- 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
- Code is NOT tested
- 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:

1. `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.
2. `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.
3. `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.
4. `native.text`: a text version of native, it is now deprecated. Convert your `rmr` 1.1 data quick and move on.
5. `csv`: added support in version 1.1. A family of concrete formats modeled after R's own `read.table`
6. `text`: for english text. key is `NULL` and value is a string, one per line. Please don't use it for anything else.

Custom formats

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

$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: 0x7fed0fd672d0>

$streaming.format
NULL
```

R data types natively work without additional effort (for matrices, functions, models, promises it is true from v1.2, hopefully all bases covered now)

```
my.data <- list(TRUE, list("nested list", 7.2), seq(1:3), letters[1:4], matrix(1:25, nrow = 5,ncol = 5))
```

Put into HDFS: `r hdfs.data <- to.dfs(my.data)` `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: please note the use of `input.format = "text"`.

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

under revision from here to end

```
myTSVReader <- function(line){
    delim <- strsplit(line, 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

```
mrResult <- mapreduce(input = hdfsData,
    textinputformat = myTSVReader,
    map = function(k,v) keyval(v[[1]], 1),
    reduce = function(k,vv) sapply(vv, sum(unlist(vv))
    )
```

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

```
mySpecificTSVReader <- function(line){
    delim <- strsplit(line, 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 `r mrResult <- mapreduce(input = hdfsData, textinputformat = mySpecificTSVReader, 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 textinputformat -must define a textoutputformat

```
myCSVOutput <- function(k, v){
    keyval(paste(k, paste(v, collapse = ","), sep = ","))
}
```

In v1.1 this should be as simple as

```
myCSVOutput = csvtextoutputformat(sep = ",")
```

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

```
mapreduce(input = hdfsData,
    output = "/rhadoop/output/",
    textoutputformat = myCSVOutput,
    map = function(k,v){
        # complicated function here
    },
    reduce = function(k,v) {
        #complicated function here
    }
    )
```

Save output to the local filesystem

```
hdfs.get("/rhadoop/output/", "/home/rhadoop/filesystemoutput/")
```

Within /home/rhadoop/filesystemoutput/ will now be your CSV data (likely split into multiple part-files according to the Hadoop way).