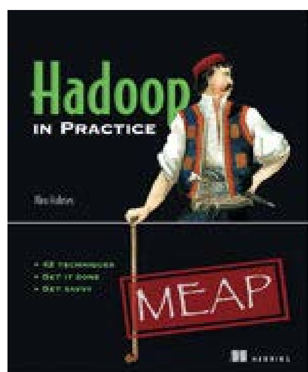


Hadoop in Practice
By Alex Holmes

MapReduce patterns help you write effective code and make efficient use of your data and your Hadoop cluster. 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. In this article based on chapter 13 of Hadoop in Practice, you can learn and laugh at mistakes that the author 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.

You may also be interested in…

MapReduce Anti-patterns

When you're running MapReduce in production, you can guarantee that someday you'll receive a call about a failing job. 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.

In this article, you'll learn some MapReduce anti-patterns so you'll be aware of what practices you should avoid.

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, 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 `InputFormat`s, 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 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. 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[1] and Sqoop[2] 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 shows a map-only job that is writing output to a side-effect file:

[1] DistCp is a useful tool for copying HDFS data between clusters; see http://hadoop.apache.org/common/docs/current/distcp.html.
[2] Sqoop is a tool to import and export database data to and from HDFS.

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

  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.

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

If you run this job, you should observe two output files, one written to be the `Record-Writer`, 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.[3] *Hadoop in Action*

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

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.

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, 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

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.

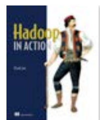
Too many counters

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 if you dynamically created a counter for each input record in the map task—it would only take a few million records to have a noticeable memory impact and overall slowdown in the JobTracker.

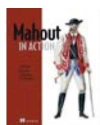
Summary

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 article, 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.

Here are some other Manning titles you might be interested in:



Hadoop in Action
Chuck Lam



Mahout in Action
Sean Owen, Robin Anil, Ted Dunning, and Ellen Friedman

Tika in Action
Chris A. Mattmann and Jukka L. Zitting

Last updated: September 11, 2012