Hadoop IN PRACTICE

Alex Holmes

INCLUDES 85 TECHNIQUES

SAMPLE CHAPTER

MANNING

Hadoop in Practice

by Alex Holmes

Chapter 13

brief contents

13 Testing and debugging

In this chapter

- Using design and testing techniques to write bulletproof MapReduce code
- Learning to debug issues in production and isolate problem inputs
- Avoiding MapReduce anti-patterns

When you're running MapReduce in production you can guarantee that some day you'll receive a call about a failing job. The goal of this chapter is to help you put in as many measures as possible to avoid the chance of this happening. We'll look at how to provide adequate unit testing for MapReduce code and examine some defensive coding techniques to minimize badly behaving code.

All the preparation and testing in the world doesn't guarantee you won't encounter any problems, and in the event that you do, we'll look at how to debug your job to figure out what went wrong. In this chapter we'll focus on testing and debugging user space MapReduce.

13.1 Testing

In this section we'll look at the best methods to test your MapReduce code, and also look at some design aspects to consider when writing MapReduce to help in your testing efforts.

13.1.1 Essential ingredients for effective unit testing

It's important to make sure unit tests are easy to write, and to ensure that they cover a good spectrum of positive and negative scenarios. Let's take a look at the impact that test-driven development, code design, and data have on writing effective unit tests.

TEST-DRIVEN DEVELOPMENT

When it comes to writing Java code, I'm a big proponent of test-driven development (TDD),[1] and with MapReduce things are no different. Test-driven development emphasizes writing unit tests ahead of writing the code, and recently has gained in importance as quick development turnaround times become the norm rather than the exception. Applying test-driven development to MapReduce code is crucial, particularly when such code is part of a critical production application.

Writing unit tests prior to writing your code forces your code to be structured in a way that easily facilitates testing.

CODE DESIGN

When you write code, it's important to think about the best way to structure it so you can easily test it. Leveraging concepts such as abstraction and dependency injection[2] will go a long way to reaching this goal.

When you write MapReduce code, it's a good idea to abstract away the code doing the work, which means you can test that code in regular unit tests without having to think about how to work with Hadoop-specific constructs. This is true not only for your map and reduce functions, but also for your `InputFormats`, `OutputFormats`, data serialization, and partitioner code.

Let's look at a simple example to better illustrate this point. The following code shows a reducer that calculates the mean for a stock:

```
public static class Reduce
    extends Reducer<Text, DoubleWritable, Text, DoubleWritable> {

  DoubleWritable outValue = new DoubleWritable();
  public void reduce(Text stockSymbol, Iterable<DoubleWritable> values,
                     Context context)
      throws IOException, InterruptedException {

    double total = 0;
    int instances = 0;
    for (DoubleWritable stockPrice : values) {
      total += stockPrice.get();
      instances++;
    }
    outValue.set(total / (double) instances);
    context.write(stockSymbol, outValue);
  }
}
```

This is a trivial example, but the way the code is structured means you can't easily test this in a regular unit test because MapReduce has constructs such as `Text`, `DoubleWritable`,

[1] See http://en.wikipedia.org/wiki/Test-driven_development.
[2] See http://en.wikipedia.org/wiki/Dependency_injection.

and the Context class that get in your way. If you were to structure the code to abstract away the work, you could easily test the user space code that's doing your work, as the following code shows:

```
public static class Reduce2
    extends Reducer<Text, DoubleWritable, Text, DoubleWritable> {

  SMA sma = new SMA();
  DoubleWritable outValue = new DoubleWritable();
  public void reduce(Text key, Iterable<DoubleWritable> values,
                     Context context)
      throws IOException, InterruptedException {
    sma.reset();
    for (DoubleWritable stockPrice : values) {
      sma.add(stockPrice.get());
    }
    outValue.set(sma.calculate());
    context.write(key, outValue);
  }
}

public static class SMA {
  protected double total = 0;
  protected int instances = 0;

  public void add(double value) {
    total += value;
    instances ++;
  }

  public double calculate() {
    return total / (double) instances;
  }

  public void reset() {
    total = 0;
    instances = 0;
  }
}
```

With this improved code layout you can now easily test the SMA class that's adding and calculating the simple moving average, without the Hadoop code getting in your way.

IT'S THE DATA, STUPID

When you write unit tests, you try to discover how your code handles both positive and negative input data. In both cases it's best if the data you're testing with is a representative sample from production.

Often, no matter how hard you try, issues in your code in production will arise from unexpected input data. Later, in section 13.2.2, we'll look at how to identify when this occurs in production jobs. It's important that when you do discover input data that causes a job to blow up, you not only fix the code to handle the unexpected data, but you also pull the data that caused the blowup and use it in a unit test to prove that the code can now correctly handle that data.

13.1.2 MRUnit

MRUnit is a test framework you can use to unit test MapReduce code. It was developed by Cloudera (a vendor with its own Hadoop distribution) and is currently an Apache project in incubator status. It should be noted that MRUnit supports both the old (`org.apache.hadoop.mapred`) and new (`org.apache.hadoop.mapreduce`) MapReduce APIs.

TECHNIQUE 79 Unit Testing MapReduce functions, jobs, and pipelines

In this technique we'll look at writing unit tests that leverage each of the four types of tests provided by MRUnit:

1. A map test that only tests a map function (supported by the `MapDriver` class).
2. A reduce test that only tests a reduce function (supported by the `ReduceDriver` class).
3. A map and reduce test that tests both the map and reduce functions (supported by the `MapReduceDriver` class).
4. A pipeline test that allows a series of MapReduce functions to be exercised (supported by the `TestPipelineMapReduceDriver` class).

Problem

You want to test map and reduce functions, as well as MapReduce pipelines.

Solution

Learn how MRUnit's `MapDriver`, `ReduceDriver`, `MapReduceDriver` and `PipelineMapReduce-Driver` classes can be used as part of your unit tests to test your MapReduce code.

Discussion

MRUnit has four types of unit tests—we'll start with a look at the map tests.

MAP TESTS

Let's kick things off by writing a test to exercise a map function. Before starting, let's look at what you need to supply to MRUnit to execute the test, and in the process learn about how MRUnit works behind the scenes.

Figure 13.1 shows the interactions of the unit test with MRUnit, and how, in turn, it interacts with the mapper you're testing.

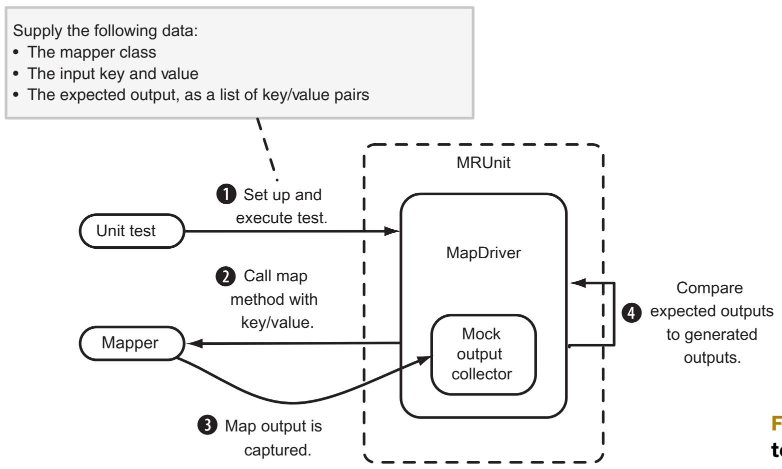

Supply the following data:
• The mapper class
• The input key and value
• The expected output, as a list of key/value pairs
MRUnit
1 Set up and execute test.
Unit test
MapDriver
2 Call map method with key/value.
Mapper
Mock output collector
4 Compare expected outputs to generated outputs.
3 Map output is captured.

Figure 13.1 MRUnit test using `MapDriver`

The following[3] code is a simple unit test of the (identity) mapper class in Hadoop:

```
public class IdentityMapTest extends TestCase {

  private Mapper<Text, Text, Text, Text> mapper;
  private MapDriver<Text, Text, Text, Text> driver;

  @Before
  public void setUp() {
    mapper = new Mapper<Text, Text, Text, Text>();
    driver = new MapDriver<Text, Text, Text, Text>(mapper);

  }




        .runTest();

  }
}
```

Create the map object you're testing. Here you're using Hadoop's built-in IdentityMapper, which outputs the input data without any transformations.

The MRUnit driver class you'll use in your test. This is the MapDriver, and as such you need to specify the key/value input and output types for the mapper you're testing in this class.

The withInput method is used to specify an input key/value, which will be fed to the IdentityMapper.

Run the test. If a failure's encountered, it logs the discrepancy, and throws an exception.

The withOutput method is used to specify the output key/value, which MRUnit will compare against the output generated by the mapper being tested.

MULTIPLE INPUT SUPPORT

Be aware that MRUnit doesn't support multiple input records. If you call the `withInput` method more than once, it will overwrite the key and value from the previous call to `withInput`.

MRUnit is not tied to any specific unit testing framework, so if it finds an error it logs the error and throws an exception. Let's see what would happen if your unit test had specified output that didn't match the output of the mapper, as in the following code:

```
driver.withInput(new Text("foo"), new Text("bar"))
      .withOutput(new Text("foo"), new Text("bar2"))
      .runTest();
```

If you run this test, your test will fail, and you'll see the following log output:

```
ERROR Received unexpected output (foo, bar)
ERROR Missing expected output (foo, bar2) at position 0
```

One of the powerful features of JUnit and other test frameworks is that when tests fail, the failure message includes details on the cause of the failure. Unfortunately, MRUnit

[3] **GitHub source**—https://github.com/alexholmes/hadoop-book/blob/master/src/test/java/com/manning/hip/ch13/mrunit/IdentityMapTest.java

LOGGING CONFIGURATION

Because MRUnit uses the Apache Commons logging, which defaults to using log4j, you'll need to have a log4j.properties file in the classpath that's configured to write to standard out, similar to the following:

```
log4j.rootLogger=WARN, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=
➥ %-5p [%t][%d{ISO8601}] [%C.%M] - %m%n
```

logs and throws a nondescriptive exception, which means you need to dig through the test output to determine what failed.

What if you wanted to leverage the power of MRUnit, and also leverage the informative errors that JUnit provides when assertions fail? You could modify your code[4] to do that, and bypass MRUnit's testing code:

```
@Test
public void testIdentityMapper() throws IOException {
  List<Pair<Text, Text>> results = driver
      .withInput(new Text("foo"), new Text("bar"))
      .run();

  assertEquals(1, results.size());
  assertEquals(new Text("foo"), results.get(0).getFirst());
  assertEquals(new Text("bar"), results.get(0).getSecond());
}
```

The run method executes the map function and returns a list of all the output records emitted by the function. Also note that there was no need to call the withOutput method, because you'll do the validation yourself.

You assert the size and contents of the records.

With this approach, if there's a mismatch between the expected and actual outputs, you get a more meaningful error message, which report-generation tools can use to easily describe what failed in the test:

```
junit.framework.AssertionFailedError: expected:<bar2> but was:<bar>
```

To cut down on the inevitable copy-paste activities with this approach, I wrote a simple helper class[5] to use JUnit asserts in combination with using the MRUnit driver. Your JUnit test now looks like this:

```
@Test
public void testIdentityMapper() throws IOException {
  List<Pair<Text, Text>> results = driver
      .withInput(new Text("foo"), new Text("bar"))
      .withOutput(new Text("foo"), new Text("bar"))
      .run();

  MRUnitJUnitAsserts.assertOutputs(driver, results);
}
```

You're calling withOutput because the helper function can extract the outputs directly from the driver.

Call the helper function that uses JUnit asserts to test the contents of the expected output with the generated output.

4 **GitHub source**—https://github.com/alexholmes/hadoop-book/blob/master/src/test/java/com/manning/hip/ch13/mrunit/IdentityMapJUnitTest.java

5 **GitHub source**—https://github.com/alexholmes/hadoop-book/blob/master/src/test/java/com/manning/hip/ch13/mrunit/IdentityMapJUnitAssertsTest.java

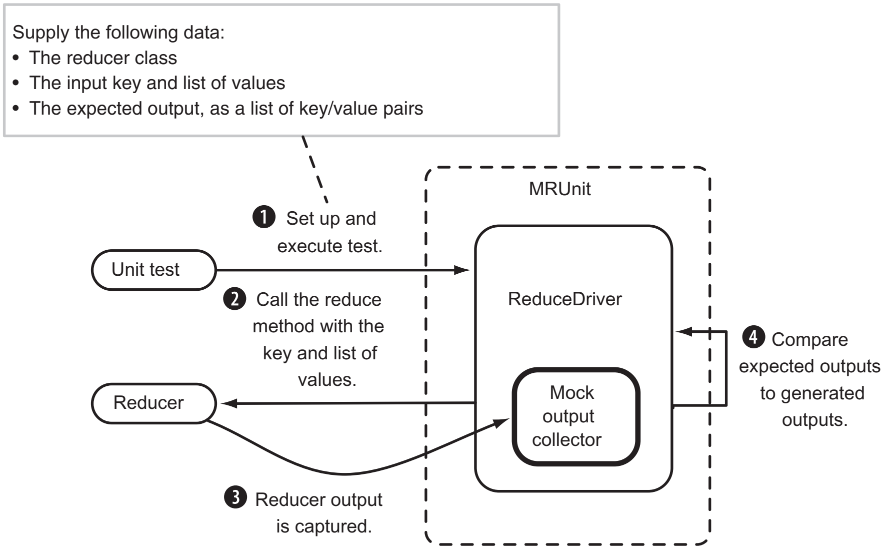

Supply the following data:
• The reducer class
• The input key and list of values
• The expected output, as a list of key/value pairs
MRUnit
1 Set up and execute test.
Unit test
ReduceDriver
2 Call the reduce method with the key and list of values.
4 Compare expected outputs to generated outputs.
Reducer
Mock output collector
3 Reducer output is captured.

Figure 13.2 **MRUnit test using `ReduceDriver`**

This is much cleaner and removes any mistakes that arise from the copy-paste anti-pattern.

REDUCE TESTS

Now that we've looked at map function tests, let's look at reduce function tests. The MRUnit framework takes a similar approach for reduce testing. Figure 13.2 shows the interactions of your unit test with MRUnit, and how it in turn interacts with the reducer you're testing.

The following code[6] is a simple unit test for testing the (identity) reducer class in Hadoop:

```
public class IdentityReduceTest extends TestCase {

  private Reducer<Text, Text, Text, Text> reducer;
  private ReduceDriver<Text, Text, Text, Text> driver;

  @Before
  public void setUp() {
    reducer = new Reducer<Text, Text, Text, Text>();
    driver = new ReduceDriver<Text, Text, Text, Text>(reducer);
  }

  @Test
  public void testIdentityMapper() throws IOException {
    List<Pair<Text, Text>> results = driver
        .withInput(new Text("foo"),
            Arrays.asList(new Text("bar1"), new Text("bar2")))
        .withOutput(new Text("foo"), new Text("bar1"))
```

With the identity reducer you specified two value inputs so you expect two outputs.

When testing the reducer you specify a list of values that MRUnit sends to your reducer.

6 **GitHub source**—https://github.com/alexholmes/hadoop-book/blob/master/src/test/java/com/manning/hip/ch13/mrunit/IdentityMapJUnitAssertsTest.java

```
            .withOutput(new Text("foo"), new Text("bar2"))
        .run();

    MRUnitJUnitAsserts.assertOutputs(driver, results);
  }
}
```

Add the expected output for the second value.

Use the helper class you wrote earlier in the map section.

Now that we've completed our look at the individual map and reduce function tests, let's look at how to test a map and reduce function together.

MAPREDUCE TESTS

MRUnit also supports testing the map and reduce functions in the same test. You feed MRUnit the inputs, which in turn are supplied to the mapper. You also tell MRUnit what reducer outputs you expect.

Figure 13.3 shows the interactions of your unit test with MRUnit, and how, in turn, it interacts with the mapper and reducer you're testing.

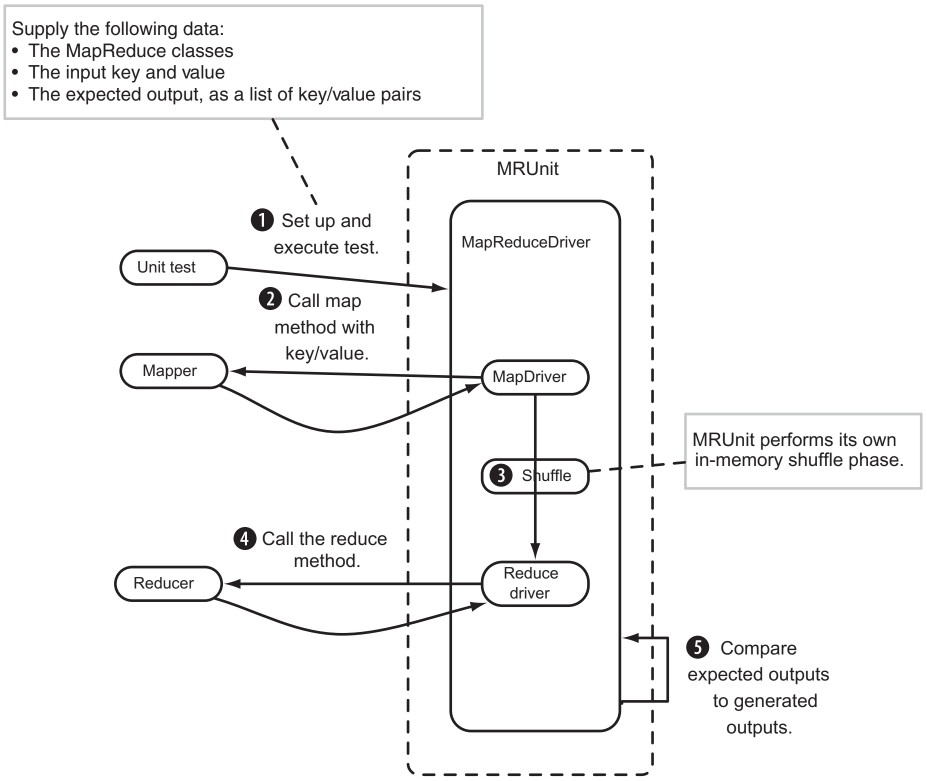

Supply the following data:
• The MapReduce classes
• The input key and value
• The expected output, as a list of key/value pairs
MRUnit
1 Set up and execute test.
Unit test
MapReduceDriver
2 Call map method with key/value.
Mapper
MapDriver
MRUnit performs its own in-memory shuffle phase.
3 Shuffle
4 Call the reduce method.
Reducer
Reduce driver
5 Compare expected outputs to generated outputs.

Figure 13.3 MRUnit test using `MapReduceDriver`

The following code[7] is a simple unit test for testing the (identity) mapper and reducer classes in Hadoop:

```
public class IdentityMapReduceTest extends TestCase {

  private Reducer<Text, Text, Text, Text> reducer;
  private Mapper<Text, Text, Text, Text> mapper;
  private MapReduceDriver<Text, Text, Text, Text, Text, Text> driver;

  @Before
  public void setUp() {
    mapper = new Mapper<Text, Text, Text, Text>();
    reducer = new Reducer<Text, Text, Text, Text>();
    driver =
      new MapReduceDriver<Text, Text, Text, Text, Text, Text>(
      mapper, reducer);
  }

  @Test
  public void testIdentityMapper() throws IOException {
    List<Pair<Text, Text>> results = driver
        .withInput(new Text("foo"), new Text("bar"))
        .withInput(new Text("foo2"), new Text("bar2"))
        .withOutput(new Text("foo"), new Text("bar"))
        .withOutput(new Text("foo2"), new Text("bar2"))
        .run();

    MRUnitJUnitAsserts.assertOutputs(driver, results);
  }
}
```

With the MapReduce driver you need to specify six types, the map input and output key/value types, as well as the reducer key/value output types.

Supply the map inputs. In contrast to the MapDriver and ReduceDriver, the MapReduceDriver supports multiple inputs.

Set the expected reducer outputs.

Now we'll look at our fourth and final type of test that MRUnit supports, pipeline tests, which are used to test multiple MapReduce jobs.

PIPELINE TESTS

MRUnit supports testing a series of map and reduce functions—these are called *pipeline tests*. You feed MRUnit one or more MapReduce functions, the inputs to the first map function, and the expected outputs of the last reduce function. Figure 13.4 shows the interactions of your unit test with MRUnit pipeline driver.

The following code[8] is a unit test for testing a pipeline containing two sets of (identity) mapper and reducer classes in Hadoop:

```
public class PipelineTest extends TestCase {

  private Mapper<Text, Text, Text, Text> mapper1;
  private Reducer<Text, Text, Text, Text> reducer1;
  private Mapper<Text, Text, Text, Text> mapper2;
  private Reducer<Text, Text, Text, Text> reducer2;
```

[7] **GitHub source**—https://github.com/alexholmes/hadoop-book/blob/master/src/test/java/com/manning/hip/ch13/mrunit/IdentityMapReduceTest.java

[8] **GitHub source**—https://github.com/alexholmes/hadoop-book/blob/master/src/test/java/com/manning/hip/ch13/mrunit/PipelineTest.java

```
  private PipelineMapReduceDriver<Text, Text, Text, Text> driver;

  @Before
  public void setUp() {
    mapper1 = new IdentityMapper<Text, Text>();
    reducer1 = new IdentityReducer<Text, Text>();
    mapper2 = new IdentityMapper<Text, Text>();
    reducer2 = new IdentityReducer<Text, Text>();
    driver = new PipelineMapReduceDriver<Text, Text, Text, Text>();
    driver.addMapReduce(
      new Pair<Mapper, Reducer>(mapper1, reducer1));
    driver.addMapReduce(
      new Pair<Mapper, Reducer>(mapper2, reducer2));
  }

  @Test
  public void testIdentityMapper() throws IOException {
    List<Pair<Text, Text>> results = driver
        .withInput(new Text("foo"), new Text("bar"))
        .withInput(new Text("foo2"), new Text("bar2"))
        .withOutput(new Text("foo"), new Text("bar"))
        .withOutput(new Text("foo2"), new Text("bar2"))
        .run();

    MRUnitJUnitAsserts.assertOutputs(driver, results);
  }
}
```

Add the first map and reduce pair to the pipeline.

Add the second map and reduce pair to the pipeline.

As with the MapReduceDriver, the PipelineMapReduceDriver supports multiple input records.

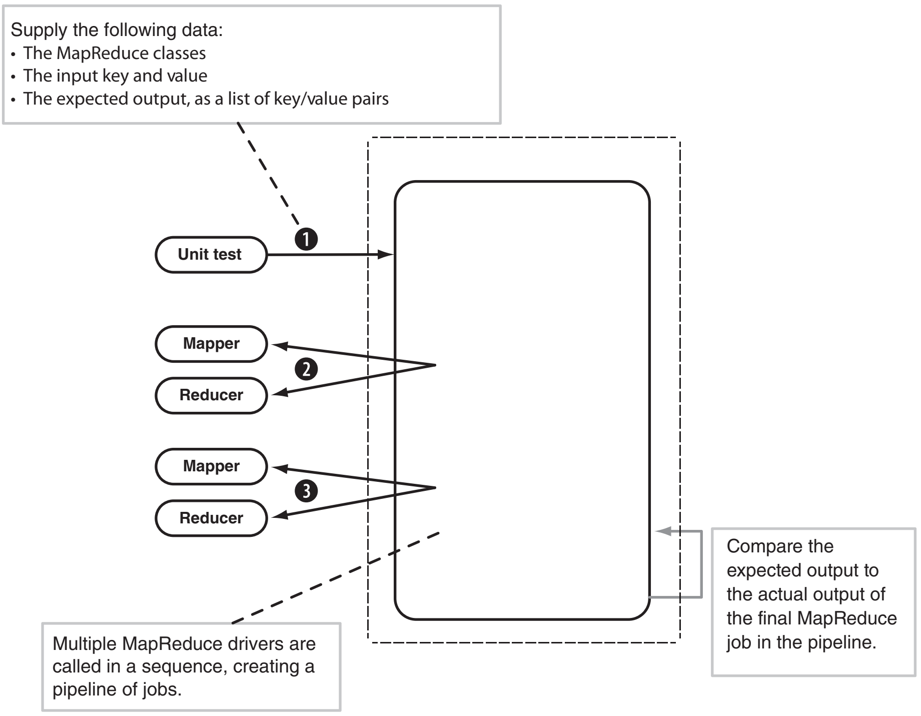

Supply the following data:
• The MapReduce classes
• The input key and value
• The expected output, as a list of key/value pairs
Unit test
1
Mapper
2
Reducer
Mapper
3
Reducer
Compare the expected output to the actual output of the final MapReduce job in the pipeline.
Multiple MapReduce drivers are called in a sequence, creating a pipeline of jobs.

Figure 13.4 **MRUnit test using `PipelineMapReduceDriver`**

Note that the PipelineMapReduceDriver is the only driver in MRUnit that doesn't come in both old and new MapReduce API versions, which is why the previous code uses the old MapReduce API.

Summary

What type of test should you use for your code? Take a look at table 13.1 for some pointers. MRUnit has a few limitations, some of which we touched upon in this technique:

- The MapDriver and ReduceDriver support only a single key as input, which can make it more cumbersome to test map and reduce logic that requires multiple keys, such as those that cache the input data.
- MRUnit isn't integrated with unit test frameworks that provide rich error-reporting capabilities for quicker determination of errors.
- The pipeline tests only work with the old MapReduce API, so MapReduce code that uses the new MapReduce API can't be tested with the pipeline tests.
- There's no support for testing data serialization, or InputFormat, RecordReader, OutputFormat, or RecordWriter classes.

Table 13.1 MRUnit tests and when to use them

Type of test	Works well in these situations
Map	You have a map-only job, and you want low-level unit tests where the framework takes care of testing the expected map outputs for your test map inputs.
Reduce	Your job has a lot of complexity in the reduce function and you want to isolate your tests to only that function.
MapReduce	You want to test the combination of the map and reduce functions. These are higher-level unit tests.
Pipeline	You have a MapReduce pipeline where the input of each MapReduce job is the output from the previous job.

Notwithstanding these limitations, MRUnit is an excellent test framework to help you test at the granular level of individual map and reduce functions; MRUnit also can test a pipeline of MapReduce jobs. And because it skips the InputFormat and OutputFormat steps, your unit tests will execute quickly.

Next we'll look at how you can use the LocalJobRunner to test some MapReduce constructs that are ignored by MRUnit.

13.1.3 LocalJobRunner

In the last section we looked at MRUnit, a great lightweight unit test library. But what if you want to test not only your map and reduce functions, but also the InputFormat, RecordReader, OutputFormat, and RecordWriter code as well as the data serialization between the map and reduce phases? This becomes important if you've written your own input/output format classes because you want to make sure you're testing that code, too.

Hadoop comes bundled with the `LocalJobRunner` class, which Hadoop and related projects (such as Pig and Avro) use to write and test their MapReduce code. `LocalJobRunner` allows you to test all the aspects of a MapReduce job, including the reading and writing of data to and from the filesystem.

TECHNIQUE 80 Heavyweight job testing with the LocalJobRunner

Tools like MRUnit are useful for low-level unit tests, but how can you be sure that your code will play nicely with the whole Hadoop stack?

Problem

You want to test the whole Hadoop stack in your unit test.

Solution

Leverage the `LocalJobRunner` class in Hadoop to expand the coverage of your tests to include code related to processing job inputs and outputs.

Discussion

Using the `LocalJobRunner` makes your unit tests start to feel more like integration tests, because what you're doing is testing how your code works in combination with the whole MapReduce stack. This is great because you can use this to test not only how your user space MapReduce code plays with MapReduce, but also to test `InputFormats`, `OutputFormats`, partitioners, and advanced sort mechanisms. The code in the next listing[9] shows an example of how you can leverage the `LocalJobRunner` in your unit tests.

Using `LocalJobRunner` to test a MapReduce job

```
public class IdentityTest {

  @Test
  public void run() throws Exception {
    Path inputPath = new Path("/tmp/mrtest/input");
    Path outputPath = new Path("/tmp/mrtest/output");

    Configuration conf = new Configuration();
    conf.set("mapred.job.tracker", "local");

    conf.set("fs.default.name", "file:///");

    FileSystem fs = FileSystem.get(conf);
    if (fs.exists(outputPath)) {
      fs.delete(outputPath, true);
    }
    if (fs.exists(inputPath)) {
      fs.delete(inputPath, true);
    }
    fs.mkdirs(inputPath);

    String input = "foo\tbar";
    DataOutputStream file = fs.create(new Path(inputPath, "part-" + 0));
```

You force use of the LocalJobRunner by setting mapred.job.tracker to local (which is the default).

Force the filesystem to be local (which is the default).

Retrieve the filesystem. By default this will be the local filesystem. The next few lines of code delete the output and input directories to remove any lingering data from other tests.

Write the job inputs into a file.

[9] **GitHub source**—https://github.com/alexholmes/hadoop-book/blob/master/src/test/java/com/manning/hip/ch13/localjobrunner/IdentityTest.java

```
      file.writeBytes(input);
      file.close();

      Job job = runJob(conf, inputPath, outputPath);

      assertTrue(job.isSuccessful());

      List<String> lines =
         IOUtils.readLines(fs.open(new Path(outputPath, "part-r-00000")));

      assertEquals(1, lines.size());
      String[] parts = StringUtils.split(lines.get(0), "\t");
      assertEquals("foo", parts[0]);
      assertEquals("bar", parts[1]);
    }

    public Job runJob(Configuration conf, Path inputPath, Path outputPath)
        throws ClassNotFoundException, IOException, InterruptedException {
      Job job = new Job(conf);
      job.setInputFormatClass(KeyValueTextInputFormat.class);
      job.setMapOutputKeyClass(Text.class);
      FileInputFormat.setInputPaths(job, inputPath);
      FileOutputFormat.setOutputPath(job, outputPath);
      job.waitForCompletion(false);
      return job;
    }
  }
```

Run an identity MapReduce job.

Assert that the job completed successfully.

Read the job output from the filesystem.

Verify the job output.

Writing this test is more involved because you need to handle writing the inputs to the filesystem, and also reading them back out. That's a lot of boilerplate code to have to deal with for every test, and probably something that you want to factor out into a reusable helper class.

Here's an example of a utility class to do that; the following code[10] shows how `IdentityTest` code can be condensed into a more manageable size:

```
  @Test
  public void run() throws Exception {

    TextIOJobBuilder builder = new TextIOJobBuilder()
        .addInput("foo", "bar")
        .addExpectedOutput("foo", "bar")
        .writeInputs();

    Job job = runJob(
        builder.getConfig(),
        builder.getInputPath(),
        builder.getOutputPath());

    assertTrue(job.isSuccessful());

    builder.verifyResults();
  }
```

Set the job inputs.

Set the expected job outputs.

Write the inputs to the filesystem.

Delegate testing the expected results with the results to the utility class.

[10] **GitHub source**—https://github.com/alexholmes/hadoop-book/blob/master/src/test/java/com/manning/hip/ch13/localjobrunner/IdentityWithBuilderTest.java

Summary

So what are some of the limitations to be aware of when using `LocalJobRunner`?

- `LocalJobRunner` runs only a single reduce task, so you can't use it to test partitioners.
- As you saw, it's also more labor intensive; you need to read and write the input and output data to the filesystem.
- Jobs are also slow because much of the MapReduce stack is being exercised.
- Finally, it'll be tricky to use this approach to test `InputFormats` and `OutputFormats` that aren't file-based.

Despite these limitations, `LocalJobRunner` is the most comprehensive way to test your MapReduce code, and as such will provide the highest level of assurance that your jobs will run the way you expect them to in Hadoop clusters.

13.1.4 Integration and QA testing

Using the TDD approach, you wrote some unit tests using the techniques in this section. You next wrote the MapReduce code and got it to the point where the unit tests were passing. Hurray. Before you break out the champagne, you still want assurances that the MapReduce code is working prior to running it in production. The last thing you want is your code to fail in production and have to debug it over there. But why, you ask, would my job fail if all of my unit tests pass? Good question, and it could be due to a variety of factors:

- The data you used for your unit tests doesn't contain all of the data aberrations and variances of the data used in production.
- The volume and/or data skew issues cause side effects in your code.
- Differences in Hadoop and other libraries result in behaviors different from those in your build environment.
- Hadoop and operating system configuration differences between your build host and production cause problems.

Because of these factors, when you build integration or QA test environments it's crucial to ensure that the Hadoop version and configurations mirror those of the production cluster. Different versions of Hadoop will behave differently, as will the same version of Hadoop configured in different ways. When you're testing changes in test environments, you want to ensure a smooth transition to production, so do as much as you can to make sure that version and configuration are as close as possible to production.

After your MapReduce jobs are successfully running in integration and QA, you can push them into production, knowing there's a much higher probability that your jobs will work as expected.

This wraps up our look at testing MapReduce code. We looked at some TDD and design principles to help write and test your Java code, and also covered some unit test libraries that make it easier to unit test MapReduce code. Next we'll move into the complex world of debugging problems in MapReduce jobs.

13.2 Debugging user space problems

In this section we'll walk through the steps to isolate and fix problems in your MapReduce user space code. What's meant by *user space code* is code that developers write.

Your MapReduce jobs can fail due to a number of problems, including the following:

- You can run out of memory because you're caching too much data.
- Your logic, which parses input records, may not handle all inputs correctly.
- A logic error exists in your code or in a third-party library.
- Your custom `RecordReader` or `RecordWriter` code may have a serialization or deserialization bug.
- Your custom partitioner isn't partitioning records correctly.
- Your custom comparator for primary or secondary sort isn't working as expected.

The list goes on and on. You'll need to take a structured approach to debugging a problem MapReduce job. Figure 13.5 shows a decision tree you can use to narrow down a problem in your MapReduce code.

In the remainder of this section, we'll address the three areas highlighted in figure 13.5 to help with your debugging efforts:

- Examining task logs for details on what's causing the problems
- Finding the inputs that are breaking your code
- Looking at logging and coding guidelines to help you effectively debug your code

We'll kick things off with a look at the task logs.

13.2.1 Accessing task log output

Accessing your task logs is the first step to figuring out what issues you're having with your MapReduce job. Depending on the exact issue, the logs in their current form may or may not help you. For example, if there's a subtle serialization bug in your code, unless the steps in section 13.2.4 were followed, there's a good chance the logs won't be much help in pinpointing serialization as the problem.

TECHNIQUE 81 Examining task logs

In this technique we'll look at ways to access task logs in the event that you have a problem MapReduce job you want to debug.

Problem

Your MapReduce job is failing, or generating unexpected outputs, and you want to determine if the logs can help you figure out the problem.

Solution

Learn how to use the JobTracker UI to view task logs. You will also look at how you can SSH to individual TaskTracker nodes and examine the logs directly.

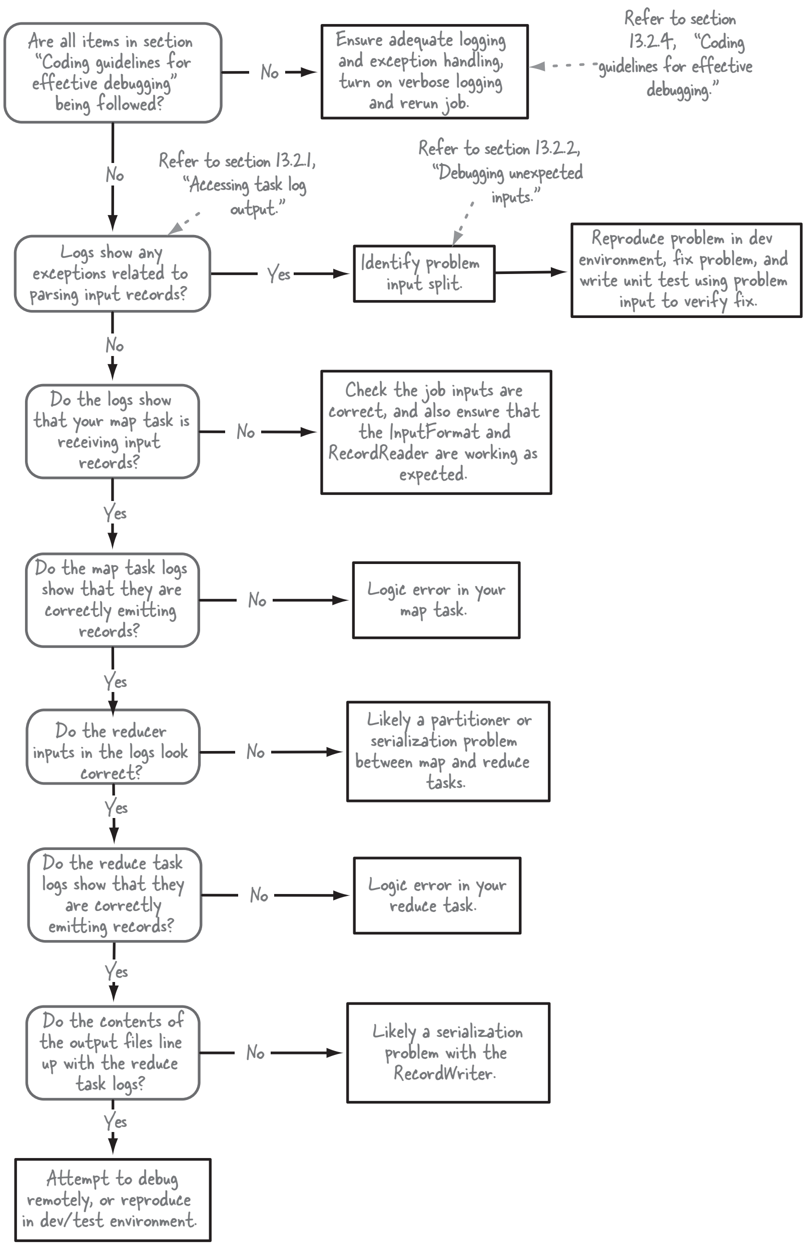
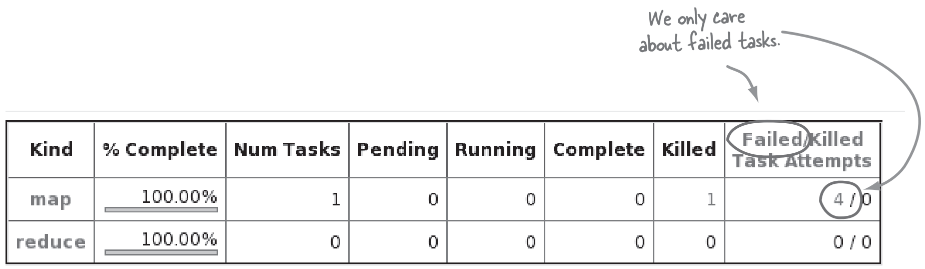

Are all items in section "Coding guidelines for effective debugging" being followed?
No
Ensure adequate logging and exception handling, turn on verbose logging and rerun job.
Refer to section 13.2.4, "Coding guidelines for effective debugging."
No
Refer to section 13.2.1, "Accessing task log output."
Refer to section 13.2.2, "Debugging unexpected inputs."
Logs show any exceptions related to parsing input records?
Yes
Identify problem input split.
Reproduce problem in dev environment, fix problem, and write unit test using problem input to verify fix.
No
Do the logs show that your map task is receiving input records?
No
Check the job inputs are correct, and also ensure that the InputFormat and RecordReader are working as expected.
Yes
Do the map task logs show that they are correctly emitting records?
No
Logic error in your map task.
Yes
Do the reducer inputs in the logs look correct?
No
Likely a partitioner or serialization problem between map and reduce tasks.
Yes
Do the reduce task logs show that they are correctly emitting records?
No
Logic error in your reduce task.
Yes
Do the contents of the output files line up with the reduce task logs?
No
Likely a serialization problem with the RecordWriter.
Yes
Attempt to debug remotely, or reproduce in dev/test environment.

Figure 13.5 A MapReduce debugging workflow

Discussion

So, a job has failed and you want to find out information about the cause of the failure. When a job fails, it's useful to look at the logs to see if they tell you anything about the failure. Each map and reduce task has its own logs, so you need to identify the

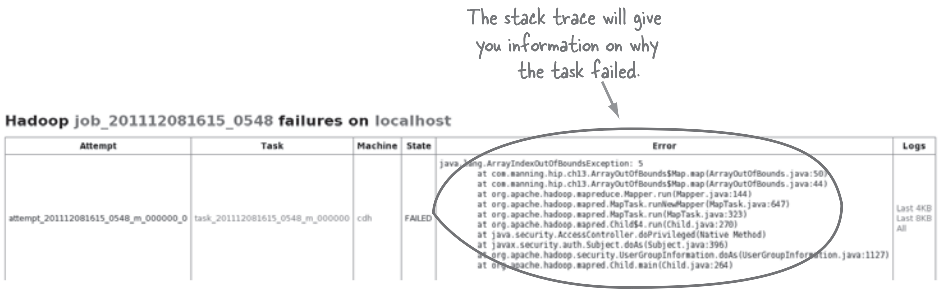


Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	1	0	0	0	1	4 / 0
reduce	100.00%	0	0	0	0	0	0 / 0

Figure 13.6 JobTracker job summary page showing tasks

tasks that failed. The easiest way to do this is to use the JobTracker UI. Select the job that failed from the main JobTracker page, and you'll be presented with some statistics about tasks, as shown in figure 13.6.

If you click on the number of a failed task you'll see a page containing all of the failed tasks and a stack trace for each task, an example of which is shown in figure 13.7.


The stack trace will give you information on why the task failed.

Hadoop job_201112081615_0548 failures on localhost

Attempt	Task	Machine	State	Error	Logs
attempt_201112081615_0548_m_000000_0	task_201112081615_0548_m_000000	cdh	FAILED	java.lang.ArrayIndexOutOfBoundsException: 5 at com.manning.hip.ch13.ArrayOutOfBounds$Map.map(ArrayOutOfBounds.java:50) at com.manning.hip.ch13.ArrayOutOfBounds$Map.map(ArrayOutOfBounds.java:44) at org.apache.hadoop.mapreduce.Mapper.run(Mapper.java:144) at org.apache.hadoop.mapred.MapTask.runNewMapper(MapTask.java:647) at org.apache.hadoop.mapred.MapTask.run(MapTask.java:323) at org.apache.hadoop.mapred.Child$4.run(Child.java:270) at java.security.AccessController.doPrivileged(Native Method) at javax.security.auth.Subject.doAs(Subject.java:396) at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInformation.java:1127) at org.apache.hadoop.mapred.Child.main(Child.java:264)	Last 4KB Last 8KB All

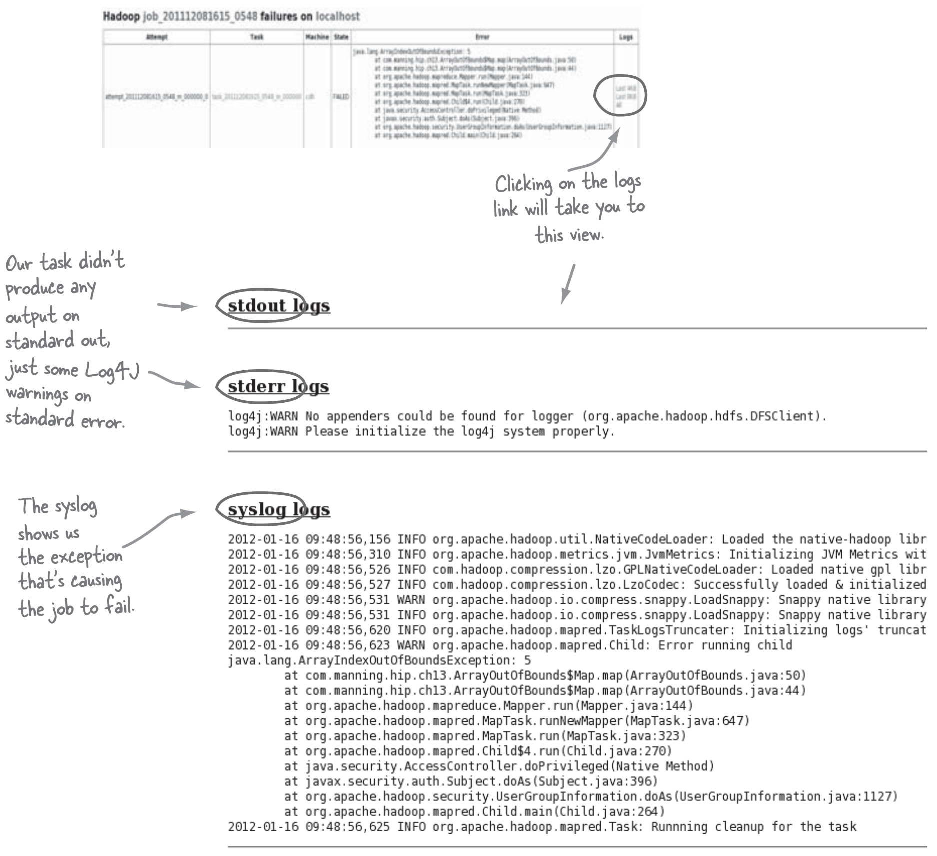
Figure 13.7 JobTracker task summary page showing failed tasks

A number of scenarios under which tasks will fail include following:

- A hardware problem related to the network or the local host running the task exists.
- The HDFS quota for the account running the job has been exceeded.
- Application caching caused the JVM to run out of memory.
- Unexpected input caused the application to fail.

Depending on the problem, you may find additional useful information in the logs, or in the standard out (stdout) or standard error (stderr) of the task process. You can view all three outputs easily by selecting the All link under the Logs column, as shown in figure 13.8.

This is all fine and dandy, but what if you don't have access to the UI? How do you figure out the failed tasks and get at their output files? The job history command-line



Hadoop job_201112081615_0548 **failures on** localhost

Attempt	Task	Machine	State	Error	Logs
attempt_201112081615_0548_m_000000_0	task_201112081615_0548_m_000000	cdh	FAILED	java.lang.ArrayIndexOutOfBoundsException: 5 at com.manning.hip.ch13.ArrayOutOfBounds$Map.map(ArrayOutOfBounds.java:50) at com.manning.hip.ch13.ArrayOutOfBounds$Map.map(ArrayOutOfBounds.java:44) at org.apache.hadoop.mapreduce.Mapper.run(Mapper.java:144) at org.apache.hadoop.mapred.MapTask.runNewMapper(MapTask.java:647) at org.apache.hadoop.mapred.MapTask.run(MapTask.java:323) at org.apache.hadoop.mapred.Child$4.run(Child.java:270) at java.security.AccessController.doPrivileged(Native Method) at javax.security.auth.Subject.doAs(Subject.java:396) at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInformation.java:1127) at org.apache.hadoop.mapred.Child.main(Child.java:264)	Last 4KB Last 8KB All

stdout logs

stderr logs

```
log4j:WARN No appenders could be found for logger (org.apache.hadoop.hdfs.DFSClient).
log4j:WARN Please initialize the log4j system properly.
```

syslog logs

```
2012-01-16 09:48:56,156 INFO org.apache.hadoop.util.NativeCodeLoader: Loaded the native-hadoop libr
2012-01-16 09:48:56,310 INFO org.apache.hadoop.metrics.jvm.JvmMetrics: Initializing JVM Metrics wit
2012-01-16 09:48:56,526 INFO com.hadoop.compression.lzo.GPLNativeCodeLoader: Loaded native gpl libr
2012-01-16 09:48:56,527 INFO com.hadoop.compression.lzo.LzoCodec: Successfully loaded & initialized
2012-01-16 09:48:56,531 WARN org.apache.hadoop.io.compress.snappy.LoadSnappy: Snappy native library
2012-01-16 09:48:56,531 INFO org.apache.hadoop.io.compress.snappy.LoadSnappy: Snappy native library
2012-01-16 09:48:56,620 INFO org.apache.hadoop.mapred.TaskLogsTruncater: Initializing logs' truncat
2012-01-16 09:48:56,623 WARN org.apache.hadoop.mapred.Child: Error running child
java.lang.ArrayIndexOutOfBoundsException: 5
        at com.manning.hip.ch13.ArrayOutOfBounds$Map.map(ArrayOutOfBounds.java:50)
        at com.manning.hip.ch13.ArrayOutOfBounds$Map.map(ArrayOutOfBounds.java:44)
        at org.apache.hadoop.mapreduce.Mapper.run(Mapper.java:144)
        at org.apache.hadoop.mapred.MapTask.runNewMapper(MapTask.java:647)
        at org.apache.hadoop.mapred.MapTask.run(MapTask.java:323)
        at org.apache.hadoop.mapred.Child$4.run(Child.java:270)
        at java.security.AccessController.doPrivileged(Native Method)
        at javax.security.auth.Subject.doAs(Subject.java:396)
        at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInformation.java:1127)
        at org.apache.hadoop.mapred.Child.main(Child.java:264)
2012-01-16 09:48:56,625 INFO org.apache.hadoop.mapred.Task: Runnning cleanup for the task
```

Clicking on the logs link will take you to this view.
Our task didn't produce any output on standard out, just some Log4J warnings on standard error.
The syslog shows us the exception that's causing the job to fail.

Figure 13.8 TaskTracker page showing output from standard output, standard error, and the logs

interface (CLI) can help because it will include in its output a list of all the failed map and reduce tasks, and for each task the thrown exception:

```
$ hadoop job -history all output
...
FAILED MAP task list for job_201112081615_0548
TaskId          StartTime     FinishTime     Error     InputSplits
====================================================
task_201112081615_0548_m_000000
java.lang.ArrayIndexOutOfBoundsException: 5
    at com.manning.hip.ch13.ArrayOutOfBounds$Map.map(ArrayOutOfBo...
    at com.manning.hip.ch13.ArrayOutOfBounds$Map.map(ArrayOutOfBo...
    at org.apache.hadoop.mapreduce.Mapper.run(Mapper.java:144)
    at org.apache.hadoop.mapred.MapTask.runNewMapper(MapTask.java...
    ...
    http://cdh:50060/tasklog?attemptid=attempt_.
```

The last argument is the output directory of the job, which is used to extract the job history details. The all option gives you verbose output for all tasks.

The failed task name.

A URL that can be used to retrieve all the outputs related to the task. You can also figure out the host that executed the task by examining the host in the URL.

This output is informative: not only do you see the exception, but you also see the task name and the host on which the task was executed. The only data related to the task output you see here is the exception, so to view all of the task outputs you'll need to use the URL in this output:

```
$ curl http://cdh:50060/tasklog?attemptid=attempt_...

<html> <title>Task Logs: 'attempt_201112081615_0548_m_000000_0'</title>
<body> <h1>Task Logs: 'attempt_201112081615_0548_m_000000_0'</h1><br>
<br><b><u>stdout logs</u></b><br>
<pre>
</pre></td></tr></table><hr><br>
<br><b><u>stderr logs</u></b><br>
<pre> log4j:WARN No appenders could be found ...
log4j:WARN Please initialize the log4j system properly.
</pre></td></tr></table><hr><br>
<br><b><u>syslog logs</u></b><br>
<pre> ...
2012-01-16 09:48:56,623 WARN org.apache.hadoop.mapred.Child:

Error running child
java.lang.ArrayIndexOutOfBoundsException: 5
    at com.manning.hip.ch13.ArrayOutOfBounds$Map.map(ArrayOutOfBounds...
    at com.manning.hip.ch13.ArrayOutOfBounds$Map.map(ArrayOutOfBounds...
...
</pre></td></tr></table><hr><br> </body></html>
```

Using curl to download the outputs from the TaskTracker URL

An HTML tag indicating the start of the standard output.

The start of the standard error.

The start of the logs.

TASKTRACKER ACCESSIBILITY

For the curl command to work you'll need to run it from a host that has access to the TaskTracker node.

It'll be easier to parse the output by saving the HTML to a file (by adding `-o [filename]` to the curl command), copying that file to your local host, and using a browser to view the file.

What if you're working in an environment where you don't have access to the JobTracker or TaskTracker UI? This may be the case if you're working in clusters that have firewalls blocking access to the UI ports from your laptop or desktop. What if you only have SSH access to the cluster? One option is to run Lynx, a text-based web browser, from inside your cluster. If you don't have Lynx you'll have to know how to access the task logs directly. You know the hostname from the URL, so you'll need to first SSH to that host. The logs for each task are contained in the Hadoop logs directory.

LOCATION OF HADOOP LOGS

By default this directory is $HADOOP_HOME/logs, but check your `hadoop-env.xml`, because this can be overridden with the `HADOOP_LOG_DIR` environment variable.

In the example, you're running on a packaged CDH installation and CDH doesn't override the default logs directory, so because HADOOP_HOME is `/usr/lib/hadoop`,

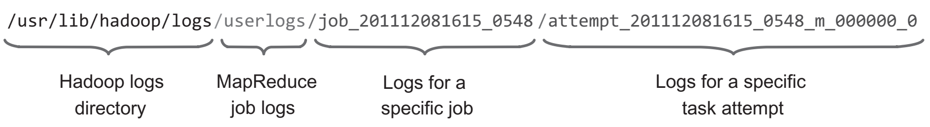

/usr/lib/hadoop/logs/userlogs/job_201112081615_0548/attempt_201112081615_0548_m_000000_0
Hadoop logs directory
MapReduce job logs
Logs for a specific job
Logs for a specific task attempt

Figure 13.9 **Location of task output files on the TaskTracker node**

your logs are under `/usr/lib/hadoop/logs`.[11] Figure 13.9 shows the entire path to your failed task.

Under this directory you'll find at least the following three files:

- `stderr`, containing standard error output
- `stdout`, containing standard output
- `stdlog`, containing the logs

You can use your favorite editor or simple tools like `cat` or `less` to view the contents of these files.

Summary

Often, when things start going wrong in your jobs the task logs will contain details on the cause of the failure. This technique looked at how you could use the JobTracker and, alternatively, the Linux shell to access your logs.

If the data in the logs suggests that the problem with your job is with the inputs (which can be manifested by a parsing exception), you need to figure out what kind of input is causing the problem.

13.2.2 Debugging unexpected inputs

In the previous section, you saw how to access failed task output files to help you figure out the root cause of the failure. In the example, the outputs didn't contain any additional information, which means that you're dealing with some MapReduce code that wasn't written to handle error conditions.

If it's possible to easily modify the MapReduce code that's failing, go ahead and skip to section 13.2.4 and look at the strategies to update your code to better handle and report on broken inputs. Roll these changes into your code, push your code to the cluster, and rerun the job. Your job outputs now will contain enough details for you to be able to update your code to better handle the unexpected inputs.

If this isn't an option, read on; we'll look at what to do to isolate the input data that's causing your code to misbehave.

TECHNIQUE 82 Pinpointing a problem Input Split

Imagine you have a job that's reading Twitter tweets from a number of input files. Some of the tweets aren't formed correctly (could be a syntax problem or an unexpected value that you're unaware of in your data dictionary), which leads to failure in your processing logic.

[11] `/usr/lib/hadoop/logs` is a symbolic link that points to `/var/log/hadoop-[version]`.

By examining the logs, you're able to determine that there's some data in your input files which is causing your parsing code to fail. But your job has numerous input files and they're all large, so your challenge is to narrow down where the problem inputs exist.

Problem

You want to identify the specific input split that's causing parsing issues.

Solution

Use the `keep.failed.task.files` MapReduce configuration parameter to stop Hadoop from cleaning-up task metadata, and use this metadata to understand information about the input splits for a failing task.

Discussion

Take the following three steps to fix the situation:

1. Identify the bad input record(s).
2. Fix your code to handle the bad input records.
3. Add additional error handling capabilities to your code to make this easier to debug in the future.

In this technique we'll focus on the first item, because it will help you to fix your code. We'll cover future-proofing your code for debugging in section 13.2.4.

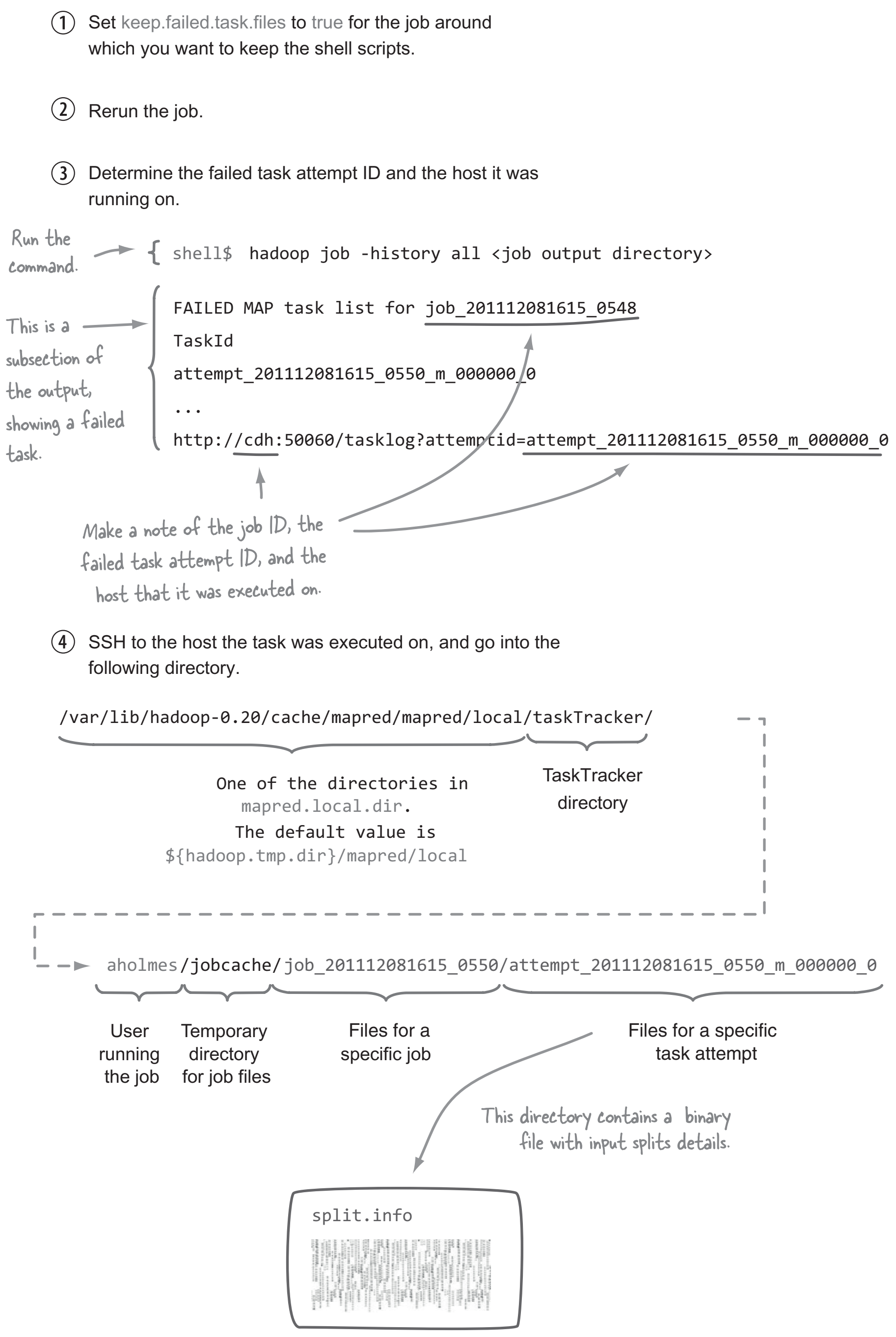
The first step you need to do is determine what file contains the bad input record, and even better, find a range within that file, if the file's large. Unfortunately, Hadoop by default wipes out task-level details, including the input splits after the tasks have completed. You'll need to disable this by setting the `keep.failed.task.files` to `true`. You'll also have to rerun the job that failed, but this time you'll be able to extract additional metadata about the failing task.

After rerunning the failed job you'll once again need to use the `hadoop job -history` command discussed in the previous section to identify the host and job or task IDs. With this information in hand, you'll need to use the shell to log into the TaskTracker node, which ran the failed task, and then navigate to the task directory, which contains information about the input splits for the task. Figure 13.10 shows how to do that.

The trick here is, if you have multiple directories configured for `mapred.local.dir`, you'll need to find which directory contains the task directory. This can be easily accomplished with a `find`, as follows:

```
$ cd <mapred.local.dir>
$ find <task-attempt-id>
```

When you've located this directory, you'll see a number of files related to the task, including a file called split.info. This file contains information about the location of the input split file in HDFS, as well as an offset that's used to determine which of the input splits this task is working on. Both the task and job split files are a mixture of text and binary content, so unfortunately, you can't crack out your command-line editor to easily view their contents.


① Set keep.failed.task.files to true for the job around which you want to keep the shell scripts.
② Rerun the job.
③ Determine the failed task attempt ID and the host it was running on.
Run the command.
This is a subsection of the output, showing a failed task.
Make a note of the job ID, the failed task attempt ID, and the host that it was executed on.
④ SSH to the host the task was executed on, and go into the following directory.
One of the directories in mapred.local.dir. The default value is ${hadoop.tmp.dir}/mapred/local
TaskTracker directory
User running the job
Temporary directory for job files
Files for a specific job
Files for a specific task attempt
This directory contains a binary file with input splits details.
split.info

```
shell$ hadoop job -history all <job output directory>
```

```
FAILED MAP task list for job_201112081615_0548
TaskId
attempt_201112081615_0550_m_000000_0
...
http://cdh:50060/tasklog?attemptid=attempt_201112081615_0550_m_000000_0
```

```
/var/lib/hadoop-0.20/cache/mapred/mapred/local/taskTracker/
```

```
aholmes/jobcache/job_201112081615_0550/attempt_201112081615_0550_m_000000_0
```

Figure 13.10 Accessing the files related to a task on a TaskTracker node

To help with this situation, I've written a utility that can read the input split file for a task and use that information to open the job split file in HDFS, jump into the task-specific offset, and read the task-specific split information. Be warned that there's a good chance this won't work with versions other than Hadoop 0.20.x.

If you run this on the `input.split` file for your failed task you'll gain some insight into the file and data about the start and end of the split (assuming the input for the task is a file):

```
$ bin/run.sh com.manning.hip.ch13.TaskSplitReader split.info
ToString on split = hdfs://localhost/user/aholmes/users.txt:0+110
Reflection fields = FileSplit[
file=hdfs://localhost/user/aholmes/users.txt,
start=0,
length=110,
hosts=<null> ]
```

At this point you have a couple of options: you can copy the file into your local box and write a unit test that runs the MapReduce job with the file. You also can modify your code to catch an exception, which will allow you to set a breakpoint in your IDE and observe the input that's causing your exception.

Alternatively, depending on the Hadoop distribution you're running, Hadoop comes with a tool called `IsolationRunner`, which can re-execute a specific task with its input split. Unfortunately, `IsolationRunner` is broken on 0.20.x[12] and on older Hadoop distributions, but it should work on versions 0.21 and newer. *Hadoop in Action* by Chuck Lam contains an example of how to use the `IsolationRunner`, as does the Hadoop tutorial at http://goo.gl/FRv1H. You can enable some options so that the task JVM runs with the Java debug agent enabled, and connect to the task via your IDE or `jdb`.

Summary

We used this technique to identify the input splits for a task that's failing due to a problem with some input data. Next we'll look at how you get at the JVM arguments you used to launch your task—useful when you suspect there's an issue related to the JVM environment.

13.2.3 Debugging JVM settings

This technique steps somewhat outside of the realm of your user space MapReduce debugging, but it's a useful technique in situations where you suspect there's an issue with the startup JVM arguments for tasks. For example, sometimes the classpath ordering of JARs is significant and issues with it can cause class loading problems. Also, if a job has dependencies on native libraries, the JVM arguments can be used to debug issues with `java.library.path`.

[12] If you're feeling adventurous, there's a 0.20 patch in the JIRA ticket https://issues.apache.org/jira/browse/HADOOP-4041.

TECHNIQUE 83 Figuring out the JVM startup arguments for a task

The ability to examine the various arguments used to start a task can be helpful in debugging task failures. For example, let's say you're trying to use a native Hadoop compression codec, but your MapReduce tasks are failing and the errors complain that the native compression libraries can't be loaded. In this case review the JVM startup arguments to determine if all of the required settings exist for native compression to work.

Problem

You suspect that a task is failing due to missing arguments when a task is being launched, and want to examine the JVM startup arguments.

Solution

Use the `keep.failed.task.files` MapReduce configuration parameter to stop Hadoop from cleaning-up task metadata, and use this metadata to view the shell script used to launch the MapReduce map and reduce tasks.

Discussion

As the TaskTracker prepares to launch a map or reduce task, it also creates a shell script that's subsequently executed to run the task. The problem is that MapReduce by default removes these scripts after a job has completed. So, during the executing of a long-running job or task, you'll have access to these scripts, but if tasks and the job are short-lived (which they may well be if you're debugging an issue that causes the task to fail off the bat), you will once again need to set `keep.failed.task.files` to true. Figure 13.11 shows all of the steps required to gain access to the task shell script.

If you were investigating an issue related to the native Hadoop compression, there's a good chance that when you viewed the taskjvm.sh file, you'd notice that it's missing the necessary `-Djava.library.path`, which points to the directories containing the native Linux compression libraries. If this is the case, you can remedy the problem by adding the native path, which you do by exporting `JAVA_LIBRARY_PATH` in `hadoop-env.sh`.

Summary

This technique is useful in situations where you want to be able to examine the arguments used to launch the task JVM. Next we'll look at some coding practices that can help with debugging activities.

13.2.4 Coding guidelines for effective debugging

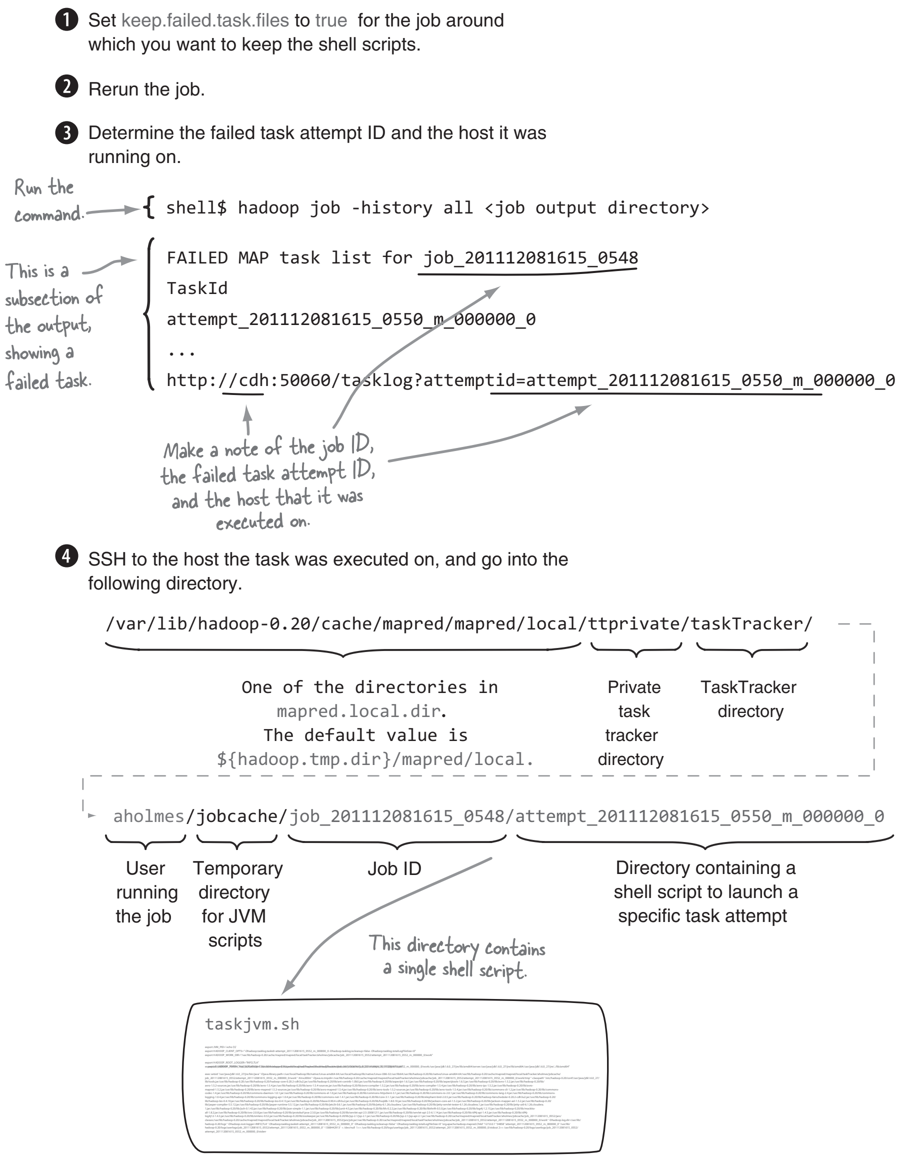
Debugging MapReduce code in production can be made a whole lot easier if you follow a handful of logging and exception-handling best practices.

TECHNIQUE 84 Debugging and error handling

Debugging a poorly written MapReduce job consumes a lot of time, and can be challenging in production environments where access to cluster resources is limited.

Problem

You want to know the best practices to follow when writing MapReduce code.


1 Set keep.failed.task.files to true for the job around which you want to keep the shell scripts.
2 Rerun the job.
3 Determine the failed task attempt ID and the host it was running on.
Run the command.
This is a subsection of the output, showing a failed task.
Make a note of the job ID, the failed task attempt ID, and the host that it was executed on.
4 SSH to the host the task was executed on, and go into the following directory.
One of the directories in mapred.local.dir. The default value is ${hadoop.tmp.dir}/mapred/local.
Private task tracker directory
TaskTracker directory
User running the job
Temporary directory for JVM scripts
Job ID
Directory containing a shell script to launch a specific task attempt
This directory contains a single shell script.
taskjvm.sh

```
shell$ hadoop job -history all <job output directory>
```

```
FAILED MAP task list for job_201112081615_0548
TaskId
attempt_201112081615_0550_m_000000_0
...
http://cdh:50060/tasklog?attemptid=attempt_201112081615_0550_m_000000_0
```

```
/var/lib/hadoop-0.20/cache/mapred/mapred/local/ttprivate/taskTracker/
aholmes/jobcache/job_201112081615_0548/attempt_201112081615_0550_m_000000_0
```

Figure 13.11 How to get to the private directory for a task attempt

Solution

Look at how counters and logs can be used to enhance your ability to effectively debug and handle problem jobs.

Discussion

Add the following features into your code:

- Include logs to capture data related to inputs and outputs to help isolate where problems exist.
- Catch exceptions and provide meaningful logging output to help track down problem data inputs and logic errors.
- Think about whether you want to rethrow or swallow exceptions in your code.
- Leverage counters and task statuses that can be utilized by driver code and humans alike to better understand what happened during the job execution.

In the following code,[13] you'll see applied a number of the previously described principles.

A mapper job with some best practices applied to assist debugging

```
public static class Map
    extends Mapper<Text, Text, Text, Text> {
  protected Text outputValue = new Text();
  protected int failedRecords;
  public static enum Counters {
    FAILED_RECORDS
  }

  @Override
  protected void setup(Context context)
      throws IOException, InterruptedException {
    super.setup(context);
    log.info("Input split = {}", context.getInputSplit());
  }


  @Override
  protected void map(Text key, Text value, Context context)
      throws IOException, InterruptedException {

    if(log.isDebugEnabled()) {
      log.debug("Input K[{}],V[{}]", key, value);
    }


    try {
      String id = StringUtils.split(value.toString())[5];
      outputValue.set(id);
      if(log.isDebugEnabled()) {
        log.debug("Output K[{}],V[{}]", key, value);
      }
      context.write(key, outputValue);
    } catch(Exception t) {
      processError(context, t, key, value);
    }
  }

  protected void processError(Context c, Throwable t, Text k, Text v) {
    log.error("Caught exception processing key[" +
      k + "], value[" + v + "]", t);
```

When the task starts, write the input split details to the log. This will tell you the input file for each specific task and the byte offset within that input file that was used to read map input records.

If the logger's in debug mode (which it should never be in production environments unless you're debugging a job), write out the input record key and value. You wouldn't want this to be a System.out or log.info since that would dramatically slow down your job. Note that you enclose both the key and value with square brackets so you can easily identify leading and trailing whitespaces. Also note that this is important because it helps isolate potential problems in your input data, or InputFormat/RecordReader classes.

You also log the map output key and value. This can be compared to reducer inputs to help determine if there's a serialization or partitioning problem between map and reduce tasks.

Catch any exceptions thrown in your code.

Write out the key and value to the logs. Note that you enclosed both strings with square brackets to easily track down leading or trailing whitespaces.

[13] **GitHub source**—https://github.com/alexholmes/hadoop-book/blob/master/src/main/java/com/manning/hip/ch13/OptimizedMRForDebugging.java

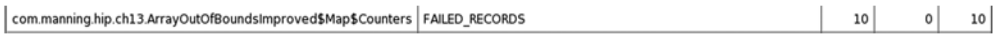
```
        c.getCounter(Counters.FAILED_RECORDS).increment(1);

        c.setStatus("Records with failures = " +
          (++failedRecords));
      }
    }
```

Increment a counter to signal that you hit an error.

Set that task status to indicate you hit an issue with a record, including a count of the total number of failed records this task encountered.

The reduce task should have similar debug log statements added to write out each reduce input key and value, and the output key and value. Doing so will help identify any issues between the map and reduce side, in your reduce code, or a problem with the `OutputFormat/RecordWriter`.

You used counters to count the number of bad records you encountered. The JobTracker UI can be used to view the counter values, as shown in figure 13.12.



com.manning.hip.ch13.ArrayOutOfBoundsImproved$Map$Counters	FAILED_RECORDS	10	0	10

Figure 13.12 Screenshot of the counter in JobTracker's job summary page

Depending on how you executed the job, you'll see the counters dumped on standard out. You have programmatic access to counters, and the job history command will also include the counters:

```
$ hadoop job -history all output

Counters:

|Group Name                    |Counter name                  |

Map Value |Reduce Value|Total Value|
----------------------------------------------------------------
...
|com.manning.hip.ch13.ArrayOutOfBoundsImproved$Map$Counters|

FAILED_RECORDS                |10         |0          |10
...
```

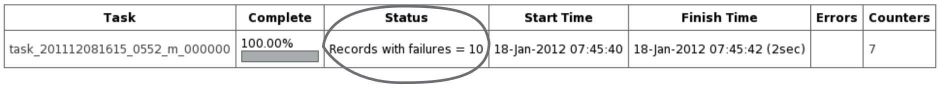
If you look at the logs for your tasks, you'll also see some informative data related to the task:

```
Input split = hdfs://localhost/user/aholmes/users.txt:0+110

Caught exception processing key[anne], value[22 NY]
```

This tells you what file the task was working on, as well as the input split range.

Write out the key and value. Note that because you used square brackets to encapsulate your strings, any leading or whitespace issues will be easily identified.

Because you also updated the task status in your code, you can use the JobTracker UI to easily identify the tasks that had failed records, as shown in figure 13.13.



Task	Complete	Status	Start Time	Finish Time	Errors	Counters
task_201112081615_0552_m_000000	100.00%	Records with failures = 10	18-Jan-2012 07:45:40	18-Jan-2012 07:45:42 (2sec)		7

Figure 13.13 JobTracker UI showing map task and status

Summary

We looked at a handful of simple yet useful coding guidelines for your MapReduce code. If they're applied and you hit a problem with your job in production, you'll be in a great position to quickly narrow down possibilities on the root cause of the issue. If the issue's related to the input, your logs will contain details about how the input caused your processing logic to fail. If the issue is related to some logic error, or errors in serialization/deserialization, you can enable debug-level logging and better understand where things are going awry.

SHOULD EXCEPTIONS BE SWALLOWED?

In the previous code example you caught any exception in your code and then made sure to write the exception to the logs along with as much contextual information as possible (such as the current key and value that the reducer was working on). The big question is, should you rethrow the exception, or swallow it?

Rethrowing the exception is tempting because you'll be immediately aware of any issues in your MapReduce code. But if your code is running in production and fails every time it encounters a problem such as some input data that's not handled correctly, ops, dev, and QA will be spending quite a few cycles addressing each issue as it comes along.

Writing code as you did to swallow exceptions has its own problems—for example, what if you encounter an exception on all inputs to the job? If you write code to swallow exceptions, the correct approach is to increment a counter (as in the code example), which the driver class should use after job completion to ensure that most of the input records within some tolerable threshold were successfully processed. If they weren't, the workflow being processed should probably be terminated and the appropriate alerts be sent to notify operations.

Another approach is to not swallow exceptions and to configure record skipping with a call to `setMapperMaxSkipRecords` and/or `setReducerMaxSkipGroups`, indicating the number of records that you can tolerate losing if an exception is thrown when they're processed. This is covered in more detail in *Hadoop in Action* by Chuck Lam.

13.3 *MapReduce gotchas*

To complete this chapter we'll examine some common missteps in MapReduce that often lead to hours of debugging. The intent here is to learn by examining practices that should be avoided in MapReduce.

TECHNIQUE 85 MapReduce anti-patterns

Throughout this book I've covered a number of patterns to help you write and execute MapReduce jobs. It can be just as useful to learn from anti-patterns, which are patterns that are commonly used but are either ineffective or worse, detrimental in practice.

Problem

You want to learn some MapReduce anti-patterns so you'll be aware of what practices you should avoid.

Solution

Learn and laugh at mistakes that I've made in MapReduce on production clusters, which range from loading too much data into memory in tasks to going crazy with counters and bringing down the JobTracker.

Discussion

Here are some practices that are best avoided.

TOO MUCH CACHE

Caching data in map and reduce tasks is required for many kinds of operations, such as data joins. But in Java the memory overhead of caching is significant (see chapter 6 for specific details), and if your cache becomes too large to fit in Java's heap, your task will fail with an `OutOfMemoryError` exception.

Data join packages that perform caching (such as the Hadoop contribution `org.apache.hadoop.contrib.utils.join` package) attempt to mitigate this by limiting the maximum number of records that will be cached. This is an approach worth considering, albeit it assumes the records are not overly large (bear in mind that even if you cap the number of records to a small size, it only takes a handful of large records to blow out your memory).

If you're implementing some strategies (such as capping how many records are being cached), make sure you use counters to identify that you're performing that capping, and ideally, by how much (count how many records aren't being cached), so you can better understand data that's being skipped. If you're working with variable-length records, it may be useful to log records over a certain size—again, to better understand your data and to help you make future caching decisions.

LARGE INPUT RECORDS

Stop to think about the input data to your MapReduce jobs. If each input record isn't a fixed size, there's a chance you could encounter records that are possibly too large to fit into memory. Take, for example, a simple case of a job that reads lines from a text file. You'll likely be using `TextInputFormat` (the default `InputFormat` in MapReduce) or `KeyValueTextInputFormat`. In either case there's no cap on the maximum length of a line, so if you have a line that's millions of characters in length, there's a chance it won't fit into memory (or if it does, any operation you attempt to perform on that string will exhaust your memory).

Luckily, `TextInputFormat` and `KeyValueTextInputFormat` use the same `RecordReader` class, which contains a configuration you can set that limits the maximum line size, `mapred.linerecordreader.maxlength`. It will also log cases where it encountered lines that are over this length, including the byte offset in the input file.

If you're working with other `InputFormats`, you should check to see if they have any mechanisms to limit the size of input records. Similarly, if you're writing an `InputFormat`, think about adding support for limiting the size of records you feed to a map task.

OVERWHELMING EXTERNAL RESOURCES

There's nothing that can stop you from writing MapReduce jobs to pull data from databases, or web servers, or any other data source external to HDFS. Keep in mind, though, the use of these external data sources, both by other users as well as by the MapReduce job. It's possible that the data source you're working with doesn't scale to support hundreds or thousands of concurrent reads or writes, and your single MapReduce job may bring it to its knees. I recommend you limit the number of map and/or reduce tasks to a small number to minimize the likelihood of this occurring.

SPECULATIVE EXECUTION RACE CONDITIONS

Speculative execution is a mechanism used in MapReduce to guard against slow nodes in a cluster. As the map and reduce phases of a job near completion, MapReduce will launch duplicate tasks that work off of the same inputs as the remaining tasks.

This is fine if your job is writing its outputs using the standard MapReduce output mechanism (and assuming the `InputFormat` being used is correctly handling output committing (see chapter 3 for more details). But what if your job is writing to a database or some other external resource, or directly to a file in HDFS? Now you have multiple tasks both writing the same data, which is probably not what you want.

One approach that tools such as distCp[14] and Sqoop[15] use to guard against this is to disable speculative execution:

```
conf.set("mapred.map.tasks.speculative.execution", "false");
conf.set("mapred.reduce.tasks.speculative.execution", "false");
```

If you're using an `OutputFormat` that's based on the `FileOutputFormat`, and you want to write additional output to HDFS, the best approach is to write into the task's attempt directory. Each task's reduce (or map if a no-reduce job is being run) is written to a temporary attempt directory, and only if the task succeeds are the files moved into the job output directory. The following code[16] shows a map-only job that is writing output to a side-effect file:

```
public static class Map
    extends Mapper<Text, Text, Text, Text> {
```

[14] DistCp is a useful tool for copying HDFS data between clusters; see http://hadoop.apache.org/docs/r1.0.3/distcp.html.

[15] Sqoop is a tool to import and export database data to and from HDFS. More details on Sqoop can be found in chapter 2.

[16] **GitHub source**—https://github.com/alexholmes/hadoop-book/blob/master/src/main/java/com/manning/hip/ch13/SideEffectJob.java

```
    OutputStream sideEffectStream;

    @Override
    protected void setup(Context context)
        throws IOException, InterruptedException {
      Path attemptDir =
        FileOutputFormat.getWorkOutputPath(context);
      String filename = context.getTaskAttemptID()
          .getTaskID().toString();

      Path sideEffectFile = new Path(attemptDir, filename);

      sideEffectStream = FileSystem.get(context.getConfiguration())
          .create(sideEffectFile);
    }

    @Override
    protected void map(Text key, Text value, Context context)
        throws IOException, InterruptedException {

      IOUtils.write(key.toString(), sideEffectStream);

      context.write(key, value);
    }

    @Override
    protected void cleanup(Context context)
        throws IOException, InterruptedException {
      sideEffectStream.close();
    }
  }
```

The OutputStream for the file in HDFS you'll be writing to.

Ask the FileOutputFormat for the HDFS working directory for this attempt.

Extract the attempt ID to be used as your filename.

Create a file in the attempt's working directory in HDFS.

Write the input key to your file.

Remember to close the file after your task has completed.

If you run this job you should observe two output files, one written to be the `RecordWriter`, and the other written by you directly from your map task:

```
$ bin/run.sh com.manning.hip.ch13.SideEffectJob users.txt output

$ hadoop fs -ls output

/user/aholmes/output2/_SUCCESS
/user/aholmes/output2/_logs
/user/aholmes/output2/part-m-00000
/user/aholmes/output2/task_201112081615_0558_m_000000
```

NOT HANDLING BAD INPUT

Working with bad input is often the norm in MapReduce, but if you have code that doesn't expect the unexpected, it may start failing when it sees data it doesn't expect. Ideally, the code should be able to handle these situations, but there is a workaround, without having to touch the code, via the `SkipBadRecords` class.[17] *Hadoop in Action* by Chuck Lam has more details on how to use this class, but at a basic level this feature allows you to specify the tolerance for the number of records surrounding a bad record that can be discarded.

[17] See http://hadoop.apache.org/common/docs/r1.0.0/api/org/apache/hadoop/mapred/SkipBadRecords.html.

CLUSTERS WITH DIFFERENT HADOOP VERSIONS AND CONFIGURATION SETTINGS

It's not uncommon for code that works in unit tests to fail in a cluster. But if you're running multiple clusters, make an effort to ensure that the Hadoop versions, and Hadoop configurations, align as closely as possible. Hadoop's many configuration settings can cause jobs to behave differently, and keeping discrepancies down to a minimum will ensure that a job succeeding on one cluster will work on another cluster.

TESTING AND DEBUGGING WITH LARGE DATASETS

When you're developing and testing MapReduce, Pig, or Hive scripts, it's tempting to work directly with the full set of input data. But doing so flies in the face of rapid development—rather than quickly iterating the development and test cycles, you'll be sitting around waiting for the results of your job, and at the same time needlessly chewing up cluster resources. Instead, look at the sampling techniques presented in chapters 4, 10, and 11 to work on a subset of the input data, and leave the execution against the full set of data until such a time as you're happy with the results using the smaller dataset.

NOT HANDLING PARSING OR LOGIC ERRORS

We already covered this topic in section 13.2.4, but I want to reemphasize that a high percentage of problems you'll encounter in your job are due to unexpected input, and can be as simple an issue as leading or trailing whitespace characters that cause parsing issues. Including measures to be able to quickly debug these issues is crucial.

TOO MANY COUNTERS

Counters are a great mechanism to communicate numerical data to some driver code that's running your MapReduce job. Be warned that each counter incurs some amount of memory overhead in the JobTracker. For each individual counter the memory footprint may be small, but if you use counters carelessly this could lead to memory exhaustion in the JobTracker. An example of this situation would be where you dynamically created a counter for each input record in the map task—it would only take a few million records for there to be a noticeable memory impact and overall slowdown in the JobTracker.

Summary

We covered a few of the bumps you'll face when you work with MapReduce. You'll never be able to foresee all of the potential problems you could encounter, but understanding some of the more common issues we've highlighted in this technique, coupled with a well-thought-out implementation of your MapReduce functions, can go a long way to avoiding those 2 a.m. production debugging sessions.

13.4 Chapter summary

This chapter only scratched the surface when it comes to testing and debugging. We laid the groundwork for how to test and debug user space MapReduce, but there's much more to testing and debugging outside of the scope of user space MapReduce.

If you're running any critical MapReduce code in production, it's crucial to follow at least the steps in the testing section of this chapter, where I showed you how to best design your code so it easily lends itself to basic unit testing methodologies outside the scope of Hadoop. We also covered how the MapReduce-related parts of your code could be tested in both lightweight (without MapReduce stack involvement via MRUnit) and more heavyweight (with `LocalTestRunner`) setups.

We also emphasized how to debug issues that result in failing MapReduce jobs, as well as jobs that aren't generating the results you'd expect. We wrapped things up with some examples of badly written MapReduce jobs with the hope that we all can learn from the mistakes of others (including the author).

DATABASES/PROGRAMMING

Hadoop IN PRACTICE

Alex Holmes

Hadoop is an open source MapReduce platform designed to query and analyze data distributed across large clusters. Especially effective for big data systems, Hadoop powers mission-critical software at Apple, eBay, LinkedIn, Yahoo, and Facebook. It offers developers handy ways to store, manage, and analyze data.

Hadoop in Practice collects 85 battle-tested examples and presents them in a problem/solution format. It balances conceptual foundations with practical recipes for key problem areas like data ingress and egress, serialization, and LZO compression. You'll explore each technique step by step, learning how to build a specific solution along with the thinking that went into it. As a bonus, the book's examples create a well-structured and understandable codebase you can tweak to meet your own needs.

What's Inside

- Conceptual overview of Hadoop and MapReduce
- 85 practical, tested techniques
- Real problems, real solutions
- How to integrate MapReduce and R

This book assumes the reader knows the basics of Hadoop.

Alex Holmes is a senior software engineer with extensive expertise in solving big data problems using Hadoop. He has presented at JavaOne and Jazoon and is a technical lead at VeriSign.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/HadoopinPractice

Free eBook
SEE INSERT

"Interesting topics that tickle the creative brain."
—Mark Kemna, Brillig

"Ties together the Hadoop ecosystem technologies."
—Ayon Sinha, Britely

"Comprehensive ... high-quality code samples."
—Chris Nauroth
The Walt Disney Company

"Covers all of the variants of Hadoop, not just the Apache distribution."
—Ted Dunning
MapR Technologies

"Charts a path to the future."
—Alexey Gayduk
Grid Dynamics

ISNB 13: 978-1-61729-023-7
ISBN 10: 1-61729-023-8

54999

9 781617 290237

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]