Version 0.4-0

Guide to the kazaam Package

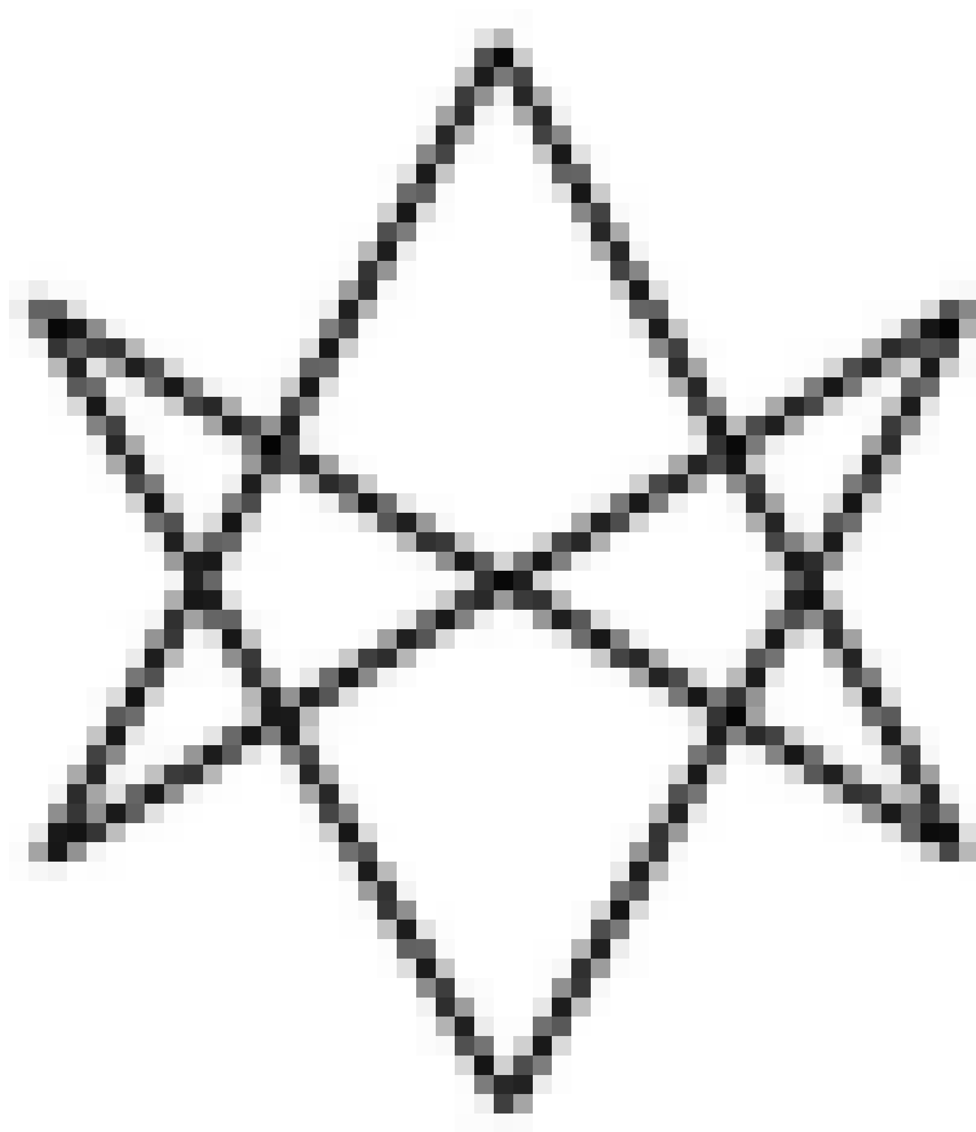
Don't get all hysterical! Say thank you for your miracle!

Drew Schmidt

Guide to the kazaam Package

November 1, 2024

Drew Schmidt
wrathematics@gmail.com



Version 0.1-0

Acknowledgements and Disclaimer

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The findings and conclusions in this article have not been formally disseminated by the U.S. Department of Health & Human Services nor by the U.S. Department of Energy, and should not be construed to represent any determination or policy of University, Agency, Administration and National Laboratory.

The **kazaam** package logo is derived from this image, which is licensed CC0, public domain.

This manual may be incorrect or out-of-date. The author(s) assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

This publication was typeset using LaTeX.

Permission is granted to make and distribute verbatim copies of this vignette and its source provided the copyright notice and this permission notice are preserved on all copies.

Contents

1 Introduction **1**
- 1.1 Installation 1

2 Background and Motivation **1**
- 2.1 Assumptions 2
- 2.2 Performance Considerations 2
- 2.3 Conventions 3

3 Creating a shaq **4**
- 3.1 Using expand() and collapse() 4
- 3.2 Using the Random Constructor 4
- 3.3 Using the Explicit Constructor 5

4 Matrix Computing and Statistics **6**

References **8**

1 Introduction

The **kazaam** package [5] is a set of utilities for creating, managing, and doing analysis with very tall, skinny, distributed matrices in R. It is not the first such package to handle distributed matrices in R (the authors even have another; more on that in a moment). However, it has some distinct advantages over the competition, assuming your problem fits its various assumptions well.

In contrast to the **pbdDMAT** package [10, 11] which uses ScaLAPACK [2], most of the linear algebra and statistics operations in **kazaam** are performed locally. This gives the advantage of being faster, at the expense of numerical accuracy. Additionally, **pbdDMAT** can handle square, as well as short/wide matrices, both of which are bad fits for **kazaam**. So if you believe your data to be ill-conditioned or its dimension are not appropriate, then the **pbdDMAT** package may be a much better fit.

While the **kazaam** package has these deficiencies, it does have some advantages. For one, it generally performs very well on very large, tall/skinny matrices. So long as the data is well-conditioned (and it probably is, frankly), then the performance should be quite good. Additionally, ScaLAPACK indexes the global number of rows/columns by a signed 32-bit integer, making it impossible to use with extremely tall matrices; but this is no problem for **kazaam**. Finally, because the distribution schema is *much* simpler than ScaLAPACK's (and whence **pbdDMAT**'s) 2d block cyclic distribution, we find that **kazaam** is much easier to work with in general.

No one package can solve every problem. This is especially true in HPC. However, if your data truly is large and you need to run in an HPC environment, we believe that the various tradeoffs and compromises among other similar sorts of packages leave **kazaam** looking very competitive.

1.1 Installation

You can install the stable version from CRAN using the usual `install.packages()`:

```
1 install.packages("kazaam")
```

The development version is maintained on GitHub, and can easily be installed by any of the packages that offer installations from GitHub:

```
1 ### Pick your preference
2 devtools::install_github("RBigData/kazaam")
3 ghit::install_github("RBigData/kazaam")
4 remotes::install_github("RBigData/kazaam")
```

To simplify installation on cloud systems, we also have a Docker container available.

2 Background and Motivation

Our tall/skinny/distributed matrices are called "shaqs", which stands for Super Huge Analytics done Quickly. This of course has nothing at all to do with esteemed actor Shaquille O'Neal, who is very tall. And since the package is so easy to use, it sometimes looks like a magic trick. And "kazaam!" is something a magician might say. It is by mere coincidence that Shaquille O'Neal starred in a movie titled *Kazaam*.

Many large scale data science applications ("big data") look like small scale ones, only with way more data. Often, these are "tall and skinny". When distributing these matrices, it makes the most sense (from both a performance and ease-of-use perspective) to distribute these across processors by row. Meaning, each process should own all of the columns of the data, but only a subset of the rows. In the following subsection, we make precise all of the assumptions required for using **kazaam**; but that is the big one.

The primary motivation for the authors was a *very* tall dataset on which we needed to compute a few principal components. But the layout was so easy to work with that we couldn't help ourselves in adding other methods and model fitters.

2.1 Assumptions

Throughout the package, we make a few key assumptions:

- The data local to each process has *the same number of columns*. The number of rows can vary freely, or be identical across ranks.
- Codes should be *run in batch*. Communication is handled by the **pbdMPI** package [3], which (as the name suggests) uses MPI [6].
- Finally, *adjacent ranks in the MPI communicator* as reported by 'comm.rank()' (e.g., ranks 2 and 3, 20 and 21, 1000 and 1001, ...) should store *adjacent pieces of the matrix*.

In order to get good performance, there are several other considerations:

- The number of rows m should be *very large*. If you only have a few hundred thousand rows (and few columns), you're probably better off with (non-distributed) base R matrices.
- The number of columns n should be *very small*. A shaq with 10,000 colums is pushing it.
- For most operations, the local problem size should be *as big as possible* so that the local BLAS/LAPACK operations can dominate over communication. This also keeps the total number of MPI ranks minimal, which cuts down on communication.

Because of these assumptions, we get a few distinct advantages over other, similar frameworks:

- Communication is very minimal. Generally it amounts to a single `allreduce()` of an $n \times n$ matrix. With even a few hundred MPI ranks, this is basically instantaneous. And since most of the work is local, operations should complete very quickly.
- The total number of rows can be as large as you like, even if that's more than can fit in a signed 32-bit integer, or $2^{31} - 1$.

2.2 Performance Considerations

All of the communication is handled by MPI. For better or worse, and no matter how loudly mapreduce zealots scream, MPI is still the gold standard for this kind of thing. As noted above, if you set your problem up right (and if it is indeed a good fit for **kazaam**), then most operations amount to fairly minimal communication. This is good, because *communication is expensive*. At large scales, the most expensive part of codes are generally the I/O (reading from/writing to disk) and the inter-node communication.

Most of the local operations eventually, if only in part, offload to the BLAS [8] and/or LAPACK [1]. Using a high quality BLAS implementation such as **MKL** [7] or **OpenBLAS** [14] with R will greatly improve the performance of these operations. The issue of linking R with different BLAS is a well-trod path and so we do not discuss it here. We suggest the reader refer to [4] for more details.

However, there is an additional point of consideration when using a high performance BLAS implementation. Generally speaking, these libraries are multithreaded. This will actually seriously negatively impact performance if you are using more than one MPI rank per node and do not adjust the environment variable `OMP_NUM_THREADS` accordingly.

To make this more concrete, say you have 4 cores per node. If you launch 4 MPI ranks, then you should set `OMP_NUM_THREADS=1` for best performance. If you launch 2 MPI ranks per node, then you would

want to set `OMP_NUM_THREADS=2`. Finally, you guessed it, if you launch 1 MPI rank per node, then you should set `OMP_NUM_THREADS=4`. There may be a good reason to stray from this schema; but if you can't articulate one, then this is the pattern you should follow.

Table 1 shows the results of a simple benchmark using the **kazaam** function `logistic.fit()`. Note the extreme jump in run time when the number of OMP threads is set inappropriate with the number of MPI ranks. We can also see that in this case, the best performing case was when we saturated the node with MPI processes, each using only 1 OMP thread. However, we note that this will not always be the case.

MPI Ranks	OMP Threads	Time
4	1	0.361
2	2	0.409
1	4	0.667
4	4	6.883

Table 1: Timing `logistic.fit()` on a single node with 4 total cores. The problem size remains the same each time, but we vary the number of OMP threads and MPI ranks.

Finally, we would be remiss if we did not discuss the batch aspect of the package. Most parallel R frameworks assume a manager/work (or using the older terminology, master/slave) pattern. However, **kazaam** is meant to be used in Single Program Multiple Data (SPMD) fashion. This is why we require it to be run in batch. And while the loss of interactivity may seem like a great and unbearable cost, this gives some great advantages. In addition to generally being much easier to write SPMD codes, it is often faster. Additionally, it is possible to use the packages interactively, using the pbdR client/server system [13, 12]. But that is a very lengthy topic in its own right, and we do not cover it here.

2.3 Conventions

Throughout the remainder of the document, all code examples will follow the convention that a variable name with no preceding "d" is not distributed (e.g., `x`, `y`), but one with a preceding "d" is a distributed shaq (e.g., `dx`, `dy`). However we note that we do not generally stick to this convention elsewhere (say in the package tests or other documentation).

As discussed in Section 2.1, these examples must all be run in batch via `mpirun` (or its equivalent; e.g., `aprun` on a Cray). Also, for brevity and ease of reading, in each code example we will omit the boilerplate that each script requires. So to run any one example, one should preface each example by the appropriate library load call and end with a call to `finalize()`:

```
suppressPackageStartupMessages(library(kazaam))

### script goes here ...

finalize()
```

And then run it with

```
mpirun -np 2 Rscript my_script.r
```

Changing 2 above to your desired number of processes.

Finally, when an example code below has some output given inline as a comment, then that was generated with 2 MPI ranks.

3 Creating a shaq

Correctly creating the distributed object is the most difficult part of using the **kazaam** package. So we need to spend a bit of time discussing some finer points about the various components of the shaq construction API.

3.1 Using expand() and collapse()

If the matrix is not particularly large and can comfortably fit on one process, then one can use the `expand()` and `collapse()` functions. This is probably not particularly useful in practice since the whole point of the package is to handle enormous matrices, which may not fit into memory of any one of the nodes. However, for testing and experimentation (and for extremely computationally expensive problems, like a robust PCA), this can be beneficial.

The `expand()` function assumes that all of the data is owned by MPI rank 0, and that every other process has something worth ignoring (but it has to have something!). By convention, we use `NULL` for that ignorable something. Using the function is fairly basic, but requires a bit of rank-checking boilerplate:

```
if (comm.rank() == 0){
  x = matrix(rnorm(30), 10)
} else {
  x = NULL
}

dx = expand(x)
dx
## # A shaq: 10x3 on 2 MPI ranks
##             [,1]       [,2]       [,3]
## [1,]  1.0235321 -1.2847391 -0.4941115
## [2,] -0.4801970 -0.5547932  0.9545184
## [3,] -0.8849440 -1.5026188 -1.4448291
## [4,] -1.2209445 -1.1273592 -1.6025001
## [5,]  0.1442414 -0.4165442 -1.3283896
## # ...
```

Likewise, one can easily go from an “expanded” matrix back to the original via `collapse()`:

```
y = collapse(dx)
comm.print(all.equal(x, y))
## [1] TRUE
```

This will glue the matrix back together on rank 0, while all other processes will store `NULL`.

3.2 Using the Random Constructor

It is also very simple to generate random shaqs via the `ranshaq()` constructor. When using this, one should pay careful attention to random seed use. See `?pbdMPI::comm.set.seed` for details. For these examples, we will assume that `comm.set.seed(1234, diff=TRUE)` has been called before any of the generation takes place.

The `ranshaq()` function may look strange, but it is very simple to use. It takes as its first argument a generating function, like `runif()` or `rnorm()` for example. It then uses the supplied dimension information to generate the local data. So for example, to generate a 10×3 shaq of random normal values, we could call:

```
dx = ranshaq(rnorm, 10, 3)
```

We can pass in other kinds of functions as well. Say for example that we wanted to generate a distributed vector of labels for a logistic regression. We could then call:

```
dy = ranshaq(function(i) sample(0:1, size=i, replace=TRUE), 10)
dy
## # A shaq: 10x1 on 2 MPI ranks
##       [,1]
## [1,]     0
## [2,]     1
## [3,]     1
## [4,]     0
## [5,]     0
## # ...
```

Another occasionally useful trick with the random constructor is to set the input parameter `local=TRUE` (default is `local=FALSE`). This tells the constructor that the number of rows supplied is the *local* number of rows. So if you have 4 MPI ranks and pass **`nrows=5`**, then the shaq will have 20 total rows. It is similarly simple to use:

```
dx = ranshaq(rnorm, 10, 3, local=TRUE)
```

3.3 Using the Explicit Constructor

Discussing these small examples is useful in understanding conceptually how shaqs are “glued” together and for basic testing and experimentation. But for a real data analysis, most truly big jobs will not look like any of these examples. For these, one should read in the data from a parallel file system, such as lustre, onto a collection of processes, and then distribute them to the whole grid. The **pbdIO** package [9] can be helpful here.

To handle this last case, there is an explicit shaq constructor. It can be used to create very simple objects if it is passed a vector, like a 10×3 shaq of 1's:

```
dx = shaq(1, 10, 3)
dx
## # A shaq: 10x3 on 2 MPI ranks
##       [,1] [,2] [,3]
## [1,]     1    1    1
## [2,]     1    1    1
## [3,]     1    1    1
## [4,]     1    1    1
## [5,]     1    1    1
## # ...
```

But whenever it is given a matrix, it assumes that you are passing the entirety of the local submatrix to it. So if you are wanting to create an $m \times 3$ shaq, and the local matrix you pass to `shaq()` only has 2 columns, then very bad things will happen. The purpose for this is that it makes significant communication savings possible by putting the burden onto the programmer rather than the package. Welcome to high performance computing.

To demonstrate this, we will show an example that should be run with a total of 2 MPI ranks, constructing the shaq explicitly:

```
m.local = 5
m = 5 * comm.size()
n = 3

if (comm.rank() == 0){
  x = matrix(1:15, m.local, n)
} else if (comm.rank() == 1){
  x = matrix(15:1, m.local, n)
} else {
  stop("too many MPI ranks")
}

dx = shaq(x, m, n)

comm.print(Data(dx), all.rank=TRUE)
## COMM.RANK = 0
##       [,1] [,2] [,3]
## [1,]     1    6   11
## [2,]     2    7   12
## [3,]     3    8   13
## [4,]     4    9   14
## [5,]     5   10   15
## COMM.RANK = 1
##      [,1] [,2] [,3]
## [1,]   15   10    5
## [2,]   14    9    4
## [3,]   13    8    3
## [4,]   12    7    2
## [5,]   11    6    1
```

4 Matrix Computing and Statistics

If the hard part is getting the data into a shaq, then this is the easy part. Generally, we try to make the method functions as close to native R code as possible. We believe that this makes the transition from a regular matrix to a distributed one much simpler. So you can prototype on a subset, and then run the full code on the large distributed object in batch.

We generally try to avoid communication, unless it is required or makes things simpler for the user. One of the side effects of trying to make things easier is that often, small output objects from a shaq operation will no longer be distributed; and in fact, every process will own a copy of the object. For example, in a singular value decomposition, constructed by `svd()`, the singular values are just an n length vector, and the right singular vectors are an $n \times n$ matrix. Remember that we are assuming that n is small, so there is no reason to distribute the object any longer. We could perhaps improve the performance some by making it so that only rank 0 had a copy of the matrix. However, as soon as you wanted to do something with the matrix and a shaq, like multiply them, then you would have to 1. know you needed to distribute it, and 2. know *how* to distribute it. So for simplicity, we opt to ensure that every MPI rank has a copy.

Of course, there's no free lunch. If you wish to print something, then you will likely need to be aware of what kind of object it is. You do not need to use `comm.print()` on a shaq (although this is safe), but you really should use it for vectors and matrices, otherwise every rank will print the values (and not necessarily in order!). Below is an example using the SVD:

```
dx = ranshaq(rnorm, 10, 3)
ret = svd(dx)

# a vector
comm.print(ret$d)
## [1] 4.832578 4.380649 1.968638

# a shaq
ret$u
## # A shaq: 10x3 on 2 MPI ranks
##              [,1]        [,2]        [,3]
## [1,]  0.69426448  0.29722839 -0.52002822
## [2,] -0.34960480  0.16684400 -0.29989357
## [3,]  0.44037583 -0.50982744  0.07714539
## [4,]  0.08126058  0.51866794  0.22956647
## [5,]  0.21030075  0.08756975  0.08398620
## # ...

# a matrix
comm.print(ret$v)
##            [,1]       [,2]       [,3]
## [1,] -0.6790577  0.4704660  0.5635090
## [2,] -0.7152984 -0.2515092 -0.6519903
## [3,] -0.1650116 -0.8458161  0.5073128
```

Most operations on shaqs are only on a single shaq object. However, some operate on multiple shaqs at a time, such as the regression fitters. For *any* such operation, we assume *without checking* (because that would be expensive) that the shaqs are distributed *identically*. This probably means exactly what it sounds like to you. However, we can precisely define this.

First, adjacent MPI ranks should hold adjacent rows. So if the last row that rank `k` owns is `i`, then the first row that rank `k+1` owns should be row `i+1`. Additionally, any method that operates on two (or more) shaq objects, the two shaqs should be distributed identically. By this we mean that if the number of rows shaq `A` owns on rank `k` is `k_i`, then the number of rows shaq `B` owns on rank `k` should also be `k_i`. Finally, each MPI rank should own at least one row.

We conclude with an example of fitting a logistic regression model:

```
dx = ranshaq(rnorm, 10, 3)
dy = ranshaq(function(i) sample(0:1, size=i, replace=TRUE), 10)

# Somewhat akin to calling glm.fit(x, y, family=binomial(logit))
fit = logistic.fit(dx, dy)

comm.print(fit)
## $par
## [1] -0.5012429 -0.4395908 -1.9234366
##
## $value
## [1] 0.5625917
##
## $counts
## function gradient
```

```
16 ##       201       101
17 ##
18 ## $convergence
19 ## [1] 1
20 ##
21 ## $message
22 ## NULL
```

References

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[2] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[3] W.-C. Chen, G. Ostrouchov, D. Schmidt, P. Patel, and H. Yu. pbdmpi: Programming with big data – interface to mpi, 2012. R Package.

[4] Andrie de Vries. How the MKL speeds up Revolution R Open. http://blog.revolutionanalytics.com/2014/10/revolution-r-open-mkl.html, 2014.

[5] Mike Matheson Drew Schmidt, Wei-Chen Chen and George Ostrouchov. *kazaam: Tools for Tall Distributed Matrices*, 2017. R package version 0.1-0.

[6] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA, USA: MIT Press Scientific And Engineering Computation Series, 1994.

[7] Intel Corporation. Intel Math Kernel Library (Intel MKL), 2010.

[8] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[9] George Ostrouchov. *pbdIO: Programming with Big Data – Input-Output*, 2016. R package version 0.1-0.

[10] D. Schmidt, W.-C. Chen, G. Ostrouchov, and P. Patel. pbdbase: Programming with big data – core pbd classes and methods, 2017. R Package version 0.5-0.

[11] D. Schmidt, W.-C. Chen, G. Ostrouchov, and P. Patel. pbddmat: Programming with big data – distributed matrix algebra computation, 2017. R Package version 0.5-0.

[12] Drew Schmidt and Wei-Chen Chen. *pbdCS: 'pbdR' Client/Server Utilities*, 2015. R package version 0.1-0.

[13] Drew Schmidt and Wei-Chen Chen. *remoter: Remote R: Control a Remote R Session from a Local One*, 2015. R package version 0.1-1.

[14] Zhang Xianyi, Wang Qian, and Zaheer Chothia. OpenBLAS. *URL: http://xianyi. github. io/OpenBLAS*, 2012.