

*Deborah Nolan, University of California, Berkeley*  
*Duncan Temple Lang, University of California, Davis*

---

# ***Data Science in R: A Case Studies Approach to Computational Reasoning and Problem Solving***

Copyrighted Material – Taylor & Francis

---

# Strategies for Analyzing a 12-Gigabyte Data Set: Airline Flight Delays

---

**Michael Kane**  
*Yale University*

## CONTENTS

5.1	Introduction .....	217
5.1.1	Computational Topics .....	218
5.2	Acquiring the Airline Data Set .....	218
5.3	Computing with Massive Data: Getting Flight Delay Counts ...	219
5.3.1	The <i>R</i> Programming Environment .....	219
5.3.2	The <i>UNIX</i> Shell .....	221
5.3.3	An <i>SQL</i> Database with <i>R</i> .....	223
5.3.4	The <i>bigmemory</i> Package with <i>R</i> .....	226
5.4	Explorations Using Parallel Computing: The Distribution of Flight Delays .....	229
5.4.1	Writing a Parallelizable Loop with <i>foreach</i> .....	230
5.4.2	Using the Split-Apply-Combine Approach for Better Performance .....	231
5.4.3	Using Split-Apply-Combine to Find the Best Time to Fly .....	232
5.5	From Exploration to Model: Do Older Planes Suffer Greater Delays? .....	236
	Bibliography .....	237

---

## 5.1 Introduction

Anyone who has dealt with flight delays at the airport understands the associated inconvenience and aggravation. And while we might hope that delays are rare, they are probably more common than you think. Since October 1987, there have been over 50 million flights in the United States that failed to depart at their scheduled times. Around 200,000 of those flights were at least two hours late; some were much later. From these two simple facts we can surmise that delays are not isolated, rare events; they are routine. Since 1987 the number of flights per year has steadily increased and as this trend continues we expect to see more inconvenience, more aggravation, and more time lost.

But why do flight delays occur? Is it simply because there are more flights now than in previous years? Are delays caused by bad weather? Is enough time being scheduled for flights? Do single flight delays cause multiple flight delays later in the day? A better

understanding of the cause of flight delays could allow the airline industry to intelligently react to issues such as bad weather, providing more flights with fewer delays.

This chapter presents a means for understanding flight delays by analyzing data. In 2009, the American Statistical Association (ASA) Section on Statistical Computing and Statistical Graphics released the “Airline on-time performance” data set [17] for their biannual data exposition. The data set was compiled and organized by Hadley Wickham [16] from the official releases from the US government’s Bureau of Transportation Research and Innovative Technology Administration (RITA) Web site ([http://www.transtats.bts.gov/DL\\_SelectFields.asp?Table\\_ID=236](http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236)). The data include commercial flight 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, each with 29 variables related to flight time, delay time, departure airport, arrival airport, and so on. In total, the uncompressed data set is about 12 gigabytes (GB) in size. The data set is so large that it is difficult to analyze using the standard tools and techniques we have come to rely upon. As a result, new approaches need to be utilized to understand the structure of these data. This chapter presents some of these new approaches along with an initial exploration of airline flight delays.

These new computational approaches are illustrated by taking the reader through the process of acquiring, exploring, and modeling the airline data set. The first section describes how to acquire the data. The second section demonstrates the use of 3 different computing environments to manage and access large data sets: *R* [7], the *UNIX* shell, and *SQL* databases. The third section offers easy-to-use parallel computing techniques for basic data exploration. No prior experience with parallel computing is required for this section. The fourth section provides an approach to answering the question, “Do older planes suffer greater delays?” This section synthesizes earlier material and demonstrates how to construct linear models with a potentially massive data set.

### 5.1.1 Computational Topics

- Big Data strategies.
- Shell commands and pipes.
- Relational databases.
- Parallel computing.
- External data representation.
- The split-apply-combine approach.

Question: What if the reader runs into software questions while trying to run the sample code?

Answer: Throughout the chapter an attempt has been made to anticipate some of the implementation issues that will arise as the reader works through the examples. These issues are presented as questions along with their solutions in boxes like this one.

---

## 5.2 Acquiring the Airline Data Set

The airline on-time performance data can be found at <http://stat-computing.org/dataexpo/2009/the-data.html>. The Web page contains the compressed airline delay data in comma-separated values (CSV) files, organized by year. The page also provides descriptions of each of the 29 variables associated with each flight. The following sections will assume that you have downloaded the data files and decompressed each of them, e.g., using the `bunzip2` shell command or potentially a point-and-click graphical interface.

---

## 5.3 Computing with Massive Data: Getting Flight Delay Counts

This section introduces methods for managing and accessing data sets that require more than a computer's available RAM (Random Access Memory). At the time this chapter was written, a 12Gb data set, such as the airline data, presents a computational challenge to most statisticians because of its sheer size. Admittedly, this may not always be the case. A future reader may scoff at the idea of having difficulty managing 12 gigabytes of data. However, it is assumed that there will still be a data set which, by virtue of its size, frustrates even this well-equipped statistician.

To begin our exploration, let's consider two simple questions:

- How many flights are in the data set for 1987?
- How many Saturday flights appear in the entire data set, i.e., all years?

The next four subsections show distinct approaches to answering these questions. The first uses the *R* programming environment with its native data structures and capabilities. The second uses the *UNIX* shell, independent of *R* or other programming environments. The third uses *R* to connect to a database where *SQL* queries are constructed to answer the posed questions. The fourth and final approach utilizes some of *R*'s more advanced functionality to show how the `bigmemory` package can be used to explore the entire airline data set from within the *R* environment.

### 5.3.1 The *R* Programming Environment

The downloaded files provide data for more than 20 years of airline traffic. Each file holds information for one year and each year contains information for approximately 5 million flights. The aggregate data set is larger than the amount of RAM on most single computers. However, to compute the flight count for 1987, we do not need to load the entire data set for all years into *R*. We can simply load the 1987 data and compute the number of observations with

```
x <- read.csv("1987.csv")
nrow(x)
```

```
[1] 1311826
```

To compute the number of Saturday flights across multiple files, we need to determine how many values in the column labeled `DayOfWeek` correspond to Saturday in each of the

data files. The values in this column have values from 1 through 7, with 1 corresponding to Monday and 7 for Sunday. The value 6 indicates Saturday. At the same time, we need to be wary of the size of the data set. We can find the total number of Saturday flights by working on one file at a time, rather than trying to read all of the files at once. After a single file is read into a data frame, the number of Saturday flights will be calculated and saved as an intermediate result. This will be done for each file and, finally, these intermediate counts will be added together to get the total. This approach is sometimes called the *incremental* or *chunking* approach because a single, manageable “chunk” of data is processed and an incremental result is stored before aggregation, which yields the final result. Here, the individual files are natural chunks. In other cases, we divide a large collection of observations into smaller chunks by just reading/processing chunks of the observations sequentially. For this airline data, we can compute the total number of Saturday flights with

```
totalSat <- 0
for (year in 1987:1988) {
  x <- read.csv(paste(year, ".csv", sep=''))
  totalSat <- totalSat + sum(x$DayOfWeek == 6)
}
totalSat

[1] 15915382
```

We could improve the speed of this by specifying a vector of types for each column via the `colClasses` parameter. This helps `read.csv()` (and ultimately `read.table()`) so that it doesn't have to infer the type of each column and potentially have to reallocate memory if it guesses incorrectly from the first  $k$  observations. We could also provide the total number of observations via `nrows` if we already knew these, as we did for the 1987 data. Again, this helps R to allocate memory efficiently by doing it just once for each column/vector of values.

Instead of the `for` loop, we could also use

```
counts <- sapply(sprintf("%d.csv", 1987:1988),
  function(f)
    sum(read.csv(f)$DayOfWeek == 6))
sum(counts)
```

Each call to our function reads the appropriate CSV file and then discards it after computing the number of observations. This is the critical aspect of both approaches, i.e., to avoid having more than one year's data frame in memory at any time. The `sapply()` approach enables this; the `for` loop approach actually has two data frames in memory at times (when and why?). With the `sapply()` code, we also have the counts for each individual year and can examine these to see how they are distributed, e.g., does the number of flights in a year increase over the years? The `for` loop code is easily modified to create this vector also.

Although these questions were easy to answer, the approach of loading files as needed can be somewhat limiting. Also, the approach assumes that each data file can be stored as a data frame. When operations cannot be expressed incrementally/cumulatively, this approach becomes cumbersome. In the next sections, we will focus on approaches where the data are managed and accessed from a single source, rather than a set of files, allowing us to perform more sophisticated analyses.

Question: Can *R* objects be destroyed explicitly when we are done with them for better performance?

Answer: It is possible to force *R* to remove a variable and run the garbage collector, freeing all memory associated with that variable. For the vast majority of cases, little is gained by doing this. Even in cases where this helps, we will not see a dramatic increase in speed nor a dramatic decrease in memory usage.

A call to the garbage collector may help in special cases where the variables held in an *R* session have memory requirements greater than the size of a computer's available RAM. When programs require more RAM than what is available, variables may be "swapped" from RAM to the disk and retrieved when needed. In some cases, even often-used variables can be swapped to disk. This causes calculations to run slower because accessing the data from the disk takes much longer than accessing them from RAM. In a few cases, calling the garbage collector may free unused memory in RAM, allowing other variables to take their place and reducing the amount of time spent swapping. It may also reduce memory fragmentation, which can be important.

The function to remove a variable is `rm()` and the garbage collector is called with the `gc()` function. The following code shows how to use these two functions to ensure that only one data frame exists in RAM at any time when calculating the number of Saturday flights:

```
totalSat <- 0
for (year in 1987:1989) {
  x <- read.csv(paste(year, '.csv', sep=''))
  totalSat <- totalSat + sum(x$DayOfWeek == 6)
  rm(x)
  gc()
}
totalSat

[1] 15915382
```

Here we explicitly remove the data frame object and force the garbage collector to run. *R* does this implicitly.

### 5.3.2 The *UNIX* Shell

This section uses the *UNIX* shell to answer the previously posed questions. The shell is a powerful interactive and scripting programming environment with a reasonably simple language and a rich set of computational resources. Many common operations are provided as built-in utilities, analogous to functions in *R*. For example, finding the number of flights in 1987 can easily be accomplished with the `wc` utility. `wc` stands for "word count" which is not quite what we want but the `-l` option can be used to count lines rather than words. To find out how many flights in 1987 were recorded we can simply type:

```
wc -l 1987.csv
1311827 1987.csv
```

Again we see that there are 1,311,826 flights that were recorded for 1987. The count from the shell command indicates one more, but this is because it includes the first (header) line of the file that lists the names of each column. The utility executes more quickly than the solution from the previous section and it was expressed with less code. In this subsection we

Shell

will see that the shell is often a very good tool for file manipulation, simple subset selection, and simple summaries. Furthermore, we can invoke shell commands from *R* (and other languages) and read the resulting output back into *R*, giving us the best of both worlds.

For the rest of this section, we would like to work with a single CSV file, which contains all of the airline data. This file can be created by appending each of the airline files to a new file. However, when the files are being concatenated we need to make sure that the header information, which is given on the first line of each file, appears only once at the beginning of the new file.

The task of combining each of the CSV files into a single file can be accomplished with the following commands:

Shell

```
cp 1987.csv AirlineDataAll.csv
for year in {1988..2008}
do
  tail -n+2 $year.csv >> AirlineDataAll.csv
done
```

The resulting file begins with a header line containing the names of each column and then contains observations for each flight for every year.

This small script illustrates some of the features of the shell. We copied the first file (*1987.csv*) to our target file and then appended the contents of the other files to this new file. We can specify the years to iterate over using a list constructor `1988..2008`. This is analogous to `1988:2008` in *R*. Then, we can create an argument specifying a file name by appending `.csv` to the value of the shell variable `year`, i.e. `$year.csv`<sup>1</sup>. Next, we can extract the “tail end” of a file with the *UNIX* `tail` utility. The `-n+2` option specifies that `tail` will return all but the first line of the file. Finally, we can use the `>>` operator to redirect the output from the `tail` command so that it is appended to the *AirlineDataAll.csv* file.

The loop above is not our only option for aggregating information from a file. The `tail` command itself can take multiple files as input, outputting the specified lines for each of the files. For example, if we were interested in creating a single file, with no header information, for all files from the twentieth century we could simply use the command:

Shell

```
tail -n+2 19*.csv > All1900.csv
```

Like the last example, this one uses `tail` to return all but the first line of each of a set of files. Unlike the last example, we used `19*.csv` to specify all files that start with 19 and end with `.csv`. This example also made use of the `>` operator, which overwrites the existing contents of *All1900.csv* or creates the file if it does not already exist. This version does not include the header line at the top of the file. Depending on how we use this file, this may or may not be important, but it is something we need to know. Now that we have two different approaches for creating a single file, holding all of the airline data, let’s use the *UNIX* shell to find the number of Saturday flights.

Calculating the total number of Saturday flights is trickier than simply counting the number of lines in a file. The day of the week column is the fourth one in *AirlineDataAll.csv*. (Examine the first line of any of the original data files to verify this.) So, we would like to extract only that column of data values from the file and see how many times 6 appears as the extracted value (corresponding to Saturday). To do this we’ll need to introduce a few more *UNIX* shell utilities and concepts. First, we can use the `cut` utility, which extracts and outputs sections from each line of a file. We’ll use it to extract the fourth column of each line and discard all other columns. This will create a large collection of values ranging from 0 to 6, i.e., the day of the week as a number, each on its own line of output. We can then use

<sup>1</sup>The shell knows that the variable is named `year` since a shell variable cannot have a “.” in its name.

the `grep` command to match only those lines that contain the character 6. This produces a collection of lines that is a subset of that produced by the `cut` command. Finally, we can use `wc`, with which we are already familiar, to compute the total count for Saturday flights by counting the number of lines output by the `grep` command. We could implement this with 3 separate commands and output the results of each to intermediate files, e.g.,

```
cut -d , -f 4 AirlineDataAll.csv > tmp
grep 6 tmp > tmp1
wc -l tmp1
rm tmp tmp1
```

Shell

However, the shell explicitly supports directing the *output* from one command as *input* to another command without (explicitly) using intermediate files. This the purpose of the pipe operator `|`. We can write the entire computation to calculate the number of Saturday flights with the command:

```
cut -d , -f 4 AirlineDataAll.csv | grep 6 | wc -l
15915382
```

Shell

In this example the `cut` utility is passed 3 sets of parameters. The first set is `-d`, which specifies that the file consists of columns, each of which is delimited (`-d`) by a comma. The second set `-f 4` specifies that we want to extract the fourth column/field. The third specifies the name of the file whose contents we want to process. The output of the `cut` command is the fourth column of `AirlineDataAll.csv`. The lines from the `cut` command are then passed as lines of input for the `grep` command, which outputs all the lines that consist of the single value 6. All 15,915,382 of those rows are passed to `wc -l`, which counts the number and prints the result on the console.

We should note that we don't actually need to create this single file containing the data for all of the years in order to easily calculate the number of Saturday flights. We can have `cut` operate on all of the files with the command `cut -d , -f 4 *.csv` and then pipe this to `grep` and `cut` as before. However, in the next section we will need and use this single file containing all of the data.

The `UNIX` shell capabilities go far beyond the examples shown here. There are many other utilities included with the shell and users can even create their own utilities using almost any programming language. As a result, the `UNIX` shell is often a good choice for file manipulation and basic summaries. Moreover, we can use these shell commands from `R` via the `system()` and `system2()` functions.

### 5.3.3 An SQL Database with `R`

The `R` programming environment approach to answering the posed questions is simple for someone who is familiar with `R` and its functionality could easily be expanded to do more than simply tally the number of flights in 1987 or on Saturdays. However, the approach is somewhat limited in that it assumes each file is relatively small, i.e., the contents can be held in memory. The `UNIX` shell approach is also simple, if you are already familiar with its syntax and computational model. With only a single line we were able to answer each of the posed questions. Also, the results were returned more quickly than with `R`. However, the `UNIX` shell is not a familiar environment to many statisticians, it has a limited computational model, and it does not have built-in capabilities for performing statistical analyses. As a data exploration requires more sophisticated analyses, the corresponding `UNIX` scripts may become difficult to write. In practice, we often combine `R` and the shell to process data. However, there is an important alternative approach – databases.



A database provides a general solution for managing a large data set and extracting meaningful information. Where the previous approaches required that we manually open files and extract the data of interest each time we process the data, a database provides a means to ingest and structure the data once and reuse that structure each time we access the data. A database also provides the Structured Querying Language (*SQL*) that allows us to specify and efficiently extract the subset of data of interest with a powerful query. This approach creates a rich and general way of not only extracting potentially large and complex subsets of the data but also for computing basic summaries of these subsets. Just as we can call the shell from *R* or other languages, we can interact with a database from *R*, sending *SQL* commands to the database and accessing the output as *R* objects.

The examples in this section use SQLite [1], a lightweight (*SQL*) database engine. All interactions with the database are done in *R* with the RSQLite [2] package. However, these examples are not specific to SQLite or RSQLite. They will work with any *SQL* database engine and corresponding *R* database connector package.

Question: How do I import the airline delay data into a database?

Answer: We'll use SQLite as the database engine. You must have SQLite installed on the machine. The software can be downloaded from the SQLite Web page (<http://www.sqlite.org/>). You must also have the RSQLite package installed on your machine; it can be found on the CRAN Web site or installed directly with `install.packages("RSQLite")`. The following instructions are based on those given by Hadley Wickham on the 2009 Data Expo Web page.

- Inside the directory where the data (CSV files) reside, type the command:

```
sqlite3 AirlineDelay.sqlite3
```

- This will create the database and put you into a *SQL* console/read-eval-print-loop (REPL), with a prompt `sqlite>`. Next, create the table and its fields with the command

```
CREATE TABLE AirlineDelay (
  Year int,
  Month int,
  DayofMonth int,
  DayOfWeek int,
  DepTime int,
  CRSDepTime int,
  ArrTime int,
  CRSArrTime int,
  UniqueCarrier varchar(5),
  FlightNum int,
  TailNum varchar(8),
  ActualElapsedTime int,
  CRSElapsedTime int,
  AirTime int,
  ArrDelay int,
  DepDelay int,
  Origin varchar(3),
  Dest varchar(3),
  Distance int,
  TaxiIn int,
```

Shell

SQL

```
TaxiOut int,
Cancelled int,
CancellationCode varchar(1),
Diverted varchar(1),
CarrierDelay int,
WeatherDelay int,
NASDelay int,
SecurityDelay int,
LateAircraftDelay int );
```

- Now that the table has been created, import the data from the *AirlineDataAll.csv* file. This may take from 10 minutes to over an hour, depending on the speed of your machine.

```
.separator ,
.import AirlineDataAll.csv AirlineDelay
```

SQL

To extract all of the flights in 1987, start by opening an *R* session and connecting to the database:

```
library(RSQLite)
delay.con <- dbConnect("SQLite", dbname = "AirlineDelay.sqlite3")
```

The `delay.con` variable holds a connection to the database that we can use in subsequent commands.

Queries are expressed via *SQL* statements. These statements allow a user to describe the data of interest and perform operations (such as counting) on them. The `SELECT` statement is used to retrieve entries from a database and we can use it to retrieve all data of the flights from 1987:

```
delays87 <- dbGetQuery(delay.con,
                       "SELECT * FROM AirlineDelay WHERE Year=1987")
```

`dbGetQuery()` is an *R* function. It sends an *SQL* query to the database to be processed there and `dbGetQuery()` waits for the result. It does not examine the query as that is written in a different language (*SQL*). The query above returns all of the variables in a data frame for those flights whose value for the year variable is equal to 1987. Now, we can find the number of 1987 flights in a familiar way:

```
nrow(delays87)
[1] 1311826
```

Equivalently, the *SQL* engine can do the counting for us by utilizing the `COUNT()` aggregator function:

```
dbGetQuery(delay.con, "SELECT COUNT(*), Year FROM AirlineDelay
                       WHERE Year=1987")
[1] 1311826
```

It is reasonably clear from the examples above that *SQL* statements are useful as we perform more complex queries. What if, instead of getting the flight count for a single year, we want to get the flight count for each year in the database? We can do this using the `GROUP BY` clause in *SQL*. The following query groups all of the rows of the data set by year and counts the number of rows in each of these groups:

```
dbGetQuery(delay.con,
            "SELECT COUNT(*), Year FROM AirlineDelay GROUP BY Year")
```

	COUNT(*)	Year
1	1311826	1987
2	5202096	1988
3	5041200	1989
4	5270893	1990
5	5076925	1991
6	5092157	1992
7	5070501	1993
8	5180048	1994
9	5327435	1995
10	5351983	1996
11	5411843	1997
12	5384721	1998
13	5527884	1999
14	5683047	2000
15	5967780	2001
16	5271359	2002
17	6488540	2003
18	7129270	2004
19	7140596	2005
20	7141922	2006
21	7453215	2007
22	7009728	2008

To find the number of Saturday flights, we can use a query similar to the previous one, but rather than grouping by the *Year* variable, we subset/filter by day of week using the *WHERE* clause and perform the calculations in *SQL* with

```
dbGetQuery(delay.con,
            "SELECT COUNT(*), DayOfWeek FROM AirlineDelay
            WHERE DayOfWeek = 6")
```

	COUNT(*)	DayOfWeek
1	15915382	6

(Note the use of the single = operator in *SQL* for testing equality, different from == in *R*.)

A database provides a useful way of managing large data sets and it supports complex queries. However, it also comes with challenges. First, it requires the creation of the database and the importing of data. For the airline data this was not difficult, but it did involve an extra step. Second, the database approach also requires that the statistician is familiar with *SQL* to perform even simple operations. Finally, if an analysis is done in *R* and the data set is small, we would use a `data.frame` or `matrix` to manage it. When the data set gets big and we need a new mechanism to handle the larger volume, a database connection cannot simply be “swapped-in” in place of the familiar *R* data structures. Significant changes must be made in the code in order for the analysis to work with a database. Wouldn’t it be nice if there was an *R* data structure that behaved like a `matrix`, but at the same time, managed large data sets for you?

### 5.3.4 The `bigmemory` Package with `R`

The `R` package `bigmemory` [4] provides matrix-like functionality for data sets that could be much larger than a computer's available RAM. This approach to computing with data provides several advantages when compared with other approaches. First, `bigmemory` provides data structures that hold entire, possibly massive, sets of data. As a result, there is no need to manually load and unload data from files. Second, the data structures provided by `bigmemory` are accessed and manipulated in the same way as `R`'s `matrix` objects. Our experience with `R` has prepared us to work with `bigmemory` and there is no need to learn a new language, such as `SQL`, to access and manipulate data. Third, `bigmemory` works with many of `R`'s standard functions with little or no modification, minimizing the amount of time required to perform standard manipulations and analyses. Finally, `bigmemory` was designed from the ground up for use in parallel and distributed computing environments. Later on we will see how `bigmemory` can be used as a basis for implementing scalable analyses for data that may be much larger than even the airline data set.

The essential data structure provided by `bigmemory` is the `bigmatrix`. A `bigmatrix` maintains a binary data file on the disk called a *backing file* that holds all of the values in a data set. When values from a `bigmatrix` object are needed by `R`, a check is performed to see if they are already in RAM (cached). If they are, then the cached values are returned. If they are not cached, then they are retrieved from the backing file, cached, and then returned. These caching operations reduce the amount of time needed to access and manipulate the data across separate calls, and they are transparent to the statistician. A `bigmatrix` object is designed to be a convenient and intuitive tool for computing with massive data. When using these data structures the emphasis is on the exploration, not the underlying technology.

As mentioned before, `bigmatrix` looks like a standard `R matrix`. It has rows and columns, and subsets of the elements of a `bigmatrix` can be read and set using the standard subsetting operator (`[]`). Like `R`'s `matrix`, a `big.matrix` object requires that all elements are of the same type. However, this leads to a challenge with the airline data set since it has columns that are character type as well as numeric. Before we can read the airline data into a `bigmatrix`, some preprocessing must be done so that all the columns and their values are numeric. For columns with character data, this preprocessing step creates a mapping between a unique numeric value and the character value for each row, much like `R`'s `factor` data type. Preprocessing is left as an exercise for the reader or a preprocessing script is available from the author of this chapter. For the rest of the chapter, we will assume that the preprocessing step has been performed and that the preprocessed file has been named `airline.csv`.

A user can create a `bigmatrix` from a CSV file with the function `read.big.matrix()` that is similar to `R`'s `read.csv()` function, e.g.,

```
x <- read.big.matrix("airline.csv", header = TRUE,
                    backingfile = "airline.bin",
                    descriptorfile = "airline.desc",
                    type = "integer", extraCols = "age")
```

The `extraCols` and `descriptorfile` parameters used in the example will be explained later.

As we said, a `big.matrix` object `x` acts like a regular `R matrix` and commands such as `dim()` and `head()` give the appropriate results, e.g.,

```
dim(x) # How big is x?
```

```
[1] 123534969      30
```

```
x[1:6,1:6] # Show the first 6 rows and columns.
```

```
      Year Month DayOfMonth DayOfWeek DepTime CRSDepTime
[1,] 1987    10         14          3      741         730
[2,] 1987    10         15          4      729         730
[3,] 1987    10         17          6      741         730
[4,] 1987    10         18          7      729         730
[5,] 1987    10         19          1      749         730
[6,] 1987    10         21          3      728         730
```

At this point, we can compute the number of flights in 1987 in a familiar way:

```
sum(x[, "Year"] == 1987)
```

```
[1] 1311826
```

Similarly, the number of Saturday flights can be found using the command

```
sum(x[, "DayOfWeek"] == 6)
```

```
[1] 15915382
```

Depending on your hardware, the `read.big.matrix()` function could have taken over 30 minutes to complete. The prospect of waiting to load these data in future *R* sessions is very unappealing. Wouldn't it be nice if subsequent sessions could create a `bigmatrix` by simply attaching to the existing backing file and not have to wait to create the big matrix object? Fortunately, we can do this by using a descriptor file. This file contains all of the information needed to create a new `bigmatrix` from an available backing file. A new `bigmatrix`, named `y`, which uses the airline backing file, can be rapidly created with the `attach.big.matrix()` function:

```
y <- attach.big.matrix("airline.desc")
```

It is important to realize that the variables `x` and `y` now point to the same data set. This means that changes made in `x` will be reflected in `y`. To illustrate this point, let's create a new `bigmatrix` object which has 3 rows, 3 columns, and holds zero integer values.

```
foo <- big.matrix(nrow = 3, ncol = 3, type = "integer", init = 0)
```

We can look at the contents of `foo` by typing:

```
foo
```

```
      [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0
```

Now, let's create another variable `bar`:

```
bar <- foo
```

If `foo` and `bar` were *R* matrices, then `bar` would be assigned a copy of `foo`. However, since `foo` is a `bigmatrix` object, the assignment causes `bar` to point to the same data as `foo`. This is easily verified with

```
bar[1,1] <- 1
foo

      [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    0    0
[3,]    0    0    0
```

The fact that `big.matrix` objects can reference the same data can be extremely useful. By preventing *R* from making copies of a potentially large data set, calculations can be made more efficient, both in terms of memory usage and computing time. However, this functionality comes at a price. Because they do not follow the same copy semantics as *R* matrices, a statistician may have to make small modifications so that `bigmatrix` objects work correctly with existing *R* code.

**Q.1** Using the *UNIX* shell, create the `AirlineDataAll.csv` file without using a loop.

**Q.2** Write a preprocessing script (using the shell, *R*, *Python* or any tools) to create a file that can be used with `bigmemory`, i.e., convert the non-numeric values to numeric values in some well-defined manner.

**Q.3** How many flights were there for each day of the week?

**Q.4** For each year, how many flights were there for each day of the week?

**Q.5** For each year, how many of the tail codes are listed as NA?

**Q.6** Which year had the greatest proportion of late flights? Is this result significant?

**Q.7** Which flight day is best for minimizing departure delays? Which time of day?




---

## 5.4 Explorations Using Parallel Computing: The Distribution of Flight Delays

Many calculations executing quickly on small data sets take proportionally longer on larger ones. When execution time becomes an issue, parallel computing can be used to reduce the time required by computationally intensive calculations. *R* offers several different packages for executing code in parallel, each relying on different underlying technologies. Because these parallel mechanisms, or *backends*, are different, their configuration and use are also slightly different. As a result, it has traditionally been cumbersome to migrate sequential code to a parallel platform, and even after this migration was successful, the resulting parallel code was usually specific to a single parallel backend. The `foreach` package [11] addresses this issue by providing one approach to standardizing the syntax for describing parallel calculations. The `foreach` package decouples the function calls needed to run

code in parallel from the underlying technology executing the code. This approach allows a statistician to write and debug sequential code and then run it in parallel by registering an appropriate package such as `multicore` [14], `snow` [13] or `nws` [12]. Using the `foreach` package, R code can run sequentially on a single machine, in parallel on a single machine, or in parallel on a cluster of machines with no (or very minimal) code changes.

Although parallel computing can dramatically decrease execution time for many calculations, you should be aware that there are limitations. There are even cases where parallelization can increase execution time, not decrease it. When deciding whether or not to parallelize code there are a few things to keep in mind. First, there is some additional overhead associated with executing any parallel code (e.g., launching the worker processes, copying data to them, collecting the results). As a result, code should only be parallelized when each of the tasks run in parallel takes a sufficient amount of time to compute so that this overhead becomes a negligible part of the overall computational time. Second, you can expect a speed-up that is at most linear in the number of processor cores. If a snippet of code takes  $t$  seconds to execute on a single core and it is run in parallel on two cores, it will take more than  $t/2$  seconds to execute. Speed gains often diminish as the number of cores being utilized increases. Finally, each R object in the main R session that is used in the parallel computations is typically copied to each of the parallel processes. If these copies cannot all be stored in RAM, then there will be significant overhead as the operating system uses the hard drive to manage these copied data structures.

Now that you are aware of the issues with parallel computing, we are going to explore the process of creating parallel code. The next subsection discusses the process writing code that can be executed in parallel. The following section presents an approach for optimizing potentially parallel code called *split-apply-combine*. The third and final subsection explores the use of the `foreach` and `bigmemory` packages to perform more sophisticated analyses with the airline data set in parallel.

### 5.4.1 Writing a Parallelizable Loop with `foreach`

Let's go back to the question from the last section's exercises, "For each day of the week, how many flights are recorded?" To compute the solution, we could use a `big.matrix` object to store the airline data and a `for` loop to iterate over each day, finding the number of flights, e.g.,

```
x <- attach.big.matrix("airline.desc")
dayCount = integer(7)
for (i in 1:7)
  dayCount[i] <- sum(x[, "DayOfWeek"] == i)

dayCount
[1] 18136111 18061938 18103222 18083800 18091338
[5] 15915382 17143178
```

You may notice that computing the number of Monday flights is completely independent of computing the number of Tuesday flights. Wouldn't it be nice if we could take advantage of the multiple cores in a machine to calculate day of the week counts at the same time? Well, we can; and loops like this, where each iteration is independent of other iterations, are so easy to execute in parallel that they are sometimes called "embarrassingly parallel."

It is important to understand that not all loops are embarrassingly parallel and some calculations must be run sequentially. A single Markov chain simulation is generally impossible to run in parallel. As an example, consider the following random walk on the integers with an initial state of zero, implemented with

```
state <- numeric(10)
for (i in 2:10)
  state[i] <- state[i - 1] + sample( c(-1, 1), 1 )
state

[1] 0 1 0 1 2 3 2 3 4 5 6
```

For the value of `state` at time `i` to be calculated, the value of `state` at time `i - 1` must be known. Information must be shared across loop iterations to perform the simulation, and as a result, iterations of the loop must be computed sequentially, not in parallel.

Getting back to the question at hand, let's start with a sequential solution to the problem of tallying flights in a given day. Unlike the previous implementation, let's use the `foreach` [11] package. This package allows us to define embarrassingly parallel loops either sequentially or in parallel. The new code uses the `foreach()` function and the previously created `big.matrix` object:

```
library(foreach)
dayCount <- foreach(i = 1:7, .combine=c) %do% {
  sum(x[, "DayOfWeek"] == i)
}
```

Like the previous code example, loop iterations are indexed by an integer `i` going from 1 to 7. In each iteration of the loop, the number of times `DayOfWeek` is identical to the loop counter is calculated. Unlike the previous example, the calculated value is simply returned, not appended to the `dayCount` variable. The `.combine` parameter tells `foreach()` to combine the results from each iteration of the loop into a vector, which is returned and stored in the `dayCount` variable. You should also notice that after the `foreach()` statement, there is a `%do%` operator that tells the function to perform each loop iteration *sequentially*. (We'll use `%dopar%` later to perform the loop in parallel.) The result of this example and the previous one are the same; `dayCount` holds the number of flights recorded for each day of the week in a vector of numeric values.

Both the `for` and `foreach()` loop in this subsection process the entire `DayOfWeek` column 7 times to extract the number of delays for each day. For small data sets, each of these passes happen very quickly and the corresponding delay may go unnoticed. However, as the number of rows in the data set grows, each extraction requires more time, eventually delaying the exploration. This delay would be even more pronounced if we were finding the delay count for each day of the month. A day of the month count would require 31 separate passes through the data set and would take more than 4 times longer than finding the delay count for the day of the week. Wouldn't it be nice if we could perform these calculations while only passing through the data once to get the rows of interest and once to perform the calculation? In the next subsection, we will explore a different approach that only requires a single pass through the data offering significant performance gains.

### 5.4.2 Using the Split-Apply-Combine Approach for Better Performance

The task of counting the number of delays by the day of the week can be recast into separating all of the observations into 7 groups, one for each day of the week and then counting the number in each group. We can do this generally for records using the `split()` function, which passes through the data once. The `split()` function returns a named list. The names of the list, in our case, correspond to the day of the week. For each of day of the week, the list contains a vector of indices corresponding to the rows for that day:



```

# Split the rows of x by days of the week.
dow <- split(1:nrow(x), x[, "DayOfWeek"])
# Rename the names of dow
names(dow) <- c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")
# Get the first 6 rows corresponding to Monday flights.
dow$Mon[1:6]

```

```
[1] 5 11 19 26 32 38
```

Now that we have the row numbers for each day of the week, we can write a `foreach()` loop to get the counts for each day. Since all of the information to get the delay counts is contained in the `dow` variable, we don't need to include the data set in the calculation.

```

dayCount <- foreach(dayInds = dow, .combine = c) %do% {
  length(dayInds)
}

```

```
dayCount
```

```
[1] 18136111 18061938 18103222 18083800 18091338
[6] 15915382 17143178
```

There are many calculations, like this, that can be accomplished by grouping data (called the *split*), performing a single calculation on each group (the *apply*), and returning the results in a specified format (the *combine*). The term “split-apply-combine” was coined by Hadley Wickham [18] but the approach has been available in a number of different computing environments for some time under different names.

There are several advantages to the split-apply-combine approach over a traditional `for` loop. First, as already mentioned, split-apply-combine is computationally efficient. It only requires two passes through the data: one to create the groups and one to perform the calculation. Admittedly, storing the groups from the split step requires extra memory. However, this overhead is usually manageable. In contrast, a `for` loop makes a costly pass through the data for each group, which can be significant if the number of groups is large. Second, calculations that can be expressed within the split-apply-combine framework are guaranteed to be embarrassingly parallel. Because the split defines groups on the rows of a data set, calculations for each group are guaranteed to be independent. When the calculation being applied to a group is intensive, as in the next section, parallel computing can dramatically reduce execution time.

### 5.4.3 Using Split-Apply-Combine to Find the Best Time to Fly

Now that we have gained some familiarity with `foreach()` and parallel computing, let's move on to the more difficult question, “Which is the best hour of the day to fly to minimize departure delays?” This was originally posed as one of the ASA Data Expo challenges and the solution lends itself to the split-apply-combine approach.

To answer this question, we need to start by determining what is meant by the “best” hour. About half of the flights in the data set do not have departure delays. Of the flights with departure delays, most are only a few minutes late. Is the best hour the one minimizing the chance of any delay? Is the best hour the one that tends to have fewer long delays? In this situation, we have all of the flights, not a sample. Accordingly, we do not need

to perform hypothesis tests that take sampling variability of the statistics into account. Instead, we can examine the population distributions directly from the data.

Let's turn our attention to the worst delays. The question can be refined by asking, "How long were the longest 1% of flight delays for a given hour?" A frequent flyer might interpret this as, "How long could my longest delay be for 99% of my flights." Along with finding the longest 1% of departure delays, let's find the longest 0.1%, 0.01%, 0.001%. That is, let's find the quantile values for probabilities of 0.9, 0.99, 0.999, and 0.9999 in the departure delays.

To find the quantiles, we will start by splitting the data based on the hour of departure for each flight. Since there is no column that gives us the hour of departure for a given flight we will need to extract this information from the `CRSDepTime` column that encodes times, such as 8:30 AM as 830. The hour of departure can be calculated with

```
# Divide CRSDepTime by 100 and take the floor to
# get the departure hour.
depHours <- floor(x[, "CRSDepTime"]/100)
# Set the departure hours listed as 24 to 0.
depHours[depHours==24] <- 0
```

Now that we have a vector holding the departure hours for each flight, we can split on it and calculate the desired quantiles:

```
# Split on the hours.
hourInds <- split(1:length(depHours), depHours)

# Create a variable to hold the quantile probabilities.
myProbs <- c(0.9, 0.99, 0.999, 0.9999)

# Use foreach to find the quantiles for each hour.
delayQuantiles <- foreach( hour = hourInds, .combine=cbind) %do% {
  require(bigmemory)
  x <- attach.big.matrix("airline.desc")
  quantile(x[hour, "DepDelay"], myProbs,
           na.rm = TRUE)
}

# Clean up the column names.
colnames(delayQuantiles) <- names(hourInds)
```

You may have noticed that for each iteration of the `foreach()` loop, we are ensuring that the `bigmemory` package is loaded and we are calling `attach.big.matrix()`. These steps are not required when the loop is run sequentially but we will see later that they are required when the loop is run in parallel.

Now that we can calculate the delay quantiles sequentially, let's parallelize the code so that it runs faster. We'll start by registering a parallel backend. When this chapter was written, there were 5 different parallel packages that were compatible with `foreach`: `doMC` [8], `doMPI` [15], `doRedis` [5], `doSMP` [9], and `doSNOW` [10]. Each of these packages allow *R* users to exploit distinct parallel programming technologies. Packages like `doMC` and `doSMP` allow *R* users to take advantage of multiple cores on single machine. The other packages allow *R* users to create parallel programs for a single machine or even a cluster of machines. The `doRedis` package even supports programming in the "cloud." For this section, we are going to use the `doSNOW` package to perform parallel calculations on a single machine using multiple cores.

To run the `foreach()` loop in parallel, we need to determine the number of parallel *R* sessions that will be used to perform the calculation. In general, it is a good idea to use the total number of cores on the machine minus one. This allows the extra core to deal with some of the overhead associated with the parallel calculations. After the parallel workers are instantiated, they are registered with `foreach`. This step informs `foreach` how the iterations of the loop will be parallelized. Then, we change `%do%` clause in the `foreach()` loop to `%dopar%` in order to let the `foreach()` function know that code should be executed in parallel. We do all of this with

```

# Load the parallel package so we can find
# how many cores are on the machine.
library(parallel)

# Load our parallel backend.
library(doSNOW)

# Use the total number of cores on the
# machine minus one.
numParallelCores <- max(1, detectCores()-1)

# Create the parallel processes.
cl <- makeCluster(rep("localhost", numParallelCores),
                  type = "SOCK")

# Register the parallel processes with foreach.
registerDoSNOW(cl)

# Run the foreach loop again, this time
# with %dopar% so that it is executed in parallel.
delayQuantiles <- foreach(hour=hourInds, .combine=cbind) %dopar% {
  require(bigmemory)
  x <- attach.big.matrix("airline.desc")
  quantile(x[hour, "DepDelay"], myProbs, na.rm=TRUE)
}
colnames(delayQuantiles) <- names(hourInds)
stopCluster(cl)

```

When you run this code you should notice that it runs significantly faster than the sequential code (depending on how many cores you have available). By using `foreach()`, we have reduced the effort needed to migrate between sequential and parallel code.

It is important to understand that when a `foreach()` loop is run in parallel each iteration of the loop is run in a separate *R* session in another process, sometimes referred to as a worker process. Variables used inside the loop are copied from the master *R* session to each of the worker sessions. This presents two challenges when computing with a `big.matrix` in parallel. First, the `bigmemory` package is not automatically loaded in each of the worker sessions when they are started. However, this issue is easily remedied by requiring that the `bigmemory` package is loaded in the worker, before a calculation begins. Second, a `big.matrix` object holds a pointer to a location in memory that is only valid in the process where the pointer is created. As a result, the mechanism `foreach()` uses to copy variables from master to worker sessions doesn't work for a `big.matrix()`. This issue is also easily remedied by using the `attach.big.matrix()` function in the worker process after the `bigmemory` package has been loaded and before the calculation.

Now that we have efficiently calculated the airline quantile delays we can visualize these delays using `ggplot2`:

```
library(ggplot2)
dq <- melt(delayQuantiles)
names(dq) <- c("percentile", "hour", "delay")
qplot(hour, delay, data = dq, color = percentile, geom = "line")
```

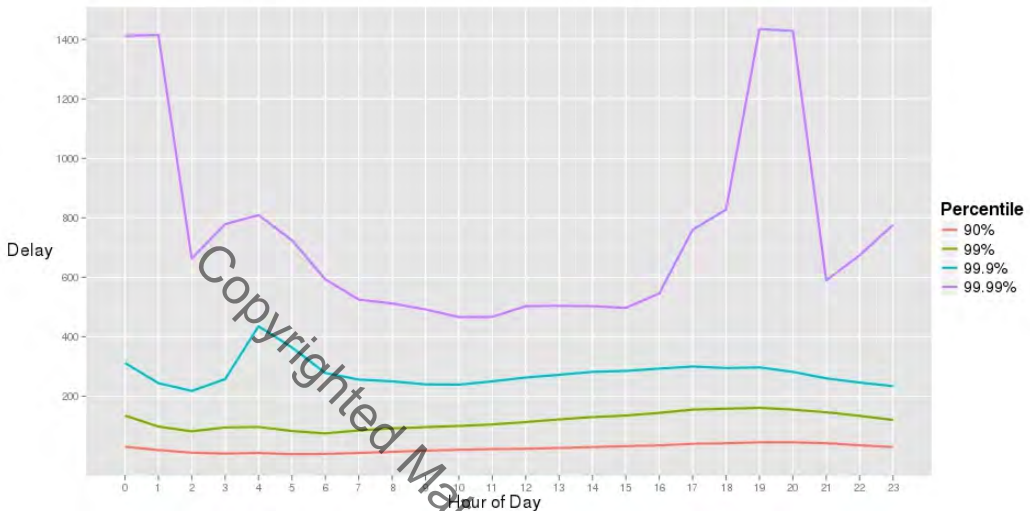


Figure 5.1: Hourly Delay Quantiles. *The airline delay quantiles (in minutes) for each hour of the day.*

Figure 5.1 shows the delays by hour. The graph indicates that delays are worse in the early hours of the morning, for these quantiles. These may correspond to red-eye flights that are delayed arriving at their destination and therefore cause delays for subsequent flights. The graph also indicates that flights leaving between 6:00 AM and 4:00 PM see the fewest lengthy delays.

For the examples in this section, a `big.matrix` object was used to hold the airline data because the data set is too large on many machines for a native `R.matrix` or `data.frame` object. A `big.matrix` object manages a large data set by caching needed data and leaving the rest on disk. It has the advantage of using less RAM than what would be needed to hold the entire data set, but this isn't its only advantage. When performing parallel calculations, variables are copied from the master `R` session to worker `R` sessions. If a variable uses a large amount of memory, then a worker `R` session incurs the overhead of waiting for these variables to be copied. Also, each worker must hold a copy of the variable potentially using up all of the available memory on one machine. A `big.matrix` object does not suffer from either of these problems. Descriptors are small and are quickly copied from one process to another. Also, a `big.matrix` object in each worker session is not a copy of the original. Each is a reference to the same data used by the master. These two qualities mean that `bigmemory` provides an efficient solution to computing in parallel with massive data and that we are now able to explore and visualize these data sets more easily than before. The `bigmemory` approach can be used for more than exploration and in the next section we will use `bigmemory` to model data and provide techniques to help answer the question “Do older planes suffer greater delays?”



Q.8 Which is the best day of the week to fly?

Q.9 Which is the best day of the month to fly?

Q.10 Are flights being given more time to reach their destination for later years?

Q.11 Which departure and arrival airport combination is associated with the worst delays?

## 5.5 From Exploration to Model: Do Older Planes Suffer Greater Delays?

Another question posed by the 2009 Data Expo was “Do older planes suffer greater delays?” The fact that the airline data does not give a plane’s age presents a difficulty in answering this. We might search for this information on the Web to see if we can find auxiliary sources of data to determine the age of each plane given its uniquely identifying tail number. However, in the absence of this auxiliary data, we can use the current data to approximate it. The year and month of each flight are available, and we have each plane’s unique tail code. For each flight, we can get the number of months the plane has been used since the first time it appears in the data set. This approach does have an issue with censoring: if a plane appears in the first year and month of the data set, we don’t know if the plane started service in that month (January, 1988) or sometime before then. Nonetheless, this approach is reasonable given the limited data we have.

How do we calculate the age of a plane? Using the `big.matrix` object from before, which holds the entire data set, we can quickly find that there are 13,536 unique tail codes that appear in the data set:

```
length(unique(x[, "TailNum"]))
```

```
[1] 13536
```

The task of finding the first time a tail code appears (in months A.D.) is independent across tail codes, so we’ll split the data by the `TailNum` variable and use `foreach()` to find this value for each `TailNum` group:

```
planeStart <- foreach(tailInds = tailSplit, .combine=c) %dopar% {
  require(bigmemory)
  x <- attach.big.matrix("airline.desc")

  # Get the first year this tail code appears in the
  # data set.
  minYear <- min(x[tailInds, "Year"], na.rm = TRUE)

  # Get the rows that have the same year.
  minYearTailInds <-
    tailInds[which(x[tailInds, "Year"] == minYear)]

  # The first month this tail code appears is the
  # minimum month for rows indexed by minYearTailInds.
  minMonth <- min(x[minYearTailInds, "Month"], na.rm = TRUE)
```

```

    # Return the first time the tail code appears
    # in months A.D.
    12*minYear + minMonth
  }

```

Remember the `extraCols = "age"` argument that was specified when the `bigmatrix` object was created in Section 5.3.4 (page 227)? This argument created an extra column named `age` in the `bigmatrix` object and this can be assigned a value with the single `R` command

```
x[, "age"] <- x[, "Year"] * 12 + x[, "Month"] -
           planeStart[x[, "TailNum"]]
```


Now that we have created a variable holding the age of a plane, what should we do with it? One approach to answering the posed question is to create a linear model with arrival delay modeled as a linear function of airplane age to see if there is an association between older planes and larger arrival delays. While the `lm()` function will not, in general, be able to handle this much data, there is a function, called `biglm()` in the `biglm` [6] package, designed to perform regressions in this setting. The `biglm()` function works on subsets of rows at a time so that a linear model can be updated incrementally with new rows of data. A wrapper for this function has been implemented in the `biganalytics` [3] package for creating linear models with `bigmatrix` objects. Regressing arrival delay as a function of age can be implemented as

```
library(biganalytics)
blm <- biglm.big.matrix( ArrDelay ~ age, data = x )
```

and calling `summary(blm)` gives a summary similar to that of an `lm` object:

```
Large data regression model: biglm(formula = formula,
  data = data, ...)
Sample size = 84216580
              Coef      (95%      CI)      SE p
(Intercept) 6.8339 6.8229 6.8448 0.0055 0
age          0.0127 0.0126 0.0129 0.0001 0
```

The model indicates that older planes are associated with large delays. However, the effect is very small and there may also be effects that are not accounted for in the model. At this point though you have the tools and techniques to begin evaluating these issues, as well as pursuing your own exploration and analysis for the airline data as well as other massive data sets.

**Q.12** One of the examples in this section creates a vector, named `planeStart`, which gives the first month in which a plane with a given tail code appears in the data set. Estimate the amount of time the loop to create this vector will take to run sequentially? and in parallel? 

**Q.13** How many of the planes ages are censored?

**Q.14** How much do weather delays contribute to arrival delay?

**Q.15** Along with age, which other variables in the airline delay data set contribute to arrival delays?

---

## Bibliography

- [1] The SQLite Web page. <http://www.sqlite.org/>, 12/9/2009.
- [2] David James. RSQLite: SQLite interface for R. <http://cran.r-project.org/package=RSQLite>, 2011. R package version 0.10.0.
- [3] Michael J. Kane and John W. Emerson. biganalytics: A library of utilities for big.matrix objects of package bigmemory. <http://cran.r-project.org/package=biganalytics>, 2010. R package version 1.0.14.
- [4] Michael J. Kane and John W. Emerson. bigmemory: Manage massive matrices with shared memory and memory-mapped files. <http://cran.r-project.org/package=bigmemory>, 2011. R package version 4.2.11.
- [5] B. Lewis. doRedis: Foreach parallel adapter for the rredis package. <http://cran.r-project.org/package=doRedis>, 2011. R package version 1.0.4.
- [6] Thomas Lumley. biglm: bounded memory linear and generalized linear models. <http://cran.r-project.org/package=biglm>, 2011. R package version 0.8.
- [7] R Development Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2012. <http://www.r-project.org>.
- [8] Revolution Analytics. doMC: Foreach parallel adaptor for the multicore package. <http://cran.r-project.org/package=doMC>, 2011. R package version 1.2.3.
- [9] Revolution Analytics. doSMP: Foreach parallel adaptor for the revoIPC package. <http://cran.r-project.org/package=doSMP>, 2011. R package version 1.0-1.
- [10] Revolution Analytics. doSNOW: Foreach parallel adaptor for the snow package. <http://cran.r-project.org/package=doSNOW>, 2011. R package version 1.0.5.
- [11] Revolution Analytics. foreach: Foreach looping construct for R. <http://cran.r-project.org/package=foreach>, 2011. R package version 1.3.1.
- [12] Revolution Computing with support and contributions from Pfizer Inc. nws: bounded memory linear and generalized linear models. <http://cran.r-project.org/package=nws>, 2010. R package version 1.7.0.1.
- [13] Luke Tierney, A Rossini, Na Li, and H. Sevcikova. snow: Simple Network of Workstations. <http://cran.r-project.org/package=snow>, 2011. R package version 0.3-8.
- [14] Simon Urbanek. multicore: Parallel processing of R code on machines with multiple cores or CPUs. <http://cran.r-project.org/package=multicore>, 2011. R package version 0.1-6.
- [15] Steve Weston. doMPI: Foreach parallel adaptor for the Rmpi package. <http://cran.r-project.org/package=doMPI>, 2010. R package version 0.1-5.
- [16] Hadley Wickham. Hadley Wickham's Homepage. <http://had.co.nz>, 19/9/2009.
- [17] Hadley Wickham. Airline on-time performance Web page. <http://stat-computing.org/dataexpo/2009/>, 2009.
- [18] Hadley Wickham. The Split-Apply-Combine Strategy for Data Analysis. *Journal of Statistical Software*, 40:1-29, 2011.