Hadoop Tutorial

David Chiang

27 Mar 2009

1 Overview

The basic Map/Reduce idea goes like this.

1. The input records, which are key/value pairs, are split into chunks and each chunk is fed into a *mapper*.
2. Each mapper repeatedly reads an input record and outputs zero or more intermediate records.
3. All the records with the same key are sent to the same *reducer*, where they are sorted by key.
4. Each reducer repeatedly reads a key and a list of values, and outputs zero or more records.

The canonical example is counting words.

$$\begin{bmatrix} \text{see spot run} \\ \text{run spot run} \end{bmatrix} \xrightarrow{\text{map}} \begin{bmatrix} \text{see} & 1 \\ \text{spot} & 1 \\ \text{run} & 1 \\ \text{run} & 1 \\ \text{spot} & 1 \\ \text{run} & 1 \end{bmatrix} \xrightarrow{\text{sort}} \begin{bmatrix} \text{run} & 1 \\ \text{run} & 1 \\ \text{run} & 1 \\ \text{see} & 1 \\ \text{spot} & 1 \\ \text{spot} & 1 \end{bmatrix} \xrightarrow{\text{reduce}} \begin{bmatrix} \text{run} & 3 \\ \text{see} & 1 \\ \text{spot} & 2 \end{bmatrix}$$

Hadoop Streaming allows you to use Hadoop without having to write a single line of Java. Instead, the mappers and reducers are implemented as processes that read key/value pairs on stdin and write them on stdout. By default, a record is a line and the key is the first tab-delimited column.

The *Hadoop Distributed Filesystem* (HDFS) is just what it sounds like, though it can't (yet) be mounted as a normal filesystem like PVFS can. It works with Map/Reduce in special ways: for instance, a program that needs to run on some bit of data can be scheduled to run on a machine close to that data.

2 Hadoop on HPC

Hadoop is installed on HPC in `/home/nlg-01/chiangd/pkg/hadoop`. There is no permanent cluster or distributed filesystem there (yet), so you have to encapsulate Hadoop clusters inside PBS jobs. We have a few ways to do this:

1. *Hadoop on Demand* tries to do everything for you. You (or your script) run a command on the login node (`hod allocate`) and it submits the PBS job for you, waits for the job to start, and creates the Hadoop cluster inside it. Then you can proceed to run Hadoop jobs. Finally, you run another command (`hod deallocate`) which retrieves your log files, cleans up the cluster, and ends the PBS job.
2. `pbs_hadoop`, written by John Heidemann, takes the minimalist approach. It expects you to start the PBS job yourself, and expects to be run inside a PBS job. It creates your Hadoop cluster, and then your script can proceed to run Hadoop jobs. If your cluster's log directories reside on the node(s), you need to retrieve them yourself before ending the job.

3. `/home/nlg-01/chiangd/hadoop/pbs_hadoop.py`, which is the same thing rewritten in Python. The configuration file generation just seemed cleaner that way. This is the script that I use and so it is maintained a little better.

Here is an example PBS script.

```
#!/bin/sh

# start.sh

# Where Hadoop lives
export HADOOP_HOME=/home/nlg-01/chiangd/pkg/hadoop

# Where the Hadoop cluster will be created.
# It should be shared among all the nodes.
CLUSTER=$HOME/hadoop-tutorial/cluster

# Create the cluster
/home/nlg-01/chiangd/hadoop/pbs_hadoop.py $CLUSTER || exit 1

# Hadoop needs to know where the cluster configuration files are
export HADOOP_CONF_DIR=$CLUSTER/conf

# Run Hadoop jobs here

# Destroy the Hadoop cluster. This isn't really necessary.
# $HADOOP_HOME/bin/stop-all.sh
```

You would submit this using something like

```
qsub -l nodes=10 start.sh
```

Or, you can run it as a step in a larger grid workflow. The fact that there are many steps run in sequence inside the script doesn't seem to be in keeping with the workflow philosophy, but the alternative is not very efficient: the workflow would create many Hadoop clusters and copy files into and out of the HDFS each time. We need a better solution.

- Run workflows on the login node. The first step in the DAG would use Hadoop on Demand to create a Hadoop cluster in the remote workflow directory, and subsequent steps would run on the login node but use the cluster. The last step in the DAG would bring the cluster down. All this would require lots of long-running processes on the login node.
- The same, but on a login node dedicated to our group that doesn't have a 30-minute time limit.
- Set up a permanent HDFS on our nodes at HPC. If we did this, then we would have the problem of (a) non-Hadoop jobs running on HDFS nodes (HDFS would take up some `/tmp` space) and (b) Hadoop jobs running on non-HDFS nodes (increased network traffic).

3 Distributed filesystem

In the following commands, pathnames can refer to the HDFS or the outside filesystem according to context. Relative HDFS pathnames are relative to your home directory (`/user/username`).

- `hadoop fs -put src dst`: copy outside file `src` into HDFS
- `hadoop fs -getmerge src dst`: concatenate all the files in `src` and copy to outside file `dst`

- `hadoop fs -rmr dir+`: like `rm -rf`
- `hadoop fs -mv src dst`: rename file in HDFS

I've only described commands that operate on HDFS directories because data is normally stored as many small files (named `part-nnnnn`) inside directories.

Hadoop uses a lot of disk space. Everything is stored in triplicate by default, and typically you need enough space to hold your input, intermediate records, and output at the same time. So the combined space in `/tmp` on all your cluster nodes should be 5–10 times the dataset you are working with. Hadoop is *not* very good at detecting when space is low. I have seen one full disk bring down the whole cluster (hopefully this has improved).

- When running on the main queue, select nodes with the `disk60g` option (i.e., `qsub -l nodes=10:disk60g`) to ensure at least 60 GB in `/tmp`. The `x2200` option will get you (coveted) nodes with 250GB. All `isi` nodes have 60 GB.
- Check (using the DFS browser, see below) that all of `/tmp` really is free on each node. Remove files in `/tmp` left over from previous jobs. `pbs_hadoop.py` does clean up files from previous incarnations of itself.

4 Running Streaming jobs

Let's compute an easy feature, the probability of a rule given its English tree (`trivial_cond_prob`). Our input looks like

```
NP(NNP("French") NNS("fries")) -> "pommes" "frites" ### id=123 count=5
```

and we want to produce output that looks like

```
123 \t e^-1.23
```

where `\t` stands for a tab character.

First, we need a mapper:

```
#!/usr/bin/env perl
use NLPRules qw(extract_lhs_safe extract_feat_safe feature_spec);

# trivial_cond_prob_m.pl
# input:  rule
# output: etree \t id \t count

while (<>) {
    chomp;
    $rule = $_;
    $e = extract_lhs_safe($rule);
    %v = feature_spec(extract_feat_safe($rule));
    print "$e\t$v{id}\t$v{count}\n";
}
```

Then we need a reducer:

```
#!/usr/bin/env python
import sys, itertools, math

# divide0.py
# Calculate conditional probability given key.
# input:  key \t ... \t count
# output: key \t prob

def input():
    """Present input stream as iterator over tuples"""
    for line in sys.stdin:
        yield line.rstrip().split("\t")

for key, records in itertools.groupby(input(), lambda record: record[0]):
    # records is an iterator over all records with the same key
    records = list(records) # turn iterator into list
    sumcount = sum(int(record[-1]) for record in records)
    for record in records:
        value = record[1:-1]
        count = record[-1]
        p = math.log(float(count)/sumcount)
        print "%s\te^%s" % ("\t".join(value), p)
```

Here's how to run it:

```
hadoop jar $HADOOP_HOME/contrib/streaming/hadoop-*-streaming.jar \
  -input rules \
  -mapper trivial_cond_prob_m.pl \
  -reducer divide0.py \
  -output feat.trivial_cond_prob
```

The input is an HDFS directory or file, and the output is an HDFS directory. Hadoop Streaming runs external programs for the mapper and reducer as follows:

- The command
  - is searched for in your PATH and then in the working directory,
  - can have arguments, i.e., `-mapper "command arg arg"`, and
  - is not run inside a shell.
- The working directory
  - is a temporary directory on the remote node.
  - To ship a file from the outside filesystem to this directory, use `-file filename`.
  - To make a file from HDFS appear in this directory, use `-cacheFile src#dst`, where `src` is the path in HDFS and `dst` is the name it will appear under. (In newer versions, this option is renamed to `-files`.)
- The environment contains
  - variables from `.profile` etc. at the time the cluster was created;
  - variables from `$HADOOP_HOME/conf/hadoop-env.sh` at the time the cluster was created;
  - a bunch of Hadoop properties, e.g., `map-input-file` is the same as the Hadoop property `map.input.file` which tells a mapper what file (in HDFS) it is working on;
  - variables specified by you using the `-cmdenv VAR=value` option.
  - Thus, in order to make these examples run, I put these lines into my `.profile` before creating the cluster:

    ```
    HADOOP_TUTORIAL=/home/rcf-12/chiangd/hadoop-tutorial
    PATH=$HADOOP_TUTORIAL:$PATH
    PERL5LIB=$HADOOP_TUTORIAL:$PERL5LIB
    ```

But in practice we would locate the mapper using either of:

```
-file trivial_cond_prob_m.pl -mapper trivial_cond_prob_m.pl
-mapper $HADOOP_TUTORIAL/trivial_cond_prob_m.pl
```

and the Perl libraries using:

```
-cmdenv PERL5LIB=$HADOOP_TUTORIAL
```

5 Inspecting the Hadoop cluster

Map/Reduce and DFS browsers You can point a web browser to `http://hpcnnnn:50030` or `http://hpcnnnn:50070`, where `hpcnnnn` is the master node, to look at the status of your jobs or the filesystem, respectively. The script `/home/nlg-01/chiangd/hadoop/hadoop-view $CLUSTER`, to be run on the HPC login node, will open Firefox to these pages for you.

If a task fails, you can use the browser to view its logs very easily. But if the cluster goes down, then you have to resort to looking at the logs directly.

Log files The first place to look is the stderr of the jobtracker (the `hadoop` process you run from the command line or PBS script). If something goes wrong, you will see something like:

```
09/03/04 15:49:09 INFO streaming.StreamJob: Running job: job_200903041548_0002
...
09/03/04 15:49:31 ERROR streaming.StreamJob: Job not Successful!
```

If there are no other clues, you have to go to the user logs. These are in

```
$CLUSTER/logs/userlogs/attempt_yyyymmddhhmm_jjjj_{m,r}_tttttt_a/{stdout,stderr,syslog}
```

where

- `yyyymmddhhmm` is a timestamp (here, 200903041548)
- `jjjj` is the job number (here, 0002)
- `m` or `r` is for map or reduce
- `tttttt` is the map or reduce task number
- `a` is the attempt number
- `stdout` is for standard output, `stderr` is for standard error, and `syslog` is Hadoop's log. In Streaming, `stdout` is always empty; `stderr` usually has what you want. A few problems (like a nonexistent mapper or reducer script) will be logged to `syslog`.

If there is a way of automatically hunting down the `stderr` of just the failed tasks, I would like to know what it is.

6 A harder example

Now let's try to calculate the probability of a rule given its root (call this feature `prob`). Our mapper looks very similar to before:

```
#!/usr/bin/env perl
use NLPRules qw(extract_root_nt extract_feat_safe feature_spec);

# prob_m.pl
# input:  rule
# output: root \t id \t count

while (<>) {
    chomp;
    $rule = $_;
    $r = extract_root_nt($rule);
    %v = feature_spec(extract_feat_safe($rule));
    print "$r\t$v{id}\t$v{count}\n";
}
```

We could use the reducer from before, but we will run into a problem: there is a line in that script

```
records = list(records) # turn iterator into list
```

that will fail if the list of values for a given key is too large to fit into memory. Since our keys are now just nonterminal symbols, this will fail. So, we need a two-pass approach: in the first pass, we calculate the root counts, and in the second pass, we will divide the rule counts by the root counts calculated in the first pass.

The first pass uses the mapper from above, and this reducer:

```
#!/usr/bin/env python
import sys, itertools

# count.py
# input:  key \t ... \t count
# output: key \t sum

def input():
    for line in sys.stdin:
        yield line.rstrip().split("\t")

for key, records in itertools.groupby(input(), lambda record: record[0]):
    sumcount = sum(int(record[-1]) for record in records)
    print "%s\t%s" % (key, sumcount)
```

The second pass is similar to `divide0.py` from before, except that it reads the sums from a file instead of calculating them:

```
#!/usr/bin/env python
import sys, itertools, math

# divide1.py sumfile
# Calculate conditional probability given key.
# input:   key \t ... \t count
# sumfile: key \t sum
# output:  key \t prob

sumcounts = {}
for line in open(sys.argv[1]):
    fields = line.rstrip().split("\t")
    key = fields[0]
    count = int(fields[1])
    sumcounts[key] = count

```

```
for line in sys.stdin:
    record = line.rstrip().split("\t")
    key = record[0]
    value = record[1:-1]
    count = int(record[-1])
    p = math.log(float(count)/sumcounts[key])
    print "%s\te^%s" % ("\t".join(value), p)
```

Here's how to run them:

```
HADOOP_STREAM="hadoop jar $HADOOP_HOME/contrib/streaming/hadoop-*-streaming.jar"

$HADOOP_STREAM \
  -input rules \
  -mapper prob_m.pl \
  -reducer count.py \
  -output /tmp/sums

hadoop fs -getmerge /tmp/sums $TMPDIR/sums

$HADOOP_STREAM \
  -input rules \
  -mapper prob_m.pl \
  -reducer "divide1.py sums" \
  -output feat.prob \
  -file $TMPDIR/sums

hadoop fs -rmr /tmp/sums
rm -f $TMPDIR/sums
```

This can be optimized in a few ways. First, we have used the `-file` option to make the sums file visible to our scripts. This could be slow if the file is large; copying it to HDFS and using `-cacheFile` might be faster (I'm not sure, actually).

Map-only jobs Second, note that the second reducer (`divide1.py`) does not actually care whether the data are grouped by keys. So we can make it part of the mapper instead:

```
-mapper "sh -c 'prob_m.pl | divide1.py sums'"
-reducer NONE
```

where NONE is a special keyword that skips the reducer, including the sort step. So this will run a lot faster. Note that we needed to wrap the mapper inside a shell to allow the use of a pipe.

Combiners Third, note that the first reducer (`count.py`) could be inefficient if there is a very large bin, because the whole bin has to be processed by a single processor. It would be a lot more efficient if each mapper, instead of sending all the rules to the reducers, calculated subtotals for its own chunk and sent those to the reducers. In Hadoop terminology this extra step is called a *combiner.* However, Hadoop Streaming doesn't have external combiners (yet), only Java-based combiners. But we can achieve exactly the same effect inside the mapper.

This script reads the output of a mapper, collects it into a buffer, and when the buffer reaches a certain size, flushes the buffer, grouping records by key.

```
#!/usr/bin/env python
import sys, getopt, collections

# precombine.py [-k <keysize>] [-b <bufsize>]
# prepare map output for input to a combiner
# <keysize> = number of key fields (default 1)
# <bufsize> = maximum number of records in buffer (default 100000)

if __name__ == "__main__":
    opts, args = getopt.gnu_getopt(sys.argv[1:], 'k:b:')
    opts = dict(opts)

    n_keys = int(opts.get('-k', 1))
    buffer_size = int(opts.get('-b', 100000))

    buffer = collections.defaultdict(list)
    count = 0
    for line in sys.stdin:
        record = line.rstrip().split('\t')
        key = tuple(record[:n_keys])
        buffer[key].append(record)
        count += 1

        if count >= buffer_size:
            for key, records in buffer.iteritems():
                for record in records:
                    print "\t".join(record)
            buffer.clear()
            count = 0

    for key, records in buffer.iteritems():
        for record in records:
            print "\t".join(record)
```

Then, instead of using just `prob_m.pl` as the mapper, we can use

```
-mapper "sh -c 'prob_m.pl | precombine.py | count.py'"
-reducer count.py
```

Thus the output of the mapper will be considerably smaller.

7 An even harder example

The first set of scripts worked if the bins were not too big; the second set of scripts handled the case of large bins by precomputing the sums of all the bins. Now, what happens if we have a large number of large bins, as with our `phrase_pfe` feature (formerly known as `lex_pfe`)? Simplifying a bit, let's say that this is the probability of a rule given its English words (excluding all variables and nonterminals).[1] Once again we need a mapper:

```
#!/usr/bin/env perl
use NLPRules qw(extract_lhs_safe target_words extract_feat_safe feature_spec);

# phrase_pfe_m.pl
# This is not the same as the real phrase_pfe.
# input:  rule
# output: ewords \t id \t count

while (<>) {
    chomp;
```

[1]The real `phrase_pfe` is the probability of the French words given the English words.

```
    $rule = $_;
    $e = join(" ", target_words(extract_lhs_safe($rule)));
    %v = feature_spec(extract_feat_safe($rule));
    print "$e\t$v{id}\t$v{count}\n";
}
```

We need to precompute the sums just as before, but we can't (or don't want to) load the resulting file into memory. Instead, we're going to use a program that performs an operation similar to SQL JOIN.

$$\begin{bmatrix} \text{dog} & \text{rule 1} & \text{count=1} \\ \text{dog} & \text{rule 2} & \text{count=2} \\ \text{dog} & \text{rule 4} & \text{count=1} \\ \text{cat} & \text{rule 3} & \text{count=2} \\ \text{cat} & \text{rule 5} & \text{count=3} \end{bmatrix} \bowtie \begin{bmatrix} \text{dog} & \text{sum=4} \\ \text{cat} & \text{sum=5} \end{bmatrix} = \begin{bmatrix} \text{dog} & \text{rule 1} & \text{count=1} & \text{sum=4} \\ \text{dog} & \text{rule 2} & \text{count=2} & \text{sum=4} \\ \text{dog} & \text{rule 4} & \text{count=1} & \text{sum=4} \\ \text{cat} & \text{rule 3} & \text{count=2} & \text{sum=5} \\ \text{cat} & \text{rule 5} & \text{count=3} & \text{sum=5} \end{bmatrix}$$

Now the sum is available in every record and doesn't need to be looked up in some huge file.

Hadoop has a facility to do exactly this, but we need the join to be done in such a way that one of the joined tables (the leftmost one) can have arbitrarily large value sets. I've written a script `join.py` that does the proper incantations to make this happen robustly. Its usage is

```
join.py <input>+ -o <output> [-r <reducer>] [-k <keysize>]
```

where `<reducer>` is an optional reducer to be applied to the output of the join, and `<keysize>` is the number of tab-delimited fields in the key.

After performing the join, we run the following, which is similar to `divide1.py`, except that it expects to find the sum appended to each record.

```
#!/usr/bin/env python
import sys, itertools, math

# divide2.py
# Calculate conditional probability given key.
# input:  key \t ... \t count \t sum
# output: key \t prob

for line in sys.stdin:
    record = line.rstrip().split("\t")
    key = record[0]
    value = record[1:-2]
    count = int(record[-2])
    sumcount = int(record[-1])
    p = math.log(float(count)/sumcount)
    print "%s\te^%s" % ("\t".join(value), p)
```

Here's how to run them:

```
HADOOP_STREAM="hadoop jar $HADOOP_HOME/contrib/streaming/hadoop-*-streaming.jar"

$HADOOP_STREAM \
  -input rules \
  -mapper phrase_pfe_m.pl \
  -reducer NONE \
  -output /tmp/keys

$HADOOP_STREAM \
  -input /tmp/keys \
  -mapper /bin/cat \
  -reducer count.py \
13     -output /tmp/sums
14
15   join.py \
16     /tmp/keys \
17     /tmp/sums \
18     -r divide2.py \
19     -o feat.phrase_pfe
20
21   hadoop fs -rmr /tmp/keys /tmp/sums
```

Note that very large bins will take a long time to process, just as before. Although the combiner trick would work on the first Map/Reduce (lines 3–13), it wouldn't work on the join. For an efficient *and* bulletproof solution a further refinement is needed, which is left as an exercise for the reader.

How the join works Given a bunch of input tables, the mapper concatenates them, tagging each record with an identifier that says which input table it came from. This information is available in the environment variable `map_input_file`. (We need to watch out for the case where an input is a directory containing many parts.) This tag is placed between the key and the value. It is chosen so that the large table has the highest tag.

$$\begin{bmatrix} \text{dog} & \text{rule 1} & \text{count=1} \\ \text{dog} & \text{rule 2} & \text{count=2} \\ \text{dog} & \text{rule 4} & \text{count=1} \\ \text{cat} & \text{rule 3} & \text{count=2} \\ \text{cat} & \text{rule 5} & \text{count=3} \end{bmatrix} \bowtie \begin{bmatrix} \text{dog} & \text{sum=4} \\ \text{cat} & \text{sum=5} \end{bmatrix} \xrightarrow{\text{map}} \begin{bmatrix} \text{dog} & 1 & \text{rule 1} & \text{count=1} \\ \text{dog} & 1 & \text{rule 2} & \text{count=2} \\ \text{dog} & 1 & \text{rule 4} & \text{count=1} \\ \text{cat} & 1 & \text{rule 3} & \text{count=2} \\ \text{cat} & 1 & \text{rule 5} & \text{count=3} \\ \text{dog} & 0 & \text{sum=4} & \\ \text{cat} & 0 & \text{sum=5} & \end{bmatrix}$$

Now the records are sent, by a stage called the *partitioner*, to different reducers. All the records with the same key (dog or cat) go to the same reducer. But inside the reducer, we want them to be sorted so that input 0 comes first.

$$\xrightarrow{\text{partition}} \begin{matrix} \begin{bmatrix} \text{cat} & 1 & \text{rule 3} & \text{count=2} \\ \text{cat} & 1 & \text{rule 5} & \text{count=3} \\ \text{cat} & 0 & \text{sum=5} & \end{bmatrix} \\ \begin{bmatrix} \text{dog} & 1 & \text{rule 1} & \text{count=1} \\ \text{dog} & 1 & \text{rule 2} & \text{count=2} \\ \text{dog} & 1 & \text{rule 4} & \text{count=1} \\ \text{dog} & 0 & \text{sum=4} & \end{bmatrix} \end{matrix} \xrightarrow{\text{sort}} \begin{matrix} \begin{bmatrix} \text{cat} & 0 & \text{sum=5} & \\ \text{cat} & 1 & \text{rule 3} & \text{count=2} \\ \text{cat} & 1 & \text{rule 5} & \text{count=3} \end{bmatrix} \\ \begin{bmatrix} \text{dog} & 0 & \text{sum=4} & \\ \text{dog} & 1 & \text{rule 1} & \text{count=1} \\ \text{dog} & 1 & \text{rule 2} & \text{count=2} \\ \text{dog} & 1 & \text{rule 4} & \text{count=1} \end{bmatrix} \end{matrix}$$

To do this, we tell the partitioner that our keys are one field wide, but we tell the sorter that they are two fields wide:

```
-partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner
-jobconf num.key.fields.for.partition=1
-jobconf stream.map.output.key.fields=2
```

Finally, the reducer performs the join for each key, discarding the source tags. It calculates the cross-product (here trivial) of the records coming from the non-large tables, and then it can cross those with the records from the large table. Because the large table is last, its records don't have to be stored in memory at once.

$$\xrightarrow{\text{reduce}} \begin{bmatrix} \text{cat} & \text{rule 3} & \text{count}=2 & \text{sum}=5 \\ \text{cat} & \text{rule 5} & \text{count}=3 & \text{sum}=5 \end{bmatrix} \begin{bmatrix} \text{dog} & \text{rule 1} & \text{count}=1 & \text{sum}=4 \\ \text{dog} & \text{rule 2} & \text{count}=2 & \text{sum}=4 \\ \text{dog} & \text{rule 4} & \text{count}=1 & \text{sum}=4 \end{bmatrix}$$

8 Putting it all together

Finally, we need to paste the three features back onto the rules. We can do this using `join.py` as well.

```
hadoop jar $HADOOP_HOME/contrib/streaming/hadoop-*-streaming.jar \
  -input rules \
  -mapper paste_m.pl \
  -reducer NONE \
  -output /tmp/idrules

join.py \
  /tmp/idrules \
  feat.trivial_cond_prob \
  feat.prob \
  feat.phrase_pfe \
  -r "paste_r.pl trivial_cond_prob prob phrase_pfe" \
  -o rules.final
```

```
#!/usr/bin/env perl
use NLPRules qw(extract_feat_safe feature_spec);

# paste_m.pl
# input:  rule
# output: id \t rule

while (<>) {
    chomp;
    $rule = $_;
    %v = feature_spec(extract_feat_safe($rule));
    print "$v{id}\t$rule\n";
}
```

```
#!/usr/bin/env perl

# paste_r.pl featname+
# input:  id \t rule (\t feat)+
# output:       rule (" " featname=feat)+

while (<STDIN>) {
    chomp;
    @fields = split(/\t/);
    $out = $fields[1];
    for ($i=0; $i<@ARGV; $i++) {
        $out = $out . " $ARGV[$i]=$fields[$i+2]";
    }
    print "$out\n";
}
```

9 Controlling Map/Reduce execution

Number of mappers The default number of mappers is one per input HDFS block (64 MB). This is normally okay, but if your mapper has a long startup time, or if it generates vastly more output than it inputs, you may want to cut down. The number of reducers is set by `pbs_hadoop` to 1.75 times the number of nodes and I've never needed to change it.

```
-jobconf mapred.map.tasks=123
-jobconf mapred.reduce.tasks=123
```

Speculative execution This is a crazy feature, on by default, by which Hadoop starts multiple mapper or reducer tasks on the same chunk, uses the output of the winner, and kills the loser. Kind of like DARPA does. Hadoop knows not to let duplicate records get into the output, but if a task has side-effects (like writing files to the outside filesystem), you need to turn this feature off, using one of:

```
-jobconf mapred.map.tasks.speculative.execution=false
-jobconf mapred.reduce.tasks.speculative.execution=false
```

There are many more options. Hadoop Streaming options can be found in the official tutorial; Hadoop options are poorly documented, and the best place to look is usually `$HADOOP_HOME/conf/hadoop-default.xml`.

10 Map/Reduce Summary

In the course of these examples we have mentioned several refinements to the basic Map/Reduce story. Here is the whole story:

1. split data into splits which are each fed to a mapper
2. mapper:
   (a) map: user-defined operation on individual key/value pairs
   (b) partition the output of each mapper into one partition for each reducer
   (c) combine: preliminary reduction within a single mapper
3. shuffle each partition from each mapper to the reducer it is destined for
4. reducer:
   (a) sort key/value pairs by key
   (b) group key/value pairs by key
   (c) reduce: user-defined operation on a key and its values

In Hadoop Streaming, only Java combiners are currently allowed, but you can simulate a combiner inside the mapper (`precombine.py` may help). The grouping step is irrelevant because the reducer receives its input as a stream of records and must do its own grouping.