Part 1 Section 2: R Data IO

Aaron Robotham

Load libraries needed for this chapter:

```
library(magicaxis, quietly=TRUE)
library(data.table, quietly=TRUE)
library(Rcpp, quietly=TRUE)
library(microbenchmark, quietly=TRUE)
library(arrow, quietly=TRUE)
library(dplyr, quietly=TRUE)
library(fst, quietly=TRUE)
library(hdf5r, quietly=TRUE)
library(RSQLite, quietly=TRUE)
library(duckdb, quietly=TRUE)
```

There are many useful formats for data storing and loading in **R**. Here we will cover the most useful ones, and discuss how you can create your own IO interface.

Data Formats

Simply put, IO and data formats refer to how data you want to analyse is stored on disk. There are two main paradigms: human readable and binary. Human readable means you can open up the file in a text editor and read the contents (with perhaps some difficulty). Binary format means the data is in something closer to raw machine code- you can still open it up in a text editor, but it will look very weird.

The mains pros and cons are that a human readable format is highly cross platform and language- if you can read it, it should be easy to write some software in your language of choice to read it too (although for most popular formats such routines will probably already exist). Binary formats are usually more compact and faster to read and write.

Within binary formats there are also two paradigms: open standards (e.g. **HDF5**, covered later) and propriety ones (e.g. Excel). There are few good reasons to use proprietary formats unless you have no choice. Even within open standards you might find there are limited libraries available in your language of choice, so just because they are open does not make them trivially cross platform and language (in the way that human readable formats always are).

Big Data = Big Confusion

You will see a lot of mentions online of terms like **Spark**, **Arrow**, **Hadoop**, **HDFS**, **Parquet** and **SQL**, and it all gets a bit confusing. Here is a brief round-up.

- **Hadoop** is a **Java** based distributed on-disk file system format that primarily uses a file format called **HDFS**. It allows for the efficient access of large scale data stored across multiple computing clusters.
- **Spark** is a distributed in-memory file system format that allows for efficient analysis of data.
- **Parquet** is a columnar (column major, discussed later) on-disk file format popular in the world of big data.
- **Arrow** is a columnar (column major, discussed later) in-memory file format popular in the world of big data.
- **SQL** is a non-distributed data base system language It allows for the efficient access of large scale data stored on a single computing clusters.

As you can see, few of these popular paradigms / formats directly compete. For that reason they tend to be used in complement to each other. In particular it is common to see **Hadoop** combined with **Spark**. **R** (like most other modern high level languages) has support for all of these formats and paradigms, making it in principle easy to use **R** to control big data processing operations (similar to **Python**, which is also popular in this regard). Since they will not be discussed more in this course, for **Hadoop** check out the **RHadoop** package, and for **Spark** see **sparklyr**.

Perhaps the biggest recent confusion has come from **Arrow**, since it has recently become able to write on-disk too much like **Parquet**. Both are supported by the open source **Apache** foundation, so there is a bit of confusion as to when you should use which. The advice is that **Parquet** is considered archival grade (if you save the file now you will be able to read it in 10 years) whereas **Arrow** is ephemeral and will change with time in order to maintain the speed of its in-memory format (which is the priority, not longterm file storage). **Parquet** files are also able to be compressed, so tend to be much smaller and therefore useful when data is being processed across physically separated computing clusters.

In RAM Memory Format

It is important to realise how **R** holds data in memory, especially matrices and tables. The two ways you can do this with column major and row major data, where a contiguous stream of data in RAM is treated as first filling with columns or rows before incrementing. This is obvious if we create a matrix in **R** with both strategies:

```
matrix(1:9,3, byrow=TRUE) # row major filling
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

```
matrix(1:9,3, byrow=FALSE) # column major filling
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
matrix(1:9,3) # default in R (column major)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Why does this matter? Data access becomes appreciably faster if you subset or sum with the cut in the same direction as the major ordering, e.g.:

```
bigmatrix = matrix(1:1e6, 1000)
microbenchmark(
  bigmatrix[50:60,],
  bigmatrix[,50:60],
  rowSums(bigmatrix),
  colSums(bigmatrix)
)
```

```
## Unit: microseconds
##                 expr      min        lq       mean    median        uq      max
##  bigmatrix[50:60, ]   21.515   37.4815   45.52174   45.2245   53.9760   90.807
##  bigmatrix[, 50:60]   17.212   22.1405   33.53242   35.9890   39.8520   59.410
##  rowSums(bigmatrix) 2683.715 2806.7615 2885.64889 2888.6660 2938.8900 3234.291
##  colSums(bigmatrix)  715.870  754.9585  777.92101  773.1910  792.5775  944.229
##  neval
```

```
##     100
##     100
##     100
##     100
```

Since **R** uses a column major format it will usually be faster to subset by columns, so this might influence how you organise your data if it is clear you want to do a lot of data summing and subsetting operations. For note, **C** and **Python** (via **numpy**) uses row major and **Fortran**, **MATLAB** and **Julia** use column major for their respective standard matrix interfaces. There are pros and cons to both, but for categorical data analysis (where columns often have different types of data in them), column major is largely preferred these days. For this reason most of the popular on disk database solutions use variants of a column major format.

R has a very light weight method for creating matrices and higher dimensional arrays. They are always a stream of numbers in RAM, but with an attribute that tells **R** you can use special methods to access numbers. I.e.:

```
tempmat = matrix(1:64,8)
tempmat
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    1    9   17   25   33   41   49   57
## [2,]    2   10   18   26   34   42   50   58
## [3,]    3   11   19   27   35   43   51   59
## [4,]    4   12   20   28   36   44   52   60
## [5,]    5   13   21   29   37   45   53   61
## [6,]    6   14   22   30   38   46   54   62
## [7,]    7   15   23   31   39   47   55   63
## [8,]    8   16   24   32   40   48   56   64
```

```
tempmat[50]
```

```
## [1] 50
```

```
attributes(tempmat)
```

```
## $dim
## [1] 8 8
```

And now we can change this into a 3D array without touching the data part explicitly:

```
attributes(tempmat)$dim = c(4,4,4)
tempmat
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   17   21   25   29
## [2,]   18   22   26   30
## [3,]   19   23   27   31
## [4,]   20   24   28   32
##
## , , 3
##
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   33   37   41   45
## [2,]   34   38   42   46
## [3,]   35   39   43   47
## [4,]   36   40   44   48
##
## , , 4
##
##      [,1] [,2] [,3] [,4]
## [1,]   49   53   57   61
## [2,]   50   54   58   62
## [3,]   51   55   59   63
## [4,]   52   56   60   64
```

```
tempmat[50]
```

```
## [1] 50
```

In both the 2D and 3D format of the 64 elements, the 50^{th} objects is always 50 and the memory is not altered at all.

Built in R IO Formats

R comes with a few simple routines for reading and writing data to disk.

ASCII and csv

ASCII and csv are the most popular human readable file formats. The main functions you might use to write/read in table like data are base **R write.csv/read.csv** and **write.table/read.table**. Although note that the **data.table** package comes with the excellent and rapid **fwrite/fread** too.

As an example let us first make a small amount of toy data:

```
example_DF = data.frame(num=1:26, letters=letters, roots=sqrt(1:26))
```

We can then write out an ASCII tab (spaces separating data) and a csv version:

```
write.table(example_DF, '../data/test.tab')
write.csv(example_DF, '../data/test.csv')
```

Note in the above we have used the suffix ".tab" for the ASCII table. There is no real standard for this loosely define format, with people also often using ".dat" and ".ascii" amongst others. ".csv" is far more standardised, so it is one reason it tends to be preferred these days. With ASCII tables it is also often not clear whether the separation is made using tabs or spaces (they often look visually identical in a text editor), which can lead to mistakes when reading in the files. If you have ever seen the TV show Silicon Valley, you will know people can get pretty invested in tabs versus spaces...

In any case, let us take a look at the first couple of lines of each file:

```
readLines('../data/test.tab', n=6)
```

```
## [1] "\"num\" \"letters\" \"roots\""  "\"1\" 1 \"a\" 1"
## [3] "\"2\" 2 \"b\" 1.4142135623731"  "\"3\" 3 \"c\" 1.73205080756888"
## [5] "\"4\" 4 \"d\" 2"                "\"5\" 5 \"e\" 2.23606797749979"
```

```
readLines('../data/test.csv', n=6)
```

```
## [1] "\"\",\"num\",\"letters\",\"roots\"" "\"1\",1,\"a\",1"
## [3] "\"2\",2,\"b\",1.4142135623731"      "\"3\",3,\"c\",1.73205080756888"
## [5] "\"4\",4,\"d\",2"                    "\"5\",5,\"e\",2.23606797749979"
```

One odd thing is each line has gained a number. To be compatible with other formats it is usually better to turn this row numbering (technically naming) feature off:

```
write.table(example_DF, '../data/test.tab', row.names=FALSE)
write.csv(example_DF, '../data/test.csv', row.names=FALSE)
readLines('../data/test.tab', n=6)
```

```
## [1] "\"num\" \"letters\" \"roots\"" "1 \"a\" 1"
## [3] "2 \"b\" 1.4142135623731"       "3 \"c\" 1.73205080756888"
## [5] "4 \"d\" 2"                     "5 \"e\" 2.23606797749979"
```

```
readLines('../data/test.csv', n=6)
```

```
## [1] "\"num\",\"letters\",\"roots\"" "1,\"a\",1"
## [3] "2,\"b\",1.4142135623731"       "3,\"c\",1.73205080756888"
## [5] "4,\"d\",2"                     "5,\"e\",2.23606797749979"
```

That looks a bit more as we would expect.

We can benchmark the base **R** and **data.table** ASCII and csv IO functions:

```
microbenchmark(
  write.table(example_DF, '../data/test1.tab'), read.table('../data/test1.tab'),
  write.csv(example_DF, '../data/test.csv'), read.csv('../data/test.csv'),
  fwrite(example_DF, '../data/test2.tab', sep=' '), fread('../data/test2.tab', sep=' '),
  fwrite(example_DF, '../data/test.csv'), fread('../data/test.csv')
)
```

```
## Unit: microseconds
##                                                expr     min       lq      mean
##        write.table(example_DF, "../data/test1.tab") 379.687 428.5010 1336.2898
##                     read.table("../data/test1.tab") 353.658 431.7345  489.7666
##           write.csv(example_DF, "../data/test.csv") 413.735 474.0180 1203.8978
##                       read.csv("../data/test.csv") 337.880 416.3735  471.2092
##  fwrite(example_DF, "../data/test2.tab", sep = " ") 278.463 325.1465 1402.1742
##               fread("../data/test2.tab", sep = " ") 350.743 391.5270  461.6637
##              fwrite(example_DF, "../data/test.csv") 258.907 322.6950 1056.7035
##                          fread("../data/test.csv") 335.915 401.5820  455.5376
##    median       uq       max neval
##  471.1725 531.2505 17964.187   100
##  476.4350 514.0195  1203.197   100
##  508.7470 553.5785 18122.319   100
##  462.3735 498.9250   785.112   100
##  372.2655 473.4460 17807.650   100
##  420.4565 473.9045  2686.061   100
##  375.6380 478.7810 17861.160   100
##  442.0855 490.4290   708.546   100
```

Here it is very close, but for much larger files **data.table** becomes increasingly more rapid relative to base **R**, so I suggest you consider using it for all ASCII or csv table use cases. It is not clear to me why ASCII tables might tend to read and write faster though (in most cases). I would expect them to behave almost identically, so perhaps there are some very low level optimisations used for writing spaces (since they are the most common characters to write).

The base **R** functions will automatically assign the read in table to a data frame structure (not a matrix, which people might expect). This is almost always what you want, but for fully numeric data which you do want to be treated as a matrix (usually for processing speed, since matrix manipulation is very fast), you might need to explicitly convert things:

```
gooddf2 = data.frame(a=1:3,b=4:6,c=7:9)
gooddf2
```

```
##   a b c
## 1 1 4 7
## 2 2 5 8
```

```
## 3 3 6 9
```

```
goodmat2 = as.matrix(gooddf2)
goodmat2
```

```
##      a b c
## [1,] 1 4 7
## [2,] 2 5 8
## [3,] 3 6 9
```

Using native matrices will increase the speed of matrix-like operations (e.g. **svd**) by factors of a few usually.

Lists and Complex Data

R supports a huge variety of complex file types that can be read in or written out, but internally complex files will usually be represented as lists.

To save an individual **R** object or all **R** objects in the work-space you have three choices:

```
save.image('../data/workspace.rda') #saves everything in your workplace
```

```
list2 = list(gooddf2, goodmat2)
save(list2,file='../data/list2.rda') #just saves list2 with the name list2
saveRDS(list2,file='../data/list2.rds') #saves list2 with no name (just data)
load('../data/workspace.rda') #inherits original object names
load('../data/list2.rda') #inherits original object name
list2 = readRDS('../data/list2.rds') #has to be assigned to a named object
```

These are not especially rapid structures to access since you have to load the entire **R** structure into memory (you cannot target subsets of data). Luckily there are methods to efficiently access large data sets in **R**.

Note that rds and rda files are binary open formats, but there are only readers and writers available for **R**, i.e. you will have a hard time trying to open these file formats in **Python** (although there is some limited support).

External IO Formats

Arrow and Feather

Arrow is a fairly new in-memory columnar data format that allows for efficient access to table-like data. **Feather** was written as an on-disk complement that has become quite popular because it is rapid to read and write, and is available across many languages now (originally **R** and **Python**). As such it is a good quick way to get data from one language to another without worrying about more complex routes like **reticulate** etc.

Now we will write and read this back in:

```
microbenchmark(
  write_feather(example_DF, '../data/test.ft'), read_feather('../data/test.ft')
)
```

```
## Unit: milliseconds
##                                       expr      min       lq     mean
##  write_feather(example_DF, "../data/test.ft") 1.921285 2.115752 4.605523
##               read_feather("../data/test.ft") 2.405150 2.592426 3.110045
##    median       uq      max neval
##  2.247730 2.641802 35.06764   100
##  2.815745 3.041345 13.54165   100
```

Note in this example **Feather** is actually slower! This is a very small table- the speed up usually becomes noticeable once you have a few hundred rows.

To help remove confusion (or perhaps sew more), the functions *write_arrow* and *read_arrow* are direct aliases to *write_feather* and *read_feather*.

Parquet

The **Arrow** package also comes with functions to read and write the specifically on-disk **Parquet** format. This is a bit slower to read and write than **Feather**, but has the advantage of being stable over the long-term (archival grade).

```
microbenchmark(
  write_parquet(example_DF, '../data/test.pq'), read_parquet('../data/test.pq')
)
```

```
## Unit: milliseconds
##                                          expr      min       lq     mean
##  write_parquet(example_DF, "../data/test.pq") 3.424655 3.56268 5.872143
##               read_parquet("../data/test.pq") 2.794670 3.01822 3.357548
##    median       uq      max neval
##  3.714938 4.056413 35.53802   100
##  3.139550 3.286929 10.48905   100
```

These days I use **Parquet** almost exclusively over **feather** now it is beyond v1.0.0, so in theory feature stable etc.

FST

An even faster (but currently **R** only) format comes with the **FST** package:

```
microbenchmark(write_fst(example_DF, '../data/test.fst'), read_fst('../data/test.fst'))
```

```
## Unit: microseconds
##                                       expr     min       lq     mean  median
##  write_fst(example_DF, "../data/test.fst") 178.575 197.5525 216.8333 206.910
##               read_fst("../data/test.fst")  93.784 109.7590 206.3414 120.345
##        uq      max neval
##  229.6820  430.526   100
##  133.9055 8450.181   100
```

A neat thing with **FST** is that for very large files you can just create a pointer to the file, and then access it like you would a normal **data.frame**, but it only loads the data requested (not the entire file). This makes it a very handy light-weight database:

```
fst_point = fst('../data/test.fst')
str(fst_point)
```

```
## List of 4
##  $ meta         :List of 7
##   ..$ path           : chr "/Users/aaron/Dropbox (Personal)/AstroStats_Local/data/test.fst"
##   ..$ nrOfRows       : num 26
##   ..$ keys           : NULL
##   ..$ columnNames    : chr [1:3] "num" "letters" "roots"
##   ..$ columnBaseTypes: int [1:3] 4 2 5
##   ..$ keyColIndex    : NULL
##   ..$ columnTypes    : int [1:3] 5 2 10
##   ..- attr(*, "class")= chr "fstmetadata"
##  $ col_selection: NULL
##  $ row_selection: NULL
##  $ old_format   : logi FALSE
```

So the **fst** command actually just loads a limited amount of meta data, not the actual data. And yet it behaves just like a normal **data.frame**:

```
fst_point[5:10, c('letters', 'roots')]
```

```
##   letters    roots
## 1       e 2.236068
## 2       f 2.449490
## 3       g 2.645751
## 4       h 2.828427
## 5       i 3.000000
## 6       j 3.162278
```

```
dim(fst_point)
```

```
## [1] 26  3
```

But be careful when using code that looks for **data.frames**, since:

```
is.data.frame(fst_point)
```

```
## [1] FALSE
```

In most speed tests, **FST** wipes the floor with all the competition, so if you are keeping your data within the **R** ecosystem, but perhaps moving it between machines or saving some data to analyse again later **FST** is a very good option.

HDF5

A popular very general purpose format is **HDF5** This is popular in many fields of research because it is fairly low-level and it is possible to create almost any type of hierarchical data structure using it. This flexibility is its Achilles' Heel though- it is often hard to second guess how somebody might have saved in data as simple as a 2D matrix or a table (like a **data.frame**), which can make loading somebody else's data a pain (you often have to poke around the meta data to get a good idea of the format). In practice many higher level data formats use **HDF5** as the lower level data storage mechanism, and all that complexity is abstracted away.

That said, **HDF5** is very popular and powerful, as well as fast. The **HDF5** consortium actually support the **C** API themselves, so whilst there is a standards document attached to **HDF5**, for most purposes **HDF5** *is* the **C** interface provided.

Even with this interface, it is such a large standard that there are multiple packages that support different parts of **HDF5** dotted around **R**. This also adds to the confusion since some are more flexible but complex etc, and some are very fast and others slow. After far too much of my life wasted testing all the options, I can tell you the best answer though: **hdf5r**. Just use that, and ignore that others.

That recommendation provided, it is a little bit odd to use because it makes a lot of use of **R6** classes that we only very briefly mentioned in the introduction section (sorry, we could not entirely avoid objects). This means it uses classes in a pretty unremitting manner, and I find myself having to use the documentation a lot to remind myself of how all the bits plug together (but at least it is very well documented).

So how can we save a table? With a few steps: we have to make an **HDF5** pointer to a file that we can write to, subset this pointer with the name of the group (here 'example') and then fill this with our data. Finally (and very importantly!) we have to close the file. Do not forget to do this:

```
if(file.exists('../data/test.h5')){
  file.remove('../data/test.h5')
}
```

```
## [1] TRUE
```

```
file.h5 = H5File$new('../data/test.h5', mode="w")
file.h5[['example']] = example_DF
file.h5$close_all()
```

We can now read it back:

```
file.h5 = H5File$new('../data/test.h5', mode="r")
file.h5[['example']][] #all rows
```

```
##    num letters    roots
## 1    1       a 1.000000
## 2    2       b 1.414214
## 3    3       c 1.732051
## 4    4       d 2.000000
## 5    5       e 2.236068
## 6    6       f 2.449490
## 7    7       g 2.645751
## 8    8       h 2.828427
## 9    9       i 3.000000
## 10  10       j 3.162278
## 11  11       k 3.316625
## 12  12       l 3.464102
## 13  13       m 3.605551
## 14  14       n 3.741657
## 15  15       o 3.872983
## 16  16       p 4.000000
## 17  17       q 4.123106
## 18  18       r 4.242641
## 19  19       s 4.358899
## 20  20       t 4.472136
## 21  21       u 4.582576
## 22  22       v 4.690416
## 23  23       w 4.795832
## 24  24       x 4.898979
## 25  25       y 5.000000
## 26  26       z 5.099020
```

```
file.h5[['example']][1:5] #just rows 1:5
```

```
##   num letters    roots
## 1   1       a 1.000000
## 2   2       b 1.414214
## 3   3       c 1.732051
## 4   4       d 2.000000
## 5   5       e 2.236068
```

```
file.h5$close_all()
```

In this package the tables stored as 1D, so we can access specific rows, but we always extract all columns. This makes our interfacing a bit alien, and not much like **FST** which looks so similar to a **data.frame**.

The big advantage of **HDF5** is we can make very complex structures easily though. For example:

```
if(file.exists('../data/complex.h5')){
  file.remove('../data/complex.h5')
}
```

```
## [1] TRUE
```

```
file.h5 = H5File$new('../data/complex.h5', mode="w")
file.h5$create_group('group1')
```

```
## Class: H5Group
## Filename: /Users/aaron/Dropbox (Personal)/AstroStats_Local/data/complex.h5
## Group: /group1
```

```
file.h5[['group1/example']] = example_DF
file.h5$create_attr('group1', 'test attribute info')
```

```
## Class: H5A
## Attribute: group1
## Datatype: H5T_STRING {
##       STRSIZE H5T_VARIABLE;
##       STRPAD H5T_STR_NULLTERM;
##       CSET H5T_CSET_ASCII;
##       CTYPE H5T_C_S1;
##    }
## Space: Type=Simple     Dims=1     Maxdims=1
```

```
file.h5$create_group('group2')
```

```
## Class: H5Group
## Filename: /Users/aaron/Dropbox (Personal)/AstroStats_Local/data/complex.h5
## Group: /group2
```

```
file.h5[['group2/something']] = 1:100
file.h5[['group2/else']] = 'nothing to see here'
file.h5$close_all()
```

This is a bit like an **R** list, but the big plus here is unlike **rds** files we can access subsets of this structure efficiently (i.e. without loading it all into memory first):

```
file.h5 = H5File$new('../data/complex.h5', mode="r")
file.h5$ls()
```

```
##     name     link.type  obj_type num_attrs group.nlinks group.mounted
## 1 group1 H5L_TYPE_HARD H5I_GROUP         0            1             0
## 2 group2 H5L_TYPE_HARD H5I_GROUP         0            2             0
##   dataset.rank dataset.dims dataset.maxdims dataset.type_class
## 1           NA         <NA>           <NA>               <NA>
## 2           NA         <NA>           <NA>               <NA>
##   dataset.space_class committed_type
## 1                <NA>           <NA>
## 2                <NA>           <NA>
```

```
file.h5[['group1/example']][1:5]
```

```
##   num letters    roots
## 1   1       a 1.000000
## 2   2       b 1.414214
## 3   3       c 1.732051
## 4   4       d 2.000000
## 5   5       e 2.236068
```

```
file.h5[['group2/something']][1:10]
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
file.h5[['group2/else']][]
```

```
## [1] "nothing to see here"
```

```
file.h5$attr_open('group1')$read()
```

```
## [1] "test attribute info"
```

```
file.h5$close_all()
```

This makes **HDF5** very useful for storing you own complex data, with metadata attached appropriately to explain the meaning of objects. Do not feel bad that none of the above looks obvious- it is not! The authors managed to find object class based methods to make the extremely complex **HDF5** interface more compact, with the advantage that class based methods are context aware, meaning when programming we see auto complete options (which helps to remind you how things work). With S3 it can be hard to

know what methods might exist (does it have a plot function associated?), but with R6 you can simply see or the legal functions attached directly to the object, which is confusing for functional people (which I am) but handy.

FITS Files

A very important file format in astronomy is FITS (Flexible Image Transport Specification). Despite its name, it can also store data cubes (in fact N dimensional arrays) and multi format tables.

For a few years there were various FITS packages that existed in **R**, but none of them used the well supported **CFITSIO** library that provides a **C** level interface to FITS files. In 2019 I decided to spend a few days writing a proper interface into this lower level library (and adding to it over many days since...), with the result being the **Rfits** package. Unfortunately this cannot be put on CRAN due to warnings produced by **CFITSIO** (these have been reported, but to date not fixed), but a mature and stable version of the package is available from my own **GitHub** which should be easy to install using **remotes**:

```
library(remotes)
# some sensible download options
options(download.file.method = "libcurl")
options(repos="http://cran.rstudio.com/")
install_github('asgr/Rfits')
```

And now load **Rfits**:

```
library(Rfits)
```

FITS Tables The first thing we want to look at is reading and writing table data. The package comes with some handy example data:

```
file_table = system.file('extdata', 'table.fits', package = "Rfits")
temp_table = Rfits_read_table(file_table, header=TRUE)
temp_table[1:5,1:5]
```

```
##    CATAID              OBJID       RA         DEC FIBERMAG_R
##     <int>              <i64>    <num>       <num>      <num>
## 1: 585589 588848899914203328 183.4806 -0.15822451   19.98560
## 2: 585591 588848899914203338 183.4979 -0.16766405   18.72680
## 3: 919107 588848899914203331 183.4896 -0.18662595   20.45358
## 4: 585592 588848899914203383 183.4629 -0.08433424   19.72947
## 5: 585597 588848899914203421 183.4594 -0.14743476   20.72039
```

Notice it properly supports 64 bit integers for *OBJID* (which other packages available do not).

The table you get back when running **Rfits_read_table** with *header*=TRUE looks and behaves exactly like a **data.table** (which is my preferred data table format since it is faster than **data.frame**). However, hidden away in the object attributes is all of the header information!

```
is.data.table(temp_table)
```

```
## [1] TRUE
```

```
str(attributes(temp_table), max.level=1)
```

```
## List of 15
##  $ names            : chr [1:35] "CATAID" "OBJID" "RA" "DEC" ...
##  $ row.names        : int [1:100] 1 2 3 4 5 6 7 8 9 10 ...
##  $ class            : chr [1:3] "Rfits_table" "data.table" "data.frame"
##  $ .internal.selfref:<externalptr>
##  $ keyvalues        :List of 82
##   ..- attr(*, "class")= chr "Rfits_keylist"
##  $ keycomments      :List of 82
##  $ keynames         : chr [1:82] "XTENSION" "BITPIX" "NAXIS" "NAXIS1" ...
##  $ header           : chr [1:82] "XTENSION= 'BINTABLE'           / binary table extension" "BITPI
```

```
##  $ hdr              : chr [1:164] "XTENSION" "BINTABLE" "BITPIX" "8" ...
##  $ raw              : chr "XTENSION= 'BINTABLE'           / binary table extension
##  $ nkey             : int 82
##  $ meta_col         :'data.frame':   35 obs. of  6 variables:
##  $ filename         : chr "/Library/Frameworks/R.framework/Versions/4.5-x86_64/Resources/library/
##  $ ext              : num 2
##  $ extname          : chr "/Users/aaron/GAMA2/TilingCatv46.fits#1"
```

```
attributes(temp_table)$keyvalues$NAXIS1
```

```
## [1] 155
```

```
attributes(temp_table)$keycomments$TFORM22
```

```
## [1] "format for column 22"
```

For most users, this will never matter, but it is good to know that the information is all there if needed.

It is worth carrying out some speed tests compared to **csv**, **feather** and **FST** using the same data:

```
microbenchmark(
  Rfits_write_table(temp_table, '../data/table.fits'),
  write.csv(temp_table, '../data/table.csv'),
  write_feather(temp_table, '../data/table.ft'),
  write_parquet(temp_table, '../data/table.parquet'),
  write.fst(temp_table, '../data/table.fst')
)
```

```
## Unit: microseconds
##                                                expr      min         lq
##  Rfits_write_table(temp_table, "../data/table.fits") 9851.508 10738.5585
##           write.csv(temp_table, "../data/table.csv") 2800.412  3330.7310
##        write_feather(temp_table, "../data/table.ft") 5514.506  6099.8855
##   write_parquet(temp_table, "../data/table.parquet") 9173.583  9893.4510
##           write.fst(temp_table, "../data/table.fst")  720.315   897.4005
##       mean    median        uq       max neval
##  21318.141 11193.691 12326.873 116238.90   100
##   5904.908  3481.932  3731.932  54176.48   100
##  11471.930  6378.704 19994.804  59183.31   100
##  16277.911 10363.923 11967.889  77247.51   100
##   1305.129  1019.298  1242.666  20413.88   100
```

```
microbenchmark(
  Rfits_read_table('../data/table.fits'),
  read.csv('../data/table.csv'),
  read_feather('../data/table.ft'),
  read_parquet('../data/table.parquet'),
  read.fst('../data/table.fst')
)
```

```
## Unit: microseconds
##                                     expr      min        lq      mean   median
##  Rfits_read_table("../data/table.fits") 3428.183 3719.6275 4076.6241 3903.240
##           read.csv("../data/table.csv") 1993.424 2201.7470 2324.3767 2295.499
##        read_feather("../data/table.ft") 5451.495 5950.3930 6642.6320 6125.148
##   read_parquet("../data/table.parquet") 7567.442 8207.3680 8934.9584 8430.524
##           read.fst("../data/table.fst")  440.322  548.8855  662.6613  581.023
##        uq      max neval
##  4116.585 13354.91   100
##  2434.562  2888.20   100
##  6381.787 18816.29   100
##  8729.433 20708.76   100
```

```
##   647.110  1662.28   100
```

For this example table we still find **FST** by far the fastest format, with **Parquet** actually last. This table is still quite small however- once you have more than a few thousand rows it scales better than **csv** and similar to **feather** (and **Rfits**), but **FST** still blitzes the field!

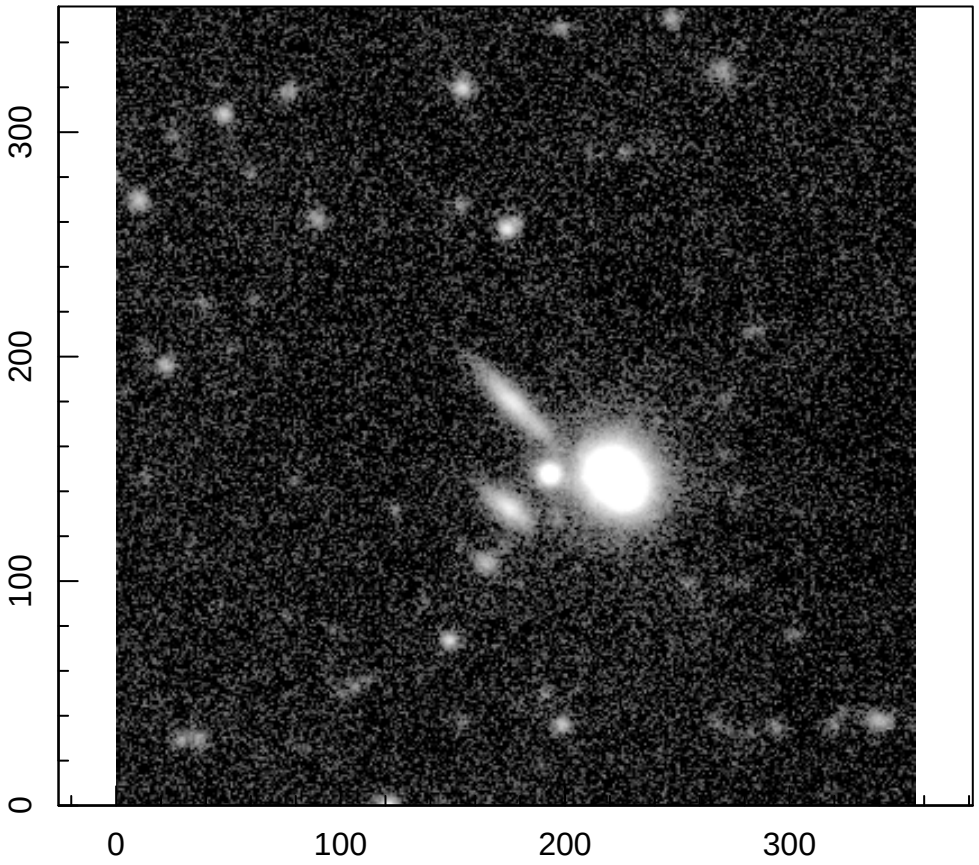
FITS Images and Cubes The other side of **FITS** (and supplying the "I" in its name) is of course image storage (and by a small extension higher dimension cubes and arrays). In this paradigm **FITS** stands apart, because all of the formats mentioned so far are really built around table storage (although **HDF5** can be utilised for image too this is unusual in practice in astronomy, but common in other fields like geology and remote sensing).

Rfits comes with some **FITS** images ready to try out:

```
file_image = system.file('extdata', 'image.fits', package = "Rfits")
temp_image = Rfits_read_image(file_image)
```

Let us have a quick look at the data:

```
magimage(temp_image$imDat)
```



For convenience we can access the matrix part directly:

```
temp_image$imDat[1:5,1:5] #explicit
```

```
##            [,1]       [,2]        [,3]       [,4]       [,5]
## [1,] -0.6531947  4.1757545  1.12943912   6.705467  4.1478391
## [2,] -8.8467922 -0.5689736  0.08958805   3.141321  7.1571307
## [3,]  5.0380459 -3.5135510 -0.51126295 -10.411980 -0.8545529
## [4,]  1.7990382 13.7815027 -1.83640981 -11.101260 -0.6462528
## [5,] 19.5163765 12.3686447 12.78296566  -4.210913  9.5418587
```

```
temp_image[1:5,1:5,header=FALSE] #convenient
```

```
##            [,1]       [,2]        [,3]       [,4]       [,5]
```

```
## [1,] -0.6531947  4.1757545  1.12943912   6.705467  4.1478391
## [2,] -8.8467922 -0.5689736  0.08958805   3.141321  7.1571307
## [3,]  5.0380459 -3.5135510 -0.51126295 -10.411980 -0.8545529
## [4,]  1.7990382 13.7815027 -1.83640981 -11.101260 -0.6462528
## [5,] 19.5163765 12.3686447 12.78296566  -4.210913  9.5418587
```

But it really is a list with a header:

```
str(temp_image)
```

```
## Classes 'Rfits_image', 'list'  hidden list of 14
##  $ imDat      : num [1:356, 1:356] -0.653 -8.847 5.038 1.799 19.516 ...
##  $ keyvalues  :List of 23
##   ..$ SIMPLE : logi TRUE
##   ..$ BITPIX : int -32
##   ..$ NAXIS  : int 2
##   ..$ NAXIS1 : int 356
##   ..$ NAXIS2 : int 356
##   ..$ EXTEND : logi TRUE
##   ..$ EQUINOX: int 2000
##   ..$ EPOCH  : int 2000
##   ..$ WCSAXES: int 2
##   ..$ CRPIX1 : int 178
##   ..$ CRPIX2 : int 178
##   ..$ CRVAL1 : num 352
##   ..$ CRVAL2 : num -31.8
##   ..$ CTYPE1 : chr "RA---TAN"
##   ..$ CTYPE2 : chr "DEC--TAN"
##   ..$ CUNIT1 : chr "deg"
##   ..$ CUNIT2 : chr "deg"
##   ..$ CD1_1  : num -9.42e-05
##   ..$ CD1_2  : int 0
##   ..$ CD2_1  : int 0
##   ..$ CD2_2  : num 9.42e-05
##   ..$ OBJECT : chr "352.2914_-31.8223"
##   ..$ RADESYS: chr "ICRS"
##   ..- attr(*, "class")= chr "Rfits_keylist"
##  $ keycomments:List of 23
##   ..$ SIMPLE : chr "file does conform to FITS standard"
##   ..$ BITPIX : chr "number of bits per data pixel"
##   ..$ NAXIS  : chr "number of data axes"
##   ..$ NAXIS1 : chr "length of data axis 1"
##   ..$ NAXIS2 : chr "length of data axis 2"
##   ..$ EXTEND : chr "FITS dataset may contain extensions"
##   ..$ EQUINOX: chr "equinox of celestial coord. system"
##   ..$ EPOCH  : chr "epoch of celestial coord. system"
##   ..$ WCSAXES: chr "Number of World Coordinate System axes"
##   ..$ CRPIX1 : chr "ref pixel x"
##   ..$ CRPIX2 : chr "ref pixel y"
##   ..$ CRVAL1 : chr "ref pixel x value"
##   ..$ CRVAL2 : chr "ref pixel y value"
##   ..$ CTYPE1 : chr "the coordinate type for the first axis"
##   ..$ CTYPE2 : chr "the coordinate type for the second axis"
##   ..$ CUNIT1 : chr "Axis unit"
##   ..$ CUNIT2 : chr "Axis unit"
##   ..$ CD1_1  : chr "pixel size x"
##   ..$ CD1_2  : chr "xy rotation"
##   ..$ CD2_1  : chr "yx rotation"
##   ..$ CD2_2  : chr "pixel size y"
```

```
##   ..$ OBJECT : chr "GAMA CATAID"
##   ..$ RADESYS: chr "Astrometric system"
##  $ keynames   : chr [1:23] "SIMPLE" "BITPIX" "NAXIS" "NAXIS1" ...
##  $ header     : chr [1:25] "SIMPLE  =                    T / file does conform to FITS standard"
##  $ hdr        : chr [1:46] "SIMPLE" "T" "BITPIX" "-32" ...
##  $ raw        : chr "SIMPLE  =                    T / file does conform to FITS standard
##  $ comment    : chr [1:2] "FITS (Flexible Image Transport System) format is defined in 'Astronomy
##  $ history    : NULL
##  $ nkey       : int 25
##  $ filename   : chr "/Library/Frameworks/R.framework/Versions/4.5-x86_64/Resources/library/Rfits/
##  $ ext        : num 1
##  $ extname    : NULL
##  $ WCSref     : chr "NULL"
```

So it seems a bit curious that temp_image[1:5,1:5] works like it does. The trick is we use the S3 class system to create a special '[' function, because '[' is really just a function, even if it looks different visually. Any **FITS** image that is read in with **Rfits** has a class of *Rfits_image*:

```
class(temp_image)
```

```
## [1] "Rfits_image" "list"
```

So that means we can create a class dependent '[' function that **R** will search for when it tries to subset the top level object:

```
Rfits:::`[.Rfits_image`
```

```
## function(x, i, j, box=201, type='pix', header=TRUE){
##
##   xdim = dim(x)[1]
##   ydim = dim(x)[2]
##
##   if(!missing(box) & missing(i) & missing(j)){
##     i = ceiling(xdim/2)
##     j = ceiling(ydim/2)
##   }
##
##   if(!missing(i)){
##     express = as.character(substitute(i))
##
##     if(express[1] == ':' & length(express) == 3L){
##       if(grepl('end',substitute(i))[3]){
##         start = express[2]
##         end = xdim
##         #i = eval(parse(text=paste0(start,':',end)))
##         i = c(start, end)
##       }
##     }
##   }
##
##   if(!missing(j)){
##     express = as.character(substitute(j))
##
##     if(express[1] == ':' & length(express) == 3L){
##       if(grepl('end',substitute(j))[3]){
##         start = express[2]
##         end = ydim
##         #j = eval(parse(text=paste0(start,':',end)))
##         j = c(start, end)
##       }
##     }
```

```
##   }
##
##
##   if(!missing(i)){
##     if(length(i)==2 & missing(j)){
##       if(i[2]-i[1] !=1){
##         j = as.numeric(i[2])
##         i = as.numeric(i[1])
##       }
##     }
##   }
##
##   if(missing(i)){i = c(1,xdim)}
##   if(missing(j)){j = c(1,ydim)}
##
##   # if(min(i) == 1L){
##   #   if(max(i) == xdim){
##   #     if(min(j) == 1L){
##   #       if(max(j) == ydim){
##   #         return(x) #do nothing!
##   #       }
##   #     }
##   #   }
##   # }
##
##   #This is Rigo's version of the above (a bit more maintainable):
##   arrays = list(i, j)
##   upper_limits = list(xdim, ydim)
##   if(all(mapply(.spans_up_to, arrays, upper_limits))){return(x)}
##
##   if(type=='coord'){
##     if(requireNamespace("Rwcs", quietly=TRUE)){
##       assertNumeric(i,len=1)
##       assertNumeric(j,len=1)
##       ij = Rwcs::Rwcs_s2p(i,j,keyvalues=x$keyvalues,pixcen='R',header=x$raw)[1,]
##       i = ceiling(ij[1])
##       j = ceiling(ij[2])
##     }else{
##       message('The Rwcs package is needed to use type=coord.')
##     }
##   }
##
##   if(length(i) == 1 & length(j) == 1){
##     if(length(box) == 1){box = c(box,box)}
##     i = ceiling(i + c(-(box[1]-1L)/2, (box[1]-1L)/2))
##     j = ceiling(j + c(-(box[2]-1L)/2, (box[2]-1L)/2))
##   }
##
##   safedim_i = .safedim(1L, xdim, min(i), max(i))
##   safedim_j = .safedim(1L, ydim, min(j), max(j))
##
##   tar = array(NA, dim=c(safedim_i$len_tar, safedim_j$len_tar))
##   if(safedim_i$safe & safedim_j$safe){
##     tar[safedim_i$tar,safedim_j$tar] = x$imDat[safedim_i$orig,safedim_j$orig]
##   }
##
##   if(header){
##     if(!isTRUE(x$keyvalues$ZIMAGE)){
```

```
##       x$keyvalues$NAXIS1 = safedim_i$len_tar
##       x$keyvalues$NAXIS2 = safedim_j$len_tar
##     }else{
##       x$keyvalues$ZNAXIS1 = safedim_i$len_tar
##       x$keyvalues$ZNAXIS2 = safedim_j$len_tar
##     }
##     if(!is.null(x$keyvalues$CRPIX1)){
##       x$keyvalues$CRPIX1 = x$keyvalues$CRPIX1 - safedim_i$lo_tar + 1L
##     }
##     if(!is.null(x$keyvalues$CRPIX2)){
##       x$keyvalues$CRPIX2 = x$keyvalues$CRPIX2 - safedim_j$lo_tar + 1L
##     }
##
##     #New keyvalues being added
##     x$keyvalues$XCUTLO = safedim_i$lo_tar
##     x$keyvalues$XCUTHI = safedim_i$hi_tar
##     x$keyvalues$YCUTLO = safedim_j$lo_tar
##     x$keyvalues$YCUTHI = safedim_j$hi_tar
##
##     #New keycomments being added
##     x$keycomments$XCUTLO = 'Low image x range'
##     x$keycomments$XCUTHI = 'High image x range'
##     x$keycomments$YCUTLO = 'Low image y range'
##     x$keycomments$YCUTHI = 'High image y range'
##
##     #New keynames being added
##     #x$keynames = c(x$keynames, 'XCUTLO', 'XCUTHI', 'YCUTLO', 'YCUTHI')
##     #x$keynames['XCUTLO'] = 'XCUTLO'
##     #x$keynames['XCUTHI'] = 'XCUTHI'
##     #x$keynames['YCUTLO'] = 'YCUTLO'
##     #x$keynames['YCUTHI'] = 'YCUTHI'
##     x$keynames = names(x$keyvalues)
##
##     #New history being added
##     x$history = c(x$history, paste0('Subset of original image: x= ',safedim_i$lo_tar,':',safedim_
##
##     x$header = Rfits_keyvalues_to_header(x$keyvalues, x$keycomments, x$comment, x$history)
##     x$raw = Rfits_header_to_raw(x$header)
##     x$hdr = Rfits_keyvalues_to_hdr(x$keyvalues)
##
##     #Now I don't think we need this, so prefer to remove it
##     #detect minimal update:
##     # updateloc_key = grep('NAXIS[1-2]|CRPIX[1-2]|ZNAXIS[1-2]', x$keynames)
##     # updateloc_header = grep(paste(x$keynames[updateloc_key],collapse='|'), x$header)
##     # if(length(updateloc_header) > 0){
##     #   x$header[updateloc_header] = Rfits_keyvalues_to_header(x$keyvalues[updateloc_key], x$keyc
##     #   x$raw = Rfits_header_to_raw(x$header)
##     # }
##
##     output = list(
##       imDat = tar,
##       keyvalues = x$keyvalues,
##       keycomments = x$keycomments,
##       keynames = x$keynames,
##       header = x$header,
##       hdr = x$hdr,
##       raw = x$raw,
##       comment = x$comment,
```

```
##       history = x$history,
##       filename = x$filename,
##       ext = x$ext
##     )
##     class(output) = "Rfits_image"
##     return(output)
##   }else{
##     return(tar)
##   }
## }
## <bytecode: 0x7fbcd61ab088>
## <environment: namespace:Rfits>
```

So temp_image[1:5,1:5] is really just a convenient shortcut to temp_image$imDat[1:5,1:5].

We use a similar trick to get the dimensions out:

```
dim(temp_image)
```

```
## [1] 356 356
```

```
Rfits:::dim.Rfits_image
```

```
## function(x){
##   if(inherits(x$imDat, 'array')){
##     return(dim(x$imDat))
##   }else{
##     return(length(x$imDat))
##   }
## }
## <bytecode: 0x7fbcb387e920>
## <environment: namespace:Rfits>
```

For convenience, we can also access the dimensions of the data for the file on disk (so no need to fully load it first, so useful for very big files):

```
Rfits_dim(file_image)
```

```
## [1] 356 356
```

Using this same class system it is easy (fairly, anyway) to create a pointer-like interface to on-disk data. This is used to access big **FITS** images without loading them into memory:

```
temp_point = Rfits_point(file_image)
```

This pointer contains little data, just some key info (the path to the **FITS** image, the extension to access) and the header:

```
temp_point
```

```
## File path: /Library/Frameworks/R.framework/Versions/4.5-x86_64/Resources/library/Rfits/extdata/im
## Ext num: 1
## Ext name:
## Class: Rfits_pointer
## Type: image
## Dim: 356 356
## Disk size: 0.4889 MB
## BITPIX: -32
## Key N: 23
```

```
str(temp_point)
```

```
## Class 'Rfits_pointer'  hidden list of 12
##  $ filename    : chr "/Library/Frameworks/R.framework/Versions/4.5-x86_64/Resources/library/Rfits
##  $ ext         : num 1
```

```
##  $ keyvalues   :List of 23
##   ..$ SIMPLE : logi TRUE
##   ..$ BITPIX : int -32
##   ..$ NAXIS  : int 2
##   ..$ NAXIS1 : int 356
##   ..$ NAXIS2 : int 356
##   ..$ EXTEND : logi TRUE
##   ..$ EQUINOX: int 2000
##   ..$ EPOCH  : int 2000
##   ..$ WCSAXES: int 2
##   ..$ CRPIX1 : int 178
##   ..$ CRPIX2 : int 178
##   ..$ CRVAL1 : num 352
##   ..$ CRVAL2 : num -31.8
##   ..$ CTYPE1 : chr "RA---TAN"
##   ..$ CTYPE2 : chr "DEC--TAN"
##   ..$ CUNIT1 : chr "deg"
##   ..$ CUNIT2 : chr "deg"
##   ..$ CD1_1  : num -9.42e-05
##   ..$ CD1_2  : int 0
##   ..$ CD2_1  : int 0
##   ..$ CD2_2  : num 9.42e-05
##   ..$ OBJECT : chr "352.2914_-31.8223"
##   ..$ RADESYS: chr "ICRS"
##   ..- attr(*, "class")= chr "Rfits_keylist"
##  $ raw         : chr "SIMPLE  =                    T / file does conform to FITS standard
##  $ header      : logi TRUE
##  $ zap         : NULL
##  $ zaptype     : chr "full"
##  $ allow_write : logi FALSE
##  $ sparse      : int 1
##  $ scale_sparse: logi FALSE
##  $ dim         : int [1:2] 356 356
##  $ type        : chr "image"
```

And yet this works:

```
temp_point[1:5,1:5,header=FALSE]
```

```
##            [,1]       [,2]        [,3]       [,4]       [,5]
## [1,] -0.6531947  4.1757545  1.12943912   6.705467  4.1478391
## [2,] -8.8467922 -0.5689736  0.08958805   3.141321  7.1571307
## [3,]  5.0380459 -3.5135510 -0.51126295 -10.411980 -0.8545529
## [4,]  1.7990382 13.7815027 -1.83640981 -11.101260 -0.6462528
## [5,] 19.5163765 12.3686447 12.78296566  -4.210913  9.5418587
```

You can perhaps guess how this works- our pointer has a particular class (Rfits_image_pointer) and it has its own '[' function:

```
Rfits:::`[.Rfits_pointer`
```

```
## function(x, i, j, k, m, box=201, type='pix', header=x$header,
##                               sparse=x$sparse, scale_sparse=x$scale_sparse, collapse=TRUE){
##
##   xdim = dim(x)[1]
##   ydim = dim(x)[2]
##   zdim = dim(x)[3]
##   tdim = dim(x)[4]
##
##   if(!missing(box) & missing(i) & missing(j)){
##     i = ceiling(xdim/2)
```

```
##     j = ceiling(ydim/2)
##   }
##
##   if(!missing(i)){
##     if(!is.matrix(i)){
##       express = as.character(substitute(i))
##
##       if(express[1] == ':' & length(express) == 3L){
##         if(grepl('end',substitute(i))[3]){
##           start = express[2]
##           end = xdim
##           i = eval(parse(text=paste0(start,':',end)))
##         }
##       }
##     }
##   }
##
##   if(!missing(j)){
##     express = as.character(substitute(j))
##
##     if(express[1] == ':' & length(express) == 3L){
##       if(grepl('end',substitute(j))[3]){
##         start = express[2]
##         end = ydim
##         j = eval(parse(text=paste0(start,':',end)))
##       }
##     }
##   }
##
##   if(!missing(k)){
##     express = as.character(substitute(k))
##
##     if(express[1] == ':' & length(express) == 3L){
##       if(grepl('end',substitute(k))[3]){
##         start = express[2]
##         end = zdim
##         k = eval(parse(text=paste0(start,':',end)))
##       }
##     }
##     k_prov = TRUE
##   }else{
##     k_prov = FALSE
##   }
##
##   if(!missing(m)){
##     express = as.character(substitute(m))
##
##     if(express[1] == ':' & length(express) == 3L){
##       if(grepl('end',substitute(m))[3]){
##         start = express[2]
##         end = tdim
##         m = eval(parse(text=paste0(start,':',end)))
##       }
##     }
##     m_prov = TRUE
##   }else{
##     m_prov = FALSE
##   }
```

```
##
##   if(!missing(i)){
##     if(is.vector(i)){
##       if(length(i)==2 & missing(j)){
##         if(i[2] - i[1] != 1){
##           j = ceiling(i[2])
##           i = ceiling(i[1])
##         }
##       }
##     }
##   }
##
##   if(type=='coord'){
##     if(requireNamespace("Rwcs", quietly=TRUE)){
##       assertNumeric(i,len=1)
##       assertNumeric(j,len=1)
##       ij = Rwcs::Rwcs_s2p(i,j,keyvalues=x$keyvalues,pixcen='R',header=x$raw)[1,]
##       i = ceiling(ij[1])
##       j = ceiling(ij[2])
##     }else{
##       message('The Rwcs package is needed to use type=coord.')
##     }
##   }
##
##   if(isTRUE(x$keyvalues$ZIMAGE)){
##     naxis1 = x$keyvalues$ZNAXIS1
##     naxis2 = x$keyvalues$ZNAXIS2
##     naxis3 = x$keyvalues$ZNAXIS3
##     naxis4 = x$keyvalues$ZNAXIS4
##     datatype = x$keyvalues$ZBITPIX
##   }else{
##     naxis1 = x$keyvalues$NAXIS1
##     naxis2 = x$keyvalues$NAXIS2
##     naxis3 = x$keyvalues$NAXIS3
##     naxis4 = x$keyvalues$NAXIS4
##     datatype = x$keyvalues$BITPIX
##   }
##
##   Ndim = 1
##   if(!is.null(naxis2)){Ndim = 2}
##   if(!is.null(naxis3)){Ndim = 3}
##   if(!is.null(naxis4)){Ndim = 4}
##
##   if(Ndim == 2){
##     if(length(box) == 1){box = c(box,box)}
##     if(!missing(i) & !missing(j)){
##       if(is.vector(i)){
##         if(length(i) == 1 & length(j) == 1){
##           if(length(box) == 1){box = c(box,box)}
##           i = ceiling(i + c(-(box[1]-1L)/2, (box[1]-1L)/2))
##           j = ceiling(j + c(-(box[2]-1L)/2, (box[2]-1L)/2))
##         }
##       }
##     }
##   }
##
##   if(!missing(i)){
##     if(is.vector(i)){
```

```
##       if(is.null(naxis1)){stop('NAXIS1 is NULL: specifying too many dimensions!')}
##       xlo = ceiling(min(i))
##       xhi = ceiling(max(i))
##     }
##   }else{
##     xlo = NULL
##     xhi = NULL
##   }
##   if(!missing(j)){
##     if(is.null(naxis2)){stop('NAXIS2 is NULL: specifying too many dimensions!')}
##     ylo = ceiling(min(j))
##     yhi = ceiling(max(j))
##   }else{
##     ylo = NULL
##     yhi = NULL
##   }
##   if(!missing(k)){
##     if(is.null(naxis3)){stop('NAXIS3 is NULL: specifying too many dimensions!')}
##     zlo = ceiling(min(k))
##     zhi = ceiling(max(k))
##   }else{
##     zlo = NULL
##     zhi = NULL
##   }
##   if(!missing(m)){
##     if(is.null(naxis4)){stop('NAXIS4 is NULL: specifying too many dimensions!')}
##     tlo = ceiling(min(m))
##     thi = ceiling(max(m))
##   }else{
##     tlo = NULL
##     thi = NULL
##   }
##
##   if(!missing(i)){
##     if(is.matrix(i)){
##       output = foreach(row = 1:dim(i)[1], .combine='c')%do%{
##         if(dim(i)[2] == 1){
##           return(Cfits_read_img_subset(filename=x$filename, ext=x$ext, datatype=datatype,
##                                        fpixel0=i[row,1],
##                                        lpixel0=i[row,1]))
##         }else if(dim(i)[2] == 2){
##           return(Cfits_read_img_subset(filename=x$filename, ext=x$ext, datatype=datatype,
##                                        fpixel0=i[row,1], fpixel1=i[row,2],
##                                        lpixel0=i[row,1], lpixel1=i[row,2]))
##         }else if(dim(i)[2] == 3){
##           return(Cfits_read_img_subset(filename=x$filename, ext=x$ext, datatype=datatype,
##                                        fpixel0=i[row,1], fpixel1=i[row,2], fpixel2=i[row,3],
##                                        lpixel0=i[row,1], lpixel1=i[row,2], lpixel2=i[row,3]))
##         }else if(dim(i)[2] == 4){
##           return(Cfits_read_img_subset(filename=x$filename, ext=x$ext, datatype=datatype,
##                                        fpixel0=i[row,1], fpixel1=i[row,2], fpixel2=i[row,3], fpix
##                                        lpixel0=i[row,1], lpixel1=i[row,2], lpixel2=i[row,3], lpix
##         }else{
##           stop('Data type not recognised!')
##         }
##       }
##       return(output)
##     }
```

```
##   }
##
##   output = Rfits_read_image(filename=x$filename, ext=x$ext, header=header,
##                             xlo=xlo, xhi=xhi, ylo=ylo, yhi=yhi, zlo=zlo, zhi=zhi,
##                             tlo=tlo, thi=thi, zap=x$zap, zaptype=x$zaptype, sparse=sparse,
##                             scale_sparse=scale_sparse, collapse=FALSE)
##
##   if(collapse){
##     if(length(dim(output)) == 3){
##       if(dim(output)[3] == 1L & k_prov){
##         output = output[,,1, collapse=TRUE]
##       }
##     }
##
##     if(length(dim(output)) == 4){
##       if(dim(output)[3] == 1L & dim(output)[4] == 1L & k_prov & m_prov){
##         output = output[,,1,1, collapse=TRUE]
##       }else if(dim(output)[4] == 1L & m_prov){
##         output = output[,,,1, collapse=TRUE]
##       }
##     }
##   }
##
##   return(output)
## }
## <bytecode: 0x7fbd34872038>
## <environment: namespace:Rfits>
```

Within this '[' function you can see a call to **Rfits_read_image** that has arguments to access subsets of images using **CFITSIO**, and this is what happens when we request the [1:5,1:5] subset. Because of how **CFITSIO** works, this means we can actually get the same subsetting result with:

```
temp_point[c(1,5),c(1,5),header=FALSE]
```

```
##            [,1]       [,2]        [,3]       [,4]       [,5]
## [1,] -0.6531947  4.1757545  1.12943912   6.705467  4.1478391
## [2,] -8.8467922 -0.5689736  0.08958805   3.141321  7.1571307
## [3,]  5.0380459 -3.5135510 -0.51126295 -10.411980 -0.8545529
## [4,]  1.7990382 13.7815027 -1.83640981 -11.101260 -0.6462528
## [5,] 19.5163765 12.3686447 12.78296566  -4.210913  9.5418587
```

Doing this is slow for small images (better just to load the whole thing into memory), but it is very useful for massive files (10s of GB is common in my work). By creating our own '[' function interface it means we can make our pointer work in functions that normally expect matrix inputs.

These **Rfits** pointers also have their own **dim** and **print** functions:

```
Rfits:::dim.Rfits_pointer
```

```
## function(x){
##   return(x$dim)
## }
## <bytecode: 0x7fbd13a067e8>
## <environment: namespace:Rfits>
```

```
Rfits:::print.Rfits_pointer
```

```
## function(x , ...){
##   cat('File path:',x$filename,'\n')
##   cat('Ext num:',x$ext,'\n')
##   cat('Ext name:',x$keyvalues[['EXTNAME']],'\n')
##   cat('Class: Rfits_pointer\n')
```

```
##   cat('Type:',x$type,'\n')
##   cat('Dim:',x$dim,'\n')
##   cat('Disk size:',round(file.size(x$filename)/(2^20),4),'MB\n')
##   cat('BITPIX:',x$keyvalues[['BITPIX']],'\n')
##   cat('Key N:',length(x$keyvalues),'\n')
## }
## <bytecode: 0x7fbda4734500>
## <environment: namespace:Rfits>
```

These examples with **Rfits** should give you a good insight into how the S3 class system works in **R**, and why you might want to use it for your own package or project (although I personally only use it in packages).

Make Our Own Clever IO

Putting together some of the above concepts, it is fairly easy to create our own easy access **HDF5** based image storage and access format.

First, we can make a simple data saving function:

```
hdf5_image_write = function(data, file='../data/test.h5'){
  if(file.exists(file)){
    file.remove(file)
  }
  file.h5 = H5File$new(file, mode="w")
  file.h5[['image']] = data$imDat
  file.h5[['header']] = data$header
  file.h5$close_all()
}
```

Let us write the image data from above, with the matrix and the simple header parts:

```
hdf5_image_write(temp_image)
```

Next we can make an image reader:

```
hdf5_image_read = function(file, xrange=NULL, yrange=NULL, header=TRUE){
  file.h5 = H5File$new(file, mode="r")
  dims = file.h5[['image']]$dims
  if(is.null(xrange)){
    xrange = 1:dims[1]
  }
  if(is.null(yrange)){
    yrange = 1:dims[2]
  }
  if(header){
    output=list(
      image = file.h5[['image']][xrange,yrange],
      header = file.h5[['header']][]
    )
  }else{
    output = file.h5[['image']][xrange,yrange]
  }
  return(output)
}
```

And just extract the bit of the on disk data we want:

```
hdf5_image_read('../data/test.h5', xrange=1:5, yrange=1:5, header=FALSE)
```

```
##            [,1]       [,2]       [,3]       [,4]       [,5]
```

```
## [1,] -0.6531947  4.1757545  1.12943912   6.705467  4.1478391
## [2,] -8.8467922 -0.5689736  0.08958805   3.141321  7.1571307
## [3,]  5.0380459 -3.5135510 -0.51126295 -10.411980 -0.8545529
## [4,]  1.7990382 13.7815027 -1.83640981 -11.101260 -0.6462528
## [5,] 19.5163765 12.3686447 12.78296566  -4.210913  9.5418587
```

We can compare the speed of this to the image subsetting available in **Rfits**:

```
microbenchmark(
  Rfits_read_image(file_image, xlo=1, xhi=5, ylo=1, yhi=5, header=FALSE),
  hdf5_image_read('../data/test.h5', xrange=1:5, yrange=1:5, header=FALSE)
)
```

```
## Unit: milliseconds
##                                                                                 expr
##  Rfits_read_image(file_image, xlo = 1, xhi = 5, ylo = 1, yhi = 5,      header = FALSE)
##    hdf5_image_read("../data/test.h5", xrange = 1:5, yrange = 1:5,      header = FALSE)
##       min       lq     mean   median       uq      max neval
##  2.777329 3.171585 3.771226 3.276685 3.459712 15.11039   100
##  4.004560 4.301907 4.806179 4.460698 4.649868 16.37301   100
```

So **Rfits** is a factor of a couple faster, but still not a bad effort given we did not have to write any low level code at all! This is the power of **HDF5**- it is easy to create your own flexible file formats that have pretty rapid IO out of the box.

As a last exercise, let us create a simple pointer system for this new format:

```
hdf5_point = function(file){
  class(file) = 'hdf5_point'
  return(file)
}

`[.hdf5_point` = function(x, i=NULL, j=NULL){
  return(hdf5_image_read(x, i, j, header=FALSE))
}

dim.hdf5_point = function(x){
  file.h5 = H5File$new(x, mode="r")
  dims = file.h5[['image']]$dims
}
```

And now we can interact with on disk data just like it is an in memory matrix:

```
test = hdf5_point('../data/test.h5')
test[1:5,1:5]
```

```
##            [,1]       [,2]        [,3]       [,4]       [,5]
## [1,] -0.6531947  4.1757545  1.12943912   6.705467  4.1478391
## [2,] -8.8467922 -0.5689736  0.08958805   3.141321  7.1571307
## [3,]  5.0380459 -3.5135510 -0.51126295 -10.411980 -0.8545529
## [4,]  1.7990382 13.7815027 -1.83640981 -11.101260 -0.6462528
## [5,] 19.5163765 12.3686447 12.78296566  -4.210913  9.5418587
```

```
dim(test)
```

```
## [1] 356 356
```

Using the S3 class system it is easy to make your own user friendly IO interface that abstracts away a lot of the complexity, and therefore reduces the amount of learning and document reading required. If something is fundamentally a matrix on disk then you should make accessing it look like manipulating a matrix with **R**.

Clever IO Update

As of 2021 the prototyped version of the above code was included and extended in **Rfits** itself. This means you can now write multi-extension FITS like HDF5 files using **Rfits** with eases:

```
file_image = system.file('extdata', 'image.fits', package = "Rfits")
temp_image = Rfits_read_image(file_image, header=TRUE)
file_table = system.file('extdata', 'table.fits', package = "Rfits")
temp_table = Rfits_read_table(file_table, header=TRUE)

data = list(temp_image, temp_table)

file_mix_temp = tempfile()

Rfits_write_all_hdf5(data, file_mix_temp)

data2 = Rfits_read_all_hdf5(file_mix_temp)
```

And just to check nothing broke whilst reading and writing (this should all be 0):

```
sum(data[[1]]$imDat - data2[[1]]$imDat)
```

```
## [1] 0
```

```
cols_check = which(sapply(temp_table[1,], is.numeric))
sum(data[[2]][,..cols_check] - data2[[2]][,..cols_check])
```

```
## [1] 0
```

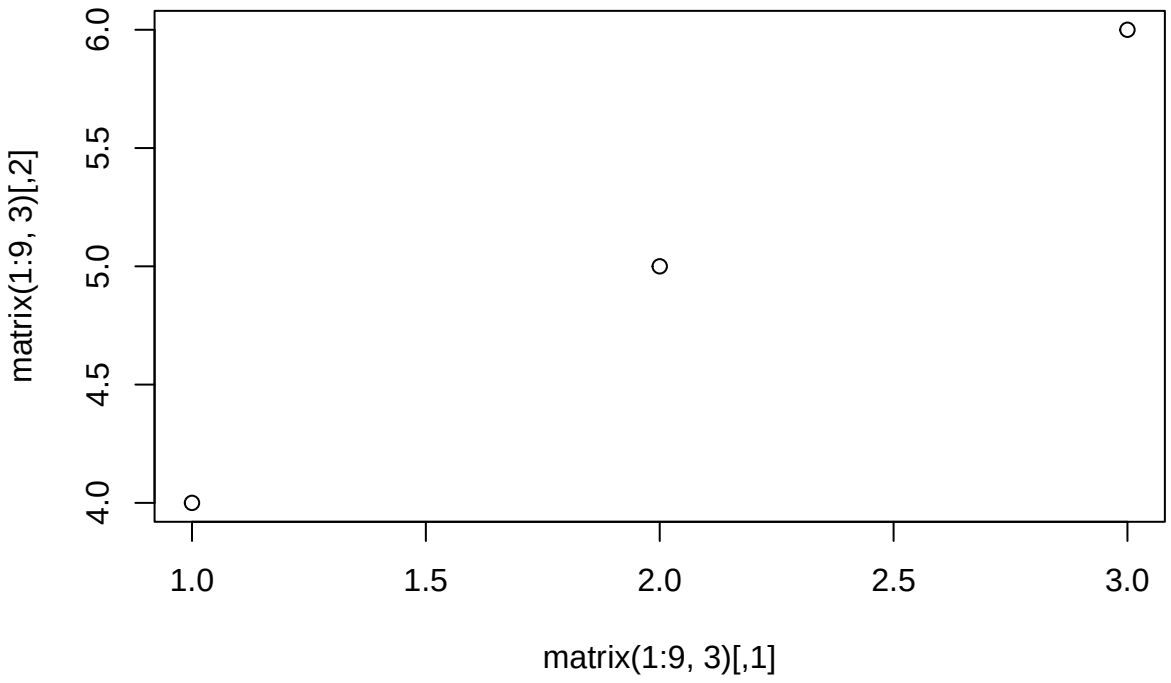
On the user end, everything should look the same as a normal FITS file within **R**:

```
print(data2)
```

```
## Multi-extension FITS loaded with Rfits_read of class Rfits_list
##
## File location: /var/folders/0j/jjlx9l6d6j75ffvr5l7mcg7w0000gn/T//RtmpiTiMGD/filee6b850d4349b
##
## Summary of extension contents:
##
##   Ext   Name       Class Mode Type       Dim Size.MB Key.N
## 1   1  data1 Rfits_image list list 356 x 356  0.9849    23
## 2   2 table2 Rfits_table list list  100 x 35  0.0975    82
```

An exercise for the reader is to make a new plot method to matrices- the default is not very sensible:

```
plot(matrix(1:9,3))
```



Databases

At database is a generic term to refer to a consistent method of accessing and querying data that can be stored in a number of different ways. In principle the file format at the back end is abstracted away, and you are left with an API to worry about.

Arrow Databases

Arrow is supported by the Apache open source foundation (https://arrow.apache.org), and is an in-memory format for data that allows consistent operations between programs (no need to keep saving and loading the data in memory as you switch between e.g. **R** and **Python**). I have spend a decent amount of time looking at low effort methods for working on Medium data, and my feeling is Apache **Arrow** is the sweet spot. It is threaded by default, and for the most part you can trust the defaults (re fiddly things like chunk sizes for data etc). **Arrow** is well supported in **R** (**arrow**), **Python** (**pyarrow**) and **Julia** (**Arrow.jl**) and many others, so it also meets the requirement to be cross-language and flexible. I prefer the speed (this is much faster in benchmarks), simplicity and flexibility of this system rather than setting up a **MiniSQL** database (or similar). Partly this is because it is easier to setup, but I also really dislike SQL syntax for data exploration.

In short, **Arrow** Datasets (their name for a database) are a way to work with large on-disk objects and treat them like they are in memory (for the most part). The backend can be a number of different formats (two popular ones, CSV and Parquet, are discussed below). Once the virtual Dataset has been made the **R arrow** package allows us to interact with it using popular **dplyr** verbs (most of them exist). When using this workflow it is common to use the pipe operator **%>%** to send the result of one operation into the next. The clever part is the lazy evaluation of **R** means only the minimal outputs that are needed are extracted, i.e. if we only want certain rows and columns then the disk is only touched where absolutely necessary to extract this data- there is no expensive full read of each table into RAM.

Arrow lets you build mini simple databases that allow you to access datasets that are larger than memory. A classic example would be a very large table, say 200 GB in size (we work with tables this big). This would be larger than most computer RAM, but if you wanted to extract out a logical subset that *does* fit inside your machine memory **Arrow** offers a simple solution. You can store multiple smaller tables in a number of formats, but **Arrow** most commonly uses csv or **Parquet** back ends.

```
set.seed(666)
tempDT = data.table(let=letters[sample(26,1e3,TRUE)], num=rnorm(1e3), loc=rep(1:10, each=100))
```

We can now make a simple 10 file Dataset:

```
Parq_dir = paste0('../data/Parq_DB/')
#partitioning tells it how to split the database
write_dataset(tempDT, Parq_dir, format='parquet', partitioning='loc')
```

Let us check what it actually wrote to disk:

```
list.files('../data/Parq_DB/', recursive=TRUE)
```

```
##  [1] "loc=1/part-0.parquet"  "loc=10/part-0.parquet" "loc=2/part-0.parquet"
##  [4] "loc=3/part-0.parquet"  "loc=4/part-0.parquet"  "loc=5/part-0.parquet"
##  [7] "loc=6/part-0.parquet"  "loc=7/part-0.parquet"  "loc=8/part-0.parquet"
## [10] "loc=9/part-0.parquet"
```

The sub folder names contain the *loc* value of the sub file, which in this case has been saved as a **Parquet**.

We can now interact with it using **dplyr** verbs:

```
Parq_DB = open_dataset(Parq_dir, format='parquet')
print(Parq_DB) #should be 101 files, because there will be a parquet metadata file
```

```
## FileSystemDataset with 10 Parquet files
## 3 columns
## let: string
## num: double
## loc: int32
##
## See $metadata for additional Schema metadata
```

```
output1 = Parq_DB |> filter(let == 'a') |> collect()
output1[1:10,]
```

```
##        let          num   loc
##     <char>        <num> <int>
##  1:      a  0.574302647     2
##  2:      a  0.146423651     2
##  3:      a  1.545056387    10
##  4:      a -0.225854361    10
##  5:      a  0.582118742    10
##  6:      a  1.086742682     1
##  7:      a -0.463683755     1
##  8:      a  0.001793171     1
##  9:      a  1.217304737     3
## 10:      a -1.460914846     3
```

```
output2 = Parq_DB |> filter(loc == 2) |> collect()
output2[1:10,]
```

```
##        let         num   loc
##     <char>       <num> <int>
##  1:      v  0.03573948     2
##  2:      b -0.05628941     2
##  3:      k  0.22915000     2
##  4:      r -0.15805058     2
##  5:      x -0.59850475     2
##  6:      e  0.67398108     2
##  7:      f  1.01429996     2
##  8:      n -1.14196110     2
##  9:      w -1.49122566     2
## 10:      i -1.11119035     2
```

```
output3 = Parq_DB |> filter(num > 0) |> collect()
output3[1:10,]
```

```
##        let        num   loc
##     <char>      <num> <int>
##  1:      q 1.31520125    10
##  2:      i 0.52400965    10
##  3:      j 0.24372615    10
##  4:      i 0.72362422    10
##  5:      l 0.80774289    10
##  6:      c 1.26240090    10
##  7:      u 0.63725347    10
##  8:      o 0.06000089    10
##  9:      l 2.12874785    10
## 10:      i 1.00393540    10
```

You can divide Datasets by multiple discrete entities. In general you want to split by the most common search type at the top level (this will be fastest). In this case searches by *loc* will be faster than those on *let* even though both are discrete. This should be pretty obvious- if we want to search $loc = 1$ we can just load the entire **Parquet** file in the 'loc=1' directory. How simple!

SQL Databases

Probably the most popular on disk database format is **SQLite** or **MySQL**, where **SQL** is just a common language to interface with the databases. **SQL** databases are heavily used for storing and accessing astronomy survey data (e.g. **SDSS** and **GAMA**) since it is open, rapid, efficient on disk, and flexible to access (you only extract the exact data you need using human readable queries).

It is a little bit different to other formats, where we first need to define an in-memory connection object that we can then use to write and read table data. This is a bit like the **HDF5** connection we saw earlier, but here it is written in a more functional manner. As as simple example we can write out our example table data with:

```
con = dbConnect(SQLite(), "../data/table.sql") #setup connection, creating file if necessary
dbWriteTable(con, "extable", as.data.frame(temp_table), overwrite=TRUE) #write database to connectio
dbDisconnect(con) #disconnect
```

Now we can make a new connection to this on-disk SQL database:

```
con = dbConnect(SQLite(), "../data/table.sql") #setup connection
```

And now we can extract the parts we want using the SQL query language (this is too complex to discuss fully here, but the below shows some simple examples which should at least be human readable):

```
dbGetQuery(con, 'SELECT * FROM extable')[1:5,1:5] #extract everything and subset in R
```

```
##   CATAID              OBJID       RA         DEC FIBERMAG_R
## 1 585589 588848899914203328 183.4806 -0.15822451   19.98560
## 2 585591 588848899914203338 183.4979 -0.16766405   18.72680
## 3 919107 588848899914203331 183.4896 -0.18662595   20.45358
## 4 585592 588848899914203383 183.4629 -0.08433424   19.72947
## 5 585597 588848899914203421 183.4594 -0.14743476   20.72039
```

```
dbGetQuery(con, 'SELECT * FROM extable LIMIT 5')
```

```
##   CATAID              OBJID       RA         DEC FIBERMAG_R  R_PETRO  U_MODEL
## 1 585589 588848899914203328 183.4806 -0.15822451   19.98560 19.45307 21.89456
## 2 585591 588848899914203338 183.4979 -0.16766405   18.72680 17.60359 19.75822
## 3 919107 588848899914203331 183.4896 -0.18662595   20.45358 20.02004 21.41414
## 4 585592 588848899914203383 183.4629 -0.08433424   19.72947 18.43912 21.37582
## 5 585597 588848899914203421 183.4594 -0.14743476   20.72039 19.62761 21.24884
##    G_MODEL  R_MODEL  I_MODEL  Z_MODEL SURVEY_CODE        Z NQ NQ2_FLAG
## 1 20.52974 19.45366 18.94580 18.57926           5  0.18341  4        0
## 2 18.38727 17.65539 17.20334 16.92550           1  0.12429  5        0
## 3 20.35667 20.06804 19.94397 19.80536           0 -1.00000  0        0
```

```
## 4 19.53055 18.33012 17.76440 17.35993           5  0.17900  4        0
## 5 20.51169 19.50901 19.12018 18.97181           5  0.43697  4        0
##               SPECID   VEL_ERR NUM_GAMA_SPEC     R_SB     SG_SEP  SG_SEP_JK
## 1     G12_Y6_057_054  32.62632             2 21.09135 0.41173172 -9.9899998
## 2 323176188157650944  10.31254             0 20.70696 1.23701096 -9.9899998
## 3                xxx -99.90000             0 21.16995 0.05849838  0.5873043
## 4     G12_Y3_022_187  35.40370             1 22.11809 1.41103745 -9.9899998
## 5     G12_Y1_GD1_015  30.84260             1 22.65192 1.56956482 -9.9899998
##     K_AUTO RADIO_FLUX HATLAS_FLAG AREA_FLAG TARGET_FLAGS SURVEY_OLDCLASS
## 1 -9.99000      -9.99           0         2        15512               6
## 2 -9.99000      -9.99           0         2        15613               7
## 3 19.19041      -9.99           0         2         8320               1
## 4 -9.99000      -9.99           0         2        15612               7
## 5 -9.99000      -9.99           0         2        15512               6
##   SURVEY_CLASS PRIORITY_CLASS NEIGHBOUR_CLASS TC_V11_ID MASK_IC_10 MASK_IC_12
## 1            5              2               0    585589          0          0
## 2            6              2               1    585591          0          0
## 3            1              5               0    919107          0          0
## 4            6              2               0    585592          0          0
## 5            5              2               1    585597          0          0
##   VIS_CLASS VIS_CLASS_USER
## 1       255            xxx
## 2         0            xxx
## 3       255            xxx
## 4         0            xxx
## 5         0            xxx
```

```
dbGetQuery(con, 'SELECT CATAID, RA FROM extable LIMIT 5')
```

```
##   CATAID       RA
## 1 585589 183.4806
## 2 585591 183.4979
## 3 919107 183.4896
## 4 585592 183.4629
## 5 585597 183.4594
```

```
dbGetQuery(con, 'SELECT CATAID, RA FROM extable WHERE "RA" > 184 LIMIT 5')
```

```
##   CATAID       RA
## 1 585665 184.0142
## 2 585683 184.0147
## 3 919251 184.0057
## 4 585687 184.1006
## 5 585688 184.1033
```

We can see how rapid SQL is for accessing certain columns compared to **Rfits** and **FST**, which can both extract specific columns too:

```
microbenchmark(
  temp_table[,c('CATAID','RA')],
  dbGetQuery(con, 'SELECT CATAID, RA FROM extable'),
  Rfits_read_table('../data/table.fits', cols = c('CATAID', 'RA')),
  read.fst('../data/table.fst', columns = c('CATAID','RA'))
)
```

```
## Unit: microseconds
##                                                          expr     min
##                                  temp_table[, c("CATAID", "RA")]  88.088
##                 dbGetQuery(con, "SELECT CATAID, RA FROM extable") 354.455
##  Rfits_read_table("../data/table.fits", cols = c("CATAID", "RA")) 735.829
##        read.fst("../data/table.fst", columns = c("CATAID", "RA")) 107.459
```

```
##        lq     mean   median       uq      max neval
##  133.9565 144.7073 149.4280 159.7300  231.576   100
##  402.6200 426.1569 419.4450 438.3730  664.523   100
##  783.0685 825.7540 822.9375 841.6095 1141.904   100
##  147.3040 159.8089 162.3135 172.5005  267.248   100
```

SQL appears to be faster than **Rfits**, but as we might expect by now it is smoked by **FST**. Unbelievably, **FST** is not even much slower than directly subsetting the *temp_table* object that is stored in RAM!

To be neat we should now close our SQL connection explicitly (although in practice, most things should work without doing this):

```
dbDisconnect(con)
```

DuckDB Databases

Almost identically to the above example with **SQLite** we can test the somewhat new **DuckDB** database format:

```
con = dbConnect(duckdb(), "../data/table.duck") #setup connection, creating file if necessary
dbWriteTable(con, "extable", as.data.frame(temp_table), overwrite=TRUE) #write database to connectio
dbDisconnect(con) #disconnect
```

The only real change is switching to connecting via the the 'duckdb()' function rather than 'SQLite()'. This works because both use the abstracted database interface package (**DBI**) to handle the link between the code you want to execute and the actual database backend. This can be a bit confusing for people, since there is a now a clear distinction between the SQL language and the actual database running in the background, which does not need to any sort of SQL system at all (e.g. MySQL, SQLite etc)!

Now we can make a new connection to this on-disk DuckDB database:

```
con = dbConnect(duckdb(), "../data/table.duck") #setup connection
```

And now we can extract the parts we want using the SQL query language (this is too complex to discuss fully here, but the below shows some simple examples which should at least be human readable):

```
dbGetQuery(con, 'SELECT * FROM extable')[1:5,1:5] #extract everything and subset in R
```

```
##   CATAID         OBJID       RA         DEC FIBERMAG_R
## 1 585589 2.651129e-269 183.4806 -0.15822451   19.98560
## 2 585591 2.651129e-269 183.4979 -0.16766405   18.72680
## 3 919107 2.651129e-269 183.4896 -0.18662595   20.45358
## 4 585592 2.651129e-269 183.4629 -0.08433424   19.72947
## 5 585597 2.651129e-269 183.4594 -0.14743476   20.72039
```

```
dbGetQuery(con, 'SELECT * FROM extable LIMIT 5')
```

```
##   CATAID         OBJID       RA         DEC FIBERMAG_R  R_PETRO  U_MODEL
## 1 585589 2.651129e-269 183.4806 -0.15822451   19.98560 19.45307 21.89456
## 2 585591 2.651129e-269 183.4979 -0.16766405   18.72680 17.60359 19.75822
## 3 919107 2.651129e-269 183.4896 -0.18662595   20.45358 20.02004 21.41414
## 4 585592 2.651129e-269 183.4629 -0.08433424   19.72947 18.43912 21.37582
## 5 585597 2.651129e-269 183.4594 -0.14743476   20.72039 19.62761 21.24884
##    G_MODEL  R_MODEL  I_MODEL  Z_MODEL SURVEY_CODE        Z NQ NQ2_FLAG
## 1 20.52974 19.45366 18.94580 18.57926           5  0.18341  4        0
## 2 18.38727 17.65539 17.20334 16.92550           1  0.12429  5        0
## 3 20.35667 20.06804 19.94397 19.80536           0 -1.00000  0        0
## 4 19.53055 18.33012 17.76440 17.35993           5  0.17900  4        0
## 5 20.51169 19.50901 19.12018 18.97181           5  0.43697  4        0
##               SPECID   VEL_ERR NUM_GAMA_SPEC     R_SB     SG_SEP  SG_SEP_JK
## 1     G12_Y6_057_054  32.62632             2 21.09135 0.41173172 -9.9899998
## 2 323176188157650944  10.31254             0 20.70696 1.23701096 -9.9899998
## 3                xxx -99.90000             0 21.16995 0.05849838  0.5873043
```

```
## 4     G12_Y3_022_187  35.40370             1 22.11809 1.41103745 -9.9899998
## 5     G12_Y1_GD1_015  30.84260             1 22.65192 1.56956482 -9.9899998
##     K_AUTO RADIO_FLUX HATLAS_FLAG AREA_FLAG TARGET_FLAGS SURVEY_OLDCLASS
## 1 -9.99000      -9.99           0         2        15512               6
## 2 -9.99000      -9.99           0         2        15613               7
## 3 19.19041      -9.99           0         2         8320               1
## 4 -9.99000      -9.99           0         2        15612               7
## 5 -9.99000      -9.99           0         2        15512               6
##   SURVEY_CLASS PRIORITY_CLASS NEIGHBOUR_CLASS TC_V11_ID MASK_IC_10 MASK_IC_12
## 1            5              2               0    585589          0          0
## 2            6              2               1    585591          0          0
## 3            1              5               0    919107          0          0
## 4            6              2               0    585592          0          0
## 5            5              2               1    585597          0          0
##   VIS_CLASS VIS_CLASS_USER
## 1       255            xxx
## 2         0            xxx
## 3       255            xxx
## 4         0            xxx
## 5         0            xxx
```

```
dbGetQuery(con, 'SELECT CATAID, RA FROM extable LIMIT 5')
```

```
##   CATAID       RA
## 1 585589 183.4806
## 2 585591 183.4979
## 3 919107 183.4896
## 4 585592 183.4629
## 5 585597 183.4594
```

```
dbGetQuery(con, 'SELECT CATAID, RA FROM extable WHERE "RA" > 184 LIMIT 5')
```

```
##   CATAID       RA
## 1 585665 184.0142
## 2 585683 184.0147
## 3 919251 184.0057
## 4 585687 184.1006
## 5 585688 184.1033
```

On this trivial example we can run a new benchmark:

```
microbenchmark(
  dbGetQuery(con, 'SELECT CATAID, RA FROM extable')
)
```

```
## Unit: microseconds
##                                          expr     min      lq     mean
##  dbGetQuery(con, "SELECT CATAID, RA FROM extable") 773.586 844.636 882.0339
##   median      uq      max neval
##  867.173 912.044 1340.515   100
```

Where **DuckDB** does not perform especially well, but really it is designed for much bigger datasets. In fact in recent benchmarks it often comes outright top in terms of speed and memory footprint.

```
dbDisconnect(con)
```