Writing composable mapreduce jobs

The theory

It's actually harder to explain it than to do it but here is an attempt to define what it takes to write a composable map reduce job. The mapreduce function is designed to allow people to chain jobs such as in

```
mapreduce(input =
    mapreduce(input = …,
                  map = somefilter(k,v) …)
              map = someotherfilter(k,v) …)
```

Suppose that, according to accepted software practices, we want to encapsulate a useful job into a new function while preserving its composability as in the above example. What properties does your composable job need to satisfy?

1. Take as an argument, ideally called "input", an object that can be only assigned from, or passed to another reusable job or `mapreduce` or `from.dfs` call:

   ```
   reusableJob = function(input, …) {
       some work here
       mapreduce(input = input, ...)}
   ```

   Since lists of such input objects are also allowed, you could in principle do manipulations on those lists, but not on the individual objects, for instance:

   ```
   multiInputReusableJob = function(inputs = c(...)) {
       mergeJob(input = lapply(inputs,
                               function(input) mapreduce(input = input ...), ...)}
   ```

2. accept an output option with a default of NULL and pass it onto the last of the jobs your are going to execute
3. Return as output the output of the last job your function executes, or a vector thereof as in the previous example.
4. Alternatively to the last two points, from.dfs the results and return them when their size makes it feasible and preferable.
   This way the results can be passed to other R functions, not to other jobs, but the jobs can still be used in complex expressions and assignments and so are considered composable.
5. accept a profilenodes option with a default of FALSE and pass it onto any job you may execute

Examples

With the following definition:

```
 mapReduceFilter = function(input, output = NULL, pred, profileNodes = FALSE) {
     mapreduce(input = input, output = output,
                              map = function(k,v) if(pred(k,v)) keyval(k,v)
                                                    else NULL),
                              profilenodes = profilenodes)}
```

We can then create a chain of filters of this sort:

```
mapReduceFilter(input = mapReduceFilter(input, output, pred1), output, pred2)
```

In this specific case it would have been easy and faster to combine the two predicates directly, but it shows how the composition is done.