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)
```