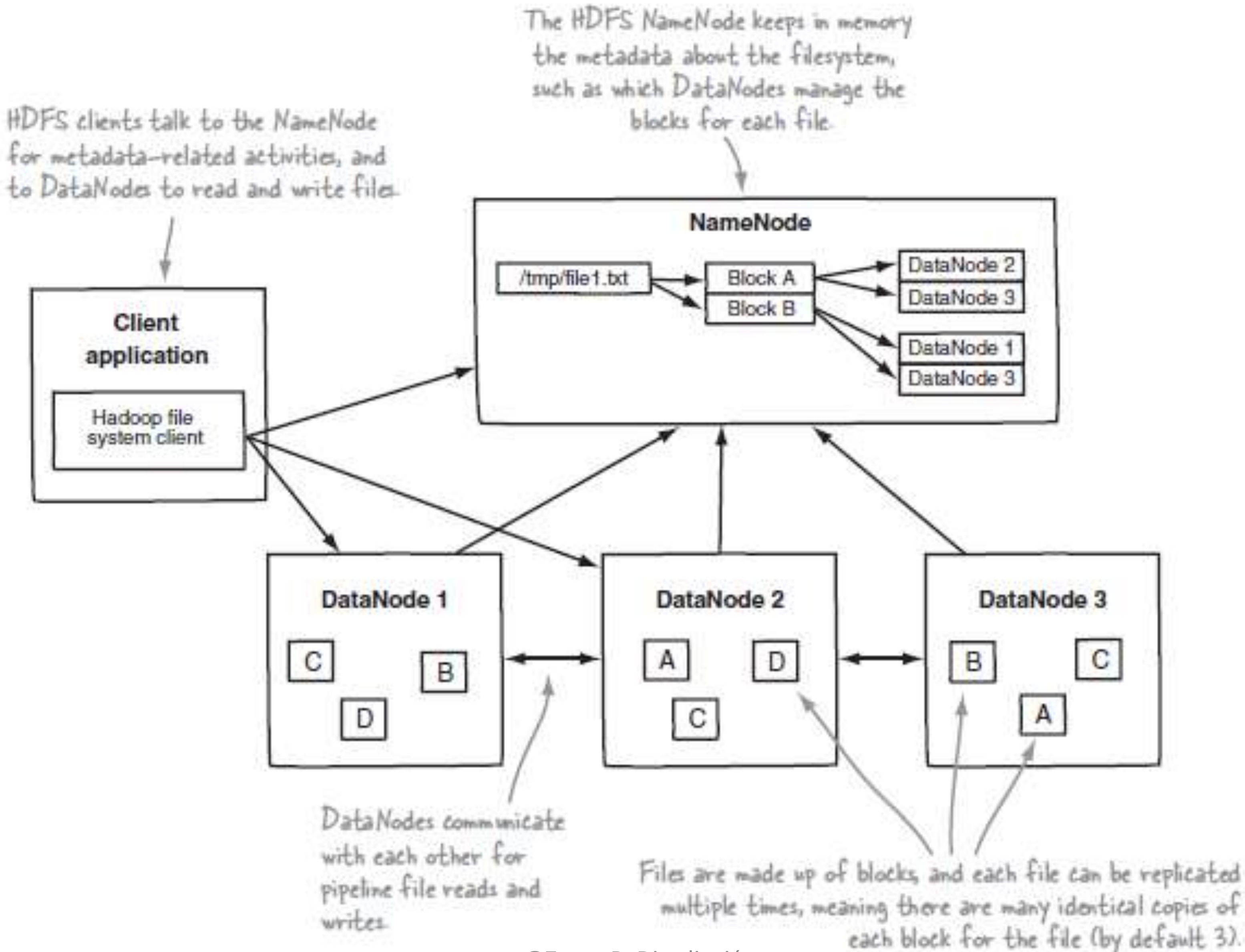
Reference

- This set of slides is based on *Hadoop in Practice* by Alex Holmes, Manning 2012

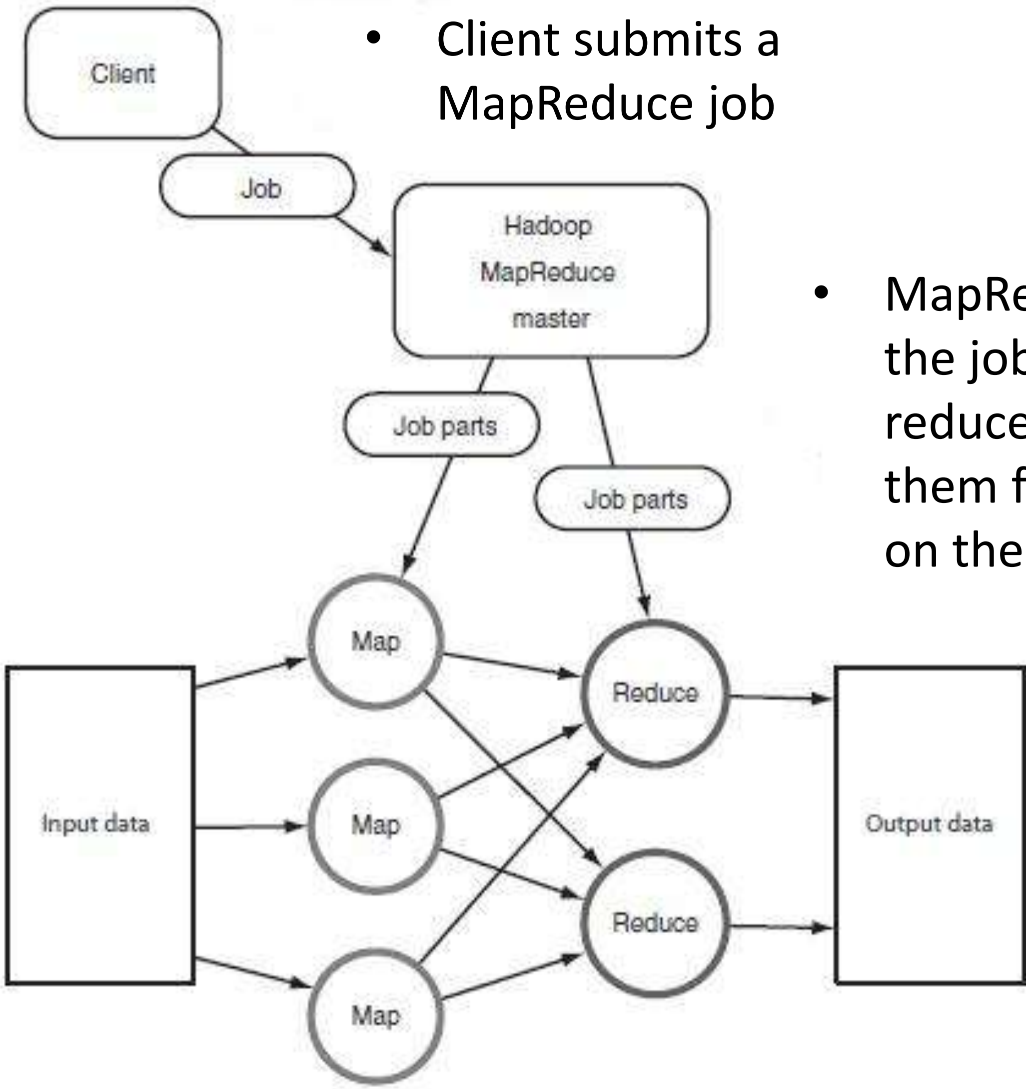
High Level Hadoop Architecture



HDFS Architecture



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.

`map()` Fuction

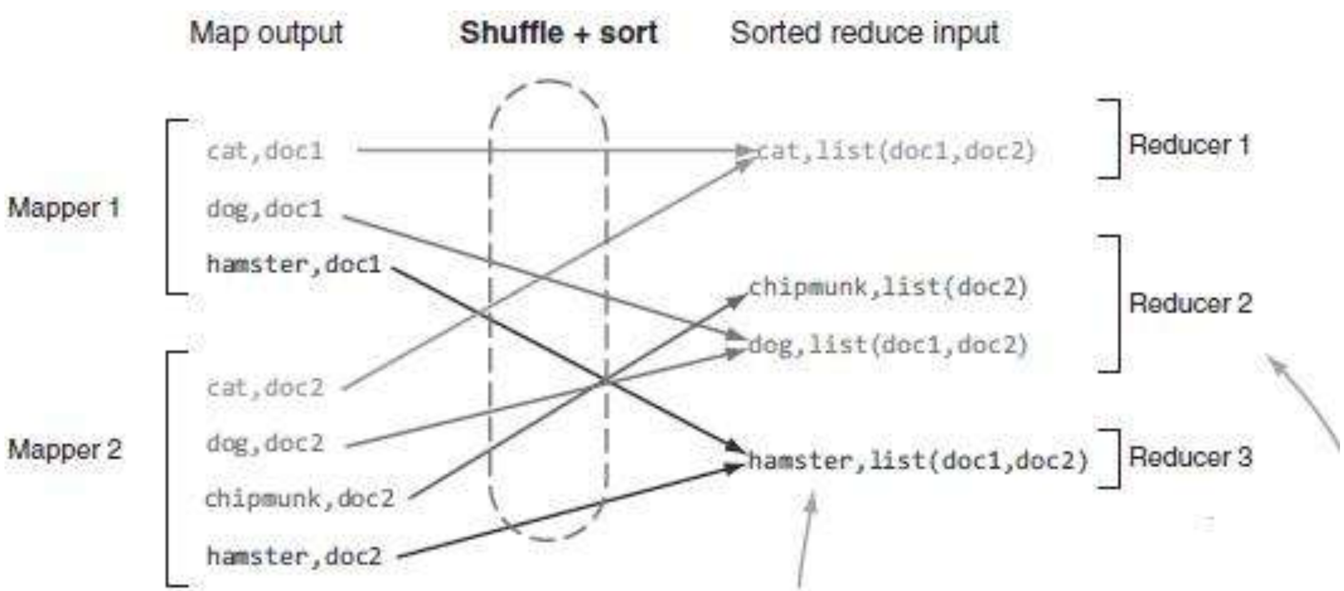
- The **map** function takes as input a key/value pair, which represents a logical record from the input data source.
- In the case of a file, this could be a line, where line number is the key and line itself the value, or if the input source is a database table, the key could be the primary key of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

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.



- Each reducer has all of its input keys 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

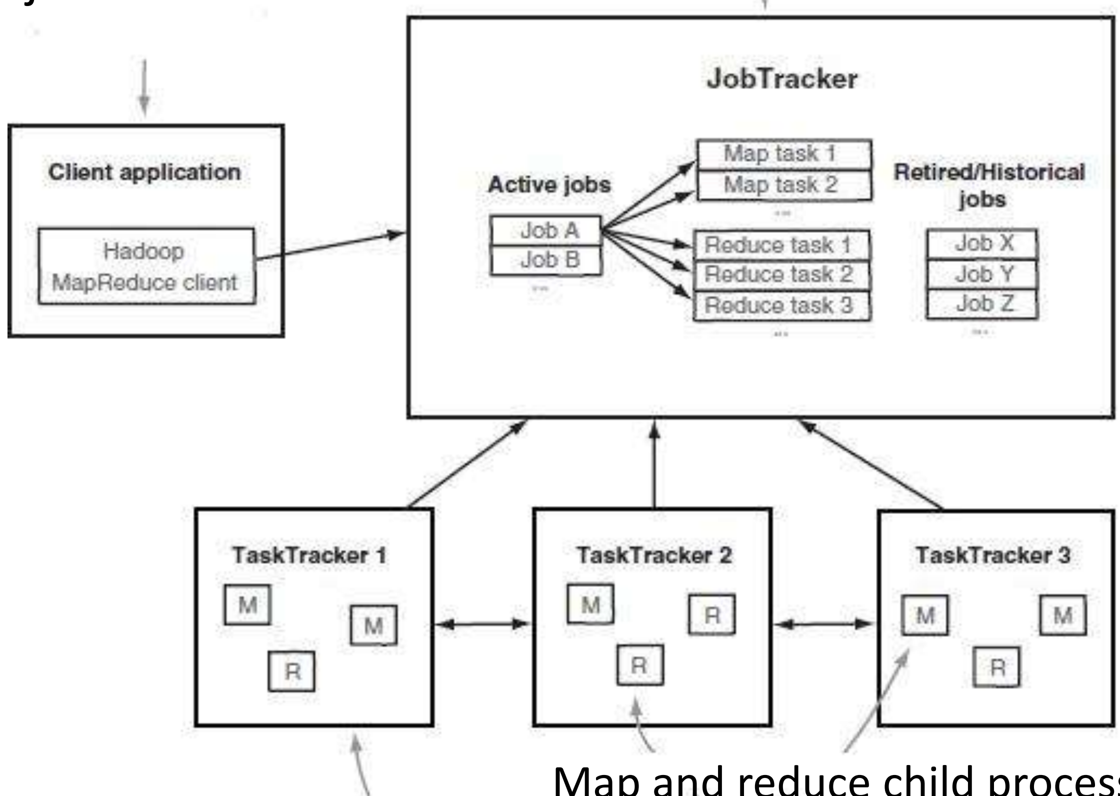
Partition Function

- 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.

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.

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
```

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
```

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]$
```

- Linux user `joe` could fetch the .txt version of James Joyce's Ulysses by issuing the following command on the command prompt:

```
wget http://www.gutenberg.org/files/4300/4300.zip
```

- Once you unzip the file, place it into ulysis directory of user `joe`

```
$ hadoop fs -put 4300.txt ulysis
$ hadoop fs -ls  ulysis
Found 1 items
-rw-r--r--  1 joe supergroup 5258688 2016-02-11 08:31 4300.txt
```

- 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.

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 `4300.txt` from HDFS and copy it into the current local working directory, we run the command

```
hadoop fs -get 4300.txt .
```

- A way to examine the data is to display data. For small files, Hadoop `cat` command is convenient.

```
hadoop fs -cat 4300.txt
```

- 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 4300.txt  | head
```

- Hadoop natively supports `tail` command for looking at the last kilobyte of a file.

```
hadoop fs -tail 4300.txt
```

Deleting files and directories

- Hadoop command for removing files is `rm`.

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

- To delete files and directories recursively us

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

- To delete empty directories use

```
hadoop fs -rmdir  directory
```

New vs. Old Map Reduce API

- Until release 0.183 Hadoop supported a particular Map Reduce API. With release 0.20 a new API is introduced which simplified coding and added some new features.
- It appears that many books and examples on the Internet are written in the old API. In order to help you use those examples we will present some MapReduce classes in both API-s and describe key differences. Differences are not huge and are easy to reconcile.
- In this set of slides we will use new API. In a subsequent lecture we will present older API and underline the differences.

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-2.6.0-cdh5.5.1.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-2.6.0-cdh5.5.1`
- 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-2.6.0-cdh5.5.1.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.
- You can unzip above `tar.gz` file on the Windows side as well with Cygwin, 7zip or WinZIP.

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);    }        }
  }
```

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);
  }
}
```

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);
  }
}
```

Compile and Run

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

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

- and found

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

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

`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 classpath
```

- 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/lib/*:/usr/lib/hadoop-yarn/.//*:/usr/lib/hadoop-
mapreduce/lib/*:/usr/lib/hadoop-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` ``

Compiling `WordCount.java`

- In the directory `examples`, 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.

An alternate way to compile

- Add one more variable in your `.bash_profile`

```
JAVA_HOME=/usr/java/jdk1.8.0_60
export JAVA_HOME
HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
export HADOOP_CLASSPATH
PATH=$PATH:$HOME/bin:${JAVA_HOME}/bin
export PATH
$ source .bash_profile
```

- Now you could compile your class with:

```
$ hadoop com.sun.tools.javac.Main WordCount.java
```

Resolving Warning

- When we compiled WordCount.java we got the warning:

```
WordCount.java:75: warning: [deprecation] Job(Configuration,String)
in Job has been deprecated
    Job job = new Job(conf, "word count");
```

- If we look in Hadoop Java API we will see that constructor Job() is deprecated. Instead we are given a few static methods to set-up the job, like:

```
// Create a new Job Job job = Job.getInstance();
job.setJarByClass(MyJob.class); // Specify job-specific parameters
job.setJobName("myjob");
job.setInputPath(new Path("in"));
job.setOutputPath(new Path("out"));
job.setMapperClass(MyJob.MyMapper.class);
job.setReducerClass(MyJob.MyReducer.class);
// Submit the job, then poll for progress until the job is complete
job.waitForCompletion(true);
```

Jaring `WordCount` MapReduce program

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

```
$ jar -cvf wordcount.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.class(in = 1946)
(out= 1035)(deflated 46%)
adding:
org/apache/hadoop/examples/WordCount$IntSumReducer.class(in =
1793) (out= 750)(deflated 58%)
$ ls
org  wordcount.jar  WordCount.java
```

- Let us fetch Ulysis by James Joice. You have that file on your system most probably

```
$  wget http://www.gutenberg.org/files/4300/4300.zip
$  unzip 4300.zip
```

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`.
- If you have not done that already, create HDFS directory `input` and copy a file, `4300.txt`, to that HDFS directory

```
$ hadoop fs -mkdir ulysis
$ hadoop fs -copyFromLocal 4300.txt ulysis
$ hadoop fs -ls ulysis
Found 1 items
-rw-r--r--   1 cloudera supergroup    1573079 2016-02-11
17:11 ulysis/4300.txt
```

- 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
```

Running `WordCount` MapReduce program

- On the command prompt, on the single line, we type:

$ hadoop jar wordcount.jar org.apache.hadoop.examples.WordCount ulysis output

```
16/02/11 19:21:56 INFO client.RMProxy: Connecting to ResourceManager at
/0.0.0.0:8032
16/02/11 19:22:00 INFO input.FileInputFormat: Total input paths to process : 1
16/02/11 19:22:01 INFO mapreduce.JobSubmitter: number of splits:1
16/02/11 19:22:01 INFO mapreduce.JobSubmitter: Submitting tokens for job:
job_1455225713108_0001
16/02/11 19:22:03 INFO impl.YarnClientImpl: Submitted application
application_1455225713108_0001
16/02/11 19:22:03 INFO mapreduce.Job: The url to track the job:
http://localhost:8088/proxy/application_1455225713108_0001/
16/02/11 19:22:03 INFO mapreduce.Job: Running job: job_1455225713108_0001
16/02/11 19:22:29 INFO mapreduce.Job: Job job_1455225713108_0001 running in uber
mode : false
16/02/11 19:22:29 INFO mapreduce.Job:  map 0% reduce 0%
16/02/11 19:22:42 INFO mapreduce.Job:  map 100% reduce 0%
16/02/11 19:22:52 INFO mapreduce.Job:  map 100% reduce 100%
16/02/11 19:22:53 INFO mapreduce.Job: Job job_1455225713108_0001 completed
successfully
16/02/11 19:22:54 INFO mapreduce.Job: Counters: 49
        File System Counters
                FILE: Number of bytes read=725062
                FILE: Number of bytes written=1673671
                FILE: Number of read operations=0
```

Examine the Results

```
$ hadoop fs -ls output
Found 2 items
-rw-r--r-- 1 cloudera supergroup       0 2016-02-11 19:22 output/_SUCCESS
-rw-r--r-- 1 cloudera supergroup  527547 2016-02-11 19:22 output/part-r-00000
$ hadoop fs -cat output/part-r-00000 | tail -20
zebra   1
zenith  3
zephyrs,        1
zero    1
zest.   1
zigzag  2
zigzagging      1
zigzags,        1
zivio,  1
zmellz  1
zodiac  1
zodiac. 1
zodiacal        2
zoe)_   1
zones:  1
zoo.    1
zoological      1
zouave's        1
zrads,  2
```

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();
```

- Object `Context` allows you to read from the environment and to write to it.
- Method `map()` accepts input key and value. It ignores the key, line number or offset, breaks the value (line of text) into tokens (words) and then writes out a pair (word, one) for each token. Variable `one` has the value of `1`.

```
  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);
    }
  }
```

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);     }
  }
```

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

`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);
```

- Methid `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);
  }
}
```

org.apache.hadoop.util.GenericOptionsParser

- `GenericOptionsParser` is a utility to parse command line arguments generic to the Hadoop framework.
- `GenericOptionsParser` recognizes several standarad 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
```

`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.

You could write MapReduce programs in local Eclipse

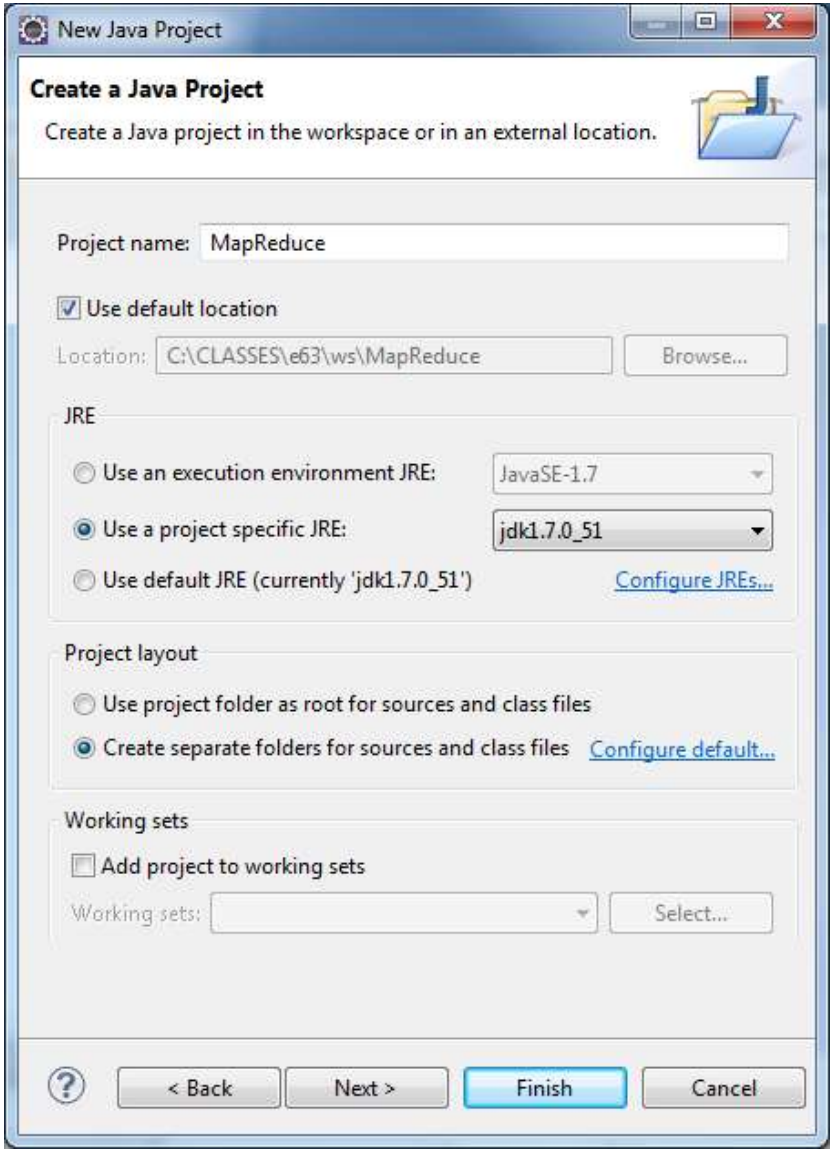
- Go to http://www.eclipse.org/downloads/
- Select your operating system and download.
- For example, I downloaded `eclipse-jeee-juno-SR2-win64.zip`
- You just unzip the file on your C: drive and you are ready to use it.
- If you are using CDH4.6 and have `JDK1.7_51` installed on your VM, please install 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.

Your First Gimps 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 and will do you good.

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.



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.
- Depending where you unzipped CDH5.5.1 tarball file, 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). You are such a bad programmer.
- 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.

Build Path

- Right click on you project `(MapReduce) > Build Path > Configure Build Path> Libraries`.
- In the widget that opens, select `Add External JARs`.
- 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-2.6.0-cdh5.5.1\share\hadoop\mapreduce1`
- there are several jars. I added `hadoop-core-2.6.0-mr1-cdh5.5.1.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-2.6.0-cdh5.5.1\share\hadoop\common\hadoop-common-2.6.0-cdh5.5.1.jar`

- Build was then complaining about missing : `org.apache.commons.cli.Options.` That is an `Apache Commons class`, not a Hadoop class. Such external classes are provided in `../share/hadoop/common/lib` directory.
- Select and add `commons-cli-1.2.jar` to you Build Path. U r done. Almost.

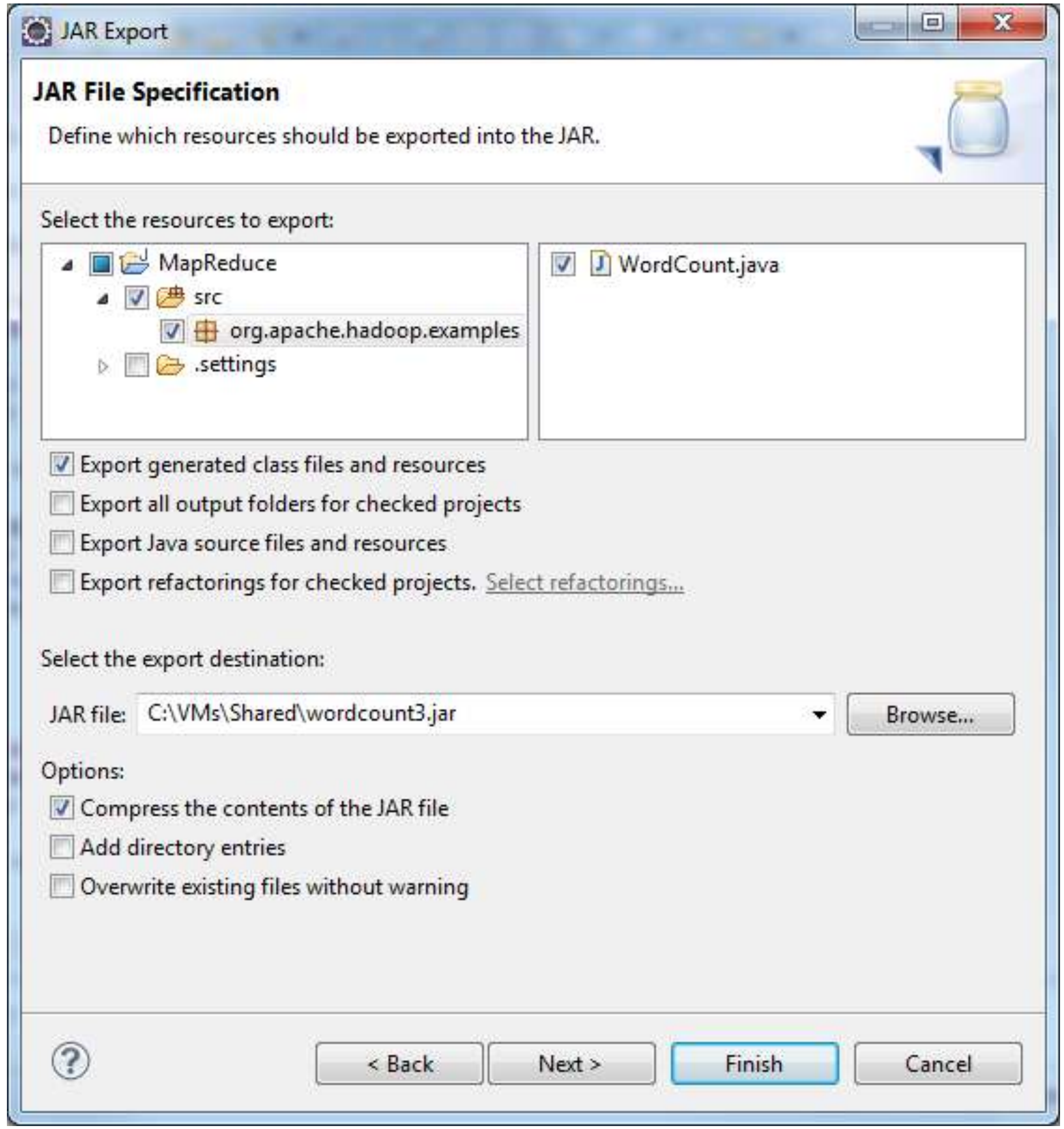
Export your Project

- 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 `wordcount2.jar` to distinguish it from the one I named `wordcount.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 `wordcount2.jar` file.
- You can copy that file to your `sharedfolder` and or transfer it to the Linux box using `scp` command. On the Linux side run it with the help of `hadoop jar` command the same way as you ran `wordcount.jar` which we generated on the Linux side.

Export jar Widget

- It is important to select: Export generated class files and resources

- Select directory destination and file name and
- Select Finish



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.

Frequently used Types

Class	Description
BooleanWritable	Wrapper for a standard Boolean variable
ByteWritable	Wrapper for a single byte
DoubleWritable	Wrapper for a Double
FloatWritable	Wrapper for a Float
IntWritable	Wrapper for a Integer
LongWritable	Wrapper for a Long
Text	Wrapper to store text using the UTF8 format
NullWritable	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.

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.

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.

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

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.

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.

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.

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.

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

Comparison between Map Reduce API-s

Reference

- These slides follow the text of the book:

  Hadoop in Action, by Chuck Lam, Manning 2011

New vs. Old Map Reduce API

- Until release 0.183 Hadoop supported a particular Map Reduce API. With release 0.20 a new API is introduced which simplified coding and added some new features.
- It appears that many books and examples on the Internet are written in the old API. In order to help you use those examples we will present some MapReduce classes in both API-s and describe key differences. Differences are not huge and are easy to reconcile.
- In the end you will end up using templates in either API and only pay attention to details of methods `map()` and `reduce()`.

Fetch Tar Balls

- The most recent tar balls (complete software packages) at Cloudera site could be found at:

```
http://www.cloudera.com/content/cloudera/en/documentation/core/la
test/topics/cdh_vd_cdh_package_tarball.html
```

Source of Data, National Bureau of Economic Research

- http://www.nber.org/data/
- http://data.nber.org/patents/
- Download acite75_99.zip (82MB) and apat63_99.zip (56Mb)

Description	Documentation	Data -- Pkzipped	
		SAS .tpt	ASCII CSV
Overview	overview.txt	--	
Pairwise citations data	Cite75_99.txt	Cite75_99.zip -- (68 Mb)	acite75_99.zip -- (82 Mb)
Patent data, including constructed variables	pat63_99.txt	pat63_99.zip -- (90Mb)	apat63_99.zip -- (56Mb)
Assignee names	coname.txt	coname.zip -- (2Mb)	aconame.zip -- (2Mb)
Contains the match to CUSIP numbers	match.txt	match.zip -- (130Kb)	amatch.zip -- (98Kb)
Individual inventor records	inventor.txt	inventor.zip -- (98Mb)	ainventor.zip -- (82Mb)
Class codes with corresponding class names	classes.txt	--	
Country codes with corresponding country names	countries.txt		
Class, technological category, and technological	class_match.txt		

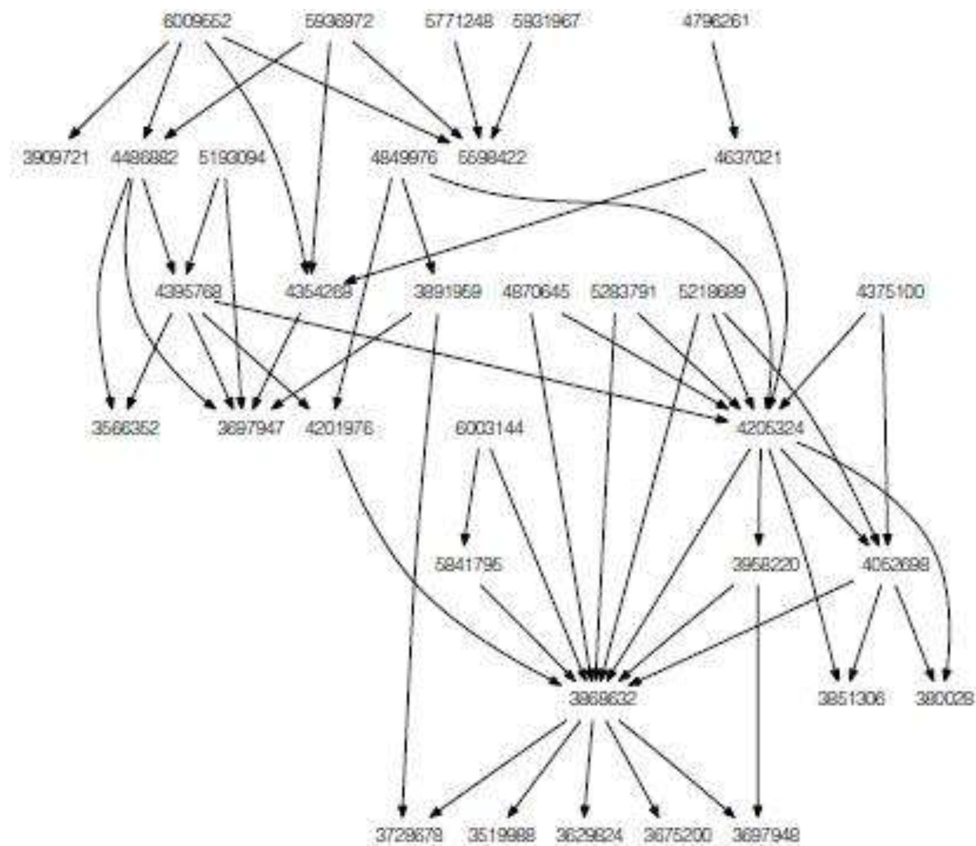
Patent Citation Data, cite75_99.txt File

- Source: US Patent office
- This file includes all US patent citations for utility patents granted in the period 1-Jan-75 to 31-Dec-99.
- No. of observations: 16,522,438
- Variable Name variable type Characters Contents
- CITING numeric 7 Citing Patent Number
- CITED numeric 7 Cited Patent Number
- The file is sorted by Citing Patent Number.

```
“CITING”, ”CITED”
6009552,5278871
6009552,5598422
6009553,4131849
6009553,4517669
6009553,4519068
6009553,4590473
6009553,4636791
6009553,5671255
................
```

Patent Citation Data

- If you're only reading the data file, the citation data appears to be a bunch of numbers.
- One way is to visualize it as a graph. Bellow we show a portion of this citation graph .
- Some patents are cited often whereas others aren't cited at all. Patents like 5936972 and 6009552 cite a similar set of patents (4354269, 4486882, 5598422) , though they don't cite each other.
- We could use Hadoop to derive descriptive statistics about this patent data, and look for interesting, non-obvious, patterns.



List patents and patents that cite them

- We want to invert citation index. Rather than listing `citing` vs. `cited`, we want to invert the data and list `cited` vs. all `citing`. We expect to generate a file that would look like:

```
"CITED"  "CITING"
1000033 4190903,4975983
1000043 4091523
1000044 4082383,4055371
1000045 4290571
1000046 5918892,5525001
1000067 5312208,4944640,5071294
```

- Patent `1000067` is cited by patents `5312208,4944640` and `5071294`

- We will accomplish this objective by writing a MapReduce class `Inverter.java.`
- That class will serve as a template for our future developments.
- In future classes, most of the lines of code will remain the same, while we will change a few lines in the `map()` and `reduce()` methods.

`Inverter.java`, Old API, Template Class

```
package edu.hu.bigdata;
import java.io.IOException;
import java.util.Iterator;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
```

Inverter.java

```
public class Inverter extends Configured implements Tool {
    public static class MapClass extends MapReduceBase
        implements Mapper<Text, Text, Text, Text> {
        public void map(Text key, Text value,
                        OutputCollector<Text, Text> output,
                        Reporter reporter) throws IOException {
            output.collect(value, key);
        }
    }
    public static class Reduce extends MapReduceBase
        implements Reducer<Text, Text, Text, Text> {
        public void reduce(Text key, Iterator<Text> values,
                           OutputCollector<Text, Text> output,
                           Reporter reporter) throws IOException {
            String csv = "";
            while (values.hasNext()) {
                if (csv.length() > 0) csv += ",";
                csv += values.next().toString();
            }
            output.collect(key, new Text(csv));
        }
    }
```

Inverter.java

```
 public int run(String[] args) throws Exception {
        Configuration conf = getConf();
        JobConf job = new JobConf(conf, Inverter.class);
        Path in = new Path(args[0]);
        Path out = new Path(args[1]);
        FileInputFormat.setInputPaths(job, in);
        FileOutputFormat.setOutputPath(job, out);
        job.setJobName("Inverter");
        job.setMapperClass(MapClass.class);
        job.setReducerClass(Reduce.class);
        job.setInputFormat(KeyValueTextInputFormat.class);
        job.setOutputFormat(TextOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        job.set("key.value.separator.in.input.line", ",");
        JobClient.runJob(job);
        return 0;
    }
    public static void main(String[] args) throws Exception {
       int res = ToolRunner.run(new Configuration(),new
Inverter(),args);
       System.exit(res);
    }
}
```

Result of Running `Inverter` job

- We compiled `Inverter.java` in Eclipse with Build Path containing:
  - `hadoop-core-2.0.0-mr1-cdh4.6.0.jar`,
  - `hadoop-common-2.0.0-chd4.6.0.jar` and `commons-cli-1.2.jar`
- Exported `inverter.jar`, moved `cite75-99.txt` to HDFS `input3` directory and ran command:

```
$ hadoop jar inverter.jar edu.hu.bigdata.Inverter input outputinv
```

- Command `hadoop fs -tail outputinv/part-00000` gave the following:

```
......
999936  5014973,5642878
999940  4022472,3876070
999941  5447466
999945  5569166,5207231
999949  5640640
999951  5374468,5316622
999957  5755359
999961  5738381,5782495,5878901,4871140,4832301,5048788,4171117,4262874
999965  5052613
999968  3916735
999971  3965843
999972  4038129
999973  5427610,4900344
999974  4728158,4560073,5464105
. . . . .
```

Programming Convention

- Standard convention is that a single class, `Inverter` in this case, completely defines each MapReduce job.
- Hadoop requires the `Mapper` and the `Reducer` to be their own static classes. These classes are quite small, and our template includes them as inner classes in the `Inverter` class.
- The advantage is that everything fits in one file, simplifying code management.
- These inner classes are independent and don't interact much with the `Inverter` class.
- Various nodes with different JVMs clone and run the Mapper and the Reducer during job execution , whereas the rest of the job class is executed *only* at the client machine.

“Client” portion of `Inverter` class, old API

- The core of `Inverter` class is within the `run()` method.
- `run()` method is also known as the *driver*.
- The driver instantiates, configures, and passes a `JobConf` object named `job` to `JobClient. runJob()` to start the MapReduce job.
- The `JobClient` class, in turn, will communicate with the `JobTracker` to start the job across the cluster.
- The `JobConf` object holds all configuration parameters necessary for the job to run.
- `JobConf` is fed configuration parameters from the `Configuration` object which is retrieved by the `getConf()` method of `Configurable` interface our main class implements.

“Client” portion of Inverter

- The driver needs to specify the basic parameters for every `job`
  - `input  Path`,
  - `output Path`,
  - `MapperClass`, and
  - `ReducerClass`.
- In addition, each `job` can reset the default properties, such as
  - `InputFormat`,
  - `OutputFormat`
- One can also call the `set()` method on the `JobConf` object to set up any configuration parameter.
- Once you pass the `JobConf` object to `JobClient.runJob()`, it’s treated as the master plan for the job. It becomes the blueprint for how the `job` will be run.

“Client” portion of `Inverter` class, new API

- The core of `Inverter` class could remain within the `run()` method.
- The driver instantiates a `Configuration` object which gets its parameters from Hadoop configuration files using `getConf()` call
- The Configuration object is passed to the static method `getInstance()` of class `Job`.
- Various parameters, properly formatted are passed to object Job.
- Call to method `job.waitForCompletion()` sets the run on.

Example of command line option use

- For example, we would normally execute the `Inverter` class using a command line like:

```
$ hadoop jar inverter.jar edu.hu.bd.Inverter input output
```

- Had we wanted to run the job only to see the mapper's output (which you may want to do for debugging purposes), we could set the number of reducers to zero with the option `-D mapred.reduce.tasks=0`.

```
$ hadoop jar inverter.jar edu.hu.bd.Inverter
      -D mapred.reduce.tasks=0   input output
```

- This works even though our program doesn't explicitly understand the `-D` option.
- By using `ToolRunner`, `Inverter` will automatically support the options understood by `GenericOptionsParser`

Mapper and Reducer

- The convention for our template is to call the Mapper class `MapClass` and the Reducer class `Reduce`.
- The naming would seem more symmetric had we called the `Mapper` class `Map`. However, Java already has a class (interface) named `Map`.
- Both the Mapper and the Reducer extend `MapReduceBase`, inold API which is a small class providing no-op implementations to the `configure()` and `close()` methods required by the two interfaces.
- The `configure()` and `close()` methods are life cycle methods and one could use them to set up and clean up the map (reduce) tasks.
- Usually you do not need to override `configure()` and `close()` except for more advanced jobs.

Signature of Mapper and Reducer, old API

- The signatures for the old Mapper and the Reducer classes are:

```
public static class MapClass extends MapReduceBase
      implements Mapper<K1, V1, K2, V2> {
      public void map(K1 key, V1 value,
                          OutputCollector<K2, V2> output,
                          Reporter reporter)
              throws IOException { }
}
public static class Reduce extends MapReduceBase
      implements Reducer<K2, V2, K3, V3> {
      public void reduce(K2 key, Iterator<V2> values,
                            OutputCollector<K3, V3> output,
                            Reporter reporter)
              throws IOException { }
}
```

Signature of Mapper and Reducer, new API

- The signatures for the new Mapper and the Reducer classes are:

```
public static class TokenizerMapper
      extends Mapper<Object, Text, Text, IntWritable>{

   public void map(Object key, Text value, Context context
                  ) throws IOException, InterruptedException {
        . . . .
        context.write(word, one);
     }
   }
 }

 public static class IntSumReducer
      extends Reducer<Text,IntWritable,Text,IntWritable> {

   public void reduce(Text key, Iterable<IntWritable> values,
                      Context context
                      ) throws IOException, InterruptedException {
      . . . . .
      context.write(key, result);
   }
 }
```

Type matching

- In addition to having consistent `K2` and `V2` types across Mapper and Reducer, you'll also need to ensure that the key and value types used in Mapper and Reducer are consistent with the input format, output key class, and output value class set in the driver.
- The use of `KeyValueTextInputFormat` in the `Inverter` means that `K1` and `V1` must both be type `Text`.
- The driver must call `setOutputKeyClass()` and `setOutputValueClass()` with the classes of `K2` and `V2`, respectively.
- All the key and value types must be subtypes of `Writable`, which means they are `Serializable` and serialization interface of Hadoop could send them around the distributed cluster.
- In fact, the key types implement `WritableComparable`, a subinterface of `Writable`. The key types need to additionally support the `compareTo()` method, as keys need to be sorted in various places in the MapReduce framework.

Counting Citations (Citings)

- Much of what we think of as statistics is counting, and many basic Hadoop jobs involve counting.
- For the patent citation data, we may want the number of citations a patent has received. The desired output would look like this:

```
1000006 1
1000007 3
1000011 7
1000017 1
```

- In each record, a patent number is associated with the number of citations it has received. We will write a MapReduce program for this task.
- We hardly ever write a MapReduce program from scratch. You have an existing MapReduce program `Inverter` that processes the data in a similar way. We copy and rename class `Inverter` into `InverterCounter` and modify it until it does what we want .

`InverterCounter`, Modify Reducer

- We can modify Inverter to output the count instead of the list of citing patents. We need the modifications only at the `Reducer`.
- If we choose to output the count as an `IntWritable`, we need to specify `IntWritable` in three places in the `Reducer` code. We called them V3 in our previous notation.

```
public static class Reduce extends MapReduceBase implements
       Reducer<Text, Text, Text, IntWritable> {
       public void reduce(Text key, Iterator<Text> values,
       OutputCollector<Text, IntWritable> output,
       Reporter reporter) throws IOException {
       int count = 0;
              while (values.hasNext()) {
               values.next(); count++;
       }
      output.collect(key, new IntWritable(count)); } }
```

- By changing a few lines and matching class types, we created a new MapReduce program.

Running `InverterCounter`

- We compile `InverterCounter` very much like class `Inverter`.
- We run it using the same input file and send output to a new directory:

```
$ hadoop jar inverter.jar edu.hu.bigdata.InverterCounter
input outputcount
```

`$ hadoop fs -tail outputcount/part-00000` gives

```
...
999945 2
999949 1
999951 2
999957 1
999961 9
999965 1
999968 1
999971 1
999972 1
999973 2
999974 3
...
```

Build Histogram of Citations

- Now that we know that many patents are cited once, twice and so forth, we would like to know the exact number of patents cited once, cited twice, and so forth.
- This type of result we call the Histogram of the Citation Counts.
- We expect a large number of patents to have been cited once, and a small number may have been cited hundreds of times.
- Initially we will use the result produced by `InverterCounter` as the input into a new MapReduce program.

Data Flow

- The first step to writing a new MapReduce program is to figure out the data flow. We start with the data produced by `InverterCounter`:

```
1000006 1
1000007 3
1000011 7
1000017 1
```

- We want to know how may times a number (`citation_count`) , for example 3, appears in the entire file and how many times number 7 appears, and so on.
- Our new mapper will read a record and ignore the patent number or rather replace it with number 1.
- The Mapper will output an intermediate key/value pair of `<citation_count, 1>` .
- The Reducer will sum up the number of 1s for each `citation_count` and output the total.

Data Types

- After figuring out the data flow, we need to decide on the types for the key/value pairs—`K1, V1, K2, V2, K3,` and `V3` for the input, intermediate, and output key/value pairs.
- We will keep using the `KeyValueTextInputFormat`, which automatically breaks each input record into key/value pairs based on a separator character.
- The input format produces `K1` and `V1` as `Text` . We choose to use `IntWritable` for `K2, V2, K3`, and `V3` because we know those data must be integers and it's more efficient to use `IntWritable` than `Text.`
- We will call our new program `CitationHistogram`. Its complete listing is given on the following slides.
- We do not plan to generate a graphical representation of that histogram using Map Reduce techniques. ☺

CitationHistogram.java, old API

```
package edu.hu.bgd;
import java.io.IOException;
import java.util.Iterator;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
```

CitationHistogram.java, old API

```
public class CitationHistogram extends Configured implements Tool {
    public static class MapClass extends MapReduceBase
        implements Mapper<Text, Text, IntWritable, IntWritable> {
         private final static IntWritable uno = new IntWritable(1);
         private IntWritable citationCount = new IntWritable();
        public void map(Text key, Text value,
                        OutputCollector<IntWritable, IntWritable> output,
                        Reporter reporter) throws IOException {

            citationCount.set(Integer.parseInt(value.toString()));
            output.collect(citationCount, uno);
        }
    }
    public static class Reduce extends MapReduceBase
        implements Reducer<IntWritable,IntWritable,IntWritable,IntWritable>    {
        public void reduce(IntWritable key, Iterator<IntWritable> values,
                           OutputCollector<IntWritable, IntWritable>output,
                           Reporter reporter) throws IOException {
            int count = 0;
            while (values.hasNext()) {
                count += values.next().get();
            }
            output.collect(key, new IntWritable(count));
        }
    }
```

CitationHistogram.java, old API

```
    public int run(String[] args) throws Exception {
        Configuration conf = getConf();
        JobConf job = new JobConf(conf, CitationHistogram.class);
        Path in = new Path(args[0]);
        Path out = new Path(args[1]);
        FileInputFormat.setInputPaths(job, in);
        FileOutputFormat.setOutputPath(job, out);
        job.setJobName("CitationHistogram");
        job.setMapperClass(MapClass.class);
        job.setReducerClass(Reduce.class);
        job.setInputFormat(KeyValueTextInputFormat.class);
        job.setOutputFormat(TextOutputFormat.class);
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(IntWritable.class);
        JobClient.runJob(job);
        return 0;
    }
    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(),
                                 new CitationHistogram(), args);
        System.exit(res);
    }
}
```

`CitationHistorgamNew.java`, new API

```
package edu.hu.bigdata;
import java.io.IOException;
//import java.util.Iterator;
import java.lang.InterruptedException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
//import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
//import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
//import org.apache.hadoop.mapred.JobClient;
//import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapreduce.Job;
//import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapreduce.lib.input.KeyValueTextInputFormat;
import org.apache.hadoop.mapreduce.Mapper;
//import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapreduce.Reducer;
```

`CitationHistorgamNew.java`, new API

```
//import org.apache.hadoop.mapred.Reporter;
//import org.apache.hadoop.mapred.TextOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class CitationHistogramNew extends Configured    implements Tool {
  public static class MapClass extends  Mapper<Text, Text,                    IntWritable, IntWritable> {
    private final static IntWritable uno = new IntWritable(1);
    private IntWritable citationCount = new IntWritable();
    public void map(Text key, Text value, Context context)
      throws IOException, InterruptedException {
        citationCount.set(Integer.parseInt(value.toString()));
        context.write(citationCount, uno);
      }
  }
```

CitationHistorgamNew.java, new API

```
public static class Reduce extends Reducer  <IntWritable,IntWritable,IntWritable,IntWritable>  {
//public void reduce(IntWritable key, Iterator<IntWritable>values,
                  public void reduce(IntWritable key, Iterable<IntWritable>  values, Context context)
          throws IOException, InterruptedException {
        int count = 0;
        // while (values.hasNext()) {
        //     count += values.next().get();
        for (IntWritable val:values){   // Iterable allows
                  count += val.get();       // for looping
        }
        context.write(key, new IntWritable(count));
      }
  }
   public int run(String[] args) throws Exception {
      Configuration conf = getConf();
      //JobConf job = new JobConf(conf, CitationHistogram);
      //job.setJobName("CitationHistogram");
      Job job = new Job(conf, "CitationHistogramNew");
      job.setJarByClass(CitationHistogramNew.class);
      Path in = new Path(args[0]);
      Path out = new Path(args[1]);
```

CitationHistorgamNew.java, new API

```
FileInputFormat.setInputPaths(job, in);
    FileOutputFormat.setOutputPath(job, out);
    job.setJobName("CitationHistogramNew");
    job.setMapperClass(MapClass.class);
    job.setReducerClass(Reduce.class);
   // job.setInputFormat(KeyValueTextInputFormat.class);
    job.setInputFormatClass(KeyValueTextInputFormat.class);
    //job.setOutputFormat(TextOutputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    job.setOutputKeyClass(IntWritable.class);
    job.setOutputValueClass(IntWritable.class);
    //JobClient.runJob(job);
    System.exit(job.waitForCompletion(true)?0:1);
    return 0;
  }
  public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new Configuration(),
           new CitationHistogramNew(), args);
    System.exit(res);
  }
}
```

`CitationHistogram`

- The class name is now `CitationHistogram`; all references to `InverterCounter` were changed to reflect the new name.
- The `main()` method is almost always the same. The driver is mostly intact.
- The input format and output format are still `KeyValueTextInputFormat` and `TextOutputFormat`, respectively.
- The main change is that the output key class and the output value class are now `IntWritable`, to reflect the new type for `K2` and `V2`.
- We've also removed this line:
  `job.set("key.value.separator.in.input.line", ",");`
- It was setting the separator character used by `KeyValueTextInputFormat` on which it was break each input line into a key/value pair.
- In classes `Inverter` and `InverterCounter` we needed a comma for processing the original citation data. If not set, this property defaults to the tab character, which is appropriate for the citation count data.

Mapper in `CitationHistogram`

- The data flow for the mapper is similar to that of the previous mappers, only here we've chosen to define and use two new class variables: `citationCount` and `uno`.
- The `map()` method has one extra line for setting `citationCount`, which is for type casting.
- The reason for defining `citationCount` and `uno` in the class rather than inside the method is purely one of efficiency. The `map()` method will be called as many times as there are records processed at every machine.
- Reducing the number of objects created inside the `map()` method can increase performance and reduce garbage collection .
- We pass `citationCount` and `uno` to `output.collect()`

Reducer in `CitationHistogram`

- The reducer sums up the values for each key. It seems inefficient because we know all values are 1-s (`uno`, to be exact).
- Unlike in `MapClass`, the call to `output.collect()` in `Reduce` instantiates a new `IntWritable` rather than reuse an existing one.

```
output.collect(key, new IntWritable(count));
```

- This is bad programming. We could improve the performance of our program by using an `IntWritable` class variable.
- The number of times `reduce()` is called in this particular program is small, probably no more than a few thousand times. So we don't have much need to optimize this particular code, but should not be so cavalier in the future.
- We compile `CitationHistogram` using the same Build Path as previously, and export the class to `histogram.jar`

Running `CitationHistogram`

- Before we run `CitationHistogram` let us clean the directory `outputcount` where we deposited the result of `InverterCounter` job.

```
$ hadoop fs -rm -R outputcount/_*
```

- The above leaves only file `part-00000` in the directory. Next we type:

```
$  hadoop jar histogram.jar
edu.hu.bigdata.CitationHistogram outputcount outputhis
```

- As input we use `outputcount` which contains the result of `InverterCounter` job . We send results to HDFS directory `outputhis`.

- Command

```
$ hadoop fs -tail
      outputhis/part-00000
```

gives:

```
. . .
631     1
633     1
654     1
658     1
678     1
716     1
779     1
```

The most cited patent is cited 779 times

- Command

```
$ hadoop fs -cat
 outputhis/part-00000 | head -5
```

gives:

```
1       921128
2       552246
3       380319
4       278438
5       210814
```

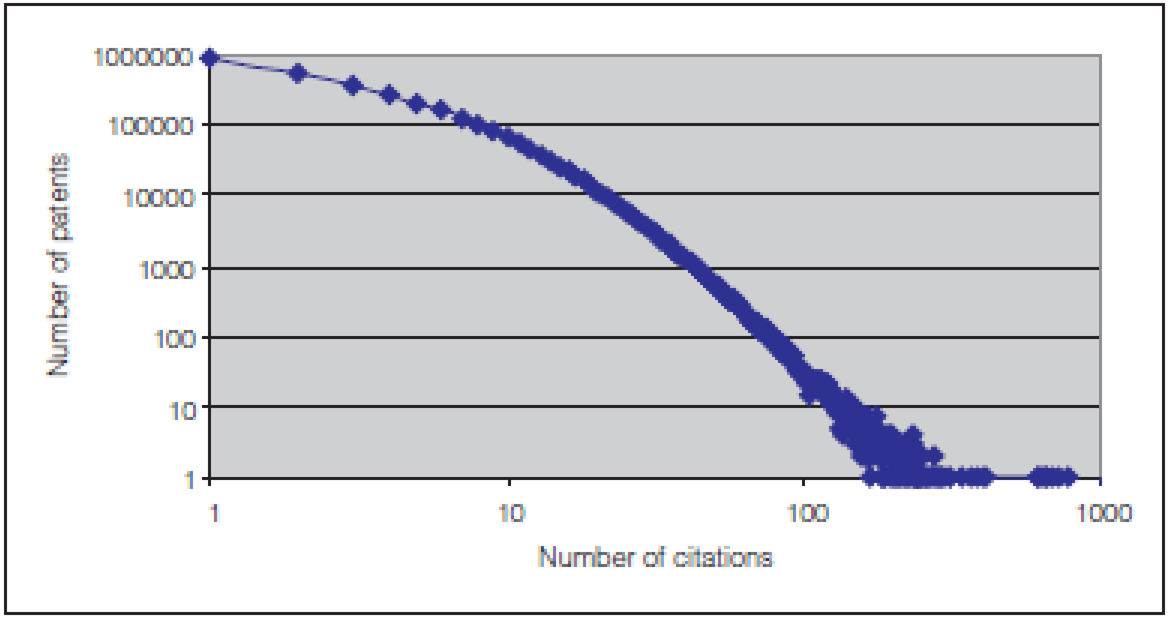
`900K` patents were cited only once

Histogram

- There are apparently only 258 lines in `part-00000`

```
$ hadoop fs -get outputhis/part-00000 .
$ cat part-00000 | wc -l
258
```

- While it might be less than convenient to generate a graph of these data using Hadoop, Excel could do it for us.
- Data are somewhat easier to understand if presented on log-log graph



Chaining MapReduce jobs

- We can execute the two MapReduce jobs manually one after the other, it would be more convenient to automate the execution sequence.
- We could chain two or more MapReduce jobs to run sequentially, with the output of one MapReduce job being the input to the next.
- Chaining MapReduce jobs is analogous to Unix pipes .

```
mapreduce-1 | mapreduce-2 | mapreduce-3 | ...
```

- Chaining is actually quite trivial. We could rename Mapper inner classes in `InverterCounter` and `CitationHistogram` classes to `MapClass1` and `MapClass2` respectively.
- Similarly we could rename to Reduce classes to `Reduce1` and `Reduce2`.
- All four inner classes could now reside inside a single class that we could call `ChainedHistogram`.
- Inside new class we will configure two jobs: `job1` and `job2`.
- `job1` will receive its input from HDFS directory `input`, and write its output to new HDFS directory called `temp`. `job2` will take its input from the `temp` directory and write its output to HDFS directory `output`.
- We will need two `createJob` methods: `createJob1` and `createJob2`. The content of those methods is identical to the content in classes `InverterCounter` and `CitationHistogram`, respectively.
- Once both jobs are done, we will delete the `temp` directory.

ChainedHistogram,createJob1,cleanup, main

```
private JobConf createJob1(Configuration conf, Path in, Path out) {
  JobConf job = new JobConf(conf, ChainedHistogram.class);
  job.setJobName("job1");
  FileInputFormat.setInputPaths(job, in);
  FileOutputFormat.setOutputPath(job, out);
  job.setMapperClass(MapClass1.class); job.setReducerClass(Reduce1.class);
  job.setInputFormat(KeyValueTextInputFormat.class);
  job.setOutputFormat(TextOutputFormat.class);
  job.set("key.value.separator.in.input.line", ",");
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);  return job;
}
private void cleanup(Path temp, Configuration conf)
   throws IOException {
        FileSystem fs = temp.getFileSystem(conf);
        fs.delete(temp, true);
}
public static void main(String[] args) throws Exception {
 int res = ToolRunner.run(new Configuration(), new ChainedHistogram(), args);
  System.exit(res);
}
```

Class `ChainedHistogram, createJob2, run`

```
private JobConf createJob2(Configuration conf, Path in, Path out) {
  JobConf job = new JobConf(conf, ChainedHistogram.class);
  job.setJobName("job2");
  FileInputFormat.setInputPaths(job, in);
  FileOutputFormat.setOutputPath(job, out);
  job.setMapperClass(MapClass2.class); job.setReducerClass(Reduce2.class);
  job.setInputFormat(KeyValueTextInputFormat.class);
  job.setOutputFormat(TextOutputFormat.class);
  job.setOutputKeyClass(IntWritable.class);
  job.setOutputValueClass(IntWritable.class); return job;
}
public int run(String[] args) throws Exception {
  Configuration conf = getConf();
  Path in = new Path(args[0]);
  Path out = new Path(args[1]);
  Path temp = new Path("chain-temp");
  JobConf job1 = createJob1(conf, in, temp);
  JobClient.runJob(job1);
  JobConf job2 = createJob2(conf, temp, out);
  JobClient.runJob(job2);
  cleanup(temp, conf);    return 0;
}
```

Migrating Code to New API

- One of the main design goals driving toward Hadoop’s major 1.0 release was a stable and extensible MapReduce API.
- At the time version 0.20 was planned to be the latest release with old API and is considered a bridge between the older API (that we used in last three example) and the upcoming stable API.
- The 0.20 release supports the future API while maintaining backward-compatibility with the old API while marking it as deprecated.
- Future releases after 0.20 are supposed to stop supporting the older API.
- Almost all the changes affect only the basic MapReduce template. We will rewrite our last class `CitationHistogram` under the new API to demonstrate the changes you need to implement.

Differences between API-s

- The most noticeable change in the new API is that many classes in `org.apache.hadoop.mapred` package have been moved elsewhere.
- Many of `org.apache.hadoop.mapred` classes are now in `org.apache.hadoop.mapreduce` package.
- Some classes are now under one of the packages in `org.apache.hadoop.mapreduce.lib`.
- After you move your code to the new API, you should not have any import statements (or full references) to any classes under `org.apache.hadoop.mapred.` All of `mapred` classes are to be deprecated.

Introduction of `Context` objects

- Another noticeable change in the new API is the introduction of *`Context`* objects.
- The most immediate impact is the replacement of `OutputCollector` and `Reporter` objects used in the `map()` and `reduce()` methods.
- In new API we output key/value pairs by calling `Context.write()` instead of `OutputCollector.collect()`.
- The long-term consequences are to unify communication between your code and the MapReduce framework, and to stabilize the Mapper and Reducer API such that the basic method signatures will not change when new functionalities are added.
- New functionalities will only add new methods on the `Context` objects. Programs written before the introduction of new functionality will be unaware of new methods. Older programs will continue to compile and run against the newer releases.

`org.apache.hadoop.mapreduce.Mapper.Context`

- Both

`org.apache.hadoop.mapreduce.Mapper` and
`org.apache.hadoop.mapreduce.Reducer`

classes have an object of type `Context, i.e.`

`org.apache.hadoop.mapreduce.Mapper.Context` and
`org.apache.hadoop.mapreduce.Reducer.Context`

For some reason those classes are poorly documented .

- `http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/class-use/Mapper.Context.html`
- `http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/class-use/Reducer.Context.html`

New abstract classes `Mapper` and `Reducer`

- The new `map()` and `reduce()` methods are contained in new *abstract classes* `Mapper` and `Reducer`, respectively.
- New classes replace the `Mapper` and `Reducer` *interfaces* (`org.apache.hadoop.mapred.Mapper` and`org.apache.hadoop.mapred.Reducer`) in the original API.
- The new abstract classes also replace the `MapReduceBase` class, which has been deprecated.
- The new `map()` and `reduce()` methods have a couple more changes.
  - They can throw `InterruptedException` instead of only `IOException`.
  - In addition, the `reduce()` method no longer accepts the list of values as an `Iterator` but as an `Iterable`, which is easier to iterate through using Java's `foreach` syntax.

Signatures of old and new `map()` and `reduce()`

```
public static class MapClass                    //OLD
    extends MapReduceBase implements Mapper<K1, V1, K2, V2> {
    public void map(K1 key, V1 value,
              OutputCollector<K2, V2> output, Reporter reporter)
    throws IOException { } }
public static class Reduce                     // OLD
    extends MapReduceBase implements Reducer<K2, V2, K3, V3> {
    public void reduce(K2 key, Iterator<V2> values,
        OutputCollector<K3, V3> output, Reporter reporter)
        throws IOException { } }
```

- The new API simplifies `map()` and `reduce()` somewhat:

```
public static class MapClass extends Mapper<K1, V1, K2, V2> {
     public void map(K1 key, V1 value, Context context)
     throws IOException, InterruptedException { }
}
public static class Reduce extends Reducer<K2, V2, K3, V3> {
  public void reduce(K2 key, Iterable<V2> values, Context context)
        throws IOException, InterruptedException { }
}
```

Driver Changes

- We also need to change the driver to support the new API.
- `JobConf` and `JobClient` classes have been replaced. Their functionalities have been pushed to the `Configuration` class (which was originally the parent class of `JobConf`) and a new class `Job`.
- The `Configuration` class purely configures a job, whereas the `Job` class defines and controls the execution of a job.
- Methods such as `setOutputKeyClass()` and `setOutputValueClass()` have moved from `JobConf` to `Job`. A job's construction and submission for execution are now under `Job`.
- Originally you would construct a job using `JobConf`:

```
JobConf job = new JobConf(conf, MyJob.class); job.setJobName("MyJob");
```

- In newer API we do it through `Job` object:

```
Job job = Jobb.getInstance(conf, “MyJob”);
```

- Previously `JobClient` submitted a job for execution:

```
JobClient.runJob(job);
```

- Now it's also done through `Job` :

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

CitationHistogramNewApi.java, **new** vs. **old**

```
package edu.hu.bgd;  import java.io.IOException; //import java.util.Iterator;
import java.lang.InterruptedException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
//import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
//import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
//import org.apache.hadoop.mapred.JobClient; //import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapreduce.Job;
//import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapreduce.lib.input.KeyValueTextInputFormat;
import org.apache.hadoop.mapreduce.Mapper; //import org.apache.hadoop.mapred.Mapper;
//import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapreduce.Reducer; // import org.apache.hadoop.mapred.Reducer;
//import org.apache.hadoop.mapred.Reporter; //import org.apache.hadoop.mapred.MapReduceBase
//import org.apache.hadoop.mapred.TextOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
```

CitationHistogramNewApi.java, **new** vs. **old**

```
public class CitationHistogramNewApi extends Configured
implements Tool {
    public static class MapClass extends
        Mapper<Text, Text, IntWritable, IntWritable> {
      private final static IntWritable uno = new IntWritable(1);
      private IntWritable citationCount = new IntWritable();
      public void map(Text key, Text value, Context context)
        throws IOException, InterruptedException {
          citationCount.set(Integer.parseInt(value.toString()));
          context.write(citationCount, uno);
        }
    }
```

CitationHistogramNewApi.java, **new** vs. **old**

```
public static class Reduce extends Reducer
       <IntWritable,IntWritable,IntWritable,IntWritable>  {
//public void reduce(IntWritable key,Iterator<IntWritable>values,
//                 OutputCollector collector, Reporter reporter)
        public void reduce(IntWritable key, Iterable<IntWritable>
                 values, Context context)
             throws IOException, InterruptedException {
          int count = 0;
          // while (values.hasNext()) {
          //     count += values.next().get();
          for (IntWritable val:values){    // Iterable allows
               count += val.get();         // for looping
          }
          context.write(key, new IntWritable(count));
       }
   }
```

CitationHistogramNewApi.java, **new** vs. **old**

```
public int run(String[] args) throws Exception {
       Configuration conf = getConf();
       //JobConf job = new JobConf(conf, CitationHistogram);
       //job.setJobName("CitationHistogram");
       Job job = new Job(conf, "CitationHistogram");
       job.setJarByClass(CitationHistogramNewApi.class);
       Path in = new Path(args[0]);
       Path out = new Path(args[1]);
       FileInputFormat.setInputPaths(job, in);
       FileOutputFormat.setOutputPath(job, out);
       job.setJobName("CitationHistogramNewApi");
       job.setMapperClass(MapClass.class);
       job.setReducerClass(Reduce.class);
       //job.setInputFormat(KeyValueTextInputFormat.class);
       job.setInputFormatClass(KeyValueTextInputFormat.class);
       //job.setOutputFormat(TextOutputFormat.class);
       job.setOutputFormatClass(TextOutputFormat.class);
       job.setOutputKeyClass(IntWritable.class);
       job.setOutputValueClass(IntWritable.class);
       //JobClient.runJob(job);
       System.exit(job.waitForCompletion(true)?0:1);
       return 0;
   }
```

CitationHistogramNewApi.java, **new** vs. **old**

```
public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(),
                      new CitationHistogramNewApi(), args);
        System.exit(res);
    }
}
```

- New class `CitationHistogramNewApi` performs identically as `CitationHistogram` class.

Appendix
Map Reduce Class Descriptions

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 processing is most efficient if the `map()` takes 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 HDFS 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.

Number of Reduce Tasks

- The right number of reduces seems to be `0.95` or `1.75 * (nodes * 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)`.

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 `InputFormat`s, typically sub-classes of `FileInputFormat`, is to split the input into *logical* `InputSplit`s 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 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.

`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.

`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.

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 key.value.separator.in.input. line property, and the default is the tab (\t) 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 Hadoopspecific 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 mapred.line.input.format. linespermap property, which defaults to one, sets N. key: LongWritable value: Text

`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 File `OutputFormat` 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.

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.

`Inverter` **`extends`** `Configured` **`implements`** `Tool`

- `Inverter is` a subclass of `Configured`, which is an implementation of the `Configurable` interface.
- All implementations of `Tool` need to implement `Configurable` (since `Tool` extends it).
- Extending, i.e. subclassing, `Configured` is a way to achieve this.
- The `run()` method obtains the `Configuration` using `getConf()` method of `Configurable` interface

`org.apache.hadoop.conf.Configured` class

```
@InterfaceAudience.Public
@InterfaceStability.Stable
public class Configured
extends Object
implements Configurable
```

`Configured` is the base class for objects that may be configured with a `Configuration`

Constructors:

`Configured()` Constructs a `Configured` object.

`Configured(Configuration conf)` Construct a `Configured` object based on specified `Configuration` object `conf`.

Methods:

`Configuration getConf()` Return the configuration used by this object.

`void setConf(Configuration conf)` Set the `Configuration` by passing a `Configuration` object `conf`.

`org.apache.hadoop.util.Tool` interface

```
@InterfaceAudience.Public
@InterfaceStability.Stable
public interface Tool extends Configurable
```

`Tool` interface supports handling of generic command-line options.

`Tool`, is the standard for any MapReduce tool/application.

The tool/application should delegate the handling of standard command-line options to `ToolRunner.run(Tool, String[])` and only handle its custom arguments.

Method Summary

`int run(String[] args)`

Execute the command with the given arguments.

`Tool` has methods inherited from interface `org.apache.hadoop.conf.Configurable`

`getConf(), setConf()`

- A typical implementation of the `Tool` interface looks like `Inverter` class

`org.apache.hadoop.util.Tool` class

```
public class Inverter extends Configured implements Tool {
    public int run(String[] args) throws Exception {  // must implement run()
        // Configuration processed by ToolRunner
        Configuration conf = getConf();
        // Create a JobConf using the provided conf
        JobConf job = new JobConf(conf, MyApp.class);
        // Process custom command-line options
        Path in = new Path(args[1]);
        Path out = new Path(args[2]);
        FileInputFormat.setInputPaths(job, in);
        FileOutputFormat.setOutputPath(job, out);
        // Specify various job-specific parameters
        job.setJobName("inverter");
        job.setInputPath(in);
        job.setOutputPath(out);
        job.setMapperClass(MapClass.class);
        job.setReducerClass(Reducer.class);
        // Submit the job, then poll for progress until the job is complete
        JobClient.runJob(job);
        return 0;
    }
    public static void main(String[] args) throws Exception {
        // Let ToolRunner handle generic command-line options
        int res = ToolRunner.run(new Configuration(), new Inverter(), args);
        System.exit(res);
      }
    }
```

`org.apache.hadoop.util.ToolRunner`

```
@InterfaceAudience.Public
@InterfaceStability.Stable
public class ToolRunner extends Object
```

- A utility to help run `Tools`.
- `ToolRunner` can be used to run classes implementing `Tool` interface. It works in conjunction with `GenericOptionsParser` to parse the generic command line arguments of command `hadoop` and modifies the `Configuration` of the `Tool`. The application-specific options are passed along without being modified.
- Constructor Summary

`ToolRunner()`

- Method Summary

  `static boolean confirmPrompt(String prompt)`
- Print out a prompt to the user, and return true if the user responds with "y" or "yes".

  `static void printGenericCommandUsage(PrintStream out)`
- Prints generic command-line argurments and usage information.

  `static int run(Configuration conf, Tool tool, String[] args)`
- Runs the given `Tool by Tool.run(String[]),` after parsing with the given generic arguments.

  `static int run(Tool tool, String[] args)`
- Runs the `Tool` with its `Configuration.`

org.apache.hadoop.mapred.JobConf

```
@InterfaceAudience.Public
@InterfaceStability.Stable
public class JobConf extends Configuration
```

- A map/reduce job configuration.
- `JobConf` is the primary interface for a user to describe a map-reduce job to the Hadoop framework for execution. The framework tries to execute the job as-is described by JobConf,
- Some configuration parameters might have been marked as final by administrators and hence cannot be altered.
- While some job parameters are straight-forward to set (e.g. `setNumReduceTasks(int)`), some parameters interact subtly with the rest of the framework and/or job-configuration and are relatively more complex for the user to control finely (e.g. `setNumMapTasks(int)`).
- `JobConf` typically specifies the `Mapper`, combiner (if any), `Partitioner`, `Reducer`, `InputFormat` and `OutputFormat` implementations to be used etc.
- Optionally `JobConf` is used to specify other advanced facets of the job such as `Comparators` to be used, files to be put in the `DistributedCache`, whether or not intermediate and/or job outputs are to be compressed (and how), debugability via user-provided scripts (`setMapDebugScript(String)/setReduceDebugScript(String)),` for doing post-processing on task logs, task's `stdout, stderr, syslog`. and etc.

org.apache.hadoop.mapred.JobConf

- The following demonstrates how to configure a job via `JobConf`:

```
// Create a new JobConf
JobConf job = new JobConf(new Configuration(),Inverter.class);

// Specify various job-specific parameters
job.setJobName("Inverter");

FileInputFormat.setInputPaths(job, new Path("in"));
FileOutputFormat.setOutputPath(job, new Path("out"));

job.setMapperClass(Inverter.MapClass.class);
job.setCombinerClass(Inverter.Reducer.class);
job.setReducerClass(Inverter.Reducer.class);

job.setInputFormat(SequenceFileInputFormat.class);
job.setOutputFormat(SequenceFileOutputFormat.class);
```

`JobConf` and `Configuration`

- The `JobConf` object has many parameters, but we don't want to program the driver to set up all of them.
- The configuration files of the Hadoop installation are a good starting point.
- When starting a job from the command line, the user may also want to pass extra arguments to alter the job configuration.
- The driver can define its own set of commands and process the user arguments itself to enable the user to modify some of the configuration parameters.
- Configuration files are represented by the `Configuration` object.
- `ToolRunner` internally runs `GenericOptionsParser` object which reads and parses the command line arguments.

`org.apache.hadoop.conf.Configuration` class

```
@InterfaceAudience.Public
@InterfaceStability.Stable
public class Configuration
extends Object
implements Iterable<Map.Entry<String,String>>, Writable
```

Provides access to configuration parameters.

Resources

Configurations are specified by resources. A resource contains a set of name/value pairs as XML data. Each resource is named by either a String or by a Path. If named by a String, then the classpath is examined for a file with that name. If named by a Path, then the local file system is examined directly, without referring to the classpath. Unless explicitly turned off, Hadoop by default specifies two resources, loaded in-order from the classpath:

`core-default.xml` : Read-only defaults for hadoop

`core-site.xml`: Site-specific configuration for a given hadoop installation.

Applications may add additional resources. On our MRv1 VM, file `core-site.xml` resides in the directory `/etc/hadoop/conf.pseudo.mr1`

`org.apache.hadoop.conf.Configuration` class

`Final Parameters`

- On our MRv1 VM, file `core-site.xml` resides in the directory `/etc/hadoop/conf.pseudo.mr1`
- Configuration parameters may be declared *final*. Once a resource declares a value final, no subsequently-loaded resource can alter that value.
- For example, one might define a final parameter with:

```
<property>
  <name>dfs.hosts.include</name>
  <value>/etc/hadoop/conf/hosts.include</value>
  <final>true</final>
</property>
```

`org.apache.hadoop.conf.Configuration` class

`Variable Expansion`

- Value strings are first processed for *variable expansion*.
- The available properties are:
  - other properties defined in this configuration file; and,
  - if a name is undefined there, properties in [`System.getProperties()`]
- For example, if a configuration resource contains the following property definitions :

```
<property>
  <name>basedir</name>
  <value>/user/${user.name}</value>
</property>
<property>
  <name>tempdir</name>
  <value>${basedir}/tmp</value>
</property>
```

- When `conf.get("tempdir")` is called, then `${basedir}` will be resolved to another property in this configuration, while `${user.name}` would ordinarily be resolved to the value of the System property with that name.

`GenericOptionsParser`, `Tool` and `ToolRunner`

- Hadoop comes with a few helper classes for making it easier to run jobs from the command line.
- `GenericOptionsParser` is a class that interprets common Hadoop command-line options and sets them on a `Configuration` object for your application to use as desired.
- You don't usually use `GenericOptionsParser` directly. It is more convenient to implement the `Tool` interface and run your application with the `ToolRunner`, which uses `GenericOptionsParser` internally:

```
public static void main(String[] args) throws Exception {
   int res = ToolRunner.run(
            new Configuration(), new Inverter(), args);
   System.exit(res);
}
```

`org.apache.hadoop.util.GenericOptionsParser`

http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/util/GenericOptionsParser.html

- `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`, and additional configuration resources.

Generic Options

- The supported generic options are:

  `-conf <configuration file>` specify a configuration file

  `-D <property=value>` user 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.

  `-archives <comma separated list of archives>` specify comma separated archives to be unarchived on the compute machines.

- Generic command line arguments **might** modify `Configuration` objects, given to constructors.
- The functionality is implemented using Commons CLI.

`org.apache.hadoop.mapred.OutputCollector`

```
@InterfaceAudience.Public
@InterfaceStability.Stable
public interface OutputCollector<K,V>
```

- Collects the `<key, value>` pairs output by `Mappers` and `Reducers.`
- `OutputCollector` is the generalization of the facility provided by the MapReduce framework to collect data output by either the `Mapper` or the `Reducer` i.e. intermediate outputs or the final output of the job.

- Method Summary

```
void collect(K key, V value) throws IOExceptionAdds
```

Adds a key/value pair to the output.