Apache Hadoop – A course for undergraduates

Lecture 5

Practical Development Tips and Techniques

Chapter 5.1

Practical Development Tips and Techniques

- **Strategies for debugging MapReduce code**
- **How to test MapReduce code locally using LocalJobRunner**
- **How to write and view log files**
- **How to retrieve job information with counters**
- **Why reusing objects is a best practice**
- **How to create Map-only MapReduce jobs**

Chapter Topics

Practical Development Tips and Techniques

- **Strategies for Debugging MapReduce Code**
- Testing MapReduce Code Locally Using LocalJobRunner
- Writing and Viewing Log Files
- Retrieving Job Information with Counters
- Reusing Objects
- Creating Map-only MapReduce Jobs

Introduction to Debugging

- **Debugging MapReduce code is difficult!**
  - Each instance of a Mapper runs as a separate task
    - Often on a different machine
  - Difficult to attach a debugger to the process
  - Difficult to catch 'edge cases'
- **Very large volumes of data mean that unexpected input is likely to appear**
  - Code which expects all data to be well-formed is likely to fail

Common-Sense Debugging Tips

- **Code defensively**
  - Ensure that input data is in the expected format
  - Expect things to go wrong
  - Catch exceptions
- **Start small, build incrementally**
- **Make as much of your code as possible Hadoop-agnostic**
  - Makes it easier to test
- **Write unit tests**
- **Test locally whenever possible**
  - With small amounts of data
- **Then test in pseudo-distributed mode**
- **Finally, test on the cluster**

Testing Strategies

- **When testing in pseudo-distributed mode, ensure that you are testing with a similar environment to that on the real cluster**
  - Same amount of RAM allocated to the task JVMs
  - Same version of Hadoop
  - Same version of Java
  - Same versions of third-party libraries

Chapter Topics

Practical Development Tips and Techniques

- Strategies for Debugging MapReduce Code
- **Testing MapReduce Code Locally Using LocalJobRunner**
- Writing and Viewing Log Files
- Retrieving Job Information with Counters
- Reusing Objects
- Creating Map-only MapReduce Jobs

Testing Locally (1)

- **Hadoop can run MapReduce in a single, local process**
  - Does not require any Hadoop daemons to be running
  - Uses the local filesystem instead of HDFS
  - Known as LocalJobRunner mode
- **This is a very useful way of quickly testing incremental changes to code**

Testing Locally (2)

- **To run in LocalJobRunner mode, add the following lines to the driver code:**

```
Configuration conf = new Configuration();
conf.set("mapred.job.tracker", "local");
conf.set("fs.default.name", "file:///");
```

  - Or set these options on the command line if your driver uses ToolRunner
    - **`-fs`** is equivalent to **`-D fs.default.name`**
    - **`-jt`** is equivalent to **`-D maprep.job.tracker`**
    - e.g.

```
$ hadoop jar myjar.jar MyDriver -fs=file:/// -jt=local \
  indir outdir
```

Testing Locally (3)

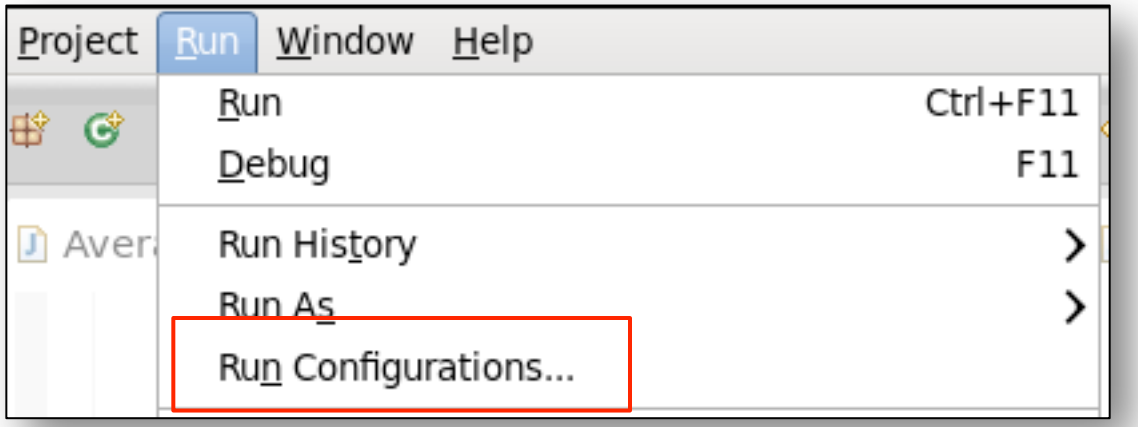
- **Some limitations of LocalJobRunner mode:**
  - Distributed Cache does not work
  - The job can only specify a single Reducer
  - Some 'beginner' mistakes may not be caught
    - For example, attempting to share data between Mappers will work, because the code is running in a single JVM

LocalJobRunner Mode in Eclipse (1)

- **Eclipse on the course VM runs Hadoop code in LocalJobRunner mode from within the IDE**
  - This is Hadoop's default behavior when no configuration is provided
- **This allows rapid development iterations**
  - 'Agile programming'

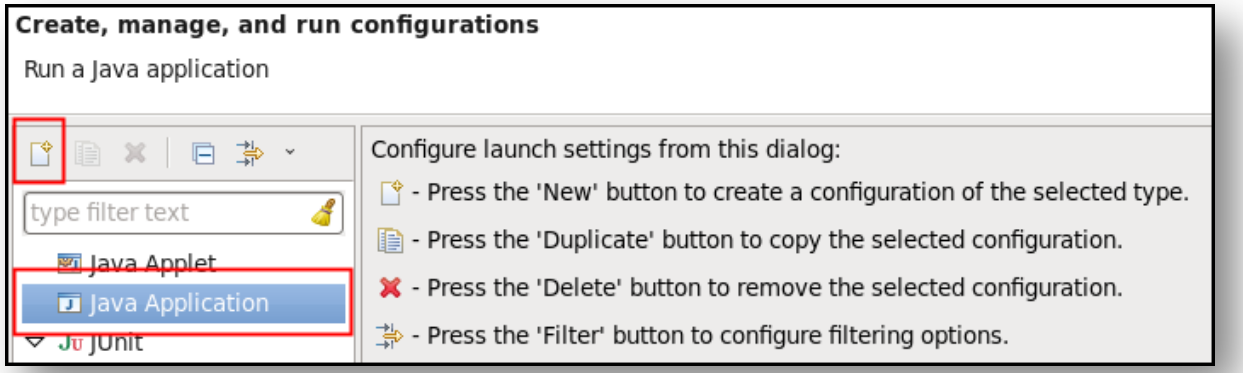
LocalJobRunner Mode in Eclipse (2)

- **Specify a Run Configuration**

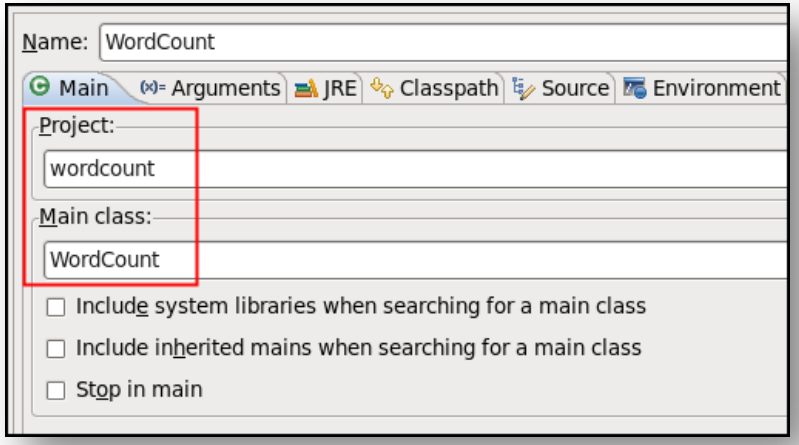


LocalJobRunner Mode in Eclipse (3)

- **Select Java Application, then select the New button**

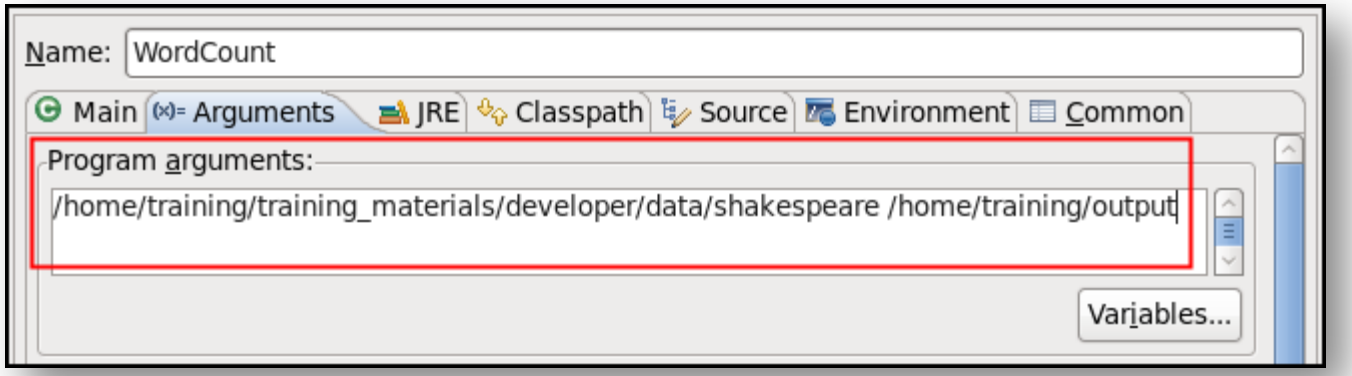


- **Verify that the Project and Main Class fields are pre-filled correctly**

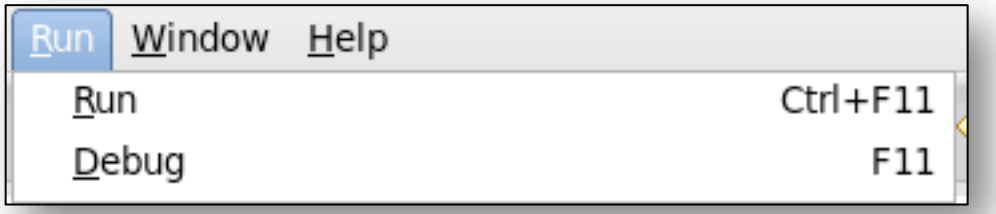


LocalJobRunner Mode in Eclipse (4)

- **Specify values in the Arguments tab**
  - Local input and output files
  - Any configuration options needed when your job runs

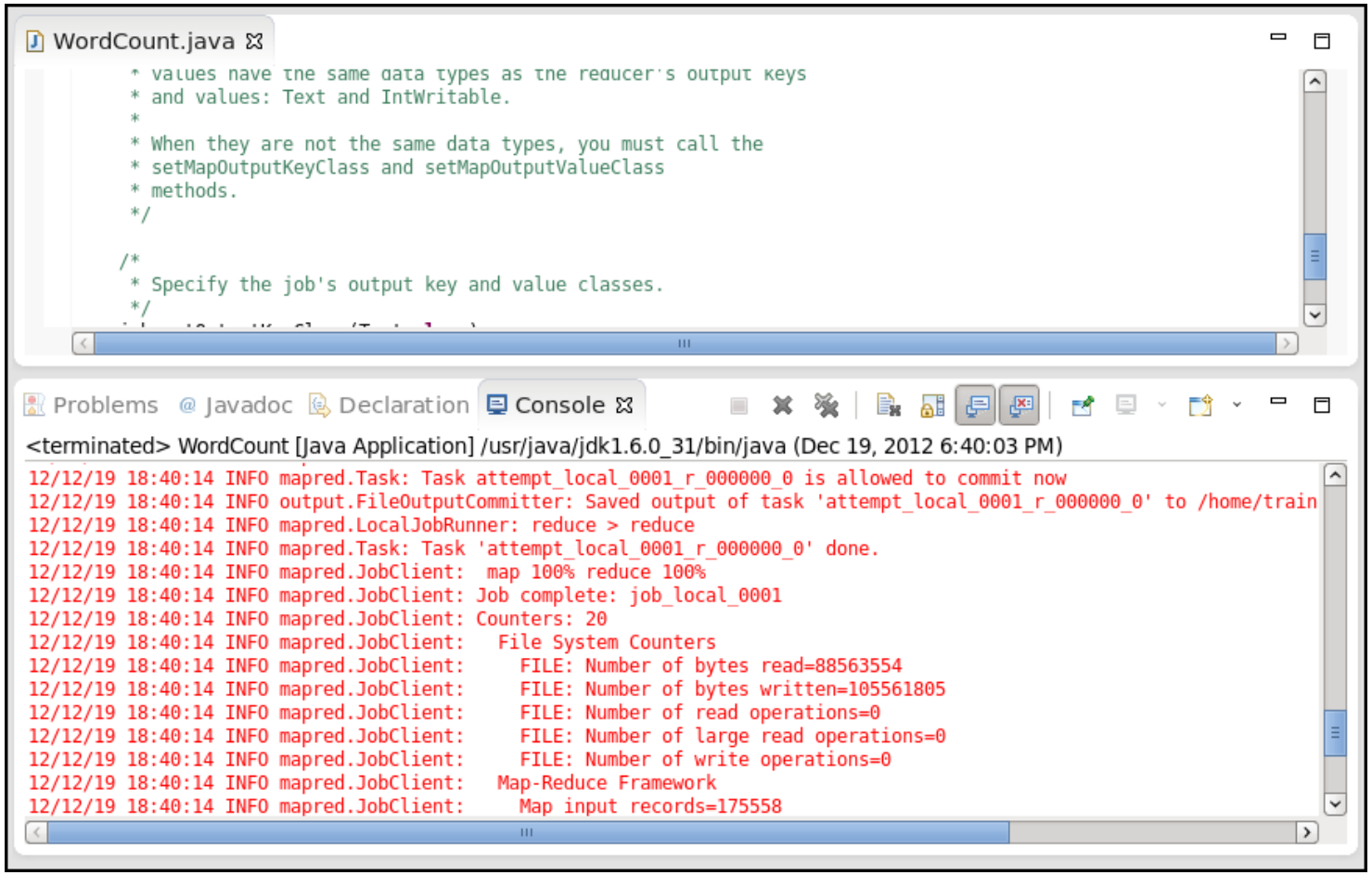


- **Define breakpoints if desired**
- **Execute the application in run mode or debug mode**



LocalJobRunner Mode in Eclipse (5)

- **Review output in the Eclipse console window**



Chapter Topics

Practical Development Tips and Techniques

- Strategies for Debugging MapReduce Code
- Testing MapReduce Code Locally Using LocalJobRunner
- **Writing and Viewing Log Files**
- Retrieving Job Information with Counters
- Reusing Objects
- Creating Map-only MapReduce Jobs

Before Logging: `stdout` and `stderr`

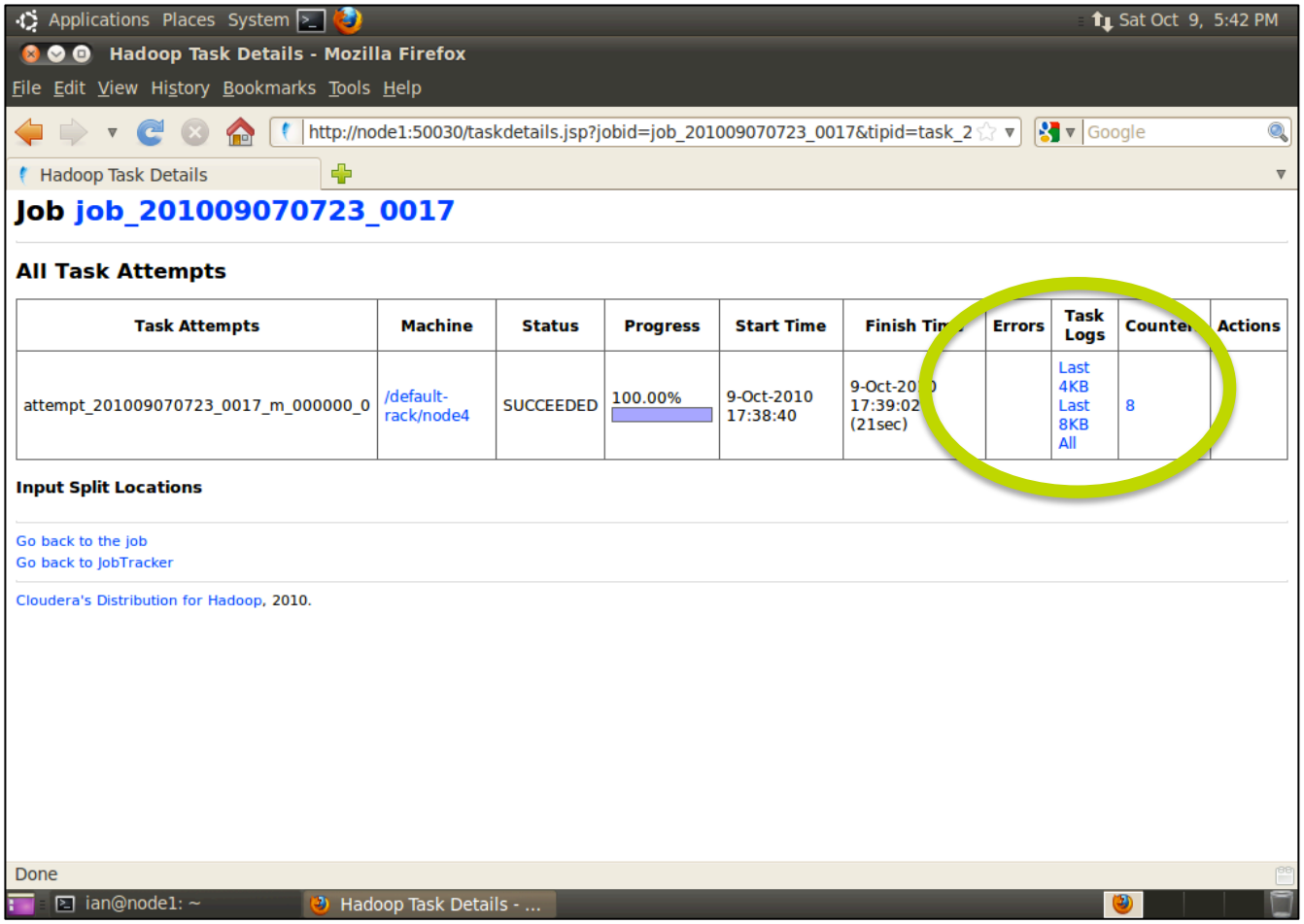
- **Tried-and-true debugging technique: write to `stdout` or `stderr`**
- **If running in LocalJobRunner mode, you will see the results of `System.err.println()`**
- **If running on a cluster, that output will not appear on your console**
  - Output is visible via Hadoop's Web UI

Aside: The Hadoop Web UI

- **All Hadoop daemons contain a Web server**
  - Exposes information on a well-known port
- **Most important for developers is the JobTracker Web UI**
  - `http://<job_tracker_address>:50030/`
  - `http://localhost:50030/` if running in pseudo-distributed mode
- **Also useful: the NameNode Web UI**
  - `http://<name_node_address>:50070/`

Aside: The Hadoop Web UI (cont'd)

- **Your instructor will now demonstrate the JobTracker UI**



Logging: Better Than Printing

- **`println` statements rapidly become awkward**
  - Turning them on and off in your code is tedious, and leads to errors

- **Logging provides much finer-grained control over:**
  - What gets logged
  - When something gets logged
  - How something is logged

Logging With `log4j`

- **Hadoop uses `log4j` to generate all its log files**
- **Your Mappers and Reducers can also use `log4j`**
  - All the initialization is handled for you by Hadoop
- **Add the `log4j.jar-<version>` file from your CDH distribution to your classpath when you reference the log4j classes**

```
import org.apache.log4j.Level;
import org.apache.log4j.Logger;

class FooMapper implements Mapper {
    private static final Logger LOGGER =
        Logger.getLogger (FooMapper.class.getName());
    ...
}
```

Logging With `log4j` (cont'd)

- **Simply send strings to loggers tagged with severity levels:**

```
LOGGER.trace("message");
LOGGER.debug("message");
LOGGER.info("message");
LOGGER.warn("message");
LOGGER.error("message");
```

- **Beware expensive operations like concatenation**
  - To avoid performance penalty, make it conditional like this:

```
if (LOGGER.isDebugEnabled()) {
   LOGGER.debug("Account info:" + acct.getReport());
}
```

log4j Configuration

- **Node-wide configuration for `log4j` is stored in `/etc/hadoop/conf/log4j.properties`**
- **Override settings for your application in your own `log4j.properties`**
  - Can change global log settings with `hadoop.root.log` property
  - Can override log level on a per-class basis, e.g.

```
log4j.logger.org.apache.hadoop.mapred.JobTracker=WARN
log4j.logger.com.mycompany.myproject.FooMapper=DEBUG
```

Full class name

- **Or set the level programmatically:**

```
LOGGER.setLevel(Level.WARN);
```

Setting Logging Levels for a Job

- **You can tell Hadoop to set logging levels for a job using configuration properties**
  - **`mapred.map.child.log.level`**
  - **`mapred.reduce.child.log.level`**
- **Examples**
  - Set the logging level to DEBUG for the Mapper

```
$ hadoop jar myjob.jar MyDriver \
  -Dmapred.map.child.log.level=DEBUG indir outdir
```

  - Set the logging level to WARN for the Reducer

```
$ hadoop jar myjob.jar MyDriver \
  -Dmapred.reduce.child.log.level=WARN indir outdir
```

Where Are Log Files Stored?

- **Log files are stored on the machine where the task attempt ran**
  - Location is configurable
  - By default:

```
/var/log/hadoop-0.20-mapreduce/
userlogs/${task.id}/syslog
```

- **You will often not have ssh access to a node to view its logs**
  - Much easier to use the JobTracker Web UI
    - Automatically retrieves and displays the log files for you

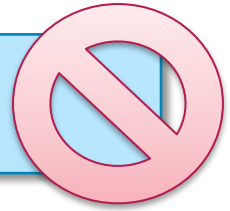
Restricting Log Output

- **If you suspect the input data of being faulty, you may be tempted to log the (key, value) pairs your Mapper receives**
  - Reasonable for small amounts of input data
  - Caution! If your job runs across 500GB of input data, you could be writing up to 500GB of log files!
  - Remember to think at scale…
- **Instead, wrap vulnerable sections of code in `try {...}` blocks**
  - Write logs in the `catch {...}` block
    - This way only critical data is logged

Aside: Throwing Exceptions

- **You could throw exceptions if a particular condition is met**
  - For example, if illegal data is found

```
throw new RuntimeException("Your message here");
```



- **Usually not a good idea**
  - Exception causes the task to fail
  - If a task fails four times, the entire job will fail

Chapter Topics

Practical Development Tips and Techniques

- Strategies for Debugging MapReduce Code
- Testing MapReduce Code Locally Using LocalJobRunner
- Writing and Viewing Log Files
- **Retrieving Job Information with Counters**
- Reusing Objects
- Creating Map-only MapReduce Jobs

What Are Counters? (1)

- **Counters provide a way for Mappers or Reducers to pass aggregate values back to the driver after the job has completed**
  - Their values are also visible from the JobTracker's Web UI
  - And are reported on the console when the job ends
- **Very basic: just have a name and a value**
  - Value can be incremented within the code
- **Counters are collected into Groups**
  - Within the group, each Counter has a name
- **Example: A group of Counters called `RecordType`**
  - Names: `TypeA`, `TypeB`, `TypeC`
  - Appropriate Counter can be incremented as each record is read in the Mapper

What Are Counters? (2)

- **Counters can be set and incremented via the method**

```
context.getCounter(group, name).increment(amount);
```

- **Example:**

```
context.getCounter("RecordType","A").increment(1);
```

Retrieving Counters in the Driver Code

- **To retrieve Counters in the Driver code after the job is complete, use code like this in the driver:**

```
long typeARecords =
    job.getCounters().findCounter("RecordType","A").getValue();

long typeBRecords =
    job.getCounters().findCounter("RecordType","B").getValue();
```

Counters: Caution

- **Do not rely on a counter's value from the Web UI while a job is running**
  - Due to possible speculative execution, a counter's value could appear larger than the actual final value
  - Modifications to counters from subsequently killed/failed tasks will be removed from the final count

Chapter Topics

Practical Development Tips and Techniques

- Strategies for Debugging MapReduce Code
- Testing MapReduce Code Locally Using LocalJobRunner
- Writing and Viewing Log Files
- **Reusing Objects**
- Creating Map-only MapReduce Jobs

Reuse of Objects is Good Practice (1)

- **It is generally good practice to reuse objects**
  - Instead of creating many new objects
- **Example: Our original WordCount Mapper code**

```
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    Strin

    for (
      if

        context.write(new Text(word), new IntWritable(1));
      }
    }
  }
}
```

Each time the `map()` method is called, we create a new `Text` object and a new `IntWritable` object.

Reuse of Objects is Good Practice (2)

- **Instead, this is better practice:**

```
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
 private final static IntWritable one = new IntWritable(1);
 private Text wordObject = new Text();

 @Overr
  publi
      t

    String line = value.toString();

    for (String word : line.split("\\W+")) {
      if (word.length() > 0) {
        wordObject.set(word);
        context.write(wordObject, one);
      }
    }
  }
}
```

Create objects for the key and value outside of your `map()` method

Reuse of Objects is Good Practice (3)

- **Instead, this is better practice:**

```
public class WordMapper extends Mapper<LongWritable, Text, Text, IntWritable>
{
 private final static IntWritable one = new IntWritable(1);
 private Text wordObject = new Text();

 @Ov
  pu

    if (word.length() > 0) {
      wordObject.set(word);
      context.write(wordObject, one);
    }
  }
 }
}
```

Within the `map()` method, populate the objects and write them out. Hadoop will take care of serializing the data so it is perfectly safe to re-use the objects.

Object Reuse: Caution!

- **Hadoop re-uses objects all the time**
- **For example, each time the Reducer is passed a new value, the same object is reused**
- **This can cause subtle bugs in your code**
  - For example, if you build a list of value objects in the Reducer, each element of the list will point to the same underlying object
    - Unless you do a deep copy

Chapter Topics

Practical Development Tips and Techniques

- Strategies for Debugging MapReduce Code
- Testing MapReduce Code Locally Using LocalJobRunner
- Writing and Viewing Log Files
- Retrieving Job Information with Counters
- Reusing Objects
- **Creating Map-only MapReduce jobs**

Map-Only MapReduce Jobs

- **There are many types of job where only a Mapper is needed**
- **Examples:**
  - Image processing
  - File format conversion
  - Input data sampling
  - ETL

Creating Map-Only Jobs

- **To create a Map-only job, set the number of Reducers to 0 in your Driver code**

```
job.setNumReduceTasks(0);
```

- **Call the `Job.setOutputKeyClass` and `Job.setOutputValueClass` methods to specify the output types**
  - *Not* the `Job.setMapOutputKeyClass` and `Job.setMapOutputValueClass` methods
- **Anything written using the `Context.write` method in the Mapper will be written to HDFS**
  - Rather than written as intermediate data
  - One file per Mapper will be written

Key Points

- LocalJobRunner lets you test jobs on your local machine
- Hadoop uses the Log4J framework for logging
- Reusing objects is a best practice
- Counters provide a way of passing numeric data back to the driver
- Create Map-only MapReduce jobs by setting the number of Reducers to zero

Bibliography

The following offer more information on topics discussed in this chapter

- **Java version selection**
  - http://wiki.apache.org/hadoop/HadoopJavaVersions
- **For an example of image processing and file format conversion in Hadoop, see**
  - http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/