Delving Deeper into the Hadoop API

Chapter 4.1

Delving Deeper into the Hadoop API

- Using the ToolRunner class
- Decreasing the amount of intermediate data with Combiners
- Setting up and tearing down Mappers and Reducers using the `setup` and `cleanup` methods
- How to access HDFS programmatically
- How to use the distributed cache
- How to use the Hadoop API's library of Mappers, Reducers, and Partitioners

Chapter Topics

Delving Deeper into the Hadoop API

- **Using the ToolRunner Class**
- Setting Up and Tearing Down Mappers and Reducers
- Decreasing the Amount of Intermediate Data with Combiners
- Accessing HDFS programmatically
- Using the Distributed Cache
- Using the Hadoop API's Library of Mappers, Reducers and Partitioners

Why Use ToolRunner?

- **You can use `ToolRunner` in MapReduce driver classes**
  - This is not required, but is a best practice
- **`ToolRunner` uses the `GenericOptionsParser` class internally**
  - Allows you to specify configuration options on the command line
  - Also allows you to specify items for the Distributed Cache on the command line (see later)

How to Implement ToolRunner: Complete Driver

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.Job;import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class WordCount extends Configured implements Tool {

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

  public int run(String[] args) throws Exception {
    if (args.length != 2) {
      System.out.printf(
          "Usage: %s [generic options] <input dir> <output dir>\n", getClass().getSimpleName());
      return -1;
    }
    Job job = new Job(getConf());
    job.setJarByClass(WordCount.class);
    job.setJobName("Word Count");

    FileInputFormat.setInputPaths(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(WordMapper.class);
    job.setReducerClass(SumReducer.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    boolean success = job.waitForCompletion(true);
    return success ? 0 : 1;
  }
}
```

How to Implement ToolRunner: Imports

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class WordCount

  public static void m
    int exitCode = Too
    System.exit(exitCo
  }

  public int run(String[] args) throws Exception {
    if (args.length != 2) {
      System.out.printf(
          "Usage: %s [generic options] <input dir> <output dir>\n",
           getClass().getSimpleName());
      return -1;
    }
    Job job = new Job(getConf());
```

Import the relevant classes. We omit the `import` statements in future slides for brevity.

How to Implement ToolRunner: Driver Class Definition

```
public class WordCount extends Configured implements Tool {

  public static void ma
    int exitCode = Tool
    System.exit(exitCo
  }

  public int run(String

    if (args.length != 2
      System.out.printf(
          "Usage: %s [generic options] <input dir> <output dir>\n", getClass().getSimpleName());
      return -1;
    }
    Job job = new Job(getConf());
    job.setJarByClass(WordCount.class); job.setJobName("Word Count");

    FileInputFormat.setInputPaths(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(WordMapper.class);
    job.setReducerClass(SumReducer.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    boolean success = job.waitForCompletion(true);
    return success ? 0 : 1;
  }
}
```

The driver class implements the `Tool` interface and extends the `Configured` class.

How to Implement ToolRunner: Main Method

```
public class WordCount extends Configured implements Tool {

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

  public int run(String
   if (args.length != 2
     System.out.printf(
         "Usage: %s [ge
     return -1;
   }
   Job job = new Job(getConf());
   job.setJarByClass(WordCount.class);
   job.setJobName("Word Count");

   FileInputFormat.setInputPaths(job, new Path(args[0]));
   FileOutputFormat.setOutputPath(job, new Path(args[1]));

   job.setMapperClass(WordMapper.class);
   job.setReducerClass(SumReducer.class);
   job.setMapOutputKeyClass(Text.class);
   job.setMapOutputValueClass(IntWritable.class);

   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(IntWritable.class);

   ...
```

The driver `main` method calls `ToolRunner.run`.

How to Implement ToolRunner: Run Method

```
public class WordCount

   public static void
     int exitCode = To
           new WordCou
     System.exit(exitC
   }
```

The driver `run` method creates, configures, and submits the job.

```
public int run(String[] args) throws Exception {
  if (args.length != 2) {
   System.out.printf(
       "Usage: %s [generic options] <input dir> <output dir>\n", getClass().getSimpleName());
   return -1;
   }
  Job job = new Job(getConf());
  job.setJarByClass(WordCount.class);
  job.setJobName("Word Count");

  FileInputFormat.setInputPaths(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));

  ...
```

ToolRunner Command Line Options

- **ToolRunner allows the user to specify configuration options on the command line**
- **Commonly used to specify Hadoop properties using the `-D` flag**
  - Will override any default or site properties in the configuration
  - But will *not* override those set in the driver code

```
$ hadoop jar myjar.jar MyDriver \
-D mapreduce.job.reduces=10 myinputdir myoutputdir
```

- **Note that `-D` options must appear before any additional program arguments**
- **Can specify an XML configuration file with `-conf`**
- **Can specify the default filesystem with `-fs` *`uri`***
  - Shortcut for `-D fs.defaultFS=`*`uri`*

Chapter Topics

Delving Deeper into the Hadoop API

- Using the ToolRunner Class
- **Setting Up and Tearing Down Mappers and Reducers**
- Decreasing the amount of intermediate data with Combiners
- Accessing HDFS programmatically
- Using the Distributed Cache
- Using the Hadoop API's Library of Mappers, Reducers and Partitioners

The `setup` Method

- **It is common to want your Mapper or Reducer to execute some code before the `map` or `reduce` method is called for the first time**
  - Initialize data structures
  - Read data from an external file
  - Set parameters
- **The `setup` method is run before the `map` or `reduce` method is called for the first time**

```
public void setup(Context context)
```

The `cleanup` Method

- Similarly, you may wish to perform some action(s) after all the records have been processed by your Mapper or Reducer
- The `cleanup` method is called before the Mapper or Reducer terminates

```
public void cleanup(Context context) throws
    IOException, InterruptedException
```

Passing Parameters

```
public class MyDriverClass {
    public int main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        conf.setInt ("paramname",value);
        Job job = new Job(conf);
        ...
        boolean success = job.waitForCompletion(true);
        return success ? 0 : 1;
    }
}
```

```
public class MyMapper extends Mapper {

    public void setup(Context context) {
          Configuration conf = context.getConfiguration();
            int myParam = conf.getInt("paramname", 0);
            ...
    }
    public void map...
}
```

Chapter Topics

Delving Deeper into the Hadoop API

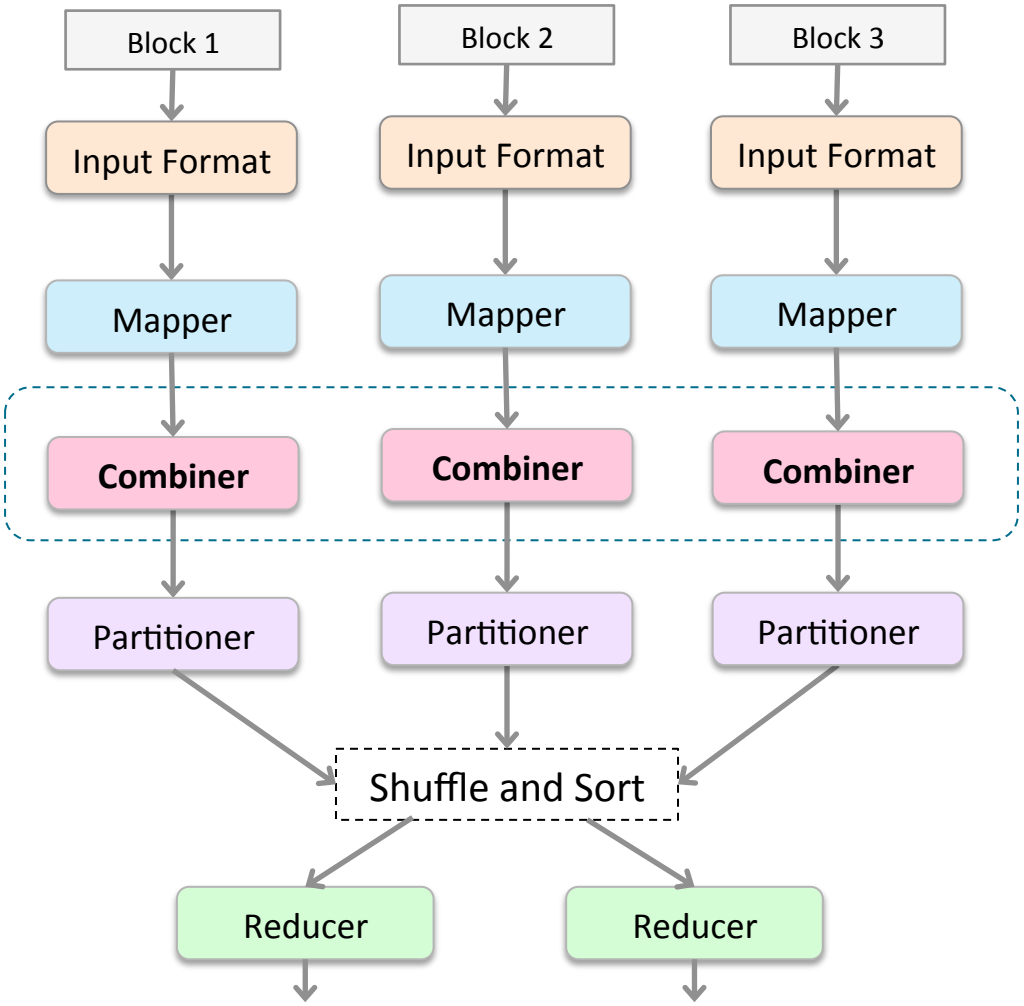
- Using the ToolRunner Class
- Setting Up and Tearing Down Mappers and Reducers
- **Decreasing the Amount of Intermediate Data with Combiners**
- Accessing HDFS programmatically
- Using the Distributed Cache
- Using the Hadoop API's Library of Mappers, Reducers and Partitioners

The Combiner

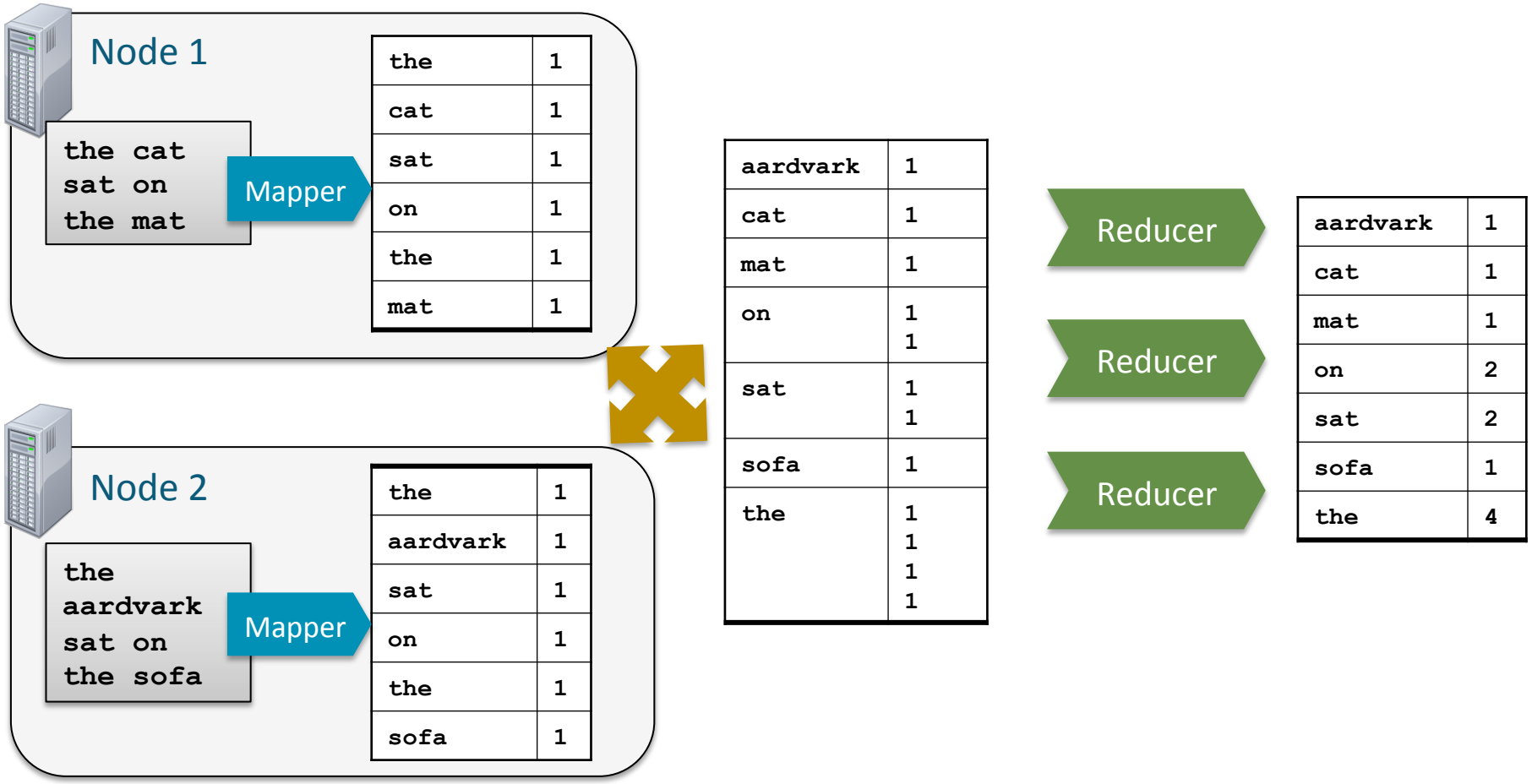
- **Often, Mappers produce large amounts of intermediate data**
  - That data must be passed to the Reducers
  - This can result in a lot of network traffic
- **It is often possible to specify a *Combiner***
  - Like a 'mini-Reducer'
  - Runs locally on a single Mapper's output
  - Output from the Combiner is sent to the Reducers
- **Combiner and Reducer code are often identical**
  - Technically, this is possible if the operation performed is commutative and associative
  - Input and output data types for the Combiner/Reducer must be identical

The Combiner

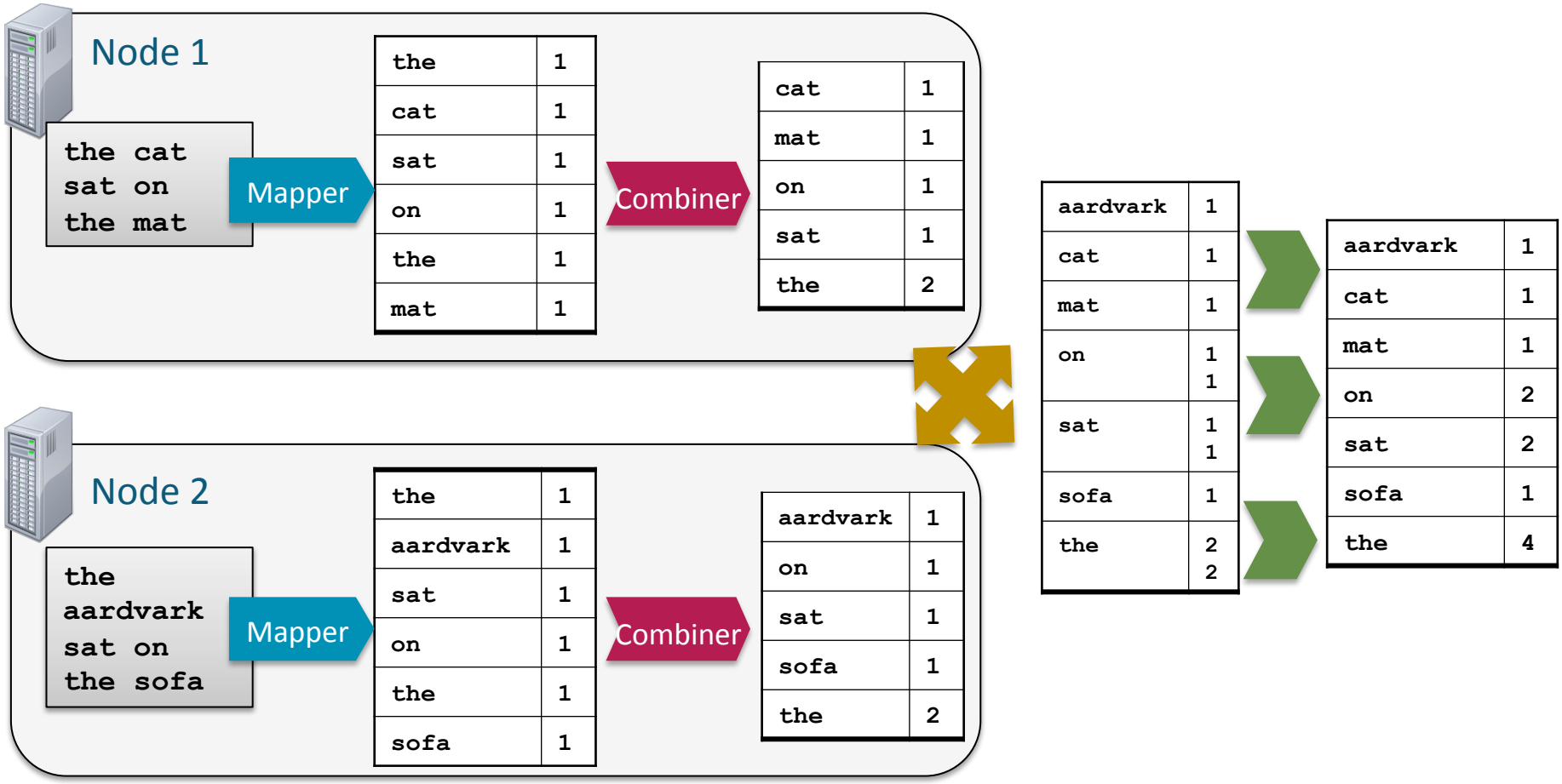
- **Combiners run as part of the Map phase**
- **Output from the Combiners is passed to the Reducers**



WordCount Revisited



WordCount With Combiner



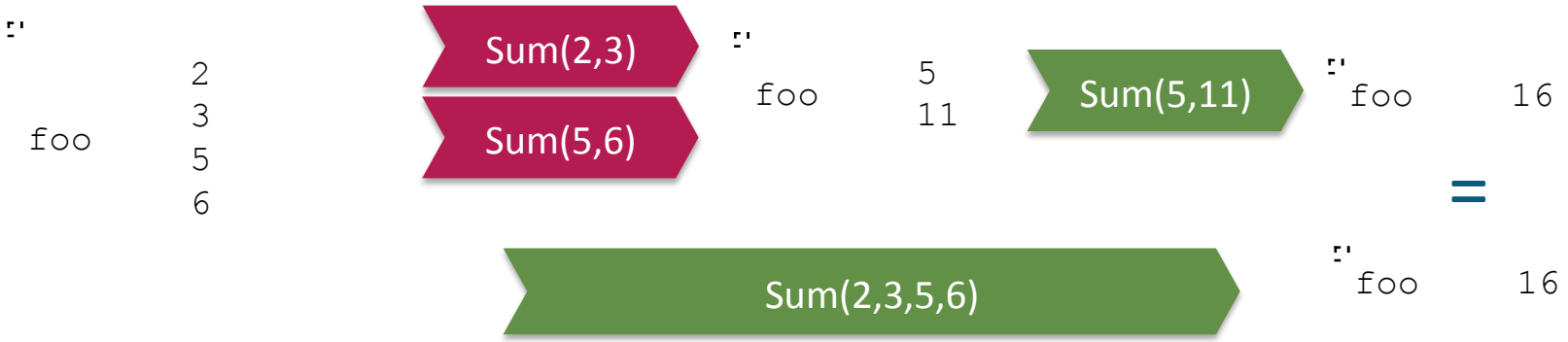
Writing a Combiner

- **The Combiner uses the same signature as the Reducer**
  - Takes in a key and a list of values
  - Outputs zero or more (key, value) pairs
  - The actual method called is the **`reduce`** method in the class

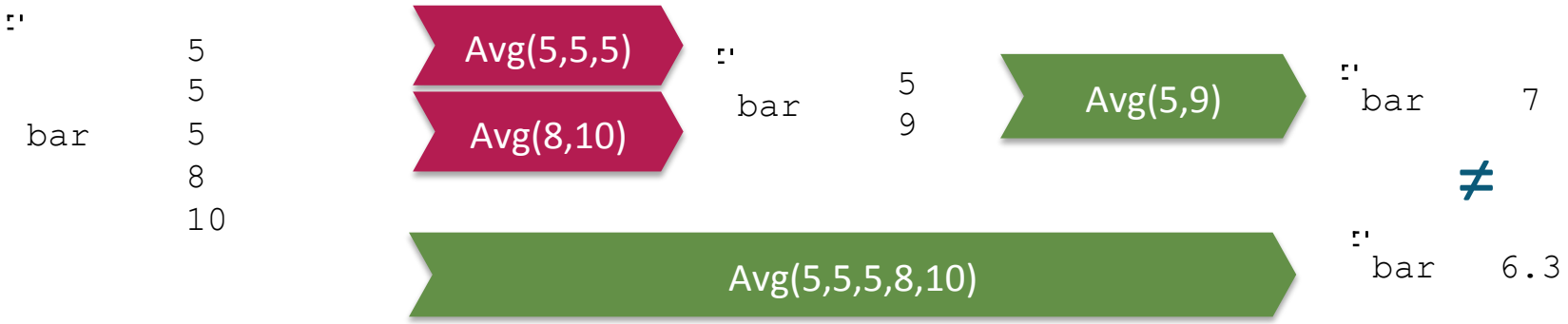
```
reduce(inter_key, [v1, v2, …]) →
                          (result_key, result_value)
```

Combiners and Reducers

- **Some Reducers may be used as Combiners**
  - If operation is associative and commutative, e.g., SumReducer



  - Some Reducers *cannot* be used as a Combiner, e.g., AverageReducer



Specifying a Combiner

- **Specify the Combiner class to be used in your MapReduce code in the driver**
  - Use the `setCombinerClass` method, e.g.:

```
job.setMapperClass(WordMapper.class);
job.setReducerClass(SumReducer.class);
job.setCombinerClass(SumReducer.class);
```

- **Input and output data types for the Combiner and the Reducer for a job must be identical**
- **VERY IMPORTANT: The Combiner may run once, or more than once, on the output from any given Mapper**
  - Do not put code in the Combiner which could influence your results if it runs more than once

Chapter Topics

Delving Deeper into the Hadoop API

- Using the ToolRunner Class
- Setting Up and Tearing Down Mappers and Reducers
- Decreasing the Amount of Intermediate Data with Combiners
- **Accessing HDFS Programmatically**
- Using the Distributed Cache
- Using the Hadoop API's Library of Mappers, Reducers and Partitioners

Accessing HDFS Programmatically

- **In addition to using the command-line shell, you can access HDFS programmatically**
  - Useful if your code needs to read or write 'side data' in addition to the standard MapReduce inputs and outputs
  - Or for programs outside of Hadoop which need to read the results of MapReduce jobs
- **Beware: HDFS is not a general-purpose filesystem!**
  - Files cannot be modified once they have been written, for example
- **Hadoop provides the `FileSystem` abstract base class**
  - Provides an API to generic file systems
    - Could be HDFS
    - Could be your local file system
    - Could even be, for example, Amazon S3

The FileSystem API (1)

- In order to use the `FileSystem` API, retrieve an instance of it

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
```

- The `conf` object has read in the Hadoop configuration files, and therefore knows the address of the NameNode
- A file in HDFS is represented by a `Path` object

```
Path p = new Path("/path/to/my/file");
```

The FileSystem API (2)

- **Some useful API methods:**
  - `FSDataOutputStream create(...)`
    - Extends `java.io.DataOutputStream`
    - Provides methods for writing primitives, raw bytes etc
  - `FSDataInputStream open(...)`
    - Extends `java.io.DataInputStream`
    - Provides methods for reading primitives, raw bytes etc
  - `boolean delete(...)`
  - `boolean mkdirs(...)`
  - `void copyFromLocalFile(...)`
  - `void copyToLocalFile(...)`
  - `FileStatus[] listStatus(...)`

The FileSystem API: Directory Listing

- **Get a directory listing:**

```
Path p = new Path("/my/path");

Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
FileStatus[] fileStats = fs.listStatus(p);

for (int i = 0; i < fileStats.length; i++) {
    Path f = fileStats[i].getPath();

    // do something interesting
}
```

The FileSystem API: Writing Data

- **Write data to a file**

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);

Path p = new Path("/my/path/foo");

FSDataOutputStream out = fs.create(p, false);

// write some raw bytes
out.write(getBytes());

// write an int
out.writeInt(getInt());

...

out.close();
```

Chapter Topics

Delving Deeper into the Hadoop API

- Using the ToolRunner Class
- Setting Up and Tearing Down Mappers and Reducers
- Decreasing the Amount of Intermediate Data with Combiners
- Accessing HDFS Programmatically
- **Using the Distributed Cache**
- Using the Hadoop API's Library of Mappers, Reducers and Partitioners

The Distributed Cache: Motivation

- **A common requirement is for a Mapper or Reducer to need access to some 'side data'**
  - Lookup tables
  - Dictionaries
  - Standard configuration values
- **One option: read directly from HDFS in the `setup` method**
  - Using the API seen in the previous section
  - Works, but is not scalable
- **The Distributed Cache provides an API to push data to all slave nodes**
  - Transfer happens behind the scenes before any task is executed
  - Data is only transferred once to each node
  - Note: Distributed Cache is read-only
  - Files in the Distributed Cache are automatically deleted from slave nodes when the job finishes

Using the Distributed Cache: The Difficult Way

- **Place the files into HDFS**
- **Configure the Distributed Cache in your driver code**

```
Configuration conf = new Configuration();
DistributedCache.addCacheFile(new URI("/myapp/lookup.dat"),conf);
DistributedCache.addFileToClassPath(new Path("/myapp/mylib.jar"),conf);
DistributedCache.addCacheArchive(new URI("/myapp/map.zip",conf));
DistributedCache.addCacheArchive(new URI("/myapp/mytar.tar",conf));
DistributedCache.addCacheArchive(new URI("/myapp/mytgz.tgz",conf));
DistributedCache.addCacheArchive(new URI("/myapp/mytargz.tar.gz",conf));
```

  - `.jar` files added with `addFileToClassPath` will be added to your Mapper or Reducer's classpath
  - Files added with `addCacheArchive` will automatically be dearchived/decompressed

Using the DistributedCache: The Easy Way

- **If you are using ToolRunner, you can add files to the Distributed Cache directly from the command line when you run the job**
  - No need to copy the files to HDFS first
- **Use the `-files` option to add files**

```
hadoop jar myjar.jar MyDriver -files file1, file2, file3, ...
```

- **The `-archives` flag adds archived files, and automatically unarchives them on the destination machines**
- **The `-libjars` flag adds jar files to the classpath**

Accessing Files in the Distributed Cache

- **Files added to the Distributed Cache are made available in your task's local working directory**
  - Access them from your Mapper or Reducer the way you would read any ordinary local file

```
File f = new File("file_name_here");
```

Chapter Topics

Delving Deeper into the Hadoop API

- Using the ToolRunner Class
- Setting Up and Tearing Down Mappers and Reducers
- Decreasing the Amount of Intermediate Data with Combiners
- Accessing HDFS Programmatically
- Using the Distributed Cache
- **Using the Hadoop API's Library of Mappers, Reducers and Partitioners**

Reusable Classes for the New API

- **The `org.apache.hadoop.mapreduce.lib.*/*` packages contain a library of Mappers, Reducers, and Partitioners supporting the new API**
- **Example classes:**
  - `InverseMapper` – Swaps keys and values
  - `RegexMapper` – Extracts text based on a regular expression
  - `IntSumReducer`, `LongSumReducer` – Add up all values for a key
  - `TotalOrderPartitioner` – Reads a previously-created partition file and partitions based on the data from that file
    - Sample the data first to create the partition file
    - Allows you to partition your data into *n* partitions without hard-coding the partitioning information
- **Refer to the Javadoc for classes available in your version of CDH**
  - Available classes vary greatly from version to version

Key Points

- **Use the ToolRunner class to build drivers**
  - Parses job options and configuration variables automatically
- **Override Mapper and Reducer `setup` and `cleanup` methods**
  - Set up and tear down, e.g. reading configuration parameters
- **Combiners are 'mini-reducers'**
  - Run locally on Mapper output to reduce data sent to Reducers
- **The FileSystem API lets you read and write HDFS files programmatically**
- **The Distributed Cache lets you copy local files to worker nodes**
  - Mappers and Reducers can access directly as regular files
- **Hadoop includes a library of predefined Mappers, Reducers, and Partitioners**

Bibliography

The following offer more information on topics discussed in this chapter

- **Combiners are discussed in TDG 3e on pages 33-36.**
- **A table describing available filesystems in Hadoop is on pages 52-53 of TDG 3e.**
- **The HDFS API is described in TDG 3e on pages 55-67.**
- **Distributed cache: See pages 289-295 of TDG 3e for more details.**

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("mapreduce.framework.name", "local");
conf.set("fs.defaultFS", "file:///");
```

  - Or set these options on the command line if your driver uses ToolRunner
    - **`-fs`** is equivalent to **`-D fs.defaultFS`**
    - **`-jt`** is equivalent to **`-D mapreduce.framework.name`**
    - e.g.

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

-fs seems flakey in YARN, so best to prepend paths with file://

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

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`

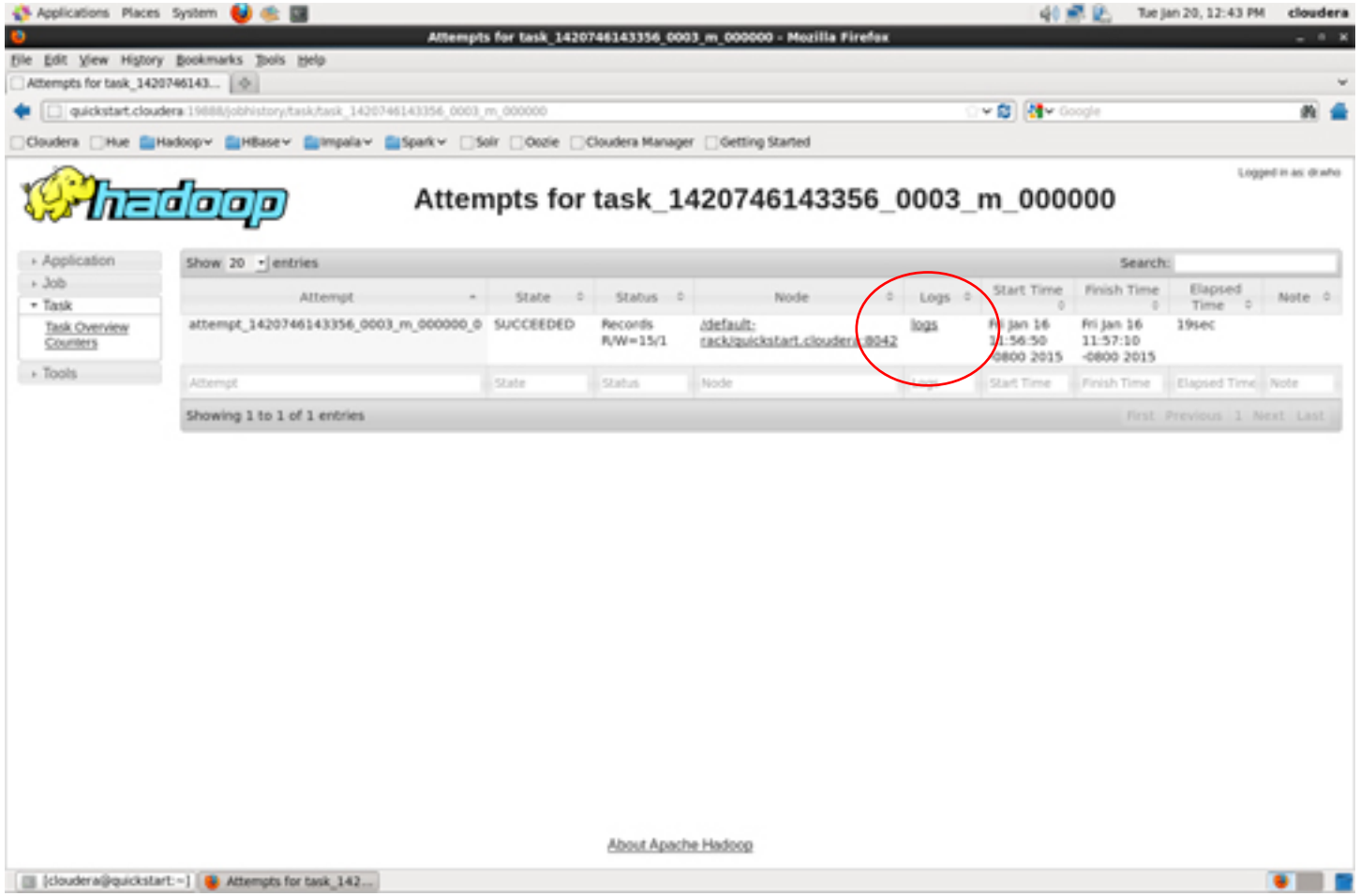
- **Tried-and-true debugging technique: write to `stdout` or `stderr`**
- **If running in LocalJobRunner mode, you will see the results of `System.err.println()`**
- **If running on a cluster, that output will not appear on your console**
  - Output is visible via Hadoop's Web UI

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 wŜă2dzNJÖŜ a l-yl-ƎŜNJ Web UI**
  - `http://<resource_manager_address>:8088/`
  - `http://localhost:8088/` 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 ResourceManager 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 ResourceManager Web UI
    - Automatically retrieves and displays the log files for you

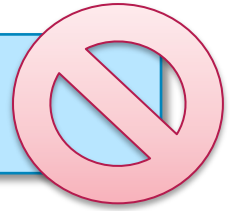
Restricting Log Output

- **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/