Mapreduce in R

My first mapreduce job

Conceptually, mapreduce is not very different than a combination of lapplys and a tapply: transform elements of a list, compute an index — key in mapreduce jargon — and process the groups thus defined. Let's start with a simple lapply example:

```
small.ints = 1:1000
lapply(small.ints, function(x) x^2)
```

The example is trivial, just computing the first 10 squares, but we just want to get the basics here, there are interesting examples later on. Now to the mapreduce equivalent:

```
small.ints = to.dfs(1:1000)
mapreduce(input = small.ints, map = function(k,v) keyval(v, v^2))
```

This is all it takes to write your first mapreduce job in rmr. There are some difference that we will go through, but the first thing to notice is that it isn't all that different, and just two lines of code. There are some superficial differences and some more fundamental ones. The first line puts the data into HDFS, where the bulk of the data has to be for mapreduce to operate on. Of course, we are unlikely to write out big data with to.dfs, certainly not in a scalable way. to.dfs is nonetheless very useful for a variety of uses like writing test cases, REPL and HPC-type uses of mapreduce — that is, small data but big CPU demands. to.dfs can put the data in a file of your own choosing, but if you don't specify one it will create tempfiles and clean them up when done. The return value is a an object that you can use as a “big data” object. You can assign it to variables, pass it to functions, mapreduce jobs or read it back in. Now onto the second line. It has mapreduce replace lapply. We prefer named arguments with mapreduce because there's quite a few possible arguments, but one could do otherwise. The input is the variable out which contains the output of to.dfs, that is our small number data set in its HDFS version, but there are other choices as we will see. The function to apply, which is called a map function in contrast with the reduce function, which we are not using here, is a regular R function with a few constraints:

1. It's a function of two arguments, a key and a value
2. It returns a key value pair as returned by the helper function keyval, which takes any two R objects as arguments; you can also return a list of such objects, or NULL.

In this example, we are not using the key at all, only the value, but we still need both to support the general mapreduce case. The return value is an object, and you can pass it as input to other jobs or read it into memory (watch out, not good for big data) with from.dfs. from.dfs is complementary to.dfs. It returns a list of key-value pairs, which is the most general data type that mapreduce can handle. If you prefer data frames to lists, you can instruct from.dfs to perform a conversion to data frames, which will cover many many important use cases but is not as general as a list of pairs (structured vs. unstructured case). from.dfs is useful in defining map reduce algorithms whenever a mapreduce job produces something of reasonable size, like a summary, that can fit in memory and needs to be inspected to decide on the next steps, or to visualize it.

My second mapreduce job

We've just created a simple job that was logically equivalent to a lapply but can run on big data. That job had only a map. Now to the reduce part. The closest equivalent in R is arguably a tapply. So here is the example from the R docs:

```
groups = rbinom(32, n = 50, prob = 0.4)
tapply(groups, groups, length)
```

This creates a sample from the binomial and counts how many times each outcome occurred. Now onto the mapreduce equivalent:

```
groups = to.dfs(groups)
from.dfs(mapreduce(input = groups, map = function(k,v) keyval(v, 1), reduce = function(k,vv) keyval(k, length(vv))))
```

First we move the data into HDFS with to.dfs. As we said earlier, this is not the normal way in which big data will enter HDFS; it is normally the responsibility of scalable data collection systems such as Flume or Sqoop. In that case we would just specify the HDFS path to the data as input to mapreduce. But in this case the input is the variable groups which points to where the data is temporarily stored, and the naming and clean up is taken care of for you. All you need to know is how you can use it. There isn't a map function, so it is set to default which is like an identity but consistent with the map requirements, that is function(k,v) keyval(k,v). The reduce function takes two arguments, one is a key and the other is a list of all the values associated with that key. Like in the map case, the reduce function can return NULL, a key value pair as generated by the function keyval or a list thereof. The default is somewhat equivalent to an identity function, under the constraints of a reduce function, that is function(k, vv) lapply(vv, function(v) keyval(k,v)). In this case they key is one possible outcome of the binomial and the values are all NULL and the only important thing is how many there are, so length gets the job done. Looking back at this second example, there are some small differences with tapply but the overall complexity is very similar.

Wordcount

The word count program has become a sort of “hello world” of the mapreduce world. For a review of how the same task can be accomplished in several languages, but always for map reduce, see this blog entry.

```
wordcount = function (input, output = NULL, pattern = " ") {
  mapreduce(input = input ,
          output = output,
          input.format = "text",
          map = function(k,v) {
                  lapply(
                    strsplit(
                            x = v,
                            split = pattern)[[1]],
                          function(w) keyval(w,1))},
            reduce = function(k,vv) {
               keyval(k, sum(unlist(vv)))},
            combine = T)}
```

We are defining a function, wordcount, that encapsulates this job. This may not look like a big deal but it is important. Our main goal was not simply to make it easy to run a mapreduce job but to make mapreduce jobs first class citizens of the R environment and to make it easy to create abstractions based on them. For instance, we wanted to be able to assign the result of a mapreduce job to a variable — and I mean *the result*, not some error code or diagnostics — and to create complex expressions including mapreduce jobs. We take the first step here by creating a function that is itself a job, can be chained with other jobs, executed in a loop etc.

Let's now look at the signature. There is an input and optional output and a pattern that defines what a word is.

The implementation is just a single call to mapreduce. Therein, the input can be an HDFS path, the return value of to.dfs or another job or a list thereof — potentially, a mix of all three cases, as in list("a/long/path", to.dfs(...), mapreduce(...), ...). The output can be an HDFS path but if it is NULL some temporary file will be generated and wrapped in a big data object like the ones generated by to.dfs. In either event, the job will return the information about the output, either the path or the big data object. So we simply pass along the input and output of thewordcount function to the mapreduce call and return whatever it returns. That way the new function also behaves like a proper mapreduce job — almost, more details here. The input.format argument allows us to specify the format of the input. The default is based on R's own serialization functions and supports all R data types. In this case we just want to read a text file, so the "text" format will create key value pairs with a NULLkey and a line of text as value. You can easily specify your own input and output formats and even accept and produce binary formats with the functions make.input.format and make.output.format.

The map function, as we know already, takes two arguments, a key and a value. The key here is not important, indeed always NULL. The value contains one line of text, which gets split according to a pattern. Here you can see that pattern is accessible in the mapper without any particular work on the programmer side and according to normal R scope rules. This apparent simplicity hides the fact that the map function is executed in a different interpreter and on a different machine than the mapreduce function. Behind the scenes the R environment is serialized, broadcast to the cluster and restored on each interpreter running on the nodes. For each word, a key value pair (w, 1) is generated with keyval and the list of all of them is the return value of the mapper.

The reduce function takes a key and a list of values as input and simply sums up all the counts and returns the pair word, count using the same helper function, keyval. Finally, specifying the use of a combiner is necessary to guarantee the scalability of this algorithm.

Logistic Regression

Now onto an example from supervised learning, specifically logistic regression by gradient descent. Again we are going to create a function that encapsulates this algorithm.

```
logistic.regression = function(input, iterations, dims, alpha){
  plane = rep(0, dims)
  g = function(z) 1/(1 + exp(-z))
  for (i in 1:iterations) {
    gradient = from.dfs(mapreduce(input,
      map = function(k, v) keyval (1, v$y * v$x * g(-v$y * (plane %*% v$x))),
      reduce = function(k, vv) keyval(k, apply(do.call(rbind,vv),2,sum)),
      combine = T))
    plane = plane + alpha * gradient[[1]]$val }
  plane }
```

As you can see we have an input with the training data. For simplicity we ask to specify a fixed number of iterations, but it would be marginally more difficult to implement a convergence criterion. Then we need to specify the dimension of the problem, which is a bit redundant because it can be inferred after seeing the first line of input, but we didn't want to put additional logic in the map function, and then we have the learning rate alpha.

We start by initializing the separating plane and defining the logistic function. As before, those variables will be used inside the map function, that is they will travel across interpreter and processor and network barriers to be available where the developer needs them and where a traditional, meaning sequential, R developer expects them to be available according to scope rules — 0 fuss and familiar, powerful behavior.

Then we have the main loop where computing the gradient of the loss function is the duty of a map reduce job, whose output is brought straight into main memory with an from.dfs — there are temp files being created and destroyed behind the scenes but you don't need to know. The only important thing you need to know is that the gradient is going to fit in memory so we can do a

Introduction to the vectorized and structured options in rmr 1.3

Goals for 1.3

The main goal for 1.3 was to do a performance review and implement changes to eliminate the main performance bottlenecks. Following the review, we determined that better support for a vectorized programming style was necessary to allow writing efficient R and hence efficient rmr programs in the case of small record size — for large record size the vectorization can happen at the record level. We also concluded that the native format parser was unacceptably slow for small objects and other parsers could benefit from a vectorized version.

We selected a number of very basic use cases to exercise different aspects of the library and the focus was on speed gains on those cases while minimizing code changes. Since API changes were necessary, it was decided to tackle at the same time the issue of better supporting the structured data case, to try and make API changes as soon and as rarely as possible and to keep the vectorized and structured options consistent. The structured data features are mostly implemented but not necessarily with great attention to speed and backward incompatible changes are possible in the future. The goal was to try and get an expanded, consistent API to the users as quickly as possible as well speed-enhancing features.

Changes Overview

We have added a vectorized option to from.dfs, mapreduce and keyval. The precise syntax and semantics will be clear from the examples and is described in the R help() but the main idea is to instruct the library that the user intends to process or generate multiple key-value pairs in one call. We have also provided vectorized implementations for the typedbytes, CSV and text formats. We have also added a structured option to the same functions that in general means that key-value pairs are provided or expected by the user as data frames.

Data types

The main data type in rmr is they key-value pair, a two element list with some attributes generated with the function keyal. Any R object can take the role of key or value (but see Caveats below). Collections of key-value pairs can be represented as lists thereof. This is a row-first representation that is close to how Hadoop does things, but not very natural or efficient in R. Therefore we support an alternative, the *vectorized* key-value pair, which is created by calling keyval with the vectorized option set to TRUE. The arguments passed to keyval represent not one but many keys or values and need to have the same length or the same number of rows if this is defined. In the unstructured case, keys and values can be provided to keyval as lists, otherwise atomic vectors or data frames are also accepted, and we plan to extend support for other data types (matrices in particular). These data types are accepted by to.dfs and generated by from.dfs, controlled by the options structured and vectorized, and they are also used as arguments and return values for the map and reduce functions, which are arguments to mapreduce.

Caveats

Because of the inefficiency of the native R serialization on small objects, even using a C implementation, we decided to switch automatically to a different serialization format, typedbytes, when the vectorized option is on. This means that users are not going to enjoy the same kind of absolute compatibility with R, but this usually is not a huge drawback when dealing with small records, typically scalars. For example, if each record contains a 100x100 submatrix of a larger matrix, the vectorized API doesn't support matrices very well but it's also not going to give a speed boost. If one is processing graphs and each record is a just an integer pair (start, stop), that's where the vectorized interface gives the biggest speed boost and typedbytes serialization is adequate for simple records. There may be “in between” cases, for instance when keys or values are very small matrices, where neither option is ideal. In consideration of that, in the future we may expand typedbytes with additional types to be more R-friendly.

Examples

First let's create some input. Input size is arbitrary but it is the one used in the automated tests included in the package and to obtain the timings. Here we are assuming a "hadoop" backend. By setting vectorized to TRUE we instruct keyval to consider its arguments collections of keys and values, not individual ones. At the same time to.dfs automatically switches to a simplified serialization format. The nice thing is that we don't have to remember if a data set was written out with one or the other serialization, as they are compatible on the read side. For testing purposes we need to create a data set in this format to maximize the benefits of the vectorized option.

```
input.size = if(rmr.options.get('backend') == "local") 10^4 else 10^6
data = keyval(rep(list(1), input.size),as.list(1:input.size), vectorized = TRUE)
input = to.dfs(data)
```

from.dfs to get it without exceeding our resources.

This map reduce job has the input we specified in the first place, the training data.

The map function simply computes the contribution of an individual point to the gradient. Please note the variables g and plane making their necessary appearance here without any work on the developer's part. The access here is read only but you could even modify them if you wanted — the semantics is copy on assign, which is consistent with how R works and easily supported by hadoop. Since in the next step we just want to add everything together, we return a dummy, constant key for each record.

The reduce function, besides the usual R fidgeting to get numbers out of lists, is just a big sum. As far as the fidgeting, we decided to support the more general lists first and then add some API convenience for the cases where a data frame would suffice, which is a special case but a very important one. The conversion is not as general as we would like it to be but you can try it (reduce.on.data.frame = T).

Since we have only one key, all the work will fall on one reducer and that's not scalable, so in this example it's important to activate the combiner, in this case it's TRUE so it's the same as the reducer. Since sums are associative and commutative that's all we need. We also support a distinct combiner function.

After the map reduce job is complete and from.dfs has copied the only record that is supposedly produced by this job into the gradient variable, we just have to upgrade the separating plane and return it after the iterations are complete.

To make this example production-level there are several things one needs to do, like having a convergence criterion instead of a fixed iteration number an an adaptive learning rate, but probably gradient descent just requires too many iterations to be the right approach in a big data context. But this example should give you all the elements to be able to implement, say, conjugate gradient instead. In general, when each iteration requires I/O of a large data set, the number of iterations needs to be modest and algorithms with O(log(N)) number of iterations are natural candidates, even if the work in each iteration may be more substantial.

K-means

We are now going to cover a simple but significant clustering algorithm and the complexity will go up just a little bit. To cheer yourself up, you can take a look at this alternative implementation which requires three languages, python, pig and java, to get the job done and is hailed as a model of simplicity.

We are talking about k-means and we are going to implement it in two parts: a function that controls the iterations and termination of the algorithm and one, essentially a mapreduce job, that does the grunt work of computing distances and electing centers. This is not a production ready implementation, but should be illustrative of the power of this package. You simply can not do this in pig or hive alone and it would take hundreds of lines of code in java.

```
kmeans =
  function(points, ncenters, iterations = 10, distfun = NULL,
           plot = FALSE, fast = F) {
    if(is.null(distfun))
      distfun =
        if (!fast) function(a,b) norm(as.matrix(a-b), type = 'F')
        else fast.dist
    if (fast) kmeans.iter = kmeans.iter.fast
    newCenters =
      kmeans.iter(
        points,
        distfun,
        ncenters = ncenters)
      newCenters = kmeans.iter(points, distfun, centers = newCenters)}
    newCenters}
```

This is the controlling program. Most of its lines are executed locally, meaning in the same interpreter where the kmeans function is called. We define a function that takes a set of points, a desired number of centers, a number of iterations (a convergence test would be better, just for illustration purposes) and a distance function defaulting to euclidean distance.

This function uses another function, kmeans.iter which implements one iteration of kmeans. It can be called in two flavors. In the first call, only the number of centers is specified. The function returns a new set of centers.

Now let's look at the main loop. It gets executed a user-set number of times, for simplicity's sake, but implementing a convergence criterion should be easy. Its body is just a call to the other flavor of kmeans.iter, where a set of centers rather than just the number of centers is provided. and the return value is, as before, a new set of centers. Let's now look at kmeans.iter, the real workhorse.

```
kmeans.iter =
  function(points, distfun, ncenters = dim(centers)[1], centers = NULL) {
    from.dfs(mapreduce(input = points,
                       map =
                         if (is.null(centers)) {
                           function(k,v) keyval(sample(1:ncenters,1),v)}
                         else {
                           function(k,v) {
                             distances = apply(centers, 1, function(c) distfun(c,v))
                             keyval(centers[which.min(distances),], v)}},
                       reduce = function(k,vv) keyval(NULL, apply(do.call(rbind, vv), 2, mean))),
           structured = T)}
```

This is one iteration of kmeans, implemented as a map reduce job. We are already familiar with all the parameters to this

- This document responds to several inquiries on data formats and how to get data in and out of the rmr system
- Still more a collection of snippets than anything organized
- Thanks Damien for the examples and Koert for conversations on the subject

Internally rmr uses R's own serialization in most cases and typedbytes serialization when in vectorized mode. The goal is to make you forget about representation issues most of the time. But what happens at the boundary of the system, when you need to get non-rmr data in and out of it? Of course rmr has to be able to read and write a variety of formats to be of any use. This is what is available and how to extend it.

Built in formats

The complete list is:

```
[1] "text"              "json"              "csv"
[4] "native"            "native.text"       "sequence.typedbytes"
```

1. text: for english text. key is NULL and value is a string, one per line. Please don't use it for anything else.
2. json-ish: it is actually so that streaming can tell key and value. This implies you have to escape all newlines and tabs in the JSON part. Your data may not be in this form, but almost any language has decent JSON libraries. It was the default in rmr 1.0, but we'll keep because it is almost standard. Parsed in C for efficiency, should handle large objects.
3. csv: A family of concrete formats modeled after R's own read.table. See examples below.
4. native: based on R's own serialization, it is the default and supports everything that R's serialize supports. If you want to know the gory details, it is implemented as an application specific type for the typedbytes format, which is further encapsulated in the sequence file format when writing to HDFS, which … Dont't worry about it, it just works. Unfortunately, it is written and read by only one package, rmr itself.
5. native.text: a text version of native, default in 1.1, it is now deprecated. Convert your rmr 1.1 data quick and move on.
6. sequence.typedbytes: based on specs in HADOOP-1722 it has emerged as the standard for non Java hadoop application talking to the rest of Hadoop.

Custom formats

A format is a triple. You can create one with make.input.format, for instance:

```
make.input.format("csv")
```

```
$mode
[1] "text"

$format
function (con, nrecs)
{
    df = tryCatch(read.table(file = con, nrows = nrecs, header = FALSE,
        ...), error = function(e) NULL)
    if (is.null(df) || dim(df)[[1]] == 0)
        NULL
    else keyval(NULL, df, vectorized = nrecs > 1)
}
<environment: 0x104dd0000>

$streaming.format
NULL
```

The mode element can be text or binary. The format element is a function that takes a connection, reads nrows records and creates a key-value pair. The streaming.format element is a fully qualified Java class (as a string) that writes to the connection the format function reads from. The default is TextInputFormat and also useful is org.apache.hadoop.streaming.AutoInputFormat. Once you have these three elements you can pass them to make.input.format and get something out that can be used as the input.format option to mapreduce and the format option to from.dfs. On the output side the situation is reversed with the R function acting first and then the Java class doing its thing.

```
make.output.format("csv")
```

```
$mode
[1] "text"

$format
function (k, v, con, vectorized)
write.table(file = con, x = if (is.null(k)) v else cbind(k, v),
    ..., row.names = FALSE, col.names = FALSE)
<environment: 0x10483a028>
```

function and we know what from.dfs does, in this case it moves new cluster centers computed by a mapreduce job and stored in HDFS into main memory. Here we are assuming we have a lot of points, as in big data, but a small number of centers, as in they fit in RAM. It's a common case but it might not cover all applications.

Next is the call actually performing a map reduce job. Its input is a set of points, as an HDFS path or big data object. We have two cases for the map function, one for the initialization, that is when there isn't a set of current cluster centers, only a desired number, and one for all the other calls from the main loop. In the former we just assign a random center to each point and generate a key value pair with the center as key and the point as value. That means, in the reduce stage, we are guaranteed that all points with the same center will end up together in the same reduce call. Of course that could mean that some reducers have an unacceptable amount of work to perform, but a simple modification of this program and the use of a combiner fixes that problem, so we are going to ignore the problem in this tutorial. In the latter we compute all the distances from a point to each center and return a key value pair that has the closest center as key and the point itself as value

To perform a sample run, we need some data. We can create it very easily from the R prompt:

```
input = to.dfs(lapply(1:1000, function(i) keyval(NULL, c(rnorm(1, mean = i%%3, sd = 0.1),
                                                         rnorm(1, mean = i%%4, sd = 0.1)))))
```

That is, create some arbitrary data in the form of a list of key value pairs, the key here doesn't really matter and write it out to hdfs.
And this is a simple test of what we've just implemented,

```
kmeans(input, 12, iterations = 5)
```

With a little extra work you can even get pretty visualizations like this one.

Linear Least Squares

We are going to build another example, LLS, that illustrates how to build map reduce reusable abstractions and how to combine them to solve a larger task. We want to solve LLS under the assumption that we have too many data points to fit in memory but not such a huge number of variables that we need to implement the whole process as map reduce job. This is sort of a hybrid solution that is made particularly easy by the seamless integration of rmr with R and an example of a pragmatic approach to big data. If we have operations A, B, and C in a cascade and the data sizes decrease at each step and we have already an in-memory solution to it, than we might get away by replacing only the first step with a big data solution and then continuing with tried and true function and pacakges. To make this as easy as possible, we need the in memory and big data worlds to integrate easily.

This is the basic equation we want to solve in the least square sense:

X β = y

We are going to do it by using the function solve as in the following expression, that is solving the normal equations.

```
solve(t(X)%*%X, t(X)%*%y)
```

But let's assume that X is too big to fit in memory, so we have to compute the transpose and matrix products using map reduce, then we can do the solve as usual on the results. This is our general plan.

We are going to adopt the following representation for matrices, here in data frame form (behind the scenes it's still lists).

```
  key1 key2       val

1    1    1  0.7595035
2    1    2  1.1731512
3    1    3  0.2112339
4    2    1  0.2305024
5    2    2 -0.5277821
6    2    3 -2.4413680
7    3    1 -0.8510856
```

The key is the pair row, col and the value is the matrix element. In practice this representation makes sense only for sparse matrices, so in a real world implementation we might want to use a representation with a submatrix for each record, but this is simpler to develop the ideas.

We start implementing the transpose with a tiny auxiliary function that swaps the elements of a two element list. You guessed right that this is going to be used to swap the raw index with the column index.

```
swap = function(x) list(x[[2]], x[[1]])
```

Then we define the map function for the transpose map reduce job. It uses a higher order function, to.map, to turn two ordinary functions into a map. This is possible because they act independently on the key and on the value. What this says is: swap the elements of the key and let the value through. We could have written it just as easily without to.map, but once you are familiar with it it is more readable this way.

Then we have the map reduce transpose job which is abstracted into a function transpose, that we can use like any other job from now on.

```
transpose = function(input, output = NULL){
```

```
  mapreduce(input = input, output = output, map = transpose.map)
}
```

It takes an input, an optional output and returns the return value of the map reduce job. It passes input and output to it and the map function we've just defined and that's all.

Detour: Relational Joins

Now we would like to tackle matrix multiplication but we need a short detour first. This takes one step further in hadoop mastery as we need to combine and process two files into one map reduce job. By default mapreduce supports merging two inputs the way hadoop does, that is once can specify multiple inputs and the only guarantee is that every record will go through one mapper. No order or grouping of any sort is guaranteed as the mappers are processing the input files.

What we need here is a very orderly merging so that we can multiply matrix elements that share an index and then sum them together. It actually looks like a join on one specific index. It turns out that joins are a very important subtask in many mapreduce algorithms and are more or less supported in a number of hadoop dialects. A generalized equijoin is part of the rmr API:

```
equijoin(
  left.input = NULL,
  right.input = NULL,
  input = NULL,
  output = NULL,
  outer = c("", "left", "right", "full"),
  map.left = to.map(identity),
  map.right = to.map(identity),
  reduce = function(k, values.left, values.right) do.call(c, lapply(values.left, function(vl)lapply(values.right, function(vr) reduce.all(k, vl, vr)))),
  reduce.all = function(k, vl, vr) keyval(k, list(left = vl, right = vr)))
```

Instead of a single input, we have a left.input and right.input, as joins normally do, but in case we want to perform a self join, we can skip the first two arguments and specify only the third, input.

Then we have an output, optional as usual, and we can specify different flavors of join such as in left outer, right outer or full outer as usual.

Now to the interesting bits. This function is a bit relational join and a bit map reduce job. Instead of specifying join keys, we specify two separate map functions, one for the left input and one for the right input. Map functions, as usual, produce a key and a value. The join will be an equijoin on the keys. For each pair of matching records there will be a shared key and two values, one coming from each side. By default, we have simple pass-throguh or identity mappers.

The reduce can be specified as usual, but an alternate interface is offered with reduce.all which is more SQL-like in that it is a join without a group by on the join key, whereas the reduce form implies a group by. This is a little advanced in a number of ways and also very reusable, so we made it part of the library even if it is built on top of mapreduce. No we are going to see its application to perform a matrix multiplication.

Linear Least Squares (continued)

Back to our matrix multiplication task, which we will implement as an application of the equijoin just shown.

$a_{ij} = \Sigma_k b_{ik} c_{kj}$

We first define the map function. It comes in two flavors, wether you want to join on the column index or on the row index, and in a matrix multiplication we need both. So here is a higher order function that generates both maps. It just produces a key-value pair with as key the desired index and as value a list with all the information, row, column and element, which we will need later on.

```
mat.mult.map = function(i) function(k,v) keyval(k[[i]], list(pos = k, elem = v))
```

And finally to the actual matrix multiplication. It is implemented as the composition of two jobs. One does the multiplications and the other the sums. There are other ways of implementing it but this is the most straightforward.

```
mat.mult = function(left, right, result = NULL) {
  mapreduce(
            input =
            equijoin(left.input = left, right.input = right,
                     map.left = mat.mult.map(2),
                     map.right = mat.mult.map(1),
                     reduce = function(k, vvl, vvr)
                       do.call(c, lapply(vvl, function(vl)
                         lapply(vvr, function(vr) keyval(c(vl$pos[[1]], vr$pos[[2]]), vl$elem*vr$elem))))),
            output = result,
            reduce = to.reduce(identity, function(x) sum(unlist(x))))}
```

The first step is a join on the the column index for the left side and the row index from the right side, so that we bring together pairs of the form (b_{ik}, c_{kj}) . In the reduce we perform the multiplication and return a record with a key of (i,j) and a value equal to the multiplication.

The following or outer mapreduce doesn't have an explicit map, that means it defaults to the pass-through one. The interesting thing is that, by default, the grouping will happen on the (i,j) pair, therefore grouping all the right products that need to be

summed together.

We now have all the elements in place to solve LLS: mapreduce transpose and matrix multiplication and old fashioned solve().

We need a simple function to turn matrices represented in list form and sparse format, that we use with mapreduce, into regular dense in memory R matrices. We rely on a feature of from.dfs that turns lists into data frames whenever possible so that this function just has to go from dataframe to dense matrix, using the Matrix package and sparseMatrix in particular.

```
to.matrix = function(df) as.matrix(sparseMatrix(i=df$key[,1], j=df$key[,2], x=df$val[,1]))
```

Then our sought after semi-big-data LLS solution

```
linear.least.squares = function(X,y) {
  Xt = transpose(X)
  XtX = from.dfs(mat.mult(Xt, X), to.data.frame = TRUE)
  Xty = from.dfs(mat.mult(Xt, y), to.data.frame = TRUE)
  solve(to.matrix(XtX),to.matrix(Xty))}
```

We start with a transpose, compute the normal equations, left and right side, and call solve on the converted data.

What we have learned

We will summarize here a few ways in which we have used the functions in the library.

Specify mapreduce jobs using familiar R functions:

```
mapreduce(..., map = function(k,v)..., reduce = function(k,vv)...)
```

Compose jobs like functions

```
mapreduce(mapreduce(...
```

store results where you want

```
mapreduce(..., output = "my-result-file", ...)
```

or in a variable

```
my.result = mapreduce(...)
```

create abstractions

```
my.job = function(x,y,z) { .... out = mapreduce(...);  ... out}
```

describe any data flow

```
out1 = my.job1(my.result); out2 = my.job2(my.result); merge.job(out1, out2)
```

move things in an out of memory and HDFS to create hybrid big-small-data algorithms, control flow and iteration, display results etc

```
if(length(from.dfs(my.job(...)))==0){...} else {...}

ggplot2(from.dfs(my.job(...)), ...)
```

Related Links

- Comparison of high level languages for mapreduce: k means
- Fast k means
- Changelog
- Design philosophy
- Efficient rmr techniques
- Writing composable mapreduce jobs
- Use cases
- Getting data in and out
- FAQ

Introduction to the vectorized and structured options in rmr 1.3

Goals for 1.3

The main goal for 1.3 was to do a performance review and implement changes to eliminate the main performance bottlenecks. Following the review, we determined that better support for a vectorized programming style was necessary to allow writing efficient R and hence efficient rmr programs in the case of small record size — for large record size the vectorization can happen at the record level. We also concluded that the native format parser was unacceptably slow for small objects and other parsers could benefit from a vectorized version.

We selected a number of very basic use cases to exercise different aspects of the library and the focus was on speed gains on those cases while minimizing code changes. Since API changes were necessary, it was decided to tackle at the same time the issue of better supporting the structured data case, to try and make API changes as soon and as rarely as possible and to keep the vectorized and structured options consistent. The structured data features are mostly implemented but not necessarily with great attention to speed and backward incompatible changes are possible in the future. The goal was to try and get an expanded, consistent API to the users as quickly as possible as well speed-enhancing features.

Changes Overview

We have added a vectorized option to from.dfs, mapreduce and keyval. The precise syntax and semantics will be clear from the examples and is described in the R help() but the main idea is to instruct the library that the user intends to process or generate multiple key-value pairs in one call. We have also provided vectorized implementations for the typedbytes, CSV and text formats. We have also added a structured option to the same functions that in general means that key-value pairs are provided or expected by the user as data frames.

Data types

The main data type in rmr is they key-value pair, a two element list with some attributes generated with the function keyal. Any R object can take the role of key or value (but see Caveats below). Collections of key-value pairs can be represented as lists thereof. This is a row-first representation that is close to how Hadoop does things, but not very natural or efficient in R. Therefore we support an alternative, the *vectorized* key-value pair, which is created by calling keyval with the vectorized option set to TRUE. The arguments passed to keyval represent not one but many keys or values and need to have the same length or the same number of rows if this is defined. In the unstructured case, keys and values can be provided to keyval as lists, otherwise atomic vectors or data frames are also accepted, and we plan to extend support for other data types (matrices in particular). These data types are accepted by to.dfs and generated by from.dfs, controlled by the options structured and vectorized, and they are also used as arguments and return values for the map and reduce functions, which are arguments to mapreduce.

Caveats

Because of the inefficiency of the native R serialization on small objects, even using a C implementation, we decided to switch automatically to a different serialization format, typedbytes, when the vectorized option is on. This means that users are not going to enjoy the same kind of absolute compatibility with R, but this usually is not a huge drawback when dealing with small records, typically scalars. For example, if each record contains a 100x100 submatrix of a larger matrix, the vectorized API doesn't support matrices very well but it's also not going to give a speed boost. If one is processing graphs and each record is a just an integer pair (start, stop), that's where the vectorized interface gives the biggest speed boost and typedbytes serialization is adequate for simple records. There may be “in between” cases, for instance when keys or values are very small matrices, where neither option is ideal. In consideration of that, in the future we may expand typedbytes with additional types to be more R-friendly.

Examples

First let's create some input. Input size is arbitrary but it is the one used in the automated tests included in the package and to obtain the timings. Here we are assuming a "hadoop" backend. By setting vectorized to TRUE we instruct keyval to consider its arguments collections of keys and values, not individual ones. At the same time to.dfs automatically switches to a simplified serialization format. The nice thing is that we don't have to remember if a data set was written out with one or the other serialization, as they are compatible on the read side. For testing purposes we need to create a data set in this format to maximize the benefits of the vectorized option.

```
input.size = if(rmr.options.get('backend') == "local") 10^4 else 10^6
data = keyval(rep(list(1), input.size),as.list(1:input.size), vectorized = TRUE)
input = to.dfs(data)
```

Read it back

The simplest possible task is to read back what we just wrote out, and we know how to do that already.

```
from.dfs(input)
```

In the next code block we switch on vectorization. That is, instead of reading in a list of key-value pairs we are going to have list of vectorized key-value pairs, that is every pair contains a list of keys and a list of values of the same length. The vectorized argument can be set to an integer as well for more precise control of how many keys and values should be stored in a key-value pair. With this change alone, from my limited, standalone mode testing we have achieved an almost 7X speed-up (raw timing data is in vectorized-API.R)

```
from.dfs(input, vectorized = TRUE)
```

Pass-through

Next on the complexity scale, or lack thereof, is the pass-through or identity map-reduce job. First in its plain-vanilla incarnation, same as rmr-1.2.

```
mapreduce(input, map = function(k,v) keyval(k,v))
```

Next we turn on vectorization. Vectorization can be turned on independently on the map and reduce side but the reduce side is not implemented yet, nor it is completely clear what the equivalent should be; we are going to let real use cases accumulate before working on vectorization on the reduce side. By tuning on map-side vectorization, the arguments k and v are going to be equal-sized lists of key and values. To write them back as they are, we need to also turn on vectorization in the keyval function, lest we make one key out of many or as an alternative to using inefficient apply family calls. The speed up here is about 4X.

```
mapreduce(input,
        map = function(k,v) keyval(k,v, vectorized = TRUE),
        vectorized = list(map = TRUE))
```

General Filter

Let's now look at a very simple but potentially useful example, filtering. We have a function returning logical, for instance the following:

```
predicate = function(k,v) unlist(v)%%2 == 0
```

with the goal of dropping all records for which predicate returns FALSE. This particular one is handy for this document because it works equally well for the vectorized and unvectorized cases. The plain-vanilla implementation is the following:

```
mapreduce(input,
        map = function(k,v) if(predicate(k,v)) keyval(k,v))
```

For the vectorized version, we want to switch on vectorization for the map function and in keyval at the same time. Remember, this is not always the case, it makes sense when we have small records in and small records out. The speedup here is about 3X.

```
mapreduce(input,
        map = function(k,v) {filter = predicate(k,v);
                keyval(k[filter], v[filter], vectorized = TRUE)},
        vectorized = list(map = TRUE))
```

This is the first case where we can show what changes by turning on the structured option. On the map side it makes sense only in conjunction with vectorized or otherwise with only one row of data there isn't much to turn into a data frame. On the reduce side the structured option is equivalent to the now deprecated reduce.on.data.frame, which turns the second argument of the reduce function into a data frame, before it is evaluated. In this case we have a single column data frame, so it is not particularly interesting, but it is possible.

```
mapreduce(input,
        map = function(k,v) {filter = predicate(k,v);
                keyval(k[filter,1], v[filter,1], vectorized = TRUE)},
        vectorized = list(map = TRUE),
        structured = list(map = TRUE))
```

Select Columns

In this example we want to select specific elements of each record or columns, if we are dealing with structured data. We need to generate slightly more complex input data so that this operation makes sense.

```
input.select = to.dfs(keyval(1:input.size,
                             replicate(input.size, list(a=1,b=2,c=3),
                                       simplify=FALSE), vectorized=TRUE))
```

The selection function takes slightly different forms for the unvectorized and vectorized cases. Here we are picking the second column just for illustration purposes.

```
select = function(v) v[[2]]
```

```
select.vec = function(v) do.call(rbind,v)[,2] #names not preserved with current impl. of typedbytes
```

In the latter case we do not have an option to refer to the column by name, as in do.call(rbind,v)[,'b']. Unfortunately names are not preserved by the simplified serialization method used when vectorization is on, a shortcoming we plan to address in the future.
In the structured case we don't even need a function to perform a column selection, just an index number.

```
field = 2
```

As usual, we start showing the plain-vanilla implementation. Nothing major to report here.

```
mapreduce(input.select,
          map = function(k,v) keyval(k, select(v)))
```

In the vectorized version, we turn on vectorization both in input and output and we switch to a vectorized selection function. The speed up is 5X.

```
mapreduce(input.select,
          map = function(k,v) keyval(k, select.vec(v), vectorized = TRUE),
          vectorized = list(map = TRUE))
```

As selecting a column is a typical operation that is well defined and natural on structured data, in the structured version we don't even need a selection function, just a column number. Unfortunately column names are lost in the current implementation, something we would like to address in the future.

```
mapreduce(input = input.select,
          map = function(k,v) keyval(k[,1], v[,field], vectorized = TRUE),
          vectorized = list(map = TRUE),
          structured = list(map = TRUE))
```

Big Sum

We now move on to the first example including a reduce. It's an extreme case of data reduction as our only goal is to perform a large sum. Let's start by generating some data.

```
input.bigsum = to.dfs(keyval(rep(1, input.size), rnorm(input.size), vectorized=TRUE))
```

This the plain-vanilla implementation. Turning on the combiner when possible is always recommended, but in this case it is mandatory: without it the reduce process would likely run out of memory.

```
mapreduce(input.bigsum,
          map  = function(k,v) keyval(1,v),
          reduce = function(k, vv) keyval(k, sum(unlist(vv))),
          combine = TRUE)
```

In its vectorized form, this program applies an additional trick, which is to start summing in the map function, which becomes an early reduce of sorts. In fact, it would be possible to use the same function for both map and reduce. Like a combine, this early reduction happens locally near the data but, in addition, it doesn't require the data to be serialized and unserialized in between. It's an extreme application of the *mantram* “reduce early, reduce often”, for a speed gain in excess of 6X.

```
mapreduce(input.bigsum,
          map  = function(k,v) keyval(1,sum(unlist(v)), vectorized = TRUE),
          reduce = function(k, vv) keyval(k, sum(unlist(vv))),
```

```
        combine = TRUE,
        vectorized = list(map = TRUE))
```

In the structured version we rely on the implicit conversion to data frame to save a couple of unlist calls, for a cleaner look.

```
mapreduce(input.bigsum,
        map  = function(k,v) keyval(1, sum(v), vectorized = TRUE),
        reduce = function(k, vv) keyval(k, sum(vv)) ,
        combine = TRUE,
        vectorized = list(map = TRUE),
        structured = TRUE)
```

Group and Aggregate

This is an example of a more realistic aggregation on a user-defined number of groups expressed in a more generic form with group and aggregate functions. First let's generate some data.

```
input.ga = to.dfs(keyval(1:input.size, rnorm(input.size), vectorized=TRUE))
```

Then pick specific group and aggregate functions to make the example fully specified and runnable. For simplicity's sake, these are written to work in all cases, plain, vectorized and structured. Some further optimizations are possible.

```
group = function(k,v) unlist(k)%%100
aggregate = function(x) sum(unlist(x))
```

What this means is that we are again calculating sums of numbers, but this time we are going to have a separate sum for each of 100 different groups. Let's start with the plain vanilla one.

```
mapreduce(input.ga,
        map = function(k,v) keyval(group(k,v), v),
        reduce = function(k, vv) keyval(k, aggregate(vv)),
        combine = TRUE)
```

In the vectorized version we could again apply the trick of in-map aggregation, but it wouldn't buy us as much as in the previous example. The speedup here is 2.5X

```
mapreduce(input.ga,
        map = function(k,v) keyval(group(k,v), v, vectorized = TRUE),
        reduce = function(k, vv) keyval(k, aggregate(vv)),
        combine = TRUE,
        vectorized = list(map = TRUE))
```

Finally, the structured version to complete these test cases.

```
mapreduce(input.ga,
        map = function(k,v) keyval(group(k,v), v[,1], vectorized = TRUE),
        reduce = function(k, vv) keyval(k, aggregate(vv)),
        vectorized = list(map = TRUE),
        structured = TRUE)
```

What's new

Major changes

- A new vectorized API supports a vectorized programming style, which can achieve C-like speeds when followed consistently. The old API is still the default and has undergone only cosmetic changes, but when efficiency is important and with small record sizes using the vectorized interface is recommended. On the other hand the old API allows simpler programs and has better compatibility with complex R objects used as keys and values. See the Introduction to the vectorized API for more information.
- In support of the vectorized API, there are completely new C implementations of serialization and deserialization from and to the typedbytes serialization format. This format is also important to exchange data with other members of the Hadoop system as typedbytes is the preferred serialization format for non-java streaming applications (HADOOP-1722). The implementation supports the most common and useful cases for R users but is not yet fully adherent to the specification yet.
- All formats are compatible with the new vectorized API but in particular “text” and “csv” readers and writers are substantially faster and the “csv” reader is even more stable when using the vectorized API.
- The support for structured data, while still evolving, has been extended to the map phase when using the vectorized API. that is we can have multiple records parsed at once and passed to the map function as a data frame. We plan to eventually have an “uninterrupted structured path” from input to output whereby the user would only manipulate data frames and costly conversions from and to data frames would be avoided. Issue #102 has been created to gather ideas and track progress on this.

Minor changes

- Updated whirr scripts run R 2.14, can optionally use lzo compression and work around unavaialbility of some packages from CRAN
- Packages are loaded on the cluster nodes with require to avoid failure when not available, nontheless your code will fail if such packages are needed by the map and reduce functions. The tradeoff here is that one doesn't have to detach packages before calling mapreduce when those packages are neither needed nor available on the nodes but the errors when packages are needed will be less intuitive (but a warning will be in stderr before the error, which should help)
- Fixed a bug (#111) whereby from.dfs would fail when tmp dir and destination are on two different volumes when the local backend is active (thanks Saar).

Read it back

The simplest possible task is to read back what we just wrote out, and we know how to do that already.

```
from.dfs(input)
```

In the next code block we switch on vectorization. That is, instead of reading in a list of key-value pairs we are going to have list of vectorized key-value pairs, that is every pair contains a list of keys and a list of values of the same length. The vectorized argument can be set to an integer as well for more precise control of how many keys and values should be stored in a key-value pair. With this change alone, from my limited, standalone mode testing we have achieved an almost 7X speed-up (raw timing data is in vectorized-API.R)

```
from.dfs(input, vectorized = TRUE)
```

Pass-through

Next on the complexity scale, or lack thereof, is the pass-through or identity map-reduce job. First in its plain-vanilla incarnation, same as rmr-1.2.

```
mapreduce(input, map = function(k,v) keyval(k,v))
```

Next we turn on vectorization. Vectorization can be turned on independently on the map and reduce side but the reduce side is not implemented yet, nor it is completely clear what the equivalent should be; we are going to let real use cases accumulate before working on vectorization on the reduce side. By tuning on map-side vectorization, the arguments k and v are going to be equal-sized lists of key and values. To write them back as they are, we need to also turn on vectorization in the keyval function, lest we make one key out of many or as an alternative to using inefficient apply family calls. The speed up here is about 4X.

```
mapreduce(input,
        map = function(k,v) keyval(k,v, vectorized = TRUE),
        vectorized = list(map = TRUE))
```

General Filter

Let's now look at a very simple but potentially useful example, filtering. We have a function returning logical, for instance the following:

```
predicate = function(k,v) unlist(v)%%2 == 0
```

with the goal of dropping all records for which predicate returns FALSE. This particular one is handy for this document because it works equally well for the vectorized and unvectorized cases. The plain-vanilla implementation is the following:

```
mapreduce(input,
        map = function(k,v) if(predicate(k,v)) keyval(k,v))
```

For the vectorized version, we want to switch on vectorization for the map function and in keyval at the same time. Remember, this is not always the case, it makes sense when we have small records in and small records out. The speedup here is about 3X.

```
mapreduce(input,
        map = function(k,v) {filter = predicate(k,v);
                keyval(k[filter], v[filter], vectorized = TRUE)},
        vectorized = list(map = TRUE))
```

This is the first case where we can show what changes by turning on the structured option. On the map side it makes sense only in conjunction with vectorized or otherwise with only one row of data there isn't much to turn into a data frame. On the reduce side the structured option is equivalent to the now deprecated reduce.on.data.frame, which turns the second argument of the reduce function into a data frame, before it is evaluated. In this case we have a single column data frame, so it is not particularly interesting, but it is possible.

```
mapreduce(input,
        map = function(k,v) {filter = predicate(k,v);
                keyval(k[filter,1], v[filter,1], vectorized = TRUE)},
        vectorized = list(map = TRUE),
        structured = list(map = TRUE))
```

Select Columns

In this example we want to select specific elements of each record or columns, if we are dealing with structured data. We need to generate slightly more complex input data so that this operation makes sense.

```
input.select = to.dfs(keyval(1:input.size,
                             replicate(input.size, list(a=1,b=2,c=3),
                                       simplify=FALSE), vectorized=TRUE))
```

The selection function takes slightly different forms for the unvectorized and vectorized cases. Here we are picking the second column just for illustration purposes.

```
select = function(v) v[[2]]
```

```
select.vec = function(v) do.call(rbind,v)[,2] #names not preserved with current impl. of typedbytes
```

In the latter case we do not have an option to refer to the column by name, as in do.call(rbind,v)[,'b']. Unfortunately names are not preserved by the simplified serialization method used when vectorization is on, a shortcoming we plan to address in the future.
In the structured case we don't even need a function to perform a column selection, just an index number.

```
field = 2
```

As usual, we start showing the plain-vanilla implementation. Nothing major to report here.

```
mapreduce(input.select,
          map = function(k,v) keyval(k, select(v)))
```

In the vectorized version, we turn on vectorization both in input and output and we switch to a vectorized selection function. The speed up is 5X.

```
mapreduce(input.select,
          map = function(k,v) keyval(k, select.vec(v), vectorized = TRUE),
          vectorized = list(map = TRUE))
```

As selecting a column is a typical operation that is well defined and natural on structured data, in the structured version we don't even need a selection function, just a column number. Unfortunately column names are lost in the current implementation, something we would like to address in the future.

```
mapreduce(input = input.select,
          map = function(k,v) keyval(k[,1], v[,field], vectorized = TRUE),
          vectorized = list(map = TRUE),
          structured = list(map = TRUE))
```

Big Sum

We now move on to the first example including a reduce. It's an extreme case of data reduction as our only goal is to perform a large sum. Let's start by generating some data.

```
input.bigsum = to.dfs(keyval(rep(1, input.size), rnorm(input.size), vectorized=TRUE))
```

This the plain-vanilla implementation. Turning on the combiner when possible is always recommended, but in this case it is mandatory: without it the reduce process would likely run out of memory.

```
mapreduce(input.bigsum,
          map  = function(k,v) keyval(1,v),
          reduce = function(k, vv) keyval(k, sum(unlist(vv))),
          combine = TRUE)
```

In its vectorized form, this program applies an additional trick, which is to start summing in the map function, which becomes an early reduce of sorts. In fact, it would be possible to use the same function for both map and reduce. Like a combine, this early reduction happens locally near the data but, in addition, it doesn't require the data to be serialized and unserialized in between. It's an extreme application of the *mantram* “reduce early, reduce often”, for a speed gain in excess of 6X.

```
mapreduce(input.bigsum,
          map  = function(k,v) keyval(1,sum(unlist(v)), vectorized = TRUE),
          reduce = function(k, vv) keyval(k, sum(unlist(vv))),
```

```
        combine = TRUE,
        vectorized = list(map = TRUE))
```

In the structured version we rely on the implicit conversion to data frame to save a couple of unlist calls, for a cleaner look.

```
mapreduce(input.bigsum,
        map  = function(k,v) keyval(1, sum(v), vectorized = TRUE),
        reduce = function(k, vv) keyval(k, sum(vv)) ,
        combine = TRUE,
        vectorized = list(map = TRUE),
        structured = TRUE)
```

Group and Aggregate

This is an example of a more realistic aggregation on a user-defined number of groups expressed in a more generic form with group and aggregate functions. First let's generate some data.

```
input.ga = to.dfs(keyval(1:input.size, rnorm(input.size), vectorized=TRUE))
```

Then pick specific group and aggregate functions to make the example fully specified and runnable. For simplicity's sake, these are written to work in all cases, plain, vectorized and structured. Some further optimizations are possible.

```
group = function(k,v) unlist(k)%%100
aggregate = function(x) sum(unlist(x))
```

What this means is that we are again calculating sums of numbers, but this time we are going to have a separate sum for each of 100 different groups. Let's start with the plain vanilla one.

```
mapreduce(input.ga,
        map = function(k,v) keyval(group(k,v), v),
        reduce = function(k, vv) keyval(k, aggregate(vv)),
        combine = TRUE)
```

In the vectorized version we could again apply the trick of in-map aggregation, but it wouldn't buy us as much as in the previous example. The speedup here is 2.5X

```
mapreduce(input.ga,
        map = function(k,v) keyval(group(k,v), v, vectorized = TRUE),
        reduce = function(k, vv) keyval(k, aggregate(vv)),
        combine = TRUE,
        vectorized = list(map = TRUE))
```

Finally, the structured version to complete these test cases.

```
mapreduce(input.ga,
        map = function(k,v) keyval(group(k,v), v[,1], vectorized = TRUE),
        reduce = function(k, vv) keyval(k, aggregate(vv)),
        vectorized = list(map = TRUE),
        structured = TRUE)
```

```
$streaming.format
NULL
```

R data types natively work without additional effort.

Put into HDFS:

my.data is coerced to a list and each element of a list becomes a record.

Compute a frequency of object lengths. Only require input, mapper, and reducer. Note that my.data is passed into the mapper, record by
record, as key = NULL, value = item.

```
result = mapreduce(
  input = hdfs.data,
  map = function(k,v) keyval(length(v), 1),
  reduce = function(k,vv) keyval(k, sum(unlist(vv))))

from.dfs(result)
```

However, if using data which was not generated with rmr (txt, csv, tsv, JSON, log files, etc) it is necessary to specify an input format.

There is a third option in between the simplicity of a string like “csv” and the full power of make.input.format, which is passing the format string to make.input.format with additional arguments that further specify the specific dialect of csv, as in make.input.format("csv", sep = ';'). csv is the only format offering this possibility as the others are fully specified and it takes the same options as read.table. The same on the output side with write.table being the model.

```
wordcount = function (input, output = NULL, pattern = " ") {
  mapreduce(input = input ,
          output = output,
          input.format = "text",
          map = function(k,v) {
                  lapply(
                    strsplit(
                            x = v,
                            split = pattern)[[1]],
                    function(w) keyval(w,1))},
             reduce = function(k,vv) {
                keyval(k, sum(unlist(vv)))},
             combine = T)}
```

To define your own input.format (e.g. to handle tsv):

```
tsv.reader = function(con, nrecs){
  lines = readLines(con, 1)
  if(length(lines) == 0)
    NULL
  else {
    delim = strsplit(lines, split = "\t")[[1]]
    keyval(delim[1], delim[-1])}} # first column is the key, note that column indexes moved by 1
```

Frequency count on input column two of the tsv data, data comes into map already delimited

```
freq.counts =
  mapreduce(
    input = tsv.data,
    input.format = tsv.format,
    map = function(k,v) keyval(v[[1]], 1),
    reduce = function(k,vv) keyval(k, sum(unlist(vv))))
```

Or if you want named columns, this would be specific to your data file

```
tsv.reader =
  function(con, nrecs){
    lines = readLines(con, 1)
    if(length(lines) == 0)
      NULL
    else {
      delim = strsplit(lines, split = "\t")[[1]]
      keyval(delim[[1]], list(location = delim[[2]], name = delim[[3]], value = delim[[4]]))}}
```

You can then use the list names to directly access your column of interest for manipulations

---

```
freq.counts =
  mapreduce(
    input = tsv.data,
    input.format = tsv.format,
    map =
      function(k, v) {
        if (v$name == "blarg"){
          keyval(k, log(v$value))}},
    reduce = function(k, vv) keyval(k, mean(unlist(vv))))
```

To get your data out - say you input file, apply column transformations, add columns, and want to output a new csv file
Just like input.format, one must define a textoutputformat

```
csv.writer = function(k, v){
  paste(k, paste(v, collapse = ","), sep = ",")}
```

And then use that as an argument to make.output.format, but why sweat it since the devs have already done the work?

```
csv.format = make.output.format("csv", sep = ",")
```

This time providing output argument so one can extract from hdfs (cannot hdfs.get from a Rhadoop big data object)

```
mapreduce(
  input = hdfs.data,
  output = "/tmp/rhadoop/output/",
  output.format = csv.format,
  map = function(k,v){
    # complicated function here
    keyval(k,v)},
  reduce = function(k, vv) {
    #complicated function here
    keyval(k, vv[[1]])})
```

Compatibility testing for rmr 1.3 (Current stable)

Please contribute with additional reports. To claim compatibility you need to run R CMD check path-to-rmr successfully. As with any new release, testing on additional platforms is under way. If you build your own Hadoop, see Which Hadoop for rmr.

Hadoop	R	OS	Compatibility	Reporter
CDH3u4	2.14.1	Ubuntu 12.04	only x86_64	Revolution Analytics
mr1-cdh4.0.0	2.15.0	OS X 10.7.4	only x86_64	Revolution Analytics
mr2-cdh4.0.0	2.15.0	OS X 10.7.4	not compatible	Revolution Analytics

We have covered a basic k-means implementation with rmr in the Tutorial. If you tried it out, though, you probably have noticed that its performance leaves to be desired and wonder if anything can be done about it. Or your have read Efficient rmr techniques and would like to see those suggestions put to work beyond the toy "large sums" example used therein. Then this document should be of interest to you since we will cover an implementation that is modestly more complex and is two orders of magnitude faster. To make the most of it, it's recommended that you read the other two documents first.

First we need to reorganize our data representation a little, creating bulkier records that contain a sizeable subset of the data set each, as opposed to a single point. To this end, instead of storing one point per record we will store a matrix, with one data point per row of this matrix. We'll set the number of rows to 1000 which is enough to reap the benefits of using "vectorised" functions in R but not big enough to hit memory limits in most cases.

```
recsize = 1000
input = to.dfs(lapply(1:100,
                function(i) keyval(NULL, cbind(sample(0:2, recsize, replace = T) + rnorm(recsize, sd = .1),
                                               sample(0:3, recsize, replace = T) + rnorm(recsize, sd = .1)))))
```

This is how a sample call would look like, with the first argument being a sample dataset.

```
  kmeans(input, 12, iterations = 5, fast = T)}
```

This creates and processes a dataset with 100,000 data points, organized in 100 records. For a larger data set you would need to increase the number of records only, the size of each record can stay the same. As you may recall, the implementation of kmeans we described in the tutorial was organized in two functions, one containing the main iteration loop and the other computing distances and new centers. The good news is the first function can stay largely the same but for the addition of a flag that tells whether to use the optimized version of the "inner" function, so we don't need to cover it here (the code is in the source under tests, only in the dev branch for now) and a different default for the distance function — more on this soon. The important changes are in the kmeans.iter.fast function, which provides an alternative to the kmeans.iter function in the original implementation. Let's first discuss why we need a different default distance function, and in general why the distance function has a different signature in the fast version. One of the most CPU intensive tasks in this algorithm is computing distances between a candidate center and each data point. If we don't implement this one in an efficient way, we can't hope for an overall efficient implementation. Since it takes about a microsecond to call the simplest function in R (vs. ~10 nanoseconds in C), we need to get a significant amount of work done for each call. Therefore, instead of specifying the distance function as a function of two points, we will switch to a function of one point and and a set thereof that returns that distance between the first argument and each element of the second. In this implementation we will us a matrix instead of a set, since there are powerful primitives available to operate on matrices. The following is the default distance function with this new signature, where we can see that we avoided any explicit loops over the rows of the matrix yy. There are two implicit loops, Reduce and lapply, but internally they used vectorized operators, that is the overhead of those explicit loops is small compared to the time taken by the vectorised operators.

```
fast.dist = function(yy, x) { #compute all the distances between x and rows of yy
    squared.diffs = (t(t(yy) - x))^2
    ##sum the columns, take the root, loop on dimension
    sqrt(Reduce(`+`, lapply(1:dim(yy)[2], function(d) squared.diffs[,d])))}
```

With fast distance computation taken care of, at least for the euclidean case, let's look at the fast implementation of the kmeans iteration.

```
kmeans.iter.fast =
  function(points, distfun, ncenters = dim(centers)[1], centers = NULL) {
```

There is no news here as far as the signature but for a different distance default, so we can move on to the body. The following function is a conversion function that allows us to work around a limitation in the RJSONIO library we are using to serialize R objects. Unserializing a deserialized matrix returns a list of vectors, which we can easily turn into a matrix again. Whenever you have doubts whether the R object you intend to use as an argument or return value of a mapper or reducer will be encoded and decoded correctly, an option is to try RJSONIO::fromJSON(RJSONIO::toJSON(x)) where x is the object of interest. This a price to pay for using a language agnostic serialization scheme.

```
  list.to.matrix = function(l) do.call(rbind,l)
```

The next is the main mapreduce call, which, as in the Tutorial, can have two different map functions: let's look at each in detail.

```
  newCenters = from.dfs(
    mapreduce(
      input = points,
```

The first of the two map functions is used only for the first iteration, when no set of cluster centers is available, only a number, and randomly assigns each point to a center, just as in the Tutorial, but here the matrix argument v represents multiple data points and we need to assign each of them to a center efficiently. Moreover, we are going to switch from computing the means of data points in a cluster to computing their sums and counts, delaying taking the ratio of the two as much as possible. This way we can apply early data reduction as will be clear soon. To achieve this, the first step in the mapper is to extend the matrix

of points with a column of counts, all initialized to one. The next line assigns points to clusters using sample. This assignment is then supplied to the function by which applies a column sum to each group of rows in the matrix v of data points, as defined by being closest to the same center. This is where we apply the sum operation at the earliest possible stage — you can see it as an in-map combiner. Also, since the first column of the matrix is now devoted to counts, we are calculating those as well. In the last line, the only thing left is to generate a list of keyvalue pairs, one per center, and return it.

For all iterations after the first, the assignment of points to centers follows a min distance criterion. The first line back-converts v to a matrix whereas the second uses the aforementioned fast.dist function in combination with apply to generate a data points x centers matrix of distances. The next assignment, which takes a few lines, aims to compute the row by row min of this distance matrix and return the index of a column containing the minimum for each row. We can not use the customary function min to accomplish this as it returns only one number, hence we would need to call it for each data point. So we need to use its parallel, less known relative pmin and apply it to the columns of the distance matrix using the combination do.call and lapply. The output of this is a two column matrix where each row contains the index of a row and the column index of the min for that row. The following assignment sorts this matrix so that the results are in the same order as the v matrix. The last few steps are the same as for the first type of map and implement the early reduction we talked about.

In the reduce function, we simply sum over the colums of the matrix of points associated with the same cluster center. Actually, since we have started adding over subgroups of points in the mapper itself, what we are adding here are already partial sums and partial counts (which we store in the first column, as you may recall). Since this is an associative and commutative operation, it can only help to also switch the combiner on. There is on subtle change necessary to do so successfully, which required some debugging even for the author of this document, allegedly a mapreduce expert. In a first version, the reducer returned key value pairs with a NULL key. After all, the reduce phase happens after the shuffle phase, so what use are keys? Not so if the combiner is turned on, as records are first shuffled into the combiner and then re-shuffled into the reducer. So the reducer has to set a key, usually keeping the one set by the mapper (the reducer has to be key-idempotent for the combiner to work)

```
    reduce = function(k, vv) {
        keyval(k, apply(list.to.matrix(vv), 2, sum))},
```

The last few lines are an optional argument to from.dfs that operates a conversion from list to dataframe when possible, the selection of centers with at least a count of one associated point and, at the very last step, converting sums into averages.

```
  newCenters = cbind(newCenters$key, newCenters$val)
  newCenters = newCenters[newCenters[,2] > 0, -1]
  (newCenters/newCenters[,1])[,-1]}
```

To recap, we started by using a slightly different representation with a performance-tunable record size. We have re-implemented essentially the same mapper and reducer functions as in the plain-vanilla version, but using only vectorised, efficient library function that acted on all the data points in one record within a single call. Finally, we have delayed the computation of means and replaced it with the computation of pairs (sum, count), which, thanks to the associativity and commutativity of sums, can be performed on arbitrary subgroups as soon as they are formed, effecting an early data reduction. These transformations exemplify only a subset of the topics covered in [[Efficient rmr techniques]] but are enough to produce a dramatic speedup in this case.