An Alternative to Hadoop: "Snowdoop"

January 2, 2015

Hadoop made it convenient to process data in very large distributed databases, and also convenient to create them, using the Hadoop Distributed File System. But eventually word got out that Hadoop is slow, and very limited in available data operations.

Both of those shortcomings are addressed to a large extent by the new kid on the block, Spark. Spark is apparently much faster than Hadoop, sometimes dramatically so, due to strong caching ability and a wider variety of available operations.

But even Spark suffers a very practical problem, shared by the others mentioned above: All of these systems are complicated. There is a considerable amount of configuration to do, worsened by dependence on infrastructure software such as Java or MPI, and in some cases by interface software such as **rJava**. Some of this requires systems knowledge that many R users may lack. And once they do get these systems set up, they may be required to design algorithms with world views quite different from R, even though technically they are coding in R.

So, do we really need all that complicated machinery? Hadoop and Spark provide efficient distributed sort operations, but if one's application does not depend on sorting, we have a cost-benefit issue here.

Here is an alternative, more of a general approach rather than a package, which I call "Snowdoop." (The name alludes to the fact that it uses the section of the **parallel** package derived from the old **snow** package.) The idea is to retain the notion of chunking files into distributed mini-files, but (a) do this on one's own, and (b) the process those files using *ordinary R code*, not fancy new functions like Hadoop and Spark require.

Let's use as our example word count, the "Hello World" of MapReduce. It determines which words are in a text file, and calculates frequency counts for each distinct word:

```
> # setclsinfo(), filechunkname(), addlists() from partools
> library(partools)
> # each node executes this function
> wordcensus <- function(basename,ndigs) {
+    # find the file chunk to be handled by this worker
+    fname <- filechunkname(basename,ndigs)
+    words <- scan(fname,what="")
+    # determine which words occur how frequently in this chunk
+    tapply(words,words,length, simplify=FALSE)
+ }
> # manager
> fullwordcount <- function(cls,basename,ndigs) {
```

```
+    setclsinfo(cls)  # give workers ID numbers, etc.
+    clusterEvalQ(cls,library(partools))
+    # have each worker execute wordcensus()
+    counts <- clusterCall(cls,wordcensus,basename,ndigs)
+    # coalesce the output for the overall counts
+    addlistssum <- function(lst1,lst2) addlists(lst1,lst2,sum)
+    Reduce(addlistssum,counts)
+ }
> test <- function() {
+    cls <- makeCluster(2)
+    # make a test file, 2 chunks
+    cat("How much wood","Could a woodchuck chuck",file="test.1",sep="\n")
+    cat("If a woodchuck could chuck wood?",file="test.2")
+    # set up cluster
+    cls <- makeCluster(2)
+    # find the counts
+    fullwordcount(cls,"test",1)
+ }
```

Output:

```
> test()
$a
[1] 2

$chuck
[1] 2

$Could
[1] 1

$How
[1] 1

$much
[1] 1

$wood
[1] 1

$woodchuck
[1] 2
...
```

All this was pure R! No Java, no configuration. Indeed, it's worthwhile comparing to the word count example in the **sparkr** distribution. There we see calls to **sparkr** functions such as **flatMap()**, **reduceByKey()** and **collect()**, none of them that deep, but in Snowdoop we avoid having to learn them, since we just use ordinary R.[1]

[1] Indeed, the **reduceByKey()** function is pretty much the same as R's famous **tapply()**.

We are achieving the parallel-read advantage of Hadoop and Spark,[2] *while avoiding the Hadoop/Spark configuration headaches and while staying with the familiar R programming paradigm.* In many cases, this should be a highly beneficial tradeoff for us.[3]

Note that we also achieve "caching," in that data will remain at the worker nodes after a Snowdoop "call." This is useful for iterative algorithms, for instance

As with Hadoop, there is a question as to how many chunks to split a file into. Generally, a chunk should fit into memory, and one should have as many chunks as we anticipate having workers (number of cores, number of machines in a physical cluster, etc.). In any event, it is assumed here that one will always set up a number of Snow workers equal to the number of chunks in the file.

[2]Note that neither Hadoop, Spark nor Snowdoop will achieve full parallel reading if the file chunks are all on the same disk.

[3]Note, though, that the fault tolerance feature of Hadhoop and Spark can be quite advantageous. And as of this writing, it is not yet clear how well this scales, e.g. how well the **parallel** package works with very large numbers of nodes.