# Efficient rmr techniques

We will try to collect here different observations, guidelines and examples about writing efficient and scalable programs with `rmr`. These observations are related to the mapreduce programming model, its specific implementation in hadoop, R as a programming language and the `rmr` package itself. To see an example where some of these techniques are applied to a simple clustering algorithm, proceed here

## Mapreduce

Considering here the abstract programming model, not the implementation specifics, the level of available parallelism in the map phase is as high as the number of input records. Each record can be, in principle, processed independently and therefore in parallel. Not so in the reduce phase. The maximum level of parallelism is determined by the cardinality of the set of keys. A single reducer has to process all the records associated with one key. This has the potential to curtail or negate the scalability of mapreduce. This may come as a surprise as it is often, incorrectly said that, once a computation is recast as a mapreduce computation, parallelism and scalability are ensured. Let's consider a very simple problem, the sum of a set of numbers, a very large one. Let's assume that there is only one key. Each value contains a single number. The trivial mapreduce implementation is as follows:

```
mapreduce(input = ..., reduce = function(k,vv) keyval(k, sum(unlist(vv))))
```

This solution is not scalable because it uses a single reducer, no matter what the configuration of the hadoop cluster is. This is not an artificial example, as this situation is common, albeit perhaps not in this extreme form. In the classic word count example, the risk is that one processor has to go through all the occurrences of a single word, even the most common. This can be a problem both from a worst case analysis point of view and in real word text analysis.

While there aren't general solutions to this problem, there are some known techniques that can help.

- Sampling. One can run a sampling job, sampling at a fixed rate, to get an estimate of the word frequencies, then run a second job sampling at a rate inversely proportional to the initial estimates to get better estimates for the low frequency words without incurring in bottlenecks. This can be generalized to other cases such as fitting one low complexity model per key etc.
- Randomized keys. The above example required multiple jobs and accepting some degree of approximation. In some cases we might be able to avoid both by artificially increasing the number of keys to achieve better load balancing and degree of parallelism. In our large sum example we could proceed as follows:

```
proc.number = 10
partial.sums = from.hdfs(mapreduce(input = ...,
                                   map = function(k,v) keyval(sample(1:proc.number,1), v),
                                   reduce = function(k,vv) keyval(k, sum(unlist(vv)))))
```

and then finish off the job in main memory. In the wordcount case one can proceed by generating keys that are a combination of the word to be counted and a random integer between one and the ideal number of reducers. That way even the most common word occurrences get split over all processors. In a second job these partial counts are accumulated word by word.

- Combiners. This problem is so common that a potential solution has been included into hadoop as a feature. Whenever the reduce function represents an operation on the records associated with one key that is associative and commutative, one can apply the reduce function on arbitrary subsets and again on the results thereof. Examples are counting as above or max and min or averaging, if we express the average as a (total, count) pair and even the median if we accept that the median of medians is a good enough approximation,

independent of the grouping. This preliminary reduce application can be triggered simply specifying that we want a combiner. Since the combiner is run right after the map, on the same node, as opposed to the multiple job solution, the cost of the shuffling phase is often drastically reduced. The combiner can be in principle a function different from the reducer, but since there isn't a guarantee that the combiner will be actually applied to each record, I have yet to see a meaningful example of this. In the large sum example we just have to write:

```
mapreduce(input = ..., reduce = function(k,vv) keyval(k, sum(unlist(vv))), combine = T)
```

# Hadoop

- Hadoop implements the mapreduce model on top of HDFS (next generation Hadoop actually generalizes over this, mapreduce becoming only one of several computational models). That means that there are phases of disk and network I/O that are integral to the computation, before and after each map and reduce phase. It's not that I/O is a new concept: the new idea is that it is integral to this type of very large computations and not limited to an initial and final phase, somewhat separate from the actual computation, as in the single machine, in RAM kind of computation we are all familiar with or in more traditional HPC (see also NUMA systems). To be more concrete, in designing algorithms for hadoop we may face tradeoffs between the total amount of CPU work and the number or type of jobs involved, hence I/O work. Since adding a job often means adding a I/O intensive phase, it may be worth reducing the number of jobs even if at the cost of increasing the total CPU work. This is an apparent contradiction with the suggestion to break a task into multiple jobs offered above, but there are tradeoffs between I/O costs, degree of parallelism and total CPU work that can play out differently for different problems and solutions. Fast, scalable, efficient, pick 2; more realistically, find the right compromise. For instance, in the tutorial we implemented logistic regression by gradient descent. Given that each step is a separate mapreduce job and that gradient descent is known to have slower convergence than other methods, it is natural to consider methods with faster convergence such as conjugate gradient, even if each iteration requires more work. Besides the number of jobs, what they actually do is also very important. All things being equal, if one has the option to effect a data reduction early rather than late in a chain of jobs, one should take it. An example could be a chain of matrix multiplication, where we can use associativity to our advantage to control the footprint of the computation, be it in RAM or on disk.
- Hadoop has a very complex configuration that may need to be optimized for `rmr` jobs, either per job or on a global basis. For instance, we observed `rmr` jobs to be mostly CPU bound (see next Section). Therefore the optimal number of concurrent tasks on each node could be as low as the number of cores. For I/O bound processes, on the other hand, this number has been set as high as 300. The current API doesn't expose any tuning parameters for simplicity's sake, to be agnostic to different backends and heeding to API design experts' advice to be wary of tuning parameters, but a pragmatic compromise may be needed for future versions of `rmr`.

# R

It is well known that the R interpreter is no speed daemon, actually more like 50X to 100X slower than C code. So what to do about it?

- Nothing. It doesn't matter. It has been said that most widespread, useful algorithms take time linear in the size of the input, so the limiting factor is I/O anyway. This argument has some merit, but doesn't seem to apply to `rmr`. Since hadoop goes to great lengths to optimize I/O, mostly by using sequential access, in limited experiments conducted so far jobs were always CPU-limited.
- Use the compiler. While this is a promising technology that is now distributed with the main R distribution, speed gains are realized only on a subset of the language. Work on the compiler is active and improvements are expected this year. Expect something like 4X

speedups as the compiler matures. There are other compiler efforts, at a more experimental level (r2c, rcc, ra-jit).
- Write in C or leverage the work of people who did. R has a convenient interface for calling C functions and many library functions are written that way. To maximize the impact of this approach, we need most computing time to be spent outside the interpreter, executing optimized C code. To achieve this, any invocation to C-implemented functions from R has to get a significant chunk of work done to offset the function call overhead and other work performed in the R interpreter. A programming style for `rmr` that makes this more feasible is one where each input record is bigger than a single unit of data. This is an efficient style for hadoop in general. By comparison, in the [Tutorial] we mostly went the opposite route for simplicity's sake, which is the dominant consideration in that context. For instance, to store a matrix, instead of a single matrix element, we could have stored a submatrix in each record. In machine learning, one could store a subset of training data points instead of a single one. By doing this, one can reduce the number of records and thus the number of calls to the map and reduce functions. Then it becomes a matter of implementing those functions efficiently by using efficient library calls or writing in C. In the case of our large sum example, we can switch to a format whereby the value is a vector of n numbers (the key is still null or constant) and modify the mapreduce call as follows:

```
mapreduce(input = ..., reduce = function(k,vv) keyval(k, sum(unlist(vv))), combine = T)
```

Actually, there is *no change* because `unlist` can transform a list of vectors into a vector. But assuming that we spent r seconds in the R interpreter and c executing C code in the first version, the new version will run in roughly in r/n + c time, approaching C speed for large enough n. Further speed gains may be achieved by setting `map = reduce`, thus effecting a data reduction earlier in the process. When manipulating the data representation is not feasible, either because processing existing data or because the record definition is mandated by the algorithm, typically in the shuffle phase, one alternative is to use the new vectorized API.

# rmr

In the first part of this project, we focused on, as they say, "getting the API right", that is trying to achieve a seamless integration of mapreduce into R, letting `rmr` interoperate and combine with other R features and language constructs, making `rmr` programs as readable as possible, minimize any "surprise factor" etc. We have not focused on efficiency but we expect that to change as we gather feedback from users. Here are some related activities and improvements that could be helpful:

- Performance testing. Testing at scale has been very limited and a new profiling feature has not been put to much use yet. We recommend that users turn on profiling of the nodes during development and share their observations. We would be interested in taking a look at the results and helping with either code optimizations or, where warranted, optimizations in `rmr` itself.
- Binary formats. Right now the default format is based on JSON (two JSON objects per line separated by a tab, for streaming compatibility). This is great for learning and debugging and also for interoperability with other data sources (after all, we expect `rmr` to work mostly from non-`rmr` sources for the foreseeable future, see for instance AVRO-808 for the different possibilities) but not the most space efficient representation. On the positive side, we convert to a and from JSON using RJSONIO, which has been optimized for large object conversion, that is, written partly in C. There are different opinions out here as to whether streaming can be used with binary formats (see this blog entry and AVRO-512) but arguably HADOOP-1722 settles the question in the positive. Dumbo, a streaming-based library for python, has access to AVRO files through a Java class that performs a conversion and can use binary formats. An alternative R library for map reduce, RHIPE, uses Google's Protocol Buffers as the preferred data format, but doesn't use hadoop streaming to the best of my understanding.
- Configuration. We have resisted adding options to fine tune hadoop on a per-job basis — things like number of mappers, number of reducers, number of concurrent tasks etc. — for a number of reasons:

- to keep the API simple and clean
- because it is not very compatible with having multiple backends (in the dev branch of `rmr` there is the option of an R-only backend for debugging and more may be added in the future, like a CUDA or Spark one)
- because of general recommendations against it by API gurus ("All tuning parameters are suspect" — Josh Bloch)
- in the hope that hadoop will mature to a point where configuration issues will become less of a concern.

A principled compromise on this issue may be necessary to achieve the full potential of `rmr`.

- Compiler: see above. As it may be soon routine to compile map and reduce functions, the same could apply to the package itself.

## Related Links

- [[Comparison of high level languages for mapreduce: k means]]
- [[Fast k means]]