iotools: High-Performance I/O Tools for R

by Taylor Arnold, Michael J. Kane, and Simon Urbanek

Abstract The **iotools** package provides a set of tools for input and output intensive data processing in R. The functions `chunk.apply` and `read.chunk` are supplied to allow for iteratively loading contiguous blocks of data into memory as raw vectors. These raw vectors can then be efficiently converted into matrices and data frames with the **iotools** functions `mstrsplit` and `dstrsplit`. These functions minimize copying of data and avoid the use of intermediate strings in order to drastically improve performance. Finally, we also provide `read.csv.raw` to allow users to read an entire dataset into memory with the same efficient parsing code. In this paper, we present these functions through a set of examples with an emphasis on the flexibility provided by chunk-wise operations. We provide benchmarks comparing the speed of `read.csv.raw` to data loading functions provided in base R and other contributed packages.

Introduction

When processing large datasets, specifically those too large to fit into memory, the performance bottleneck is often getting data from the hard-drive into the format required by the programming environment. The associated latency comes from a combination of two sources. First, there is hardware latency from moving data from the hard-drive to RAM. This is especially the case with "spinning" disk drives, which can have throughput speeds several orders of magnitude less than those of RAM. Hardware approaches for addressing latency have been an active area of research and development since hard-drives have existed. Solid state drives and redundant arrays of inexpensive disks (RAID) now provide throughput comparable to RAM. They are readily available on commodity systems and they continue to improve. The second source comes from the software latency associated with transforming data from its representation on the disk to the format required by the programming environment. This translation slows down performance for many R users, especially in the context of larger data sources.

We can compare the time needed to read, parse, and create a data frame with the time needed to just read data from disk. In order to do this, we will make use of the "Airline on-time performance" dataset, which was compiled for the 2009 American Statistical Association (ASA) Section on Statistical Computing and Statistical Graphics biannual data exposition from data released by the United States Department of Transportation (RITA, 2009). The dataset includes commercial flight arrival and departure information from October 1987 to April 2008 for those carriers with at least 1% of domestic U.S. flights in a given year. In total, there is information for over 120 million flights, with 29 variables related to flight time, delay time, departure airport, arrival airport, and so on. The uncompressed dataset is 12 gigabytes (GB) in size. While 12 GB's may not seem impressively "big" to many readers, it is still large enough to investigate the performance properties and, unlike most other large datasets, it is freely available. Supplemental materials, accessible at https://github.com/statsmaths/iotools-supplement, provide code for downloading the dataset and running the benchmarks included in this paper.

In a first step before measuring the time to load the dataset, column classes are defined so that this part of the data frame processing does not become part of our timings. These can be inferred by reviewing the dataset documentation and inspecting the first few rows of data. The column data types for this dataset are given by:

```
> col_types <- c(rep("integer", 8), "character", "integer", "character",
+                rep("integer", 5), "character", "character",
+                rep("integer", 4), "character", rep("integer", 6))
```

Now, we will read in the file `2008.csv`, which contains comma-separated values with 29 columns and 7,009,728 rows.

```
> system.time(read.csv("2008.csv", colClasses = col_types))[["elapsed"]]
[1] 68.257
```

It takes just over 68 seconds to read the file into R and parse its contents into a data frame object. We can test how much of this time is due to just loading the results from the hard-drive into R's memory. This is done using the `readBin` function to read the file as a raw string of bytes and `file.info` to infer the total size of the file; both functions are provided by the **base** package of R.

```
> system.time(readBin("2008.csv", "raw",
+             file.info("2008.csv")[["size"]]))[["elapsed"]]
[1] 0.493
```

This takes less than one half of a second. It takes about 138 times longer to read and parse the data with `read.csv` than to just read the data in as raw bytes, indicating there may be room for improvement. This is not to say `read.csv` and its associated functions are poorly written. On the contrary, they are robust and do an excellent job inferring data format and shape characteristics. They allow users to import and examine a dataset without knowing how many rows it has, how many columns it has, or its column types. Because of these functions statisticians using R are able to focus on data exploration and modeling instead of file formats and schemas.

While existing functions are sufficient for processing relatively small datasets, larger ones require a different approach. For large files, data are often processed on a single machine by first being broken into a set of consecutive rows or "chunks". Each chunk is loaded and processed individually, before retrieving the next chunk. The results from processing each chunk are then aggregated and returned. Small, manageable subsets are streamed from the disk to the processor with enough memory to represent a single chunk required by the system. This approach is common not only in single machine use but also in distributed environments with technologies such as Spark (Zaharia et al., 2010) and Hadoop MapReduce (Dean and Ghemawat, 2008). Clusters of commodity machines, such as those provided by Amazon Elastic Compute Cloud (EC2) and Digital Ocean, are able to process vast amounts of data one chunk at a time. Many statistical methods are compatible with this computational approach and are justified in a variety of contexts, including Hartigan (1975), Kleiner et al. (2014), Guha et al. (2012), and Matloff (2016).

However, base R together with the contributed packages currently does not provide convenient functionality to implement this common computing pattern. Packages such as **bigmemory** (Kane et al., 2013) and **ff** (Adler et al., 2014) provide data structures using their own binary formats over memory-mapped files stored on disk. The data structures they provide are not native R objects. They therefore do not exhibit properties such as copy-on-write behavior, which avoids making unnecessary copies of the dataset in memory (Rodeh, 2008), and, in general, they cannot be seamlessly integrated with R's plethora of user contributed packages. The **readr** package (Wickham et al., 2016) provides fast importing of `data.frame` objects but it does not support chunk-wise operations for arbitrarily large files. The **foreach** package (Revolution Analytics and Weston, 2015a), and its associated **iterators** package (Revolution Analytics and Weston, 2015b), provide a general framework for chunked processing but does not provide the low-level connection-based utilities for transforming binary data stored on the disk to those native to R.

The **iotools** package provides tools for data processing using any data source represented as a connection (Arnold and Urbanek, 2015). Users of the package can import text data into R and process large datasets iteratively over small chunks. The package can be several orders of magnitude faster when compared to R's native facilities. The package provides general tools for quickly processing large datasets in consecutive chunks and provides a basis for speeding up distributed computing frameworks including Hadoop Streaming (The Apache Software Foundation, 2013) and Spark.

The remainder of this paper introduces the use of the **iotools** package for quickly importing datasets and processing them in R. Examples center around the calculation of ordinary least squares (OLS) slope coefficients via the normal equations in a linear regression. This particular application was chosen because it balances read and write times with processing time.

Input methods and formatters

R's file operations make use of Standard C input and output operations including `fread` and `fwrite`. Data are read in, elements are parsed, and parsed values populate data structures. The **iotools** package also uses the Standard C library but it makes use of "bulk" binary operations including `memchr` and `strchr`. These functions make use of hardware specific, single instruction, multiple data operations (SIMD) and tend to be faster than their Standard I/O counterparts. (See, for example, Zhou and Ross (2002) for a complete overview of the benefits and common implementations of SIMD instructions.) As a result **iotools** is able to find and retrieve data at a higher rate. In addition, an entire dataset or chunk is buffered rather than scanned and transformed line-by-line as in the `read.table` function. Thus, by buffering chunks of data and making use of low-level, system functions **iotools** is able to provide faster data ingestion than what is available in base R.

Importing data with `dstrsplit` and `read.csv.raw`

A core function in the **iotools** package is `dstrsplit`. It takes either a `raw` or `character` vector and splits it into a data frame according to a specified separator. Each column may be parsed into a `logical`, `integer`, `numeric`, `character`, `raw`, `complex` or `POSIXct` vector. Columns of type `factor` are not supported as a method of input, though columns may be converted to a factor once the dataset is

platform	method	integer	logical	num	char	num & char	complex	raw
Ubuntu 16.04	readRDS	0.1	0.1	0.2	7.0	7.2	2.0	0.1
Ubuntu 16.04	dstrsplit	0.8	1.0	2.7	2.6	5.2	5.1	0.6
Ubuntu 16.04	read_csv	1.5	1.7	5.9	2.5	9.7	·	·
Ubuntu 16.04	read.csv	11.0	15.0	50.2	9.0	59.4	96.2	8.4
macOS Sierra	readRDS	0.2	0.2	0.3	5.4	6.5	1.9	0.1
macOS Sierra	dstrsplit	0.9	1.0	2.8	2.4	5.2	5.3	0.6
macOS Sierra	read_csv	1.4	1.5	6.2	2.0	8.0	·	·
macOS Sierra	read.csv	8.6	11.1	39.3	6.7	46.6	70.1	6.2
Windows 7	readRDS	0.1	0.1	0.3	5.6	6.1	2.3	0.1
Windows 7	dstrsplit	1.5	1.3	4.4	2.7	5.6	8.8	0.7
Windows 7	read_csv	1.3	1.9	8.9	1.7	7.6	·	·
Windows 7	read.csv	6.3	7.5	25.7	4.7	29.7	48.1	3.8

Table 1: Time in seconds (average over 10 replications) to import a data frame by element type. Each data frame has 1 million rows and 25 columns of the specified data, except for the "num & char" column which has 25 columns of character values interleaved with 25 columns of numeric columns. Note that `read_csv`, from the **readr** package, does not support `complex` and `raw` types. Linux benchmarks used a server with a 3.7 GHz Intel Xeon E5 and 32 GB of memory. Mac benchmarks used a mid-2015 Macbook Pro with 2.5 GHz Intel Core i7 and 16 GB of memory. Windows benchmarks used a desktop machine with a 3.2 GHz Intel Core i5 CPU and 8 GB of memory.

loaded. It will be shown later that `dstrsplit` can be used in a streaming context and in this case data are read sequentially. As a result, the set of factor levels cannot be deduced until the entire sequence is read. However, in most cases, a caller knows the schema and is willing to specify factor levels after loading the data or is willing to use a single pass to find all of the factor levels.

The tools in **iotools** were primarily developed to support the chunk-wise processing of large datasets that are too large to be read entirely into memory. As an additional benefit it was observed that these functions are also significantly faster when compared to the `read.table` family of functions when importing a large plain-text character separated dataset into R. The `readAsRaw` function takes either a connection or a file name and returns the contents as a `raw` type. Combining this with `dstrsplit`, we can load the `2008.csv` file significantly faster.

```
> system.time(dstrsplit(readAsRaw("2008.csv"), sep = ",",
+             col_types = col_types))[["elapsed"]]
[1] 14.087
```

This takes about 14 seconds, which is roughly a five-fold decrease when compared to the `read.csv` function. In order to simplify its usage, the function `read.csv.raw` was written as a wrapper around `dstrsplit` for users who want to use **iotools** to import data in a manner similar to `read.table`.

```
> system.time(read.csv.raw("2008.csv", colClasses = col_types))[["elapsed"]]
[1] 14.251
```

The performance is very similar to the `dstrsplit` example.

Table 1 shows the time needed to import a data file with 1,000,000 rows and 25 columns using `readRDS`, `dstrsplit`, `read_csv` (from the **readr** package), and `read.table`. Imports were performed for each of R's native types to see how their different size requirements affect performance. The return value of `read_csv` includes additional metadata because a 'tbl_df' object is returned, which is a subtype of R's native data frame. Otherwise all functions return the exact same data. The benchmarks show that, except for character vectors, `readRDS` is fastest. This is unsurprising since `readRDS` stores the binary representation of an R object and importing consists of copying the file to memory and registering the object in R. `read_csv`'s performance is reasonably close to those of **iotools** across all three platforms and data types. The **iotools** functions are generally faster on integer and numeric types, whereas the **readr** functions are slightly faster on character types. In both cases, no performance is lost when dealing with data sets that have mixed data types and results are consistent across operating systems.

Processing the model matrix

In this and the following section, we will show how to estimate based on the entire Airline on-time performance datase the slope coefficients for the follwoing linear regression model using OLS:

$$ArrDelay \sim DayOfWeek + DepTime + Month + DepDelay \tag{1}$$

The OLS slope estimates can be calculated by creating the model matrix and applying the normal equations to derive the coefficients. Given the size of the dataset in this example, it will not be possible to calculate the normal equation matrices directly in memory. Instead, the model matrix will be created sequentially over blocks of rows. As this dataset is further split with each year of data being stored in a separate file, we will also need an outer loop over the available yearly files (1988 to 2008).

We first construct the model matrix and save it in a file on disk; the slope coefficients will be calculated in a second step. Separate processing and model fitting in this case are mostly for the sake of breaking down the example into digestible bits. In many real-world data challenges it may still be a good idea, as it provides an intermediate way of checking whether problems arise while fitting the model. Regardless if problems occur either due to a bug in the code or an interruption in computing services, the model matrix does not need to be recalculated.

In order to construct a large model matrix file, we cycle over the individual data files and work on each separately. Each file is loaded using the `read.csv.raw` function, the variables `DayOfWeek` and `Month` are converted into factors. The departure time variable is given in a 24-hour format, in local time, and with the colon removed from a standard representation of time. For example, 4:30pm is given as "1630" and 5:23am is "523". Our code extracts the hour and minute and converts the time into minutes since midnight; less than 0.5% of the flights depart between midnight and 3:59am, so ignoring the circular nature of time is reasonable in this simple application. Finally, the entire output is stored as a comma separated file with the first column representing the response.

```
> out_file <- file("airline_mm.csv", "wb")
> for (data_file in sprintf("%04d.csv", 1988:2008)) {
+   df <- read.csv.raw(data_file, col_types = col_types)
+   df$DayOfWeek <- factor(df$DayOfWeek, levels = 1:7)
+   df$Month <- factor(df$Month, levels = 1:12)
+   df$DepTime <- sprintf("%04d", d$DepTime)
+   df$DepTime <- as.numeric(substr(df$DepTime, 1, 2)) * 60 +
+                 as.numeric(substr(x$DepTime, 3, 4))
+
+   mf <- model.frame(ArrDelay ~ DayOfWeek + DepTime + DepDelay + Month, df)
+   mm <- cbind(model.response(mf), model.matrix(mf, df))
+   rownames(mm) <- NULL
+   writeBin(as.output(mm, sep = ","), out_file)
+ }
> mm_names <- colnames(mm)
> close(out_file)
```

The output connection is recycled in each iteration of the loop thereby appending each year's data; the names of the model matrix are stored in memory for the next step. In the end we have one large file that contains the entire model matrix. The output is representative of the kinds of large datasets often encountered in industry applications.

Fitting the model with `mstrsplit` and `chunk.apply`

With the model matrices created, the next step is to estimate the slope coefficients β in the model

$$Y = X\beta + \varepsilon, \tag{2}$$

where $Y, \varepsilon \in \mathcal{R}^n$, and $\beta \in \mathcal{R}^d$, $n \geq d$; each element of ε is an i.i.d. random variable with mean zero; and X is a matrix in $\mathcal{R}^{n \times d}$ with full column rank. The analytic solution for estimating the OLS slope coefficients, β, is

$$\widehat{\beta} = \left(X^T X\right)^{-1} X^T Y. \tag{3}$$

platform	method	integer	logical	num	char	complex	raw
Ubuntu 16.04	`readRDS`	0.2	0.2	0.2	0.2	0.2	0.2
Ubuntu 16.04	`mstrsplit`	0.5	0.7	2.4	1.2	4.6	0.4
Ubuntu 16.04	`as.matrix`	11.2	15.2	50.6	16.0	98.3	12.0
macOS Sierra	`readRDS`	0.2	0.2	0.2	0.2	0.2	0.2
macOS Sierra	`mstrsplit`	0.6	0.8	2.7	1.3	4.8	0.4
macOS Sierra	`as.matrix`	8.8	11.3	39.7	12.4	72.1	9.3
Windows 7	`readRDS`	0.1	0.2	0.1	0.1	0.3	0.1
Windows 7	`mstrsplit`	0.7	0.9	3.0	0.8	5.7	0.5
Windows 7	`as.matrix`	6.5	7.7	26.0	10.7	50.5	8.8

Table 2: Time in seconds to import a matrix by element type. Each matrix has 1 million rows and 25 columns. Linux benchmarks used a server with a 3.7 GHz Intel Xeon E5 and 32 GB of memory. Mac benchmarks used a mid-2015 Macbook Pro with 2.5 GHz Intel Core i7 and 16 GB of memory. Windows benchmarks used a desktop machine with a 3.2 GHz Intel Core i5 CPU and 8 GB of memory.

Consider the row-wise partitioning (or chunking) of Equation 2:

$$\begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_r \end{pmatrix} = \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_r \end{pmatrix} \beta + \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_r \end{pmatrix},$$

where $Y_1, Y_2, ..., Y_r$, $X_1, X_2, ..., X_r$ and $\varepsilon_1, \varepsilon_2, ..., \varepsilon_r$ are data partitions where each chunk is composed of subsets of rows of the model matrix. Equation 3 may then be expressed as (Friedman et al., 2001),

$$\hat{\beta} = \left(\sum_{i=1}^{r} X_i^T X_i \right)^{-1} \sum_{i=1}^{r} X_i^T Y_i. \tag{4}$$

The matrices $X_i^T X_i$ and $X_i^T Y_i$ can be calculated on each chunk and then summed to calculate the slope coefficients.

In the previous step data were read into a data frame, but we now need to read data into a numeric matrix. Interestingly enough, this functionality is not provided in base R or the **Matrix** (Bates and Maechler, 2017) package. Users who wanted to read data from a file into a matrix must read it in as a data frame and then convert it using the `as.matrix` function. The **iotools** package fills this gap by providing the function `mstrsplit`, a matrix import function similar to function `dstrsplit`. Table 2 compares the performance of `mstrsplit` with `read.table` followed by a call to `as.matrix` along binary importing using `load`. As with `dstrsplit`, `mstrsplit` outperforms the base R's `read.table` benchmarks by an order of magnitude.

In order to fit the model we will need to read from `airline_mm.csv` in chunks. The function `chunk.apply`, provided in **iotools**, allows us to do this easily over an open connection. By default the data is read in as 32MB chunks, though this can be changed by the `CH.MAX.SIZE` parameter to `chunk.apply`. The parameter `CH.MERGE` describes how the outputs of all the chunks are combined. Common options include `list` or, when the result is a single vector, `c`. Here, we use `chunk.apply` to calculate the matrices $X^T y$ and $X^T X$ over chunks of the data:

```
> ne_chunks <- chunk.apply("airline_mm.csv",
+   function(x) {
+     mm <- mstrsplit(x, sep = ",", type= " numeric")
+     colnames(mm) <- mm_names
+     list(xtx = crossprod(mm[, -1]),
+          xty = crossprod(mm[, -1], mm[, 1, drop = FALSE]))
+   }, CH.MERGE = list)
```

Notice that we do not need to manually specify the chunks; this detail is abstracted away by the `chunk.apply` function, which simply selects contiguous sets of rows for our function to evaluate. The output of the chunk-wise operation can be combined using the `Reduce` function:

```
> xtx <- Reduce("+", Map(function(x) x$xtx, ne_chunks))
> xty <- Reduce("+", Map(function(x) x$xty, ne_chunks))
```

With these results, the regression function can be solved with the normal equations where d is small.

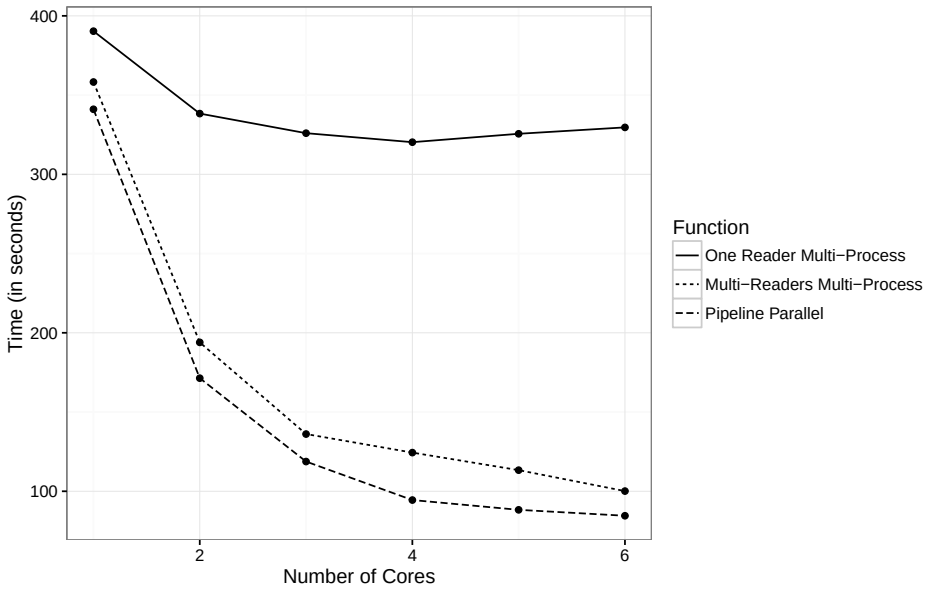

Figure 1: Average time over ten iterations to fit a linear model over the Airline on-time performance dataset as a function of the number of cores used and the number of parallel tasks used.

```
> qr.solve(xtx, xty)
                    [,1]
(Intercept)  0.5564085990
DayOfWeek2   0.5720431343
DayOfWeek3   0.8480978666
DayOfWeek4   1.2436976583
DayOfWeek5   1.0805744488
DayOfWeek6  -1.2235684080
DayOfWeek7  -0.9883340887
DepTime      0.0003022008
DepDelay     0.9329374752
Month2       0.2880436452
Month3      -0.2198123852
...
```

The technique used by the `lm` function is similar to our approach, except that the QR-decomposition there is done directly on X itself rather that on X^TX. The difference is rarely an issue unless the problem is particularly ill-conditioned. In these cases, a small ridge regression penalty can be added to stabilize the solution (Friedman et al., 2001).

Parallel processing of chunks

In the example above the `xtx` and `xty` chunks are calculated serially. The `chunk.apply` function includes a parameter, `parallel`, allowing the user to specify the number of parallel processes, taking advantage of the embarrassingly parallel nature of these calculations. However, it is worth noting that parallelism in the `chunk.apply` function is slightly different from other functions such as `mclapply`.

Most parallel functions in R work by having worker processes receive data and an expression to compute. The master process initiates the computations and waits for them to complete. For I/O-intensive computations this in general means that either the master loads data before initiating the computation or the worker processes load the data. The former case is supported in **iotools** through iterator functions (`idstrsplit` and `imstrsplit`), which are compatible with the **foreach** package. However, in this case, new tasks cannot be started until data has been loaded for each of the workers. Loading the data on the master process may become a bottleneck and it may require much more time to load the data than to process it. The latter approach is also supported in **iotools** and ensures the master process is not a bottleneck. If, however, multiple worker processes on a single machine load a large amount of data from the same disk resource contention at the system level may also cause excessive delays. The operating system has to service multiple requests for data from the same disk having limited I/O capability.

A third option, implemented in `chunk.apply`, provides *pipeline parallelism* where the master process sequentially loads data and then calls `mcparallel` to initiate the parallel computation. When the maximum number of worker processes has been reached the master process *pre-fetches* the next chunk and then blocks on the result of the running worker processes. When the result is returned a newly created worker begins processing the pre-fetched data. In this way the master does not wait idly for worker processing and there is no resource contention since only the master is retrieving data. Pipeline parallelism increases execution throughput when the computation time is around the same order as the load time. When the load time exceeds the execution time, the overhead involved in initiating worker processes and getting their results will yield less desirable performance gains from parallelization. In the case of particularly long load times, the overhead will overwhelm the process and the parallel execution may be slower than a single serial calculation.

Figure 1 shows the times required to calculate X^TX and X^TY for the normal equations in the regression described above using the three approaches described: all workers read, only the master reads, and pipeline parallelism. Pipeline parallelism performs the best overall, with all workers reading following close behind. However, all workers reading will only be able to keep pace with pipeline parallelism as long as there is sufficient hard-drive bandwidth and little contention from multiple reads. As a result, the pipeline parallel approach is likely a more general and therefore preferred strategy.

Conclusion

This paper presents the **iotools** package for the processing of data much larger than memory. Tools are included to efficiently load medium-sized files into memory and to parse raw vectors into matrices and data frames. The chunk-wise functionality is used as a building block to enable the processing of terabyte- and even petabyte-scale data. The examples emphasize computing on a single machine, however **iotools** is by no means limited to this configuration. The "chunk" functions are compatible with any object derived from a connection and can therefore be used with compressed files or even pipes and sockets. Our current work, in fact, uses **iotools** as a building block for more tightly integrating R into the Hadoop Streaming and Spark frameworks.

Bibliography

D. Adler, C. Gläser, O. Nenadic, J. Oehlschlägel, and W. Zucchini. *ff: Memory-Efficient Storage of Large Data on Disk and Fast Access Functions*, 2014. URL https://CRAN.R-project.org/package=ff. R package version 2.2-13. [p2]

T. Arnold and S. Urbanek. *iotools: I/O Tools for Streaming*, 2015. URL https://CRAN.R-project.org/package=iotools. R package version 0.1-12. [p2]

D. Bates and M. Maechler. *Matrix: Sparse and Dense Matrix Classes and Methods*, 2017. URL https://CRAN.R-project.org/package=Matrix. R package version 1.2-8. [p5]

J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. URL https://doi.org/10.1145/1327452.1327492. [p2]

J. Friedman, T. Hastie, and R. Tibshirani. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer-Verlag, 2001. URL https://doi.org/10.1007/978-0-387-21606-5. [p5, 6]

S. Guha, R. Hafen, J. Rounds, J. Xia, J. Li, B. Xi, and W. S. Cleveland. Large complex data: Divide and recombine (D&R) with RHIPE. *Stat*, 1(1):53–67, 2012. URL https://doi.org/10.1002/sta4.7. [p2]

J. A. Hartigan. Necessary and sufficient conditions for asymptotic joint normality of a statistic and its subsample values. *The Annals of Statistics*, 3(3):573–580, 1975. [p2]

M. J. Kane, J. Emerson, and S. Weston. Scalable strategies for computing with massive data. *Journal of Statistical Software*, 55(14):1–19, 2013. URL https://doi.org/10.18637/jss.v055.i14. [p2]

A. Kleiner, A. Talwalkar, P. Sarkar, and M. I. Jordan. A scalable bootstrap for massive data. *Journal of the Royal Statistical Society B*, 76(4):795–816, 2014. URL https://doi.org/10.1111/rssb.12050. [p2]

N. Matloff. Software alchemy: Turning complex statistical computations into embarrassingly-parallel ones. *Journal of Statistical Software*, 71(1):1–15, 2016. URL https://doi.org/10.18637/jss.v071.i04. [p2]

Revolution Analytics and S. Weston. *foreach: Provides Foreach Looping Construct for R*, 2015a. URL https://CRAN.R-project.org/package=foreach. R package version 1.4.3. [p2]

Revolution Analytics and S. Weston. *iterators: Provides Iterator Construct for R*, 2015b. URL https://CRAN.R-project.org/package=iterators. R package version 1.0.8. [p2]

RITA. The Airline on-time performance data set website, 2009. URL http://stat-computing.org/dataexpo/2009/. Research and Innovation Technology Administration, Bureau of Transportation Statistics. [p1]

O. Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage (TOS)*, 3(4):Article No. 2, 2008. URL https://doi.org/10.1145/1326542.1326544. [p2]

The Apache Software Foundation, 2013. Apache Hadoop Streaming, available at http://hadoop.apache.org. [p2]

H. Wickham, J. Hester, and R. Francois. *readr: Read Tabular Data*, 2016. URL https://CRAN.R-project.org/package=readr. R package version 1.0.0. [p2]

M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10:10–10, 2010. URL https://doi.org/10.1145/1755913.1755940. [p2]

J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 145–156. ACM, 2002. [p2]

Taylor Arnold
University of Richmond
28 Westhampton Way, Richmond, VA
USA
tarnold2@richmond.edu

Michael J. Kane
Yale University
300 George Street, New Haven, CT
USA
michael.kane@yale.edu

Simon Urbanek
AT&T Labs – Statistics Research
1 AT&T Way Bedminster, NJ
USA
urbanek@research.att.com