





Local MapReduce debugging

Tools, tips, and tricks

Aaron Kimball
Cloudera Inc.
July 21, 2009



Common sense debugging tips

- Build incrementally
 - Build compositionally
 - Use appropriate abstractions
 - Start with small amounts of data
 - Keep track of program state
 - Watch for the corner cases
 - (Refer to picture of Zen rock garden as necessary)
-

Testing >> debugging

- Best strategy: don't write buggy code, so no need to debug!
 - What do you mean you can't write code without bugs?
 - At least test your code to make sure it works...
 - Throwing your code at large volumes of data is a poor test
 - Long wait for results
 - Difficult to attach a debugger if necessary
 - Difficult to see what went wrong
 - Post-mortem analysis on a production cluster is hard
 - Solution: first, test locally
-

Three ways to test locally

- Inside the IDE:
 - MRUnit
 - LocalJobRunner
 - On the local machine
 - Pseudo-distributed mode
-

Unit tests

- JUnit tests are an advantageous way to test code
 - Can be run in IDE (e.g., Eclipse)
 - Can be run from command line or from automated build tools
 - Industry standard: powerful, lightweight, etc.
 - How do we make effective use of JUnit in Hadoop?
-

JUnit

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import junit.framework.JUnit4TestAdapter;

public class MyTest {
    @Test
    public void testOne() {
        assertEquals(...);
        assertTrue(...);
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(MyTest.class);
    }
}
```

My mapper...

```
private static class MyMapper
    extends MapReduceBase implements
    Mapper<LongWritable, Text, Text, Text> {

    public void map(LongWritable key, Text doc,
        OutputCollector<Text, Text> output,
        Reporter reporter) throws IOException {

        // does something really cool...

    }
}
```

JUnit + mapper = ?

```
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import junit.framework.JUnit4TestAdapter;

public class MyTest {
    @Test
    public void testOne() {
        assertEquals(...);
        assertTrue(...);
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(MyTest.class);
    }
}
```

MyMapper.map... goes where?

First, designing for testability

- Good design practices:
 - Majority of your logic should be Hadoop-agnostic
 - Don't involve Hadoop-specific classes (OutputCollector, Reporter, Writable) unless absolutely necessary
 - Connect with Hadoop API at the “interfaces”
 - Remember, compositionality and abstraction!
 - Mappers, Reducers should be relatively straightforward compositions of Hadoop-agnostic objects
 - Thus, testable separately
-

Testing Map and Reduce classes

- Mappers, Reducers still need tests even if components separately tested
 - Interaction between components
 - Type conversions (i.e., serialization and deserialization)
 - Expected output format
 - Testing a Mapper with JUnit is not straightforward
 - Input comes from data files
 - Output goes to OutputCollector
 - Testing a Reducer with JUnit is not straightforward
 - Input comes from shuffled/sorted map output
 - Output goes to OutputCollector
-

MRUnit

- A unit-test Library for Hadoop developed by Cloudera
 - Framework for sending inputs to mappers, reducers...
 - ... and verifying outputs
 - Framework preserves MapReduce semantics
 - Tests are compact and inline
 - Test harness uses mock objects where necessary
 - Reporter
 - InputSplit
 - OutputCollector
-

Example: Testing IdentityMapper

```
public class ExampleTest extends TestCase {
    private Mapper<Text, Text, Text, Text> mapper;
    private MapDriver<Text, Text, Text, Text> driver;

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

    @Test
    public void testIdentityMapper() {
        driver.withInput(new Text("foo"), new Text("bar"))
            .withOutput(new Text("foo"), new Text("bar"))
            .runTest();
    }
}
```

Test framework

- Test drivers
 - Wrappers around Mapper, Reducer, inputs, and expected outputs
 - **MapDriver** – executes tests for a Mapper
 - **ReduceDriver** – executes tests for a Reducer
 - **MapReduceDriver** – tests a mapper, shuffles/sorts outputs, then tests reducer on shuffled/sorted outputs
 - Mock objects
 - **MockOutputCollector** – delivers results back to TestDriver
 - **MockReporter** – ignores all use, but returns MockInputSplit
 - **MockInputSplit** – dummy FileInputSplit implementation
-

MRUnit test semantics: Mappers

- Mapper receives a single input (key, value) pair
- May return 0, 1, or more output (key, value) pairs
- Outputs collected by test harness
 - All types supported by Hadoop supported here

MRUnit test API: Mappers

- `withInput(K, V)`: sets the input to be sent to the mapper
 - Returns self for “chaining”
 - `run()`: runs mapper and returns actual output
 - You verify that the outputs are what you expect
 - Return type is `List<Pair<K, V>>`
 - `withOutput(K, V)`: adds an expected output pair
 - Returns self for “chaining”
 - `runTest()`: runs mapper and checks that actual outputs match expected
 - Order of emitted output elements significant

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

MRUnit test semantics: Reducers

- Reducer receives a single input key and an ordered set of values
- May emit 0, 1, or more output (key, value) pairs

MRUnit test API: Reducers

- `withInput(K, List<V>)`: sets the input to be sent to the reducer
 - Returns self for “chaining”
 - `withOutput(K, V)`: adds an expected output pair
 - `run()`, `runTest()`: same semantics as before
-

Test semantics: Map+Reduce

- Arbitrary number of (key, value) pairs are presented to mapper
 - Mapper emits arbitrary number of intermediate (key, value) pairs
 - Intermediate (key, value) pairs are sorted/shuffled (standard MapReduce semantics)
 - Sets of (key, list<value>) are sent to single reducer
 - As if `mapred.reduce.tasks = 1`
 - `run()`, `runTest()` as before
-

Recommended practices

- Create a new TestDriver, Mapper (or Reducer) in the setUp() method
- Use a single (k, v) input per test case
- Make use of IDE, debugger, stack traces, etc...

Beyond unit testing...

- MRUnit tests only cover “core” of MapReduce program
 - But this is often the easy part
 - Standard JUnit tests can be used for custom Writables
 - You still need to test all the pieces plugged together...
 - Custom InputFormat, OutputFormat, RecordReader, RecordWriter, ...
 - Driver method that initializes JobConf
-

LocalJobRunner

- Hadoop can run MapReduce entirely in a single process
 - Uses local file system instead of HDFS
 - Spills data to disk on demand

- Turning it on:

```
conf.set("mapred.job.tracker", "local");
```

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

- Then call:

```
JobClient.runJob(conf);
```

Keeping up with your process: stderr?

- That's too fancy... I just use stdout
 - Printing: the tried and true (if last-ditch) debug tactic
 - LocalJobRunner: you see output of `System.err.println()`
 - For real: where did my debug messages go?
 - `System.err.println()` does **not** go to your local console!
 - Visible on the same status page as task syslog
 - Also recorded to the same logs directory
-

Where's my output?

- Use the webapp!
Job [job_200904151511_0004](#)

All Task Attempts

Task Attempts	Machine	Status	Progress	Start Time	Finish Time	Errors	Task Logs	Counters	Actions
attempt_200904151511_0004_m_000000_0	/default-rack/jargon	SUCCEEDED	100.00%	15-Apr-2009 15:29:30	15-Apr-2009 15:29:32 (15sec)		Last 4KB Last 8KB All	5	

Input Split Locations

[Go back to the job](#)
[Go back to JobTracker](#)

[Hadoop](#), 2009.

- On the per-task drill-down page on the JobTracker status site, can access entire event log or just the last few KB
- Will also show you any attempt-killing exception, and counters
- Present in `$HADOOP_HOME/logs/userlogs/${task.id}/syslog`
 - On the machine where it ran...

Why log?

- println statements get real awkward, real fast
 - (ctrl-f println //)+
 - Logging allows much finer-grained control over...
 - What gets logged
 - When something gets logged
 - How something is logged
 - Most frequently-asked question:
 - Does it impact performance?
-

Logging with log4j

- Hadoop uses log4j to generate all its logs
- Your mappers and reducers can also use logging facilities
 - All initialization handled by Hadoop for you

```
import org.apache.commons.logging.Log;  
import org.apache.commons.logging.LogFactory;  
  
class FooMapper implements Mapper {  
    public static final Log LOGGER =  
LogFactory.getLog(FooMapper.class.getName());  
    ...  
}
```

Logging is your friend!

Perhaps single most powerful debug tool...

Logging with log4j

- Simply send strings to loggers tagged with severity levels:
 - `LOGGER.debug("message");`
 - `LOGGER.info("message");`
 - `LOGGER.warn("message");`
 - `LOGGER.error("message");`
-

Log4j configuration

- Stored in `$HADOOP_HOME/conf/log4j.properties`
 - Can change global log settings with `hadoop.root.logger` property
 - Can override log level on a per-class basis:
 - `log4j.logger.org.apache.hadoop.mapred.JobTracker=WARN`
 - `log4j.logger.com.example.myprogram.FooMapper=DEBUG`
 - Programmatically:
 - `LOGGER.setLevel(Level.WARN);`
-

Where's my log?

Job [job_200904151511_0004](#)

All Task Attempts

Task Attempts	Machine	Status	Progress	Start Time	Finish Time	Errors	Task Logs	Counters	Actions
attempt_200904151511_0004_m_000000_0	/default-rack/jargon	SUCCEEDED	100.00%	15-Apr-2009 15:29:30	15-Apr-2009 15:29:32 (13ec)		Last 4KB Last 8KB All	5	

Input Split Locations

[Go back to the job](#)
[Go back to JobTracker](#)

[Hadoop](#), 2009.

- On the per-task drill-down page on the JobTracker status site, can access entire event log or just the last few KB
- Will also show you any attempt-killing exception, and counters
- Present in `$HADOOP_HOME/logs/userlogs/${task.id}/syslog`
 - On the machine where it ran...

Recommended practice

- Separate Hadoop-specific from Hadoop-agnostic code
 - First, test (MRUnit, JUnit)
 - Then, run with LocalJobRunner (i.e., standalone mode)
 - After, run in pseudo-distributed mode
 - Now, you're ready to play at scale...
 - (skip steps once you become an expert...)
-

Conclusions

- Debugging is difficult!
- Use all the available tools
 - Unit tests
 - Eclipse debugger (with JUnit, LocalJobRunner)
 - Logs (and the web app)
 - Exceptions
- Be patient (refer to zen garden as needed)
- MRUnit can make unit tests easier to write, making your tests more thorough, saving you trouble down the road...

www.cloudera.com/hadoop-mrunit

aaron@cloudera.com



(c) 2008 Cloudera, Inc. or its licensors. "Cloudera" is a registered trademark of Cloudera, Inc.. All rights reserved. 1.0