Overview of rmr v1.2

Despite the minor release change, there are plenty of new features in this release, some backward incompatible API changes and some serious refactoring behind the scenes.

I/O

Binary I/O and new I/O formats interface

There are several reasons to support binary formats:

- They are used by other components of the Hadoop system with which we would like to exchange data
- They are potentially more space efficient than text based formats
- They allow more direct support for R's own serialization format (see next section)

We need to underline that while this development should enable running mapreduce jobs on HBase or Mahout data, it doesn't provide these capabilities straight out of the box. Work is under way to support HBase but it won't be part of 1.2. The two binary formats provided out of the box are "native" (see next section) and "sequence.typedbytes", a sequence file, the most common hadoop file format, where both key and value are of type `TypedBytes`, a Java type with an associated, largely language-independent serialization format. This format can be read and written by a Java program using the Hadoop libraries or a Python program using the Dumbo libraries.

Let's look at some of the details of the I/O subsystem API. We tried to encapsulate the specification of how I/O should be performed so as to keep the API simple for the common cases and provide a good set of predefined combinations. Gone are the several options to `mapreduce` that controlled I/O formats. There are only two arguments, `input.format` and `output.format` (just `format` for `from.dfs` and `to.dfs`) and in the simplest case they take a string as value, like `"csv"`or `"json"`, and everything (should) works.

For instance, if you run a mapreduce job on this very document you would do something like

```
>from.dfs(mapreduce("../RHadoop.wiki/rmr-v1.2-Preview.md", input.format="text"))


[[1]]
[[1]]$key
NULL

[[1]]$val
[1] "**This is a draft document, we haven't even branched a release candidate yet**"

attr(,"rmr.keyval")
[1] TRUE

[[2]]
[[2]]$key
NULL

[[2]]$val
[1] ""

attr(,"rmr.keyval")
[1] TRUE

[[3]]
[[3]]$key
NULL

[[3]]$val
[1] "#What is coming in 1.2"

attr(,"rmr.keyval")
[1] TRUE
```

If you want to process a CSV file instead, you would just have to specify `input.format = "csv"`. Well, that is not always true. The CSV format is a family of concrete formats that differ, for instance, in the character used to separate fields. The default for `read.table` is TAB and so it is in `rmr`. What if we have a really comma separated CSV format, like this

`2008,1,3,4,2003,1955,2211,2225,WN,335,N712SW,128,150,116,-14,8,IAD,TPA,810,4,8,0,,0,NA,NA,NA,NA,NA` To deal with this case, one needs the functions `make.input.format` and `make.output.format` which accept a string format definition such as "csv" but also additional parameters, modeled after `read.table` and `write.table`, such as `sep = ","`. This is specific to the CSV format, for JSON or `"text"` additional arguments have no effect, thankfully, as there aren't subtle variations for those formats. Knowing this, we are ready to process our CSV file:

```
from.dfs(
  mapreduce("../RHadoop.data/airline.1000",
            input.format = make.input.format("csv", sep = ",")),
  to.data.frame=T)

    rmr.key V2 V3 V4   V5   V6   V7   V8 V9  V10    V11  V12 V13  V14  V15  V16 V17 V18  V19  V20  V21 V22  V23 V24  V25  V26  V27  V28  V29
1      2008  1  3  4 2003 1955 2211 2225 WN  335 N712SW  128 150  116  -14    8 IAD TPA  810    4    8   0 <NA>   0 <NA> <NA> <NA> <NA> <NA>
2      2008  1  3  4  754  735 1002 1000 WN 3231 N772SW  128 145  113    2   19 IAD TPA  810    5   10   0 <NA>   0 <NA> <NA> <NA> <NA> <NA>
3      2008  1  3  4  628  620  804  750 WN  448 N428WN   96  90   76   14    8 IND BWI  515    3   17   0 <NA>   0 <NA> <NA> <NA> <NA> <NA>
4      2008  1  3  4  926  930 1054 1100 WN 1746 N612SW   88  90   78   -6   -4 IND BWI  515    3    7   0 <NA>   0 <NA> <NA> <NA> <NA> <NA>
5      2008  1  3  4 1829 1755 1959 1925 WN 3920 N464WN   90  90   77   34   34 IND BWI  515    3   10   0 <NA>   0    2    0    0    0   32
6      2008  1  3  4 1940 1915 2121 2110 WN  378 N726SW  101 115   87   11   25 IND JAX  688    4   10   0 <NA>   0 <NA> <NA> <NA> <NA> <NA>
7      2008  1  3  4 1937 1830 2037 1940 WN  509 N763SW  240 250  230   57   67 IND LAS 1591    3    7   0 <NA>   0   10    0    0    0   47
8      2008  1  3  4 1039 1040 1132 1150 WN  535 N428WN  233 250  219  -18   -1 IND LAS 1591    7    7   0 <NA>   0 <NA> <NA> <NA> <NA> <NA>
9      2008  1  3  4  617  615  652  650 WN   11 N689SW   95  95   70    2    2 IND MCI  451    6   19   0 <NA>   0 <NA> <NA> <NA> <NA> <NA>
10     2008  1  3  4 1620 1620 1639 1655 WN  810 N648SW   79  95   70  -16    0 IND MCI  451    3    6   0 <NA>   0 <NA> <NA> <NA> <NA> <NA>
```

Notice the conversion to data frame (optional) and the fact that the column names are not particularly meaningful, we can't process headers just yet.

If you want the full power and extensibility of the IO subsystem, you can create new formats with `make.input.format` and `make.output.format` and their full syntax. They accept a `mode`, `"binary"` or `"text"`, an R function and a java class as arguments. These get called in the following order on each record:

```
java input format class ->
  r input format function ->
    map ->
      reduce ->
        r output format function ->
          java output format class
```

This gives you a lot of flexibility, but you need to make sure that the Java class and the R function, both on the input and output sides, agree on a common format. For instance, one suggested route to support a variety of Hadoop-related formats is writing or using existing Java classes to convert whatever format to a simple serialization format called typedbytes (HADOOP-1722), then use the `rmr:::typed.bytes.input.format` function to get a key value pair of R objects. Alternatively, you can use JSON as the intermediate format. For instance you could use `org.apache.avro.mapred.AvroAsTextInputFormat` to convert Avro to JSON and then use `rmr:::json.input.format` to go from JSON to a key-value pair. It should be as simple as

```
mapreduce(<other args>,
  input.format = make.input.format(format = rmr:::json.input.format,
  streaming.input.format = "org.apache.avro.mapred.AvroAsTextInputFormat",
  mode = "text"))
```

but I am sure that there will be some kinks to iron out. The converse is true on the output side. Issues have been created for HBase, Avro, Mahout and Cassandra compatibility (#19, #44, #39 and #20) and now people who need those are in a position to get things done, with a little work. Work on #19 has already started. Pull requests welcome.

Internal format is now binary

As hinted above we now use internally and as default a binary format, a combination of R native serialization and deserialization and typedbytes. This gives us the highest compatibility with R, meaning any R value should be a valid key or value in the mapreduce sense. Beforehand, lacking the binary option, the native format was an ASCII version of the R native serialization with some additional escaping. It mostly worked, but, for instance, models used to cause errors. Not anymore. The goal is to support everything, if you find exceptions please do not hesitate to submit an issue. If you have data stored in the old native format, don't fret, it has now been renamed `native.text`, but we suggest to wind its use down.

Loose ends

- Support for comments in CSV format
- JSON format reads one or two JSON objects per row, uses improvements in `RJSONIO` library instead of workarounds. Nothing should change, but there should be less room for errors.

Mapreduce Galore

The odd case of the slow reduce

If you didn't know, appends are not constant time in R.

```
> t = 16000
> system.time({ll = list(); for (i  in 1:t) ll[[i]] = i})
   user  system elapsed
  0.794   0.313   1.105
> t = 32000
> system.time({ll = list(); for (i  in 1:t) ll[[i]] = i})
   user  system elapsed
  3.274   1.545   4.818
```

You see? Input doubles, time quadruples. We didn't know until a client run Really Big Reduces, and it hurt. This should be fixed right in the interpreter. In the meantime, since we don't let our users down, we cracked the code and we have fast appends in the reduce phase. You can run much bigger reduces and still go home for dinner. But let's not get complacent. The number of reduces should still scale with the size of the output of the map phase (or combine phase if you have a combiner) and we are still allocating one big list for each key, so memory is a constraint. These are the rules of engagement.

Backend specific parameters

We've always said that we want to design a tight mapreduce abstraction, to the point that it's possible to have multiple backends, the most important of which is of course hadoop. Well, real life hit and we had to punch a small hole in the abstraction, mostly for performance tuning. You can do things like setting the number of reducers on a per-job basis. See the `backend.parameters` argument to `mapreduce` for details. The dream is to have automatic performance tuning (see e.g. AMR, but don't hold your breath.

Automatic library loading

Need to use additional libraries in your map or reduce functions? If they are loaded at `mapreduce` invocation they should be available with no additional fuss, like a lapply.

``` library(rmr) library(MASS)

from.dfs( mapreduce( to.dfs(lapply(1:5, function(i) keyval(NULL,data.frame(x=rnorm(10), y = rnorm(10))))), map = function(k,v) keyval(NULL,rlm(y~x, v)))) ``Please remember this takes care of loading, not installing. You still have to install each package you need on each
```

`node. Ideally, you should have the same R environment (version and packages) on each node. Also this means that if there are loaded libraries that you don't want R to also load on the nodes, you should detach them before calling`mapreduce`.

Data.frame conversions

Hadoop doesn't impose a lot of structure on your data, which is part of what makes it so successful. To reflect that, we use lists in crucial places of the API. `from.dfs` returns a list and the reduce function accepts a list of values. But the special case where data.frames would be more than enough and more convenient is common enough to support it with specific options in `from.dfs` and `mapreduce`. The problem is that turning a list into a data.frame under weak assumptions on the contents of the list is not easy and not even well defined. We decided to aim for a data.frame with atomic cells and let it fail when it's not possible. This is work in progress, more than the rest of the package, and we found some broken cases that needed attention, and simplified a very hacky implementation. Please give it a spin and do not hesitate to give feedback.

Loose ends

Tired of that console verbiage? Set `verbose` to `FALSE` when things are running smoothly. Want to break one large file into multiple parts? Use `scatter`.

Naming conventions

We looked at the code for v1.1 and realized we had a mix of dot-separated, CamelCase and nonseparated identifiers and while I think there are more important factors to code quality, this was a relatively easy fix that brings a little more readability and writability. We went with dot-separated across the board. This will break your code in multiple places but fixing it is as simple as search and replace. For example, `reduceondataframe` becomes `reduce.on.data.frame`. The exceptions are:

- `mapreduce`: this spelling is used elsewhere often enough that I consider it a portmanteau of map and reduce. So it's a new word and doesn't need separators.
- `keyval`: used often enough that a shorter form seems warranted.

New package options API

Instead of having one call per option, we decided to go with the pair `rmr.options.set` and `rmr.options.get` to set and get any option, in preparation for future features. See the R help for details.