Unleashing the Power of R from Within

Dr. Zahid Ansari

Table of Contents


```
setwd("D:/AMU Computer Science/Courses/Big Data Analytics/Big Data Analytics Using R/Ch3")
```

In this presentation we should be able to: - Understand R's traditional limitations for Big Data analytics and how they can be resolved - Use R packages such as ff, ffbase, ffbase2, and bigmemory to enhance out-of memory performance - Apply statistical methods to large R objects through the biglm and ffbase packages - Enhance the speed of data processing with R libraries supporting parallel computing - Benefit from faster data manipulation methods available in the data.table package

1 Traditional limitations of R

While using large amounts of data, you will be faced with two traditional limitations of R: - Data must fit within the available RAM - R is generally very slow compared to other languages

1.1 Out-of-memory data

- The first of the claims against using R for Big Data is that the entire dataset you want to process has to be smaller than the amount of available RAM.
- Currently, most of the commercially sold, off-the-shelf personal computers are equipped with anything from 4GB to 16GB of RAM, meaning that these values will be the upper bounds of the size of your data which you will want to analyze with R.
- Of course, from these upper limits, you still need to deduct some additional memory resources for other processes to run simultaneously on your machine and provide extra RAM allowance for algorithms and computations you will like to perform during your analysis in the R environment.
- Realistically speaking, the size of your data shouldn't exceed a maximum of 50 to 60% of available memory, unless you want to see your machine become sluggish

and unresponsive, which may also result in R and system crashes, and even potential data loss.

- At the moment, it all may seem pretty grim. However, there are already a number of solutions and workarounds available to R users who want to do some serious data crunching on their machines, even without turning to cloud computing platforms such as Microsoft Azure, Amazon EC2, or Google Cloud Platform.
- These ready-made solutions usually come as R packages, which you can simply download and install with your R distribution to enjoy some extra processing boost.

1.2 Processing speed

- The second argument, which keeps R's antagonists going, is its processing speed.
- Although R's speed is still acceptable in some small-scale computations, it generally lags behind Python and, even more so, the C family of languages.
- There are several reasons why R cannot keep up with others.
- First, R is considered to be an interpreted language, and, as such, its slower code execution comes by definition.
- It's also interesting that, despite a large bulk of the core R being written in C language (almost 39%), a number of R functions created in C (or Fortran), are still much slower than in native code, and this may be partly due to poor memory management in R (for example, time spent on garbage collection, duplications, and vector allocation).
- Second, the core R is single-threaded, meaning that the code of a function or a computation is processed line-by-line, one at a time, engaging a single CPU.
- There are, however, several methods, including some third-party packages, which allow multi-threading.
- Third, the poor performance of the R code may come from the fact that the R language is not formally defined, and its processing speed largely depends on the designs of R implementations, rather than the R language itself.
- There is currently quite a lot of work being done to create new, much faster, alternative implementations and the recent release of Microsoft R Open distribution is an example of another, hopefully more performance optimized, implementation of R.
- Also we need to bear in mind that R is an open-source, community-run project with only a few R core development team members who are authorized to make any changes to the R internals.
- This puts serious constraints on how quickly poorly written parts of R code are altered.

- We also mustn't forget about one very important thing that underlies the whole development of the R language: it wasn't created to break computational speed records, but to provide statisticians and researchers (often with no programming or relevant IT skills) with a rich variety of robust and customizable data analysis and visualization techniques.

- In the following section we will present several techniques for squeezing this extra power out of and from within R to allow data analytics of large datasets on a single computer.

1.3 To the memory limits and beyond

We will learn three very useful and versatile packages which facilitate out-of-memory data processing: ff, ffbase, and ffbase2.

1.3.1 Data transformations and aggregations with the ff and ffbase packages

- ff package although older but still proves to be a popular solution to large data processing with R.

- The title of the package Memory-efficient storage of large data on disk and fast access functions roughly explains what it does.

- It chunks the dataset, and stores it on a hard drive, while the ff data structure (or ffdf data frame), which is held in RAM, like the other R data structures, provides mapping to the partitioned dataset.

- The chunks of raw data are simply binary flat files in native encoding, whereas the ff objects keep the metadata, which describe and link to the created binary files.

- Creating ff structures and binary files from the raw data does not alter the original dataset in any way, so there is no risk that your data may get corrupted or lost.

- The ff package includes a number of general data-processing functions, which support the import of large datasets to R, their basic transformations such as recoding levels of factors, sampling, applying other functions to rows and columns, and setting various attributes to ff objects.

- The resulting data structures can be easily exported to TXT or CSV files.

- The ffbase package, on the other hand, extends the functionality of the original ff library by allowing users to apply a number of statistical and mathematical operations, including basic descriptive statistics, and other useful data transformations, manipulations, and aggregations such as creating subsets, performing cross-tabulations, merging ff objects, and transforming ffdf data frames, converting numeric ff vectors to factors, finding duplicated rows and missing values, and many more.

- Moreover, a very versatile ffdfapply function enables users to apply any function to the created binary flat files, for example, to easily calculate any statistic of interest for each level of a factor, and so on.
- The ffbase package also makes it possible to perform selected statistical models directly on ff objects such as classifications and regressions, least-angle regressions, random-forest classifications, and clustering.
- These techniques are available due to the ffbase package's connectivity with other third-party packages, supporting Big Data analytics such as biglm, biglars, bigrf, and stream.
- In the following section, we will present several of the most widely used ff and ffbase functions, which can be used for Big Data processing and analytics.
- We will be using a flights_sep_oct15.txt dataset, which contains all flights to and from all American airports in September and October 2015.
- The data have been obtained from the Bureau of Transportation Statistics and we've selected 28 variables of interest that describe each flight, such as year, month, day of month, day of week, flight date, airline id, the names of the flight's origin and destination airports, departure and arrival times and delays, distance and air time of the flight, and several others.
- Feel free to mine as many months, years, or specific variables as you wish, but note that a complete year of data containing exactly the same 28 variables which we chose for our example will result in a file of slightly less than 1GB in size.
- The dataset used in this section is limited to two months only (951,111 rows in total), and hence its size is roughly 156MB (almost 19MB when compressed).
- This is,however, enough for us to guide you through some most interesting and relevant applications of the ff, ffbase, and ffbase2 packages.
- In addition to the main dataset, we also provide a small CSV file, which includes the full names of airlines to match with their IDs, contained within the AIRLINE_ID variable in the flight data.
- We will also present statistics related to the elapsed time and used memory for each call for our example data, as well as for a 2GB version of the dataset which covers all flights to and from all American airports between January 2013 and December 2014.
- These benchmarks will be compared with similar calls performed using functions coming from the core R and other relevant third-party packages.
- Before processing the data using packages related to ff, we first need to specify a path to a folder which will store our binary flat files-partitioned chunks of our original dataset.

- In your current working directory (which contains the data), you may explicitly create an additional folder directly from R console:

```
system("mkdir ffdf")

## [1] 1
```

Then, set the path to this newly created folder, which will store ff data chunks, for example:

```
options(fftempdir = "D:/AMU Computer Science/Courses/Big Data Analytics/Big Data Analytics Using R/Ch3/ffdf")
```

- Once this is done we may now upload the data as ff objects. Depending on the format of the data file, you may use either the read.table.ffdf() function or a convenience wrapper read.csv.ffdf() for CSV files.
- In addition to these functions, another package ETLUtils extends the ff importing capabilities to include SQL databases such as Oracle, MySQL, PostgreSQL, and Hive, through functions which use DBI, RODBC, and RJDBC connections.
- Let's then import the data to R using a standard read.table.ffdf() function:

```
library(ff)
flights.ff <- read.table.ffdf(file="flights_sep_oct15.txt", sep=",",
VERBOSE=TRUE, header=TRUE, next.rows=100000, colClasses=NA)

## read.table.ffdf 1..100000 (100000)  csv-read=0.63sec ffdf-write=0.25sec
## read.table.ffdf 100001..200000 (100000)  csv-read=0.64sec ffdf-
write=0.16sec
## read.table.ffdf 200001..300000 (100000)  csv-read=0.62sec ffdf-
write=0.14sec
## read.table.ffdf 300001..400000 (100000)  csv-read=0.64sec ffdf-
write=0.14sec
## read.table.ffdf 400001..500000 (100000)  csv-read=0.63sec ffdf-
write=0.17sec
## read.table.ffdf 500001..600000 (100000)  csv-read=0.62sec ffdf-
write=0.14sec
## read.table.ffdf 600001..700000 (100000)  csv-read=0.65sec ffdf-
write=0.14sec
## read.table.ffdf 700001..800000 (100000)  csv-read=0.62sec ffdf-
write=0.16sec
## read.table.ffdf 800001..900000 (100000)  csv-read=0.64sec ffdf-
write=0.14sec
## read.table.ffdf 900001..951111 (51111)  csv-read=0.33sec ffdf-
write=0.17sec
##  csv-read=6.02sec  ffdf-write=1.61sec  TOTAL=7.63sec
```

- The next.rows argument sets how many rows of data will be assigned to each chunk.

- From the preceding output you can see that the data have been read in nine chunks, with the last part bringing in the remaining 51,111 cases of the original data.

- The output also gives us a basic estimate of the time spent on reading the data file, and writing its ffdf copies to a disk.

- In total, it took over 40 seconds to upload this relatively small dataset, and create the ff files in the previously specified folder.

- The whole process of importing the data resulted in only one very small ffdf object (426.4 KB) being created in the R workspace, and 28 ff files of equal sizes (3.8 MB each) on a disk.

- It is also important here to mention that importing the data entailed only minimal costs in terms of RAM. We may now compare the read.table.ffdf() method with a standard read.table() procedure:

```
flights.table <- read.table("flights_sep_oct15.txt",
sep=",", header=TRUE)
```

- This more conventional method took just over 32 seconds to run; however it resulted in a much larger data.frame object created (101.9 MB) in the R workspace, and a little bit more memory usage.

- As we are working on a relatively small dataset of around 156 MB, the differences in RAM consumption will naturally be quite negligible.

- Let's then compare both approaches on much greater 2 GB-heavy data, which cover two full years (2013-2014) of flights (12,189,293 rows in total).

- The read.table.ffdf() method took almost 456 seconds to import the dataset, in 23 chunks, creating ,as a result, just one ffdf R object of only 516.5 KB in size, and 28 ff data files (48.8 MB each, so nearly 1.37 GB in total).

- What's truly impressive is that the process involved a maximum of about 380 MB of RAM at most, which is generally just slightly above the base level of an R session in RStudio.

- The read.table() approach achieved a slightly faster import (441 seconds), but remember that this method does not include any writing to a disk, so obviously we will expect it to outperform the ff package on this measurement. The real difference, though, is in the resources used to complete the operation.

- The base read.table() function created one large data.frame object (1.3 GB) at huge memory costs; during the execution of this approach the RAM consumption oscillated between 2 GB and 3.6 GB, and at times it spiked up to as much as 4.85 GB.

- After the completion of the method, 4.13 GB of RAM was still in use and only an explicit call for the garbage collection (gc() function) lowered it to 1.47 GB-still more than four times higher than following the read.table.ffdf() application.

- By this time you should see the obvious benefits of using the ff package for uploading large datasets to your R workspace. The question remains, however: What can you do with the ff or ffdf objects loaded to R?

- You can begin by inspecting the ffdf data structure just as you will do with standard data frames in R:

```
class(flights.ff)

## [1] "ffdf"

dim(flights.ff)

## [1] 951111     28

dimnames(flights.ff)

## [[1]]
## NULL
##
## [[2]]
##  [1] "YEAR"              "MONTH"             "DAY_OF_MONTH"
##  [4] "DAY_OF_WEEK"       "FL_DATE"           "UNIQUE_CARRIER"
##  [7] "AIRLINE_ID"        "TAIL_NUM"          "FL_NUM"
## [10] "ORIGIN_AIRPORT_ID" "ORIGIN"            "ORIGIN_CITY_NAME"
## [13] "ORIGIN_STATE_NM"   "ORIGIN_WAC"        "DEST_AIRPORT_ID"
## [16] "DEST"              "DEST_CITY_NAME"    "DEST_STATE_NM"
## [19] "DEST_WAC"          "DEP_TIME"          "DEP_DELAY"
## [22] "ARR_TIME"          "ARR_DELAY"         "CANCELLED"
## [25] "CANCELLATION_CODE" "DIVERTED"          "AIR_TIME"
## [28] "DISTANCE"
```

- The output of the last call will give you an understanding of how ff files on disk are mapped.
- The ffdf object is in fact a list with two components, which store virtual and physical attributes and the row names (in this case the row.names component is empty).
- The attributes hold metadata, which describe each variable and point to specific binary flat files.
- We may now use the read.csv.ffdf() function to upload supplementary information with full names of airlines:

```
airlines.ff <- read.csv.ffdf(file="airline_id.csv",
                             VERBOSE=TRUE, header=TRUE,
                             next.rows=100000,
                             colClasses=NA)

## read.table.ffdf 1..1607 (1607)  csv-read=0sec ffdf-write=0.02sec
##  csv-read=0sec  ffdf-write=0.02sec  TOTAL=0.02sec
```

- We now have both datasets in R, so we can merge them by the AIRLINE_ID variable. As the names of variables differ, we first need to rename the Code variable in the airlines.ff object to AIRLINE_ID and the Description variable to AIRLINE_NM.

```
names(airlines.ff) <- c("AIRLINE_ID", "AIRLINE_NM")
airlines.ff
```

```
## ffdf (all open) dim=c(1607,2), dimorder=c(1,2) row.names=NULL
## ffdf virtual mapping
##             PhysicalName VirtualVmode PhysicalVmode  AsIs
## AIRLINE_ID          Code      integer       integer FALSE
## AIRLINE_NM   Description      integer       integer FALSE
##             VirtualIsMatrix PhysicalIsMatrix PhysicalElementNo
## AIRLINE_ID            FALSE            FALSE                 1
## AIRLINE_NM            FALSE            FALSE                 2
##             PhysicalFirstCol PhysicalLastCol PhysicalIsOpen
## AIRLINE_ID                 1               1           TRUE
## AIRLINE_NM                 1               1           TRUE
## ffdf data
##
AIRLINE_ID
## 1    19031
## 2    19032
## 3    19033
## 4    19034
## 5    19035
## 6    19036
## 7    19037
## 8    19038
## :
:
## 1600 21672
## 1601 21673
## 1602 21674
## 1603 21677
## 1604 21692
## 1605 21693
## 1606 21694
## 1607 21697
##
AIRLINE_NM
## 1    Mackey International Inc.: MAC
## 2    Munz Northern Airlines Inc.: XY
## 3    Cochise Airlines Inc.: COC
## 4    Golden Gate Airlines Inc.: GSA
## 5    Aeromech Inc.: RZZ
## 6    Golden West Airlines Co.: GLW
## 7    Puerto Rico Intl Airlines: PRN
## 8    Air America Inc.: STZ
## :
:
```

```
## 1600 Inselair Aruba NV: 8I
## 1601 J&M Alaska Air Tours, Inc. d/b/a Alaska Air Transit: 2EQ
## 1602 Compagnie Aerienne Inter Regionale Express dba Air Antilles Express &
Air Guyane: 3SD
## 1603 Cebu Air Inc d/b/a Cebu Pacific Air: 5J
## 1604 Azerbaijan Airlines CJSC: J2
## 1605 Cavok Air LLC: 2GQ
## 1606 Silk Way West Airlines: 7L
## 1607 Orenburg Airlines: R2
```

- Let's merge both objects using the merge.ffdf() method:

```
library(ffbase)
flights.data.ff <- merge.ffdf(flights.ff, airlines.ff, by="AIRLINE_ID")
```

- The resulting flights.data.ff data frame is only 551.2 KB in size, and the merging process did not increase memory consumption.
- A similar operation executed on the 2 GB dataset with the use of the merge.ffdf() function from the ffbase package took just over 26 seconds to complete and, as a result of that, it created a ffdf data structure of 641.3 KB with minimal RAM costs.
- The large dataset previously uploaded to the R session with the standard read.table() function, and now merged with a small file containing names of airlines using the base merge() method, took more than 73 seconds to run, and increased the object size stored in RAM from 1.37 GB to 1.41 GB.
- What is even more striking is that, during this data merging, the memory usage peaked on a few occasions to 6.4 GB.
- It is clear to see how the ff approach can benefit Big Data processing and manipulation.
- The traditional core R methods, for example read.table() or merge(), applied on a dataset of just 2 GB, would produce out-of-memory errors on machines equipped with only 4 GB of RAM, and would most likely cause considerable problems even on PCs with 8 GB of RAM installed.
- The ff and ffbase packages provide, then, a very handy mechanism for avoiding memory-related issues at initial stages of large data processing.
- With ff and ffdf objects you can use a number of base R functions without the need to transform them into native R data structures such as a data.frame or a vector.
- We saw that earlier when we applied the names() function to an ffdf data frame, to rename its variables.
- In a similar way, you can use unique() to extract all the names of the states for departing flights in our dataset:

```
origin_st <- unique(flights.data.ff$ORIGIN_STATE_NM)
origin_st
```

```
## ff (open) integer length=52 (52) levels: Alabama Alaska Arizona Arkansas
California Colorado Connecticut Florida Georgia Hawaii Idaho Illinois Indiana
Iowa Kansas Kentucky Louisiana Maine Maryland Massachusetts Michigan
Minnesota Mississippi Missouri Montana Nebraska Nevada New Hampshire New
Jersey New Mexico New York North Carolina North Dakota Ohio Oklahoma Oregon
```

```
Pennsylvania Puerto Rico Rhode Island South Carolina South Dakota Tennessee
Texas U.S. Pacific Trust Territories and Possessions U.S. Virgin Islands Utah
Vermont Virginia Washington West Virginia Wisconsin Wyoming
##                  [1]                  [2]                  [3]
## Alabama             Alaska              Arizona
##                  [4]                  [5]                  [6]
## Arkansas            California          Colorado
##                  [7]                  [8]
## Connecticut         Florida                               :
##                 [45]                 [46]                 [47]
## U.S. Virgin Islands Utah                Vermont
##                 [48]                 [49]                 [50]
## Virginia            Washington          West Virginia
##                 [51]                 [52]
## Wisconsin           Wyoming
```

- Alternatively, the ffbase package provides the unique.ff() function to apply to ff vectors.
- In the same way we are able to perform a cross-tabulation with the table.ff() method, for example the count of flights for each unique state of origin:

```
orig_state_tab <- table.ff(flights.data.ff$ORIGIN_STATE_NM, exclude = NA)
orig_state_tab

##
##                                              Alabama
##                                                 4744
##                                               Alaska
##                                                 5697
##                                              Arizona
##                                                28060
##                                             Arkansas
##                                                 4374
##                                           California
##                                               117835
##                                             Colorado
##                                                37968
##                                          Connecticut
##                                                 3280
##                                              Florida
##                                                64577
##                                              Georgia
##                                                66401
##                                               Hawaii
##                                                15593
##                                                Idaho
##                                                 3190
##                                             Illinois
##                                                71549
```

```
##                                                    Indiana
##                                                       7154
##                                                       Iowa
##                                                       2793
##                                                     Kansas
##                                                       1910
##                                                   Kentucky
##                                                       6626
##                                                  Louisiana
##                                                      11615
##                                                      Maine
##                                                       1016
##                                                   Maryland
##                                                      15741
##                                              Massachusetts
##                                                      20165
##                                                   Michigan
##                                                      24208
##                                                  Minnesota
##                                                      22018
##                                                Mississippi
##                                                       2326
##                                                   Missouri
##                                                      16767
##                                                    Montana
##                                                       2847
##                                                   Nebraska
##                                                       3618
##                                                     Nevada
##                                                      27068
##                                              New Hampshire
##                                                       1185
##                                                 New Jersey
##                                                      19402
##                                                 New Mexico
##                                                       4044
##                                                   New York
##                                                      43191
##                                             North Carolina
##                                                      27226
##                                               North Dakota
##                                                       2972
##                                                       Ohio
##                                                      13184
##                                                   Oklahoma
##                                                       6055
##                                                     Oregon
##                                                      10618
##                                               Pennsylvania
##                                                      17996
```

```
##                                     Puerto Rico
##                                            3907
##                                    Rhode Island
##                                            2145
##                                  South Carolina
##                                            5347
##                                    South Dakota
##                                            1507
##                                       Tennessee
##                                           13993
##                                           Texas
##                                          110917
## U.S. Pacific Trust Territories and Possessions
##                                              78
##                             U.S. Virgin Islands
##                                             549
##                                            Utah
##                                           17647
##                                         Vermont
##                                             696
##                                        Virginia
##                                           25445
##                                      Washington
##                                           22964
##                                   West Virginia
##                                             437
##                                       Wisconsin
##                                            9235
##                                         Wyoming
##                                            1231
```

- When running table.ff() and table() functions on a ffdf structure, and a standard R data.frame object, respectively, you will see certain differences in how both functions perform.
- These differences are, again, best seen when processing a large dataset, for example 2 GB in size.
- The ff approach uses a maximum of 350 to 360 MB of RAM, but it takes almost 12 seconds to complete. The standard table() function is much faster on this data.frame object finishing the job in just over 1 second, but it increases the memory consumption by 700 MB, which, combined with the size of the data.frame already stored in RAM by R, and other earlier processes run on this object, uses up to 2.8 GB of available memory.
- It constitutes up to 10x greater RAM cost than through the ff and ffbase packages.
- Following cross-tabulations, you may easily use other generic functions on ff and ffdf objects to get some basic descriptive statistics on your data, for example, mean(), quantile(), range(), and others.

- Both ff and ffdf structures also work very well with functions contained in third-party packages such as describe() from the Hmisc package; however, in these situations, we need to explicitly tell R to treat our ffdf object as a standard data.frame using the as.data.frame.ffdf() function from the ff package:

```
library(Hmisc)
describe(as.data.frame(ffdf(flights.data.ff$DISTANCE)))

## as.data.frame(ffdf(flights.data.ff$DISTANCE))
##        n  missing distinct     Info     Mean      Gmd      .05
##   951111        0     1241        1    816.2    637.6      168
##      .10      .25      .50      .75      .90      .95
##      224      370      641     1050     1721     2239
##
## lowest :   31   36   67   68   69, highest: 4243 4502 4817 4962 4983
```

- Again, if you run describe(), or a core R summary() function, on the big data with over 12,000,000 rows using an as.data.frame.ffdf() wrapper on a ffdf object, the memory consumption is relatively small (a spike of up to 920 MB for describe()), compared with describe() applied on a standard, large data.frame (a maximum memory usage of 5.2 GB).

- This time there is also barely any difference in the processing speed

- The ff-approach allows other data manipulation methods. For example, it is possible to convert a numeric ff vector to a factor ff using the cut.ff() function.

- In our example we will transform the DAY_OF_WEEK numeric variable to a new factor variable called WEEKDAY:

```
# flights.data.ff$WEEKDAY <- cut.ff(flights.data.ff$DAY_OF_WEEK, breaks = 7,
labels = c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday", "Sunday"))
flights.data.ff$WEEKDAY <- cut.ff(flights.data.ff$DAY_OF_WEEK,
                                  breaks = 7)
levels(flights.data.ff$WEEKDAY) <- c('Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday', 'Saturday', 'Sunday')
```

- The preceding code only marginally engages the machine's resources. Even when run on a large dataset there is as little as 357 MB of RAM usage.

- A similar code applied on the standard data.frame object may use up to 4.4 GB of memory.

- What makes the ff and ffbase packages even better tools, is their ability to perform quite complex data aggregations.

- The ffdfdply() function allows us to carry out a split-apply-combine type of operation on an ffdf object.

- During the apply part of the process, you may specify any function (FUN parameter) to use in order to aggregate the data and store it as a separate ffdf object.
- In our flights example we will calculate a mean departure delay for each city of origin by calling summaryBy() from the doBy package in the FUN argument:

```
library(doBy)
DepDelayByOrigCity <- ffdfdply(flights.data.ff,
                               split = flights.data.ff$ORIGIN_CITY_NAME,
                               FUN=function(x) {
                                 summaryBy(DEP_DELAY~ORIGIN_CITY_NAME,
                                           data=x, FUN=mean,na.rm=TRUE)}
                               )

## 2022-11-01 09:39:14, calculating split sizes

## 2022-11-01 09:39:14, building up split locations

## 2022-11-01 09:39:14, working on split 1/8, extracting data in RAM of 2
split elements, totalling, 0.01483 GB, while max specified data specified
using BATCHBYTES is 0.01562 GB

## 2022-11-01 09:39:15, ... applying FUN to selected data

## 2022-11-01 09:39:15, ... appending result to the output ffdf

## 2022-11-01 09:39:15, working on split 2/8, extracting data in RAM of 3
split elements, totalling, 0.01253 GB, while max specified data specified
using BATCHBYTES is 0.01562 GB

## 2022-11-01 09:39:15, ... applying FUN to selected data

## 2022-11-01 09:39:15, ... appending result to the output ffdf

## 2022-11-01 09:39:15, working on split 3/8, extracting data in RAM of 4
split elements, totalling, 0.01346 GB, while max specified data specified
using BATCHBYTES is 0.01562 GB

## 2022-11-01 09:39:15, ... applying FUN to selected data

## 2022-11-01 09:39:15, ... appending result to the output ffdf

## 2022-11-01 09:39:15, working on split 4/8, extracting data in RAM of 6
split elements, totalling, 0.01396 GB, while max specified data specified
using BATCHBYTES is 0.01562 GB

## 2022-11-01 09:39:16, ... applying FUN to selected data

## 2022-11-01 09:39:16, ... appending result to the output ffdf

## 2022-11-01 09:39:16, working on split 5/8, extracting data in RAM of 9
split elements, totalling, 0.0151 GB, while max specified data specified
using BATCHBYTES is 0.01562 GB
```

```
## 2022-11-01 09:39:16, ... applying FUN to selected data

## 2022-11-01 09:39:16, ... appending result to the output ffdf

## 2022-11-01 09:39:16, working on split 6/8, extracting data in RAM of 19
split elements, totalling, 0.01526 GB, while max specified data specified
using BATCHBYTES is 0.01562 GB

## 2022-11-01 09:39:17, ... applying FUN to selected data

## 2022-11-01 09:39:17, ... appending result to the output ffdf

## 2022-11-01 09:39:17, working on split 7/8, extracting data in RAM of 79
split elements, totalling, 0.01561 GB, while max specified data specified
using BATCHBYTES is 0.01562 GB

## 2022-11-01 09:39:17, ... applying FUN to selected data

## 2022-11-01 09:39:17, ... appending result to the output ffdf

## 2022-11-01 09:39:17, working on split 8/8, extracting data in RAM of 183
split elements, totalling, 0.00555 GB, while max specified data specified
using BATCHBYTES is 0.01562 GB

## 2022-11-01 09:39:17, ... applying FUN to selected data

## 2022-11-01 09:39:17, ... appending result to the output ffdf
```

- The output provides a verbose explanation of how the splits are created, and presents us with informative details on the progress of the function execution.
- Depending on the size of your raw data, the output will contain more, or fewer, splits and will take a longer, or shorter, time to run.
- Performed on the large dataset of 2 GB, ffdfdply() took 181 seconds to finalize the aggregation.
- During this time the memory usage fluctuated between 250 MB and 300 MB, and it reached 459 MB only for a very short time.
- Mean departure delay for each city of origin can be seen below:

```
DepDelayByOrigCity

## ffdf (all open) dim=c(305,2), dimorder=c(1,2) row.names=NULL
## ffdf virtual mapping
##                       PhysicalName VirtualVmode PhysicalVmode  AsIs
## ORIGIN_CITY_NAME ORIGIN_CITY_NAME      integer       integer FALSE
## DEP_DELAY.mean     DEP_DELAY.mean       double        double FALSE
##                  VirtualIsMatrix PhysicalIsMatrix PhysicalElementNo
## ORIGIN_CITY_NAME           FALSE            FALSE                 1
## DEP_DELAY.mean             FALSE            FALSE                 2
##                  PhysicalFirstCol PhysicalLastCol PhysicalIsOpen
## ORIGIN_CITY_NAME                1               1           TRUE
## DEP_DELAY.mean                  1               1           TRUE
## ffdf data
```

```
##          ORIGIN_CITY_NAME        DEP_DELAY.mean
## 1   Atlanta, GA             4.5206389
## 2   Chicago, IL             6.3520662
## 3   Dallas/Fort Worth, TX  5.5119289
## 4   Denver, CO              5.1485584
## 5   Los Angeles, CA         5.6188276
## 6   Houston, TX             6.2083877
## 7   New York, NY            6.6727152
## 8   Phoenix, AZ             4.1579772
## :                       :                      :
## 298 Wrangell, AK            3.2941176
## 299 Yakutat, AK            -0.2459016
## 300 Hyannis, MA            -4.8750000
## 301 Martha's Vineyard, MA 10.5526316
## 302 Nantucket, MA         13.0095238
## 303 Ponce, PR              -1.6470588
## 304 Worcester, MA           2.8050847
## 305 Binghamton, NY          4.4516129
```

- summaryBy() was much faster than its implementation on an ffdf with ffdfdply(), as it took only 5.6 seconds to complete, but the RAM consumption momentarily sky-rocketed to reach 4.85 GB.
- As our raw data are split between several binary flat files through the ff package, many operations performed on an ffdf object will obviously be much slower than those run directly in RAM on a standard data.frame.
- The ff-approach through the ffdfdply() function requires that each very small partition of the data is initially extracted to RAM, where it is processed with a specific function (FUN) applied on the selected data.
- The result of this computation is then finally appended to the output ffdf object, in our case it was the DepDelayByOrigCity object.
- It doesn't surprise us that all these tasks may take a relatively long time to complete, but the question to answer here is: Which strategy of data aggregation are you more likely to choose?
- If you are dealing with out-of-memory data, are you able (or allowed) to compromise on the speed of processing?
- The output object of the preceding aggregation is another ffdf structure. You can convert it to a standard data.frame through the previously introduced as.data.frame.ffdf():

```
# plot1.df <- as.data.frame.ffdf(DepDelayByOrigCity)
plot1.df <- as.data.frame(DepDelayByOrigCity)
str(plot1.df)

## 'data.frame':    305 obs. of  2 variables:
##  $ ORIGIN_CITY_NAME: Factor w/ 305 levels "Abilene, TX",..: 13 41 53 56
121 93 147 162 181 26 ...
##  $ DEP_DELAY.mean  : num  4.52 6.35 5.51 5.15 5.62 ...
```

- Now the data.frame is small enough to be easily used with all functions available in core R or third-party packages.
- For example you may want to sort the cities from which the flights departed based on the average departure delay in the descending order:

```
plot1.df <- orderBy(~-DEP_DELAY.mean, data=plot1.df)
plot1.df

##                      ORIGIN_CITY_NAME DEP_DELAY.mean
## 198                    Pago Pago, TT    49.11764706
## 286                 Adak Island, AK    21.23529412
## 289                Christiansted, VI    20.46875000
## 268         North Bend/Coos Bay, OR    15.68055556
## 271                 Plattsburgh, NY    15.63333333
## 302                   Nantucket, MA    13.00952381
## 209       Scranton/Wilkes-Barre, PA    12.38565022
## 172     Jacksonville/Camp Lejeune, NC    11.03791469
## 301            Martha's Vineyard, MA    10.55263158
## 242                    Escanaba, MI    10.22115385
## 285                       Islip, NY    10.02513966
## 212               St. Augustine, FL     9.84615385
## 229               Arcata/Eureka, CA     9.58333333
## 294                    Kotzebue, AK     9.10833333
## 16                    Baltimore, MD     8.82871344
## 227                   Aguadilla, PR     8.50785340
## 28               Fort Lauderdale, FL     8.41989349
## 126    Allentown/Bethlehem/Easton, PA     8.34716157
## 19                        Miami, FL     8.28182290
## 116                White Plains, NY     7.98453608
## 140                   Brunswick, GA     7.86060606
## 18                       Dallas, TX     7.74627508
## 273                     Redding, CA     7.72131148
## 119                      Fresno, CA     7.56211656
## 143            Charleston/Dunbar, WV     7.54587156
## 85                      Lubbock, TX     7.46772229
## 258                      Juneau, AK     7.39942939
## 20                       Newark, NJ     7.25524783
## 50                        Boise, ID     7.11101124
## 280                  St. George, UT     7.08556150
## 218                     Trenton, NJ     7.06077348
## 108                  South Bend, IN     6.99513382
## 124                      Albany, GA     6.97575758
## 215                   Texarkana, AR     6.78571429
## 7                      New York, NY     6.67271518
## 197     Newport News/Williamsburg, VA     6.63963964
## 207                    Santa Fe, NM     6.37967914
## 2                       Chicago, IL     6.35206616
## 296                        Nome, AK     6.31623932
## 6                       Houston, TX     6.20838768
## 115    West Palm Beach/Palm Beach, FL     6.20499543
```

```
## 87                   Manchester, NH     6.13039797
## 22                 Philadelphia, PA     6.11639861
## 48                  Baton Rouge, LA     6.11191626
## 58                     Columbia, SC     6.09196740
## 265                    Monterey, CA     6.07889126
## 133                      Bangor, ME     6.05797101
## 154                      Duluth, MN     6.03661327
## 33                    Nashville, TN     5.94378155
## 244                   Flagstaff, AZ     5.90508475
## 9                 San Francisco, CA     5.84524070
## 104                    San Juan, PR     5.81795372
## 120                  Long Beach, CA     5.79614148
## 233                    Brainerd, MN     5.79047619
## 95                      Ontario, CA     5.78924032
## 21                      Orlando, FL     5.78474020
## 99                     Portland, ME     5.75718850
## 54                   Charleston, SC     5.72395604
## 208          Sarasota/Bradenton, FL     5.70175439
## 5                   Los Angeles, CA     5.61882756
## 3              Dallas/Fort Worth, TX     5.51192889
## 276             San Luis Obispo, CA     5.49800797
## 240                  Eau Claire, WI     5.49586777
## 17                    Charlotte, NC     5.49417387
## 12                    Las Vegas, NV     5.47688752
## 141                  Burlington, VT     5.43001443
## 178                   La Crosse, WI     5.39677419
## 15                   Washington, DC     5.34997657
## 129                   Asheville, NC     5.32258065
## 105                    Savannah, GA     5.31766490
## 11                      Detroit, MI     5.29835946
## 145             Charlottesville, VA     5.28043478
## 91                 Myrtle Beach, SC     5.25302419
## 102                    Richmond, VA     5.19193193
## 25                       Austin, TX     5.18574035
## 150              Corpus Christi, TX     5.16109422
## 4                        Denver, CO     5.14855838
## 35                      Oakland, CA     5.10058944
## 10                       Boston, MA     5.02060398
## 205    Saginaw/Bay City/Midland, MI     4.82470120
## 161                 Gainesville, FL     4.71161826
## 43                        Tampa, FL     4.68412538
## 92                      Norfolk, VA     4.67754643
## 122               Santa Barbara, CA     4.66211293
## 101                        Reno, NV     4.58347902
## 203                   Rochester, MN     4.57983193
## 147                    Columbia, MO     4.57872340
## 239                   Dickinson, ND     4.57500000
## 220                    Valdosta, GA     4.57058824
## 1                       Atlanta, GA     4.52063890
## 84                   Louisville, KY     4.51663642
```

```
## 57                  Colorado Springs, CO     4.46822742
## 37                    Raleigh/Durham, NC     4.45878449
## 305                       Binghamton, NY     4.45161290
## 103                        Rochester, NY     4.44100719
## 62                        Evansville, IN     4.42782835
## 248                             Guam, TT     4.38983051
## 60                        Des Moines, IA     4.37963636
## 121                     Palm Springs, CA     4.36883117
## 76                      Jacksonville, FL     4.36453202
## 55                       Chattanooga, TN     4.33707865
## 253                          Hibbing, MN     4.32121212
## 49                        Birmingham, AL     4.26001781
## 125                       Alexandria, LA     4.21507353
## 38                        Sacramento, CA     4.21225752
## 27                          Columbus, OH     4.18218336
## 8                            Phoenix, AZ     4.15797719
## 14                           Seattle, WA     4.15354387
## 40                          San Jose, CA     4.08739837
## 155                          Durango, CO     4.06179775
## 187                        Melbourne, FL     4.01345291
## 66                        Fort Myers, FL     3.92955053
## 46                       Albuquerque, NM     3.90998317
## 26                         Cleveland, OH     3.87567311
## 90                            Mobile, AL     3.85388128
## 189       Mission/McAllen/Edinburg, TX     3.84434968
## 64                      Fayetteville, AR     3.83560896
## 24                         San Diego, CA     3.81491086
## 107                      Sioux Falls, SD     3.79516686
## 201                       Rapid City, SD     3.75751503
## 193                       Montgomery, AL     3.75453048
## 13                       Minneapolis, MN     3.71333269
## 263                          Medford, OR     3.70083682
## 94                             Omaha, NE     3.69466437
## 114                       Valparaiso, FL     3.68948655
## 290                          Cordova, AK     3.68852459
## 106                       Shreveport, LA     3.67929760
## 148                         Columbus, GA     3.63761468
## 59                            Dayton, OH     3.62152778
## 61                           El Paso, TX     3.60601650
## 83                       Little Rock, AR     3.59036145
## 204                          Roswell, NM     3.54494382
## 110                      Springfield, MO     3.53446553
## 98                        Pittsburgh, PA     3.53214286
## 42                         St. Louis, MO     3.51101113
## 56                        Cincinnati, OH     3.49048913
## 138 Bristol/Johnson City/Kingsport, TN     3.47500000
## 210                       Sioux City, IA     3.46902655
## 34                       New Orleans, LA     3.43898574
## 70             Greensboro/High Point, NC     3.42995951
## 30                      Indianapolis, IN     3.41004647
```

```
## 41                     Santa Ana, CA     3.39916209
## 152                       Dothan, AL     3.36725664
## 100                   Providence, RI     3.36482694
## 111                     Syracuse, NY     3.35309278
## 164               Grand Junction, CO     3.34270650
## 52                       Buffalo, NY     3.30823910
## 298                     Wrangell, AK     3.29411765
## 159                 Fayetteville, NC     3.26760563
## 297                   Petersburg, AK     3.15833333
## 31                   Kansas City, MO     3.15339020
## 243                       Eugene, OR     3.11981567
## 63                         Fargo, ND     3.11007269
## 77                       Kahului, HI     3.09381181
## 211                  Springfield, IL     3.07142857
## 186                    Marquette, MI     3.05769231
## 72                      Hartford, CT     3.05689813
## 45                        Albany, NY     3.05333333
## 32                     Milwaukee, WI     3.05204101
## 109                      Spokane, WA     3.02419843
## 217                Traverse City, MI     3.02314815
## 71                         Greer, SC     2.96982397
## 156               Elmira/Corning, NY     2.93368700
## 261                      Latrobe, PA     2.91845494
## 123                      Abilene, TX     2.91731266
## 74                    Huntsville, AL     2.87467363
## 88                       Memphis, TN     2.85478200
## 134         Beaumont/Port Arthur, TX     2.85119048
## 206                   San Angelo, TX     2.84668990
## 78                     Knoxville, TN     2.83373301
## 304                    Worcester, MA     2.80508475
## 80                     Lafayette, LA     2.76588235
## 249                     Gunnison, CO     2.73913043
## 86                       Madison, WI     2.69684336
## 117                      Wichita, KS     2.69644154
## 39                   San Antonio, TX     2.67620363
## 144            Charlotte Amalie, VI     2.67220903
## 136              Bismarck/Mandan, ND     2.66771160
## 270                     Pellston, MI     2.61157025
## 184                     Longview, TX     2.58035714
## 112                       Tucson, AZ     2.56262667
## 44                         Akron, OH     2.53991597
## 89                Midland/Odessa, TX     2.48135874
## 93                 Oklahoma City, OK     2.45466035
## 260                      Laramie, WY     2.43269231
## 97                        Peoria, IL     2.42661035
## 200     Pasco/Kennewick/Richland, WA     2.41019417
## 128                     Appleton, WI     2.40280561
## 166         Harlingen/San Benito, TX     2.35283364
## 295        Newburgh/Poughkeepsie, NY     2.31404959
## 67                    Fort Wayne, IN     2.27155600
```

```
## 113                           Tulsa, OK     2.17970240
## 69                        Green Bay, WI     2.16983122
## 53          Cedar Rapids/Iowa City, IA     2.15292949
## 29                         Honolulu, HI     2.13925586
## 118                         Burbank, CA     2.12434326
## 47                        Anchorage, AK     2.10003691
## 278             Sault Ste. Marie, MI     2.08695652
## 231                         Bemidji, MN     2.00000000
## 127                        Amarillo, TX     1.94763514
## 165                  Gulfport/Biloxi, MS     1.90635452
## 68                     Grand Rapids, MI     1.89566613
## 238                     Devils Lake, ND     1.89361702
## 279                           Sitka, AK     1.86341463
## 36                         Portland, OR     1.86096613
## 96                        Pensacola, FL     1.83246618
## 225                      Wilmington, NC     1.80000000
## 267                  Niagara Falls, NY     1.77272727
## 131                  Atlantic City, NJ     1.76459144
## 226                        Aberdeen, SD     1.72307692
## 82                            Lihue, HI     1.69021424
## 216                          Toledo, OH     1.55813953
## 277                     Santa Maria, CA     1.55371901
## 214                     Tallahassee, FL     1.51119403
## 158                       Fairbanks, AK     1.47933884
## 132                         Augusta, GA     1.45995423
## 266                        Muskegon, MI     1.42975207
## 130                           Aspen, CO     1.39884393
## 223                  Wichita Falls, TX     1.38528139
## 196    New Bern/Morehead/Beaufort, NC     1.36448598
## 191                          Moline, IL     1.33846154
## 281      Sun Valley/Hailey/Ketchum, ID     1.33333333
## 176                        Key West, FL     1.29411765
## 259                       Ketchikan, AK     1.25964010
## 177                         Killeen, TX     1.16028369
## 23                   Salt Lake City, UT     1.15130313
## 192                          Monroe, LA     1.14889706
## 81                        Lexington, KY     1.13969938
## 255          International Falls, MN     1.12500000
## 167                      Harrisburg, PA     0.97701149
## 224                       Williston, ND     0.90428212
## 245                       Gillette, WY     0.89714286
## 171                         Jackson, WY     0.87983707
## 182             Lawton/Fort Sill, OK     0.87029289
## 79                             Kona, HI     0.85133690
## 252                          Helena, MT     0.84837545
## 160                      Fort Smith, AR     0.81213873
## 188                        Meridian, MS     0.78181818
## 199                     Panama City, FL     0.49460916
## 183                         Lincoln, NE     0.44915254
## 254                     Idaho Falls, ID     0.44029851
```

```
## 75              Jackson/Vicksburg, MS     0.32468553
## 222                      Waterloo, IA     0.32407407
## 292                         Eagle, CO     0.29411765
## 181                        Laredo, TX     0.12468193
## 202                       Roanoke, VA     0.12018141
## 157                          Erie, PA     0.07894737
## 274                   Rhinelander, WI     0.05232558
## 139                   Brownsville, TX     0.01827676
## 142              Champaign/Urbana, IL    -0.02255639
## 246                   Grand Forks, ND    -0.16666667
## 149                      Columbus, MS    -0.16763006
## 151                 Daytona Beach, FL    -0.17826087
## 65                          Flint, MI    -0.24559194
## 299                       Yakutat, AK    -0.24590164
## 153                       Dubuque, IA    -0.47953216
## 293                        Kodiak, AK    -0.74666667
## 237                          Cody, WY    -0.83870968
## 219                         Tyler, TX    -0.85812357
## 180                       Lansing, MI    -0.95417790
## 73                           Hilo, HI    -1.04417671
## 230                   Bakersfield, CA    -1.26912929
## 174                     Kalamazoo, MI    -1.28057554
## 51                        Bozeman, MT    -1.30411687
## 228                        Alpena, MI    -1.37373737
## 135                      Billings, MT    -1.39130435
## 137            Bloomington/Normal, IL    -1.43685300
## 291                     Deadhorse, AK    -1.46153846
## 190                      Missoula, MT    -1.47000000
## 168            Hattiesburg/Laurel, MS    -1.48076923
## 213                 State College, PA    -1.62295082
## 175                     Kalispell, MT    -1.64225352
## 303                         Ponce, PR    -1.64705882
## 269                       Paducah, KY    -1.68907563
## 284                          Yuma, AZ    -1.70392749
## 287                        Barrow, AK    -1.82580645
## 169                        Hayden, CO    -1.82758621
## 288                        Bethel, AK    -1.83030303
## 256         Iron Mountain/Kingsfd, MI    -1.84955752
## 250              Hancock/Houghton, MI    -1.88135593
## 146         College Station/Bryan, TX    -2.12765957
## 247                   Great Falls, MT    -2.21372032
## 185           Manhattan/Ft. Riley, KS    -2.24915825
## 236                    Cedar City, UT    -2.30476190
## 282                    Twin Falls, ID    -2.37058824
## 195                       Mosinee, WI    -2.38790036
## 283              West Yellowstone, MT    -2.48000000
## 275                  Rock Springs, WY    -2.66666667
## 221                          Waco, TX    -2.73404255
## 170                         Hobbs, NM    -2.76767677
## 264                         Minot, ND    -2.84761905
```

```
## 232                 Bend/Redmond, OR    -2.89690722
## 162                  Garden City, KS    -2.91596639
## 235                       Casper, WY    -3.38709677
## 179                 Lake Charles, LA    -3.57670455
## 163                 Grand Island, NE    -3.73214286
## 173                       Joplin, MO    -4.32478632
## 251                         Hays, KS    -4.52884615
## 300                      Hyannis, MA    -4.87500000
## 262                     Lewiston, ID    -5.14423077
## 194               Montrose/Delta, CO    -5.60869565
## 241                         Elko, NV    -5.77192982
## 257                    Jamestown, ND    -6.30069930
## 234                        Butte, MT    -6.53719008
## 272                    Pocatello, ID    -6.60150376
```

- Such prepared data can now be very conveniently used for visualizations or further data crunching with standard R techniques.
- The ff and ffbase packages also support subsetting of ffdf objects through the subset.ffdf() method. It takes similar arguments to the generic subset() function:

```
subs1.ff <- subset.ffdf(flights.data.ff, CANCELLED == 1,
                        select = c(FL_DATE, AIRLINE_ID,
                                   ORIGIN_CITY_NAME,
                                   ORIGIN_STATE_NM,
                                   DEST_CITY_NAME,
                                   DEST_STATE_NM,
                                   CANCELLATION_CODE))
```

- In the preceding code, we have specified that we want to subset all records with cancelled flights only, and that in our new subs1.ff ffdf object we wish to include only selected (select argument) variables from the original flights.data.ff object.
- If applied on the ffdf object, which maps to the large 2 GB-heavy data, the subset.ffdf() function consumes only minimal amounts of memory.
- As with preceding examples, the generic subset() executed on the data.frame object was more RAM-hungry; it used over 0.7 GB of available memory.
- The newly created ffdf object may be exported to a flat data file; in fact it will be saved as seven separate ff files, one for each variable (that is column) in the subs1.ff object:

```
save.ffdf(subs1.ff, overwrite=TRUE)
```

By default the save.ffdf() function saves the flat files to a new folder called ffdb, which will be created automatically in your working directory. You can, of course, change the name of the destination folder by altering the dir argument (by default set to "./ffdb"). The resulting flat files are stored with filenames in the format: $.ff. The exported ff files can be loaded back to the R session using the load.ffdf() command. If you want to see how it works, remove the subs1.ff object from your workspace and type the correct path to the destination folder with ff files in the load.ffdf() function:

```
rm(subs1.ff)
# load.ffdf("~/Desktop/data/ffdb")
load.ffdf("D:/AMU Computer Science/Courses/Big Data Analytics/Big Data
Analytics Using R/Ch3/ffdb")
```

A new old subs1.ff object should appear in the environment with all its default metadata and ffdf features. But you may also want to export it into CSV or TXT files. This functionality is also possible through the ff package. The write.csv.ffdf() or write.table.ffdf() expressions will accomplish this task for you:

```
write.csv.ffdf(subs1.ff, "subset1.csv", VERBOSE = TRUE)

## write.table.ffdf 1..

## opening ff D:/AMU Computer Science/Courses/Big Data Analytics/Big Data
Analytics Using R/Ch3/ffdb/subs1.ff$FL_DATE.ff

## opening ff D:/AMU Computer Science/Courses/Big Data Analytics/Big Data
Analytics Using R/Ch3/ffdb/subs1.ff$AIRLINE_ID.ff

## opening ff D:/AMU Computer Science/Courses/Big Data Analytics/Big Data
Analytics Using R/Ch3/ffdb/subs1.ff$ORIGIN_CITY_NAME.ff

## opening ff D:/AMU Computer Science/Courses/Big Data Analytics/Big Data
Analytics Using R/Ch3/ffdb/subs1.ff$ORIGIN_STATE_NM.ff

## opening ff D:/AMU Computer Science/Courses/Big Data Analytics/Big Data
Analytics Using R/Ch3/ffdb/subs1.ff$DEST_CITY_NAME.ff

## opening ff D:/AMU Computer Science/Courses/Big Data Analytics/Big Data
Analytics Using R/Ch3/ffdb/subs1.ff$DEST_STATE_NM.ff

## opening ff D:/AMU Computer Science/Courses/Big Data Analytics/Big Data
Analytics Using R/Ch3/ffdb/subs1.ff$CANCELLATION_CODE.ff

## 4529 (4529, 100%)  ffdf-read=0.13sec csv-write=0.04sec
##  ffdf-read=0.13sec  csv-write=0.04sec  TOTAL=0.17sec
```

So far in this section, we have presented numerous applications of several functions from the ff and ffbase packages, which you might find very useful when processing datasets that are larger than the available RAM resources.

By a rule of thumb, and following the recommendations given by the authors of ff and ffbase, the packages will generally benefit workflows executed on data up to 10 times bigger than the memory capacity.

This threshold seems very reasonable. Based on our extensive testing, all data manipulations, transformations, and aggregations performed on ff or ffdf objects and discussed in this section used a maximum of only 425.3 MB of RAM the final value of the memory resources assigned to the R session processes at the end of the R script run on flat files (the ffapproach).

On the other hand, the same data processing activities, but utilizing a more generic approach, and executed on a data.frame object which was loaded to R with the read.table() function, and as result stored in RAM, consumed as much as 3.7 GB of memory with the moment of execution of the last line of the R code. This almost ten-fold difference makes sense. The value of RAM usage in the data.frame approach will have probably been even greater if we hadn't used the garbage collection call (gc()) very frequently.

Unfortunately the ff and ffbase packages are not without their own issues and limitations. In the preceding examples, we've shown that in the ff approach, data input is not as fast as when uploaded to the R session, even using generic read.table() or read.csv().

Remember that, in order to create an ff or ffdf object, containing mapping, to the raw data, the original dataset needs to be loaded in chunks, and their content copied to binary flat files.

The processes of chunking, mapping, and writing to ff files may take considerably more time than simply loading to a data.frame using core R functions.

Later in this chapter you will learn much faster methods of getting the data into R, using, for example, the fread() function from the data.table package. Second, we've already explained that some functions may be slower in execution on ffdf objects than their counterparts on standard data frames.

Again, data first have to be retrieved from chunks and only then may a function or an operation be applied to this particular, small part of the data.

The output of the function is consequently appended, one-by-one, to a new ff or ffdf object. These several processes extend the time spent on the execution of a function. Third, the filenames of chunks are quite confusing, and definitely not user-friendly. Assuming we will like to find the name of the ff file where the AIRLINE_ID variable is stored we can obtain it as follows:

```
basename(filename(flights.data.ff$AIRLINE_ID))

## [1] "ffdf3ac43624ba3.ff"
```

This file naming convention makes working with flat files very difficult, especially when moving data around. Despite these flaws, the ff and ffbase packages offer an interesting alternative to transformations and aggregation of out-of-memory data in R. The ffbase library is currently being re-developed by Edwin de Jonge to include grammar and internal functions from a very popular data manipulation package dplyr. The re-branded version of ffbase, now called ffbase2, contains a set of transformation functions such as summarize(), group_by(), filter(), or arrange() that are known from the dplyr package, but will also be applicable to ffdf objects. As the ffbase2 library is still under development, its current version can only be installed from Edwin de Jonge's GitHub repository at https://github.com/edwindj/ffbase2 and its functionality is still very limited, but you may find some of its methods operational. In order to install and load this package you must have the recent version of the devtools library installed and ready-to-use in your RStudio:

```
install.packages("devtools")
devtools::install_github("edwindj/ffbase2")
library(ffbase2)
```

The use of ff and ffbase packages is not just limited to data transformations and aggregations. In the following section, you will learn how to perform more complex Big Data modeling and analytics operations on ffdf objects.