R Data Import/Export

Version 2.9.0 (2009-04-17)

R Development Core Team

ISBN 3-900051-10-0

Table of Contents

Acknowledgements

The relational databases part of this manual is based in part on an earlier manual by Douglas Bates and Saikat DebRoy. The principal author of this manual was Brian Ripley.

Many volunteers have contributed to the packages used here. The principal authors of the packages mentioned are

CORBA	Duncan Temple Lang
foreign	Thomas Lumley, Saikat DebRoy, Douglas Bates, Duncan Murdoch and Roger Bivand
hdf5	Marcus Daniels
ncdf	David Pierce
ncvar	Juerg Schmidli
rJava	Simon Urbanek
RMySQL	David James and Saikat DebRoy
RNetCDF	Pavel Michna
RODBC	Michael Lapsley and Brian Ripley
RSPerl	Duncan Temple Lang
RSPython	Duncan Temple Lang
SJava	John Chambers and Duncan Temple Lang
XML	Duncan Temple Lang

Brian Ripley is the author of the support for connections.

1 Introduction

Reading data into a statistical system for analysis and exporting the results to some other system for report writing can be frustrating tasks that can take far more time than the statistical analysis itself, even though most readers will find the latter far more appealing.

This manual describes the import and export facilities available either in R itself or via packages which are available from CRAN. Some of the packages described are still under development but they already provide useful functionality.

Unless otherwise stated, everything described in this manual is available on all platforms running R.

In general, statistical systems like R are not particularly well suited to manipulations of large-scale data. Some other systems are better than R at this, and part of the thrust of this manual is to suggest that rather than duplicating functionality in R we can make another system do the work! (For example Therneau & Grambsch (2000) comment that they prefer to do data manipulation in SAS and then use **survival** in S for the analysis.) Several recent packages allow functionality developed in languages such as `Java`, `perl` and `python` to be directly integrated with R code, making the use of facilities in these languages even more appropriate. (See the **SJava**, **RSPerl** and **RSPython** packages from the Omegahat project, `http://www.omegahat.org`, and the **rJava** package from CRAN.)

It is also worth remembering that R like S comes from the Unix tradition of small reusable tools, and it can be rewarding to use tools such as `awk` and `perl` to manipulate data before import or after export. The case study in Becker, Chambers & Wilks (1988, Chapter 9) is an example of this, where Unix tools were used to check and manipulate the data before input to S. R itself takes that approach, using `perl` to manipulate its databases of help files rather than R itself, and the function `read.fwf` used a call to a `perl` script until it was decided not to require `perl` at run-time. The traditional Unix tools are now much more widely available, including on Windows.

1.1 Imports

The easiest form of data to import into R is a simple text file, and this will often be acceptable for problems of small or medium scale. The primary function to import from a text file is `scan`, and this underlies most of the more convenient functions discussed in Chapter 2 [Spreadsheet-like data], page 6.

However, all statistical consultants are familiar with being presented by a client with a floppy disc or CD-R of data in some proprietary binary format, for example 'an Excel spreadsheet' or 'an SPSS file'. Often the simplest thing to do is to use the originating application to export the data as a text file (and statistical consultants will have copies of the most common applications on their computers for that purpose). However, this is not always possible, and Chapter 3 [Importing from other statistical systems], page 12 discusses what facilities are available to access such files directly from R. For Excel spreadsheets, the available methods are summarized in Chapter 8 [Reading Excel spreadsheets], page 28.

In a few cases, data have been stored in a binary form for compactness and speed of access. One application of this that we have seen several times is imaging data, which is normally stored as a stream of bytes as represented in memory, possibly preceded by a

header. Such data formats are discussed in Chapter 5 [Binary files], page 20 and Section 6.5 [Binary connections], page 24.

For much larger databases it is common to handle the data using a database management system (DBMS). There is once again the option of using the DBMS to extract a plain file, but for many such DBMSs the extraction operation can be done directly from an R package: See Chapter 4 [Relational databases], page 14. Importing data via network connections is discussed in Chapter 7 [Network interfaces], page 26.

1.2 Export to text files

Exporting results from R is usually a less contentious task, but there are still a number of pitfalls. There will be a target application in mind, and normally a text file will be the most convenient interchange vehicle. (If a binary file is required, see Chapter 5 [Binary files], page 20.)

Function `cat` underlies the functions for exporting data. It takes a `file` argument, and the `append` argument allows a text file to be written via successive calls to `cat`. Better, especially if this is to be done many times, is to open a `file` connection for writing or appending, and `cat` to that connection, then `close` it.

The most common task is to write a matrix or data frame to file as a rectangular grid of numbers, possibly with row and column labels. This can be done by the functions `write.table` and `write`. Function `write` just writes out a matrix or vector in a specified number of columns (and transposes a matrix). Function `write.table` is more convenient, and writes out a data frame (or an object that can be coerced to a data frame) with row and column labels.

There are a number of issues that need to be considered in writing out a data frame to a text file.

1. **Precision**

   Most of the conversions of real/complex numbers done by these functions is to full precision, but those by `write` are governed by the current setting of `options(digits)`. For more control, use `format` on a data frame, possibly column-by-column.

2. **Header line**

   R prefers the header line to have no entry for the row names, so the file looks like

   ```
                       dist    climb   time
   Greenmantle         2.5     650     16.083
      ...
   ```

   Some other systems require a (possibly empty) entry for the row names, which is what `write.table` will provide if argument `col.names = NA` is specified. Excel is one such system.

3. **Separator**

   A common field separator to use in the file is a comma, as that is unlikely to appear in any of the fields, in English-speaking countries. Such files are known as CSV (comma separated values) files, and wrapper function `write.csv` provides appropriate defaults. In some locales the comma is used as the decimal point (set this in `write.table` by `dec = ","`) and there CSV files use the semicolon as the field separator: use `write.csv2` for appropriate defaults.

Using a semicolon or tab (`sep = "\t"`) are probably the safest options.

4. **Missing values**

   By default missing values are output as `NA`, but this may be changed by argument `na`. Note that `NaN`s are treated as `NA` by `write.table`, but not by `cat` nor `write`.

5. **Quoting strings**

   By default strings are quoted (including the row and column names). Argument `quote` controls quoting of character and factor variables.

   Some care is needed if the strings contain embedded quotes. Three useful forms are

```
> df <- data.frame(a = I("a \" quote"))
> write.table(df)
"a"
"1" "a \" quote"
> write.table(df, qmethod = "double")
"a"
"1" "a "" quote"
> write.table(df, quote = FALSE, sep = ",")
a
1,a " quote
```

   The second is the form of escape commonly used by spreadsheets.

Function `write.matrix` in package **MASS** provides a specialized interface for writing matrices, with the option of writing them in blocks and thereby reducing memory usage.

It is possible to use `sink` to divert the standard R output to a file, and thereby capture the output of (possibly implicit) `print` statements. This is not usually the most efficient route, and the `options(width)` setting may need to be increased.

Function `write.foreign` in package **foreign** uses `write.table` to produce a text file and also writes a code file that will read this text file into another statistical package. There is currently support for export to `SPSS` and `Stata`.

1.3 XML

When reading data from text files, it is the responsibility of the user to know and to specify the conventions used to create that file, e.g. the comment character, whether a header line is present, the value separator, the representation for missing values (and so on) described in Section 1.2 [Export to text files], page 3. A markup language which can be used to describe not only content but also the structure of the content can make a file self-describing, so that one need not provide these details to the software reading the data.

The eXtensible Markup Language – more commonly known simply as XML – can be used to provide such structure, not only for standard datasets but also more complex data structures. XML is becoming extremely popular and is emerging as a standard for general data markup and exchange. It is being used by different communities to describe geographical data such as maps, graphical displays, mathematics and so on.

The **XML** package provides general facilities for reading and writing XML documents within both R and S-PLUS in the hope that we can easily make use of this technology as it evolves. Several people are exploring how we can use XML for, amongst other things, representing datasets to be shared across different applications; storing R and S-PLUS objects so

they can be shared by both systems; representing plots via SVG (Scalable Vector Graphics, a dialect of XML); representing function documentation; generating "live" analyses/reports that contain text, data and code.

A description of the facilities of the **XML** package is outside the scope of this document: see the package's Web page at http://www.omegahat.org/RSXML for details and examples. Package **StatDataML** on CRAN is one example building on **XML**.

2 Spreadsheet-like data

In Section 1.2 [Export to text files], page 3 we saw a number of variations on the format of a spreadsheet-like text file, in which the data are presented in a rectangular grid, possibly with row and column labels. In this section we consider importing such files into R.

2.1 Variations on `read.table`

The function `read.table` is the most convenient way to read in a rectangular grid of data. Because of the many possibilities, there are several other functions that call `read.table` but change a group of default arguments.

Beware that `read.table` is an inefficient way to read in very large numerical matrices: see `scan` below.

Some of the issues to consider are:

1. **Encoding**

   If the file contains non-ASCII character fields, ensure that it is read in the correct encoding. This is mainly an issue for reading Latin-1 files in a UTF-8 locale, which can be done by something like

   ```
   read.table(file("file.dat", encoding="latin1"))
   ```

   Note that this will work in any locale which can represent Latin-1 names.

2. **Header line**

   We recommend that you specify the `header` argument explicitly, Conventionally the header line has entries only for the columns and not for the row labels, so is one field shorter than the remaining lines. (If R sees this, it sets `header = TRUE`.) If presented with a file that has a (possibly empty) header field for the row labels, read it in by something like

   ```
   read.table("file.dat", header = TRUE, row.names = 1)
   ```

   Column names can be given explicitly via the `col.names`; explicit names override the header line (if present).

3. **Separator**

   Normally looking at the file will determine the field separator to be used, but with white-space separated files there may be a choice between the default `sep = ""` which uses any white space (spaces, tabs or newlines) as a separator, `sep = " "` and `sep = "\t"`. Note that the choice of separator affects the input of quoted strings.

   If you have a tab-delimited file containing empty fields be sure to use `sep = "\t"`.

4. **Quoting**

   By default character strings can be quoted by either '`"`' or '`'`', and in each case all the characters up to a matching quote are taken as part of the character string. The set of valid quoting characters (which might be none) is controlled by the `quote` argument. For `sep = "\n"` the default is changed to `quote = ""`.

   If no separator character is specified, quotes can be escaped within quoted strings by immediately preceding them by '\', C-style.

   If a separator character is specified, quotes can be escaped within quoted strings by doubling them as is conventional in spreadsheets. For example

```
'One string isn''t two',"one more"
```

can be read by

```
read.table("testfile", sep = ",")
```

This does not work with the default separator.

5. **Missing values**

   By default the file is assumed to contain the character string `NA` to represent missing values, but this can be changed by the argument `na.strings`, which is a vector of one or more character representations of missing values.

   Empty fields in numeric columns are also regarded as missing values.

   In numeric columns, the values `NaN`, `Inf` and `-Inf` are accepted.

6. **Unfilled lines**

   It is quite common for a file exported from a spreadsheet to have all trailing empty fields (and their separators) omitted. To read such files set `fill = TRUE`.

7. **White space in character fields**

   If a separator is specified, leading and trailing white space in character fields is regarded as part of the field. To strip the space, use argument `strip.white = TRUE`.

8. **Blank lines**

   By default, `read.table` ignores empty lines. This can be changed by setting `blank.lines.skip = FALSE`, which will only be useful in conjunction with `fill = TRUE`, perhaps to use blank rows to indicate missing cases in a regular layout.

9. **Classes for the variables**

   Unless you take any special action, `read.table` tries to select a suitable class for each variable in the data frame. It tries in turn `logical`, `integer`, `numeric` and `complex`, moving on if any entry is not missing and cannot be converted.[1] If all of these fail, the variable is converted to a factor.

   Arguments `colClasses` and `as.is` provide greater control. `as.is` suppresses conversion of character vectors to factors (only). Using `colClasses` allows the desired class to be set for each column in the input.

   Note that `colClasses` and `as.is` are specified *per* column, not *per* variable, and so include the column of row names (if any).

10. **Comments**

    By default, `read.table` uses '`#`' as a comment character, and if this is encountered (except in quoted strings) the rest of the line is ignored. Lines containing only white space and a comment are treated as blank lines.

    If it is known that there will be no comments in the data file, it is safer (and may be faster) to use `comment.char = ""`.

11. **Escapes**

    Many OSes have conventions for using backslash as an escape character in text files, but Windows does not (and uses backslash in path names). It is optional in R whether such conventions are applied to data files.

[1] This is normally fast as looking at the first entry rules out most of the possibilities.

Both `read.table` and `scan` have a logical argument `allowEscapes`. As from R 2.2.0 this is false by default, and backslashes are then only interpreted as (under circumstances described above) escaping quotes. If this set to be true, C-style escapes are interpreted, namely the control characters `\a, \b, \f, \n, \r, \t, \v` and octal and hexadecimal representations like `\040` and `\0x2A`. Any other escaped character is treated as itself, including backslash.

Convenience functions `read.csv` and `read.delim` provide arguments to `read.table` appropriate for CSV and tab-delimited files exported from spreadsheets in English-speaking locales. The variations `read.csv2` and `read.delim2` are appropriate for use in countries where the comma is used for the decimal point.

If the options to `read.table` are specified incorrectly, the error message will usually be of the form

```
Error in scan(file = file, what = what, sep = sep, :
        line 1 did not have 5 elements
```

or

```
Error in read.table("files.dat", header = TRUE) :
        more columns than column names
```

This may give enough information to find the problem, but the auxiliary function `count.fields` can be useful to investigate further.

Efficiency can be important when reading large data grids. It will help to specify `comment.char = ""`, `colClasses` as one of the atomic vector types (logical, integer, numeric, complex, character or perhaps raw) for each column, and to give `nrows`, the number of rows to be read (and a mild over-estimate is better than not specifying this at all). See the examples below.

2.2 Fixed-width-format files

Sometimes data files have no field delimiters but have fields in pre-specified columns. This was very common in the days of punched cards, and is still sometimes used to save file space.

Function `read.fwf` provides a simple way to read such files, specifying a vector of field widths. The function reads the file into memory as whole lines, splits the resulting character strings, writes out a temporary tab-separated file and then calls `read.table`. This is adequate for small files, but for anything more complicated we recommend using the facilities of a language like `perl` to pre-process the file.

Function `read.fortran` is a similar function for fixed-format files, using Fortran-style column specifications.

2.3 Data Interchange Format (DIF)

An old format sometimes used for spreadsheet-like data is DIF, or Data Interchange format.

Function `read.DIF` provides a simple way to read such files. It takes arguments similar to `read.table` for assigning types to each of the columns.

In Windows, spreadsheets often store spreadsheet data on the clipboard in this format; `read.DIF("clipboard")` can read it from there directly. It is slightly more robust than `read.table("clipboard")` in handling spreadsheets with empty cells.

2.4 Using scan directly

Both `read.table` and `read.fwf` use `scan` to read the file, and then process the results of `scan`. They are very convenient, but sometimes it is better to use `scan` directly.

Function `scan` has many arguments, most of which we have already covered under `read.table`. The most crucial argument is `what`, which specifies a list of modes of variables to be read from the file. If the list is named, the names are used for the components of the returned list. Modes can be numeric, character or complex, and are usually specified by an example, e.g. `0`, `""` or `0i`. For example

```
cat("2 3 5 7", "11 13 17 19", file="ex.dat", sep="\n")
scan(file="ex.dat", what=list(x=0, y="", z=0), flush=TRUE)
```

returns a list with three components and discards the fourth column in the file.

There is a function `readLines` which will be more convenient if all you want is to read whole lines into R for further processing.

One common use of `scan` is to read in a large matrix. Suppose file '`matrix.dat`' just contains the numbers for a 200 x 2000 matrix. Then we can use

```
A <- matrix(scan("matrix.dat", n = 200*2000), 200, 2000, byrow = TRUE)
```

On one test this took 1 second (under Linux, 3 seconds under Windows on the same machine) whereas

```
A <- as.matrix(read.table("matrix.dat"))
```

took 10 seconds (and more memory), and

```
A <- as.matrix(read.table("matrix.dat", header = FALSE, nrows = 200,
                          comment.char = "", colClasses = "numeric"))
```

took 7 seconds. The difference is almost entirely due to the overhead of reading 2000 separate short columns: were they of length 2000, `scan` took 9 seconds whereas `read.table` took 18 if used efficiently (in particular, specifying `colClasses`) and 125 if used naively.

Note that timings can depend on the type read and the data. Consider reading a million distinct integers:

```
writeLines(as.character((1+1e6):2e6), "ints.dat")
xi <- scan("ints.dat", what=integer(0), n=1e6)   # 0.77s
xn <- scan("ints.dat", what=numeric(0), n=1e6)   # 0.93s
xc <- scan("ints.dat", what=character(0), n=1e6) # 0.85s
xf <- as.factor(xc)                              # 2.2s
DF <- read.table("ints.dat")                     # 4.5s
```

and a million examples of a small set of codes:

```
code <- c("LMH", "SJC", "CHCH", "SPC", "SOM")
writeLines(sample(code, 1e6, replace=TRUE), "code.dat")
y <- scan("code.dat", what=character(0), n=1e6)  # 0.44s
yf <- as.factor(y)                               # 0.21s
DF <- read.table("code.dat")                     # 4.9s
DF <- read.table("code.dat", nrows=1e6)          # 3.6s
```

Note that these timings depend heavily on the operating system (the basic reads in Windows take at least as twice as long as these Linux times) and on the precise state of the garbage collector.

2.5 Re-shaping data

Sometimes spreadsheet data is in a compact format that gives the covariates for each subject followed by all the observations on that subject. R's modelling functions need observations in a single column. Consider the following sample of data from repeated MRI brain measurements

```
Status   Age    V1     V2     V3    V4
     P 23646 45190  50333  55166 56271
    CC 26174 35535  38227  37911 41184
    CC 27723 25691  25712  26144 26398
    CC 27193 30949  29693  29754 30772
    CC 24370 50542  51966  54341 54273
    CC 28359 58591  58803  59435 61292
    CC 25136 45801  45389  47197 47126
```

There are two covariates and up to four measurements on each subject. The data were exported from Excel as a file '`mr.csv`'.

We can use `stack` to help manipulate these data to give a single response.

```
zz <- read.csv("mr.csv", strip.white = TRUE)
zzz <- cbind(zz[gl(nrow(zz), 1, 4*nrow(zz)), 1:2], stack(zz[, 3:6]))
```

with result

```
      Status   Age values ind
X1         P 23646  45190  V1
X2        CC 26174  35535  V1
X3        CC 27723  25691  V1
X4        CC 27193  30949  V1
X5        CC 24370  50542  V1
X6        CC 28359  58591  V1
X7        CC 25136  45801  V1
X11        P 23646  50333  V2
...
```

Function `unstack` goes in the opposite direction, and may be useful for exporting data.

Another way to do this is to use the function `reshape`, by

```
> reshape(zz, idvar="id",timevar="var",
  varying=list(c("V1","V2","V3","V4")),direction="long")
    Status   Age var    V1 id
1.1      P 23646   1 45190  1
2.1     CC 26174   1 35535  2
3.1     CC 27723   1 25691  3
4.1     CC 27193   1 30949  4
5.1     CC 24370   1 50542  5
6.1     CC 28359   1 58591  6
7.1     CC 25136   1 45801  7
1.2      P 23646   2 50333  1
2.2     CC 26174   2 38227  2
...
```

The `reshape` has a more complicated syntax than `stack` but can be used for data where the 'long' form has more than the one column in this example. With `direction="wide"`, `reshape` can also perform the opposite transformation.

2.6 Flat contingency tables

Displaying higher-dimensional contingency tables in array form typically is rather inconvenient. In categorical data analysis, such information is often represented in the form of bordered two-dimensional arrays with leading rows and columns specifying the combination of factor levels corresponding to the cell counts. These rows and columns are typically "ragged" in the sense that labels are only displayed when they change, with the obvious convention that rows are read from top to bottom and columns are read from left to right. In R, such "flat" contingency tables can be created using `ftable`, which creates objects of class `"ftable"` with an appropriate print method.

As a simple example, consider the R standard data set `UCBAdmissions` which is a 3-dimensional contingency table resulting from classifying applicants to graduate school at UC Berkeley for the six largest departments in 1973 classified by admission and sex.

```
> data(UCBAdmissions)
> ftable(UCBAdmissions)
                 Dept   A   B   C   D   E   F
Admit    Gender
Admitted Male        512 353 120 138  53  22
         Female       89  17 202 131  94  24
Rejected Male        313 207 205 279 138 351
         Female       19   8 391 244 299 317
```

The printed representation is clearly more useful than displaying the data as a 3-dimensional array.

There is also a function `read.ftable` for reading in flat-like contingency tables from files. This has additional arguments for dealing with variants on how exactly the information on row and column variables names and levels is represented. The help page for `read.ftable` has some useful examples. The flat tables can be converted to standard contingency tables in array form using `as.table`.

Note that flat tables are characterized by their "ragged" display of row (and maybe also column) labels. If the full grid of levels of the row variables is given, one should instead use `read.table` to read in the data, and create the contingency table from this using `xtabs`.

3 Importing from other statistical systems

In this chapter we consider the problem of reading a binary data file written by another statistical system. This is often best avoided, but may be unavoidable if the originating system is not available.

3.1 EpiInfo, Minitab, S-PLUS, SAS, SPSS, Stata, Systat

The recommended package **foreign** provides import facilities for files produced by these statistical systems, and for export to Stata. In some cases these function may require substantially less memory than `read.table` would. `write.foreign` (See Section 1.2 [Export to text files], page 3) provides an export mechanism with support currently for `SPSS` and `Stata`.

EpiInfo versions 5 and 6 stored data in a self-describing fixed-width text format. `read.epiinfo` will read these '`.REC`' files into an R data frame. EpiData also produces data in this format.

Function `read.mtp` imports a 'Minitab Portable Worksheet'. This returns the components of the worksheet as an R list.

Function `read.xport` reads a file in SAS Transport (XPORT) format and return a list of data frames. If SAS is available on your system, function `read.ssd` can be used to create and run a SAS script that saves a SAS permanent dataset ('`.ssd`' or '`.sas7bdat`') in Transport format. It then calls `read.xport` to read the resulting file. (Package **Hmisc** has a similar function `sas.get`, also running SAS.) For those without access to SAS but running on Windows, the SAS System Viewer (a zero-cost download) can be used to open SAS datasets and export them to e.g. '`.csv`' format.

Function `read.S` which can read binary objects produced by S-PLUS 3.x, 4.x or 2000 on (32-bit) Unix or Windows (and can read them on a different OS). This is able to read many but not all S objects: in particular it can read vectors, matrices and data frames and lists containing those.

Function `data.restore` reads S-PLUS data dumps (created by `data.dump`) with the same restrictions (except that dumps from the Alpha platform can also be read). It should be possible to read data dumps from S-PLUS 5.x and later written with `data.dump(oldStyle=T)`.

If you have access to S-PLUS, it is usually more reliable to `dump` the object(s) in S-PLUS and `source` the dumpfile in R. For S-PLUS 5.x and 6.x you may need to use `dump(..., oldStyle=T)`, and to read in very large objects it may be preferable to use the dumpfile as a batch script rather than use the `source` function.

Function `read.spss` can read files created by the 'save' and 'export' commands in SPSS. It returns a list with one component for each variable in the saved data set. SPSS variables with value labels are optionally converted to R factors.

SPSS Data Entry is an application for creating data entry forms. By default it creates data files with extra formatting information that `read.spss` cannot handle, but it is possible to export the data in an ordinary SPSS format.

Stata '`.dta`' files are a binary file format. Files from versions 5, 6, 7/SE and 8 of Stata can be read and written by functions `read.dta` and `write.dta`. Stata variables with value labels are optionally converted to (and from) R factor.

`read.systat` reads those Systat `SAVE` files that are rectangular data files (`mtype = 1`) written on little-endian machines (such as from Windows). These have extension '`.sys`' or (more recently) '`.syd`'.

3.2 Octave

Octave is a numerical linear algebra system (`http://www.octave.org`), and function `read.octave` in package **foreign** can read in files in Octave text data format created using the Octave command `save -ascii`, with support for most of the common types of variables, including the standard atomic (real and complex scalars, matrices, and N-d arrays, strings, ranges, and boolean scalars and matrices) and recursive (structs, cells, and lists) ones.

4 Relational databases

4.1 Why use a database?

There are limitations on the types of data that R handles well. Since all data being manipulated by R are resident in memory, and several copies of the data can be created during execution of a function, R is not well suited to extremely large data sets. Data objects that are more than a (few) hundred megabytes in size can cause R to run out of memory.

R does not easily support concurrent access to data. That is, if more than one user is accessing, and perhaps updating, the same data, the changes made by one user will not be visible to the others.

R does support persistence of data, in that you can save a data object or an entire worksheet from one session and restore it at the subsequent session, but the format of the stored data is specific to R and not easily manipulated by other systems.

Database management systems (DBMSs) and, in particular, relational DBMSs (RDBMSs) *are* designed to do all of these things well. Their strengths are

1. To provide fast access to selected parts of large databases.
2. Powerful ways to summarize and cross-tabulate columns in databases.
3. Store data in more organized ways than the rectangular grid model of spreadsheets and R data frames.
4. Concurrent access from multiple clients running on multiple hosts while enforcing security constraints on access to the data.
5. Ability to act as a server to a wide range of clients.

The sort of statistical applications for which DBMS might be used are to extract a 10% sample of the data, to cross-tabulate data to produce a multi-dimensional contingency table, and to extract data group by group from a database for separate analysis.

4.2 Overview of RDBMSs

Traditionally there have been large (and expensive) commercial RDBMSs (Informix; Oracle; Sybase; IBM's DB/2; Microsoft SQL Server on Windows) and academic and small-system databases (such as MySQL, PostgreSQL, Microsoft Access, ...), the former marked out by much greater emphasis on data security features. The line is blurring, with the Open Source PostgreSQL having more and more high-end features, and 'free' versions of Informix, Oracle and Sybase being made available on Linux.

There are other commonly used data sources, including spreadsheets, non-relational databases and even text files (possibly compressed). Open Database Connectivity (ODBC) is a standard to use all of these data sources. It originated on Windows (see `http://www.microsoft.com/data/odbc/`) but is also implemented on Linux/Unix.

All of the packages described later in this chapter provide clients to client/server databases. The database can reside on the same machine or (more often) remotely. There is an ISO standard (in fact several: SQL92 is ISO/IEC 9075, also known as ANSI X3.135-1992, and SQL99 is coming into use) for an interface language called SQL (Structured Query Language, sometimes pronounced 'sequel': see Bowman *et al.* 1996 and Kline and Kline 2001) which these DBMSs support to varying degrees.

4.2.1 SQL queries

The more comprehensive R interfaces generate SQL behind the scenes for common operations, but direct use of SQL is needed for complex operations in all. Conventionally SQL is written in upper case, but many users will find it more convenient to use lower case in the R interface functions.

A relational DBMS stores data as a database of *tables* (or *relations*) which are rather similar to R data frames, in that they are made up of *columns* or *fields* of one type (numeric, character, date, currency, . . .) and *rows* or *records* containing the observations for one entity.

SQL 'queries' are quite general operations on a relational database. The classical query is a SELECT statement of the type

```
SELECT State, Murder FROM USArrests WHERE Rape > 30 ORDER BY Murder

SELECT t.sch, c.meanses, t.sex, t.achieve
  FROM student as t, school as c WHERE t.sch = c.id

SELECT sex, COUNT(*) FROM student GROUP BY sex

SELECT sch, AVG(sestat) FROM student GROUP BY sch LIMIT 10
```

The first of these selects two columns from the R data frame `USArrests` that has been copied across to a database table, subsets on a third column and asks the results be sorted. The second performs a database *join* on two tables `student` and `school` and returns four columns. The third and fourth queries do some cross-tabulation and return counts or averages. (The five aggregation functions are COUNT(*) and SUM, MAX, MIN and AVG, each applied to a single column.)

SELECT queries use FROM to select the table, WHERE to specify a condition for inclusion (or more than one condition separated by AND or OR), and ORDER BY to sort the result. Unlike data frames, rows in RDBMS tables are best thought of as unordered, and without an ORDER BY statement the ordering is indeterminate. You can sort (in lexicographical order) on more than one column by separating them by commas. Placing DESC after an ORDER BY puts the sort in descending order.

SELECT DISTINCT queries will only return one copy of each distinct row in the selected table.

The GROUP BY clause selects subgroups of the rows according to the criterion. If more than one column is specified (separated by commas) then multi-way cross-classifications can be summarized by one of the five aggregation functions. A HAVING clause allows the select to include or exclude groups depending on the aggregated value.

If the SELECT statement contains an ORDER BY statement that produces a unique ordering, a LIMIT clause can be added to select (by number) a contiguous block of output rows. This can be useful to retrieve rows a block at a time. (It may not be reliable unless the ordering is unique, as the LIMIT clause can be used to optimize the query.)

There are queries to create a table (CREATE TABLE, but usually one copies a data frame to the database in these interfaces), INSERT or DELETE or UPDATE data. A table is destroyed by a DROP TABLE 'query'.

Kline and Kline (2001) discuss the details of the implementation of SQL in SQL Server 2000, Oracle, MySQL and PostgreSQL.

4.2.2 Data types

Data can be stored in a database in various data types. The range of data types is DBMS-specific, but the SQL standard defines many types, including the following that are widely implemented (often not by the SQL name).

`float(p)` Real number, with optional precision. Often called `real` or `double` or `double precision`.

`integer` 32-bit integer. Often called `int`.

`smallint` 16-bit integer

`character(n)`
fixed-length character string. Often called `char`.

`character varying(n)`
variable-length character string. Often called `varchar`. Almost always has a limit of 255 chars.

`boolean` true or false. Sometimes called `bool` or `bit`.

`date` calendar date

`time` time of day

`timestamp`
date and time

There are variants on `time` and `timestamp`, `with timezone`. Other types widely implemented are `text` and `blob`, for large blocks of text and binary data, respectively.

The more comprehensive of the R interface packages hide the type conversion issues from the user.

4.3 R interface packages

There are several packages available on CRAN to help R communicate with DBMSs. They provide different levels of abstraction. Some provide means to copy whole data frames to and from databases. All have functions to select data within the database via SQL queries, and to retrieve the result as a whole as a data frame or in pieces (usually as groups of rows).

All except **RODBC** are tied to one DBMS, but work is in progress towards a unified 'front-end' package **DBI** (http://developer.r-project.org/db) in conjuction with a 'back-end', the most developed of which is **RMySQL**. Also on CRAN are the back-ends **ROracle**, **RPostgreSQL** and **RSQLite** (which works with the bundled DBMS `SQLite`, http://www.hwaci.com/sw/sqlite).

Two earlier packages **RmSQL** and **RPgSQL** are now unsupported and in the archive area on CRAN: the BioConductor project has updated **RdbiPgSQL** (formerly on CRAN). **PL/R** (http://www.joeconway.com/plr/) is a project to embed R into PostgreSQL.

4.3.1 Packages DBI and RMySQL

Package **RMySQL** on CRAN provides an interface to the MySQL database system (see http://www.mysql.com and Dubois, 2000.). The description here applies to version `0.5-0`: earlier versions had a substantially different interface. The current version requires the **DBI**

package, and this description will apply with minor changes to all the other back-ends to **DBI**.

MySQL exists on Unix/Linux and Windows: as from version 3.23.x (Jan 2001) it is released under GPL. MySQL is a 'light and lean' database. (It preserves the case of names where the operating file system is case-sensitive, so not on Windows.) Package **RMySQL** has been used on both Linux and Windows.

The call `dbDriver("MySQL")` returns a database connection manager object, and then a call to `dbConnect` opens a database connection which can subsequently be closed by a call to the generic function `dbDisconnect`. Use `dbDriver("Oracle")`, `dbdDriver("PostgreSQL")` or `dbDriver("SQLite")` with those DBMSs and **ROracle**, **PostgreSQL** or **RSQLite** respectively.

SQL queries can be sent by either `dbSendQuery` or `dbGetQuery`. `dbGetquery` sends the query and retrieves the results as a data frame. `dbSendQuery` sends the query and returns an object of class inheriting from `"DBIResult"` which can be used to retrieve the results, and subsequently used in a call to `dbClearResult` to remove the result.

Function `fetch` is used to retrieve some or all of the rows in the query result, as a list. The function `dbHasCompleted` indicates if all the rows have been fetched, and `dbGetRowCount` returns the number of rows in the result.

These are convenient interfaces to read/write/test/delete tables in the database. `dbReadTable` and `dbWriteTable` copy to and from an R data frame, mapping the row names of the data frame to the field `row_names` in the `MySQL` table.

```
> library(RMySQL) # will load DBI as well
## open a connection to a MySQL database
> con <- dbConnect(dbDriver("MySQL"), dbname = "test")
## list the tables in the database
> dbListTables(con)
## load a data frame into the database, deleting any existing copy
> data(USArrests)
> dbWriteTable(con, "arrests", USArrests, overwrite = TRUE)
TRUE
> dbListTables(con)
[1] "arrests"
## get the whole table
> dbReadTable(con, "arrests")
               Murder Assault UrbanPop Rape
Alabama          13.2     236       58 21.2
Alaska           10.0     263       48 44.5
Arizona           8.1     294       80 31.0
Arkansas          8.8     190       50 19.5
...
## Select from the loaded table
> dbGetQuery(con, paste("select row_names, Murder from arrests",
                        "where Rape > 30 order by Murder"))
   row_names Murder
1   Colorado    7.9
2    Arizona    8.1
3 California    9.0
4     Alaska   10.0
5 New Mexico   11.4
6   Michigan   12.1
7     Nevada   12.2
8    Florida   15.4
```

```
> dbRemoveTable(con, "arrests")
> dbDisconnect(con)
```

4.3.2 Package RODBC

Package **RODBC** on CRAN provides an interface to database sources supporting an ODBC interface. This is very widely available, and allows the same R code to access different database systems. **RODBC** runs on both Unix/Linux and Windows, and almost all database systems provide support for ODBC. We have tested Microsoft SQL Server, Access, MySQL and PostgreSQL on Windows and MySQL, Oracle, PostgreSQL and SQLite on Linux.

ODBC is a client-server system, and we have happily connected to a DBMS running on a Unix server from a Windows client, and *vice versa*.

On Windows ODBC support is normally installed, and current versions are available from http://www.microsoft.com/data/odbc/ as part of MDAC. On Unix/Linux you will need an ODBC Driver Manager such as unixODBC (http://www.unixODBC.org) or iOBDC (http://www.iODBC.org) and an installed driver for your database system.

Windows provides drivers not just for DBMSs but also for Excel ('`.xls`') spreadsheets, DBase ('`.dbf`') files and even text files. (The named applications do *not* need to be installed. Which file formats are supported depends on the the versions of the drivers.) There are versions for Excel 2007 and Access 2007 (go to http://download.microsoft.com, and search for 'Office ODBC', which will lead to '`AccessDatabaseEngine.exe`'), the '2007 Office System Driver'.

Many simultaneous connections are possible. A connection is opened by a call to `odbcConnect` or `odbcDriverConnect` (which on the Windows GUI allows a database to be selected via dialog boxes) which returns a handle used for subsequent access to the database. Printing a connection will provide some details of the ODBC connection, and calling `odbcGetInfo` will give details on the client and server.

A connection is closed by a call to `close` or `odbcClose`, and also (with a warning) when not R object refers to it and at the end of an R session.

Details of the tables on a connection can be found using `sqlTables`.

Function `sqlSave` copies an R data frame to a table in the database, and `sqlFetch` copies a table in the database to an R data frame.

An SQL query can be sent to the database by a call to `sqlQuery`. This returns the result in an R data frame. (`sqlCopy` sends a query to the database and saves the result as a table in the database.) A finer level of control is attained by first calling `odbcQuery` and then `sqlGetResults` to fetch the results. The latter can be used within a loop to retrieve a limited number of rows at a time, as can function `sqlFetchMore`.

Here is an example using PostgreSQL, for which the ODBC driver maps column and data frame names to lower case. We use a database `testdb` we created earlier, and had the DSN (data source name) set up in '`~/.odbc.ini`' under `unixODBC`. Exactly the same code worked using MyODBC to access a MySQL database under Linux or Windows (where MySQL also maps names to lowercase). Under Windows, DSNs are set up in the ODBC applet in the Control Panel ('Data Sources (ODBC)' in the 'Administrative Tools' section on 2000/XP).

```
> library(RODBC)
## tell it to map names to l/case
```

```
> channel <- odbcConnect("testdb", uid="ripley", case="tolower")
## load a data frame into the database
> data(USArrests)
> sqlSave(channel, USArrests, rownames = "state", addPK = TRUE)
> rm(USArrests)
## list the tables in the database
> sqlTables(channel)
  TABLE_QUALIFIER TABLE_OWNER TABLE_NAME TABLE_TYPE REMARKS
1                              usarrests      TABLE
## list it
> sqlFetch(channel, "USArrests", rownames = "state")
              murder assault urbanpop rape
Alabama         13.2     236       58 21.2
Alaska          10.0     263       48 44.5
    ...
## an SQL query, originally on one line
> sqlQuery(channel, "select state, murder from USArrests
           where rape > 30 order by murder")
       state murder
1 Colorado      7.9
2 Arizona       8.1
3 California    9.0
4 Alaska       10.0
5 New Mexico   11.4
6 Michigan     12.1
7 Nevada       12.2
8 Florida      15.4
## remove the table
> sqlDrop(channel, "USArrests")
## close the connection
> odbcClose(channel)
```

As a simple example of using ODBC under Windows with a Excel spreadsheet, we can read from a spreadsheet by

```
> library(RODBC)
> channel <- odbcConnectExcel("bdr.xls")
## list the spreadsheets
> sqlTables(channel)
  TABLE_CAT TABLE_SCHEM        TABLE_NAME   TABLE_TYPE REMARKS
1 C:\\bdr            NA           Sheet1$ SYSTEM TABLE      NA
2 C:\\bdr            NA           Sheet2$ SYSTEM TABLE      NA
3 C:\\bdr            NA           Sheet3$ SYSTEM TABLE      NA
4 C:\\bdr            NA Sheet1$Print_Area        TABLE      NA
## retrieve the contents of sheet 1, by either of
> sh1 <- sqlFetch(channel, "Sheet1")
> sh1 <- sqlQuery(channel, "select * from [Sheet1$]")
```

Notice that the specification of the table is different from the name returned by `sqlTables`: `sqlFetch` is able to map the differences.

5 Binary files

Binary connections (Chapter 6 [Connections], page 21) are now the preferred way to handle binary files.

5.1 Binary data formats

Packages **hdf5**, **RNetCDF** and **ncdf** on CRAN provide interfaces to NASA's HDF5 (Hierarchical Data Format, see http://hdf.ncsa.uiuc.edu/HDF5/) and to UCAR's netCDF data files (network Common Data Form, see http://www.unidata.ucar.edu/packages/netcdf/).

Both of these are systems to store scientific data in array-oriented ways, including descriptions, labels, formats, units, ... HDF5 also allows *groups* of arrays, and the R interface maps lists to HDF5 groups, and can write numeric and character vectors and matrices.

Package **ncvar** on CRAN provides a higher-level R interface to netCDF data files *via* **RNetCDF**.

There is also a package **rhdf5** available from http://www.bioconductor.org.

5.2 dBase files (DBF)

`dBase` was a DOS program written by Ashton-Tate and later owned by Borland which has a binary flat-file format that became popular, with file extension '`.dbf`'. It has been adopted for the 'Xbase' family of databases, covering dBase, Clipper, FoxPro and their Windows equivalents Visual dBase, Visual Objects and Visual FoxPro (see http://www.e-bachmann.dk/docs/xbase.htm). A dBase file contains a header and then a series of fields and so is most similar to an R data frame. The data itself is stored in text format, and can include character, logical and numeric fields, and other types in later versions (see http://clicketyclick.dk/docs/data_types.html).

Functions `read.dbf` and `write.dbf` provide ways to read and write basic DBF files on all R platforms. For Windows users `odbcConnectDbase` in package **RODBC** provides more comprehensive facilities to read DBF files *via* Microsoft's dBase ODBC driver (and the Visual FoxPro driver can also be used via `odbcDriverConnect`).

6 Connections

Connections are used in R in the sense of Chambers (1998), a set of functions to replace the use of file names by a flexible interface to file-like objects.

6.1 Types of connections

The most familiar type of connection will be a file, and file connections are created by function `file`. File connections can (if the OS will allow it for the particular file) be opened for reading or writing or appending, in text or binary mode. In fact, files can be opened for both reading and writing, and R keeps a separate file position for reading and writing.

Note that by default a connection is not opened when it is created. The rule is that a function using a connection should open a connection (needed) if the connection is not already open, and close a connection after use if it opened it. In brief, leave the connection in the state you found it in. There are generic functions `open` and `close` with methods to explicitly open and close connections.

Files compressed via the algorithm used by `gzip` can be used as connections created by the function `gzfile`, whereas files compressed by `bzip2` can be used via `bzfile`.

Unix programmers are used to dealing with special files `stdin`, `stdout` and `stderr`. These exist as *terminal connections* in R. They may be normal files, but they might also refer to input from and output to a GUI console. (Even with the standard Unix R interface, `stdin` refers to the lines submitted from `readline` rather than a file.)

The three terminal connections are always open, and cannot be opened or closed. `stdout` and `stderr` are conventionally used for normal output and error messages respectively. They may normally go to the same place, but whereas normal output can be re-directed by a call to `sink`, error output is sent to `stderr` unless re-directed by `sink, type="message")`. Note carefully the language used here: the connections cannot be re-directed, but output can be sent to other connections.

Text connections are another source of input. They allow R character vectors to be read as if the lines were being read from a text file. A text connection is created and opened by a call to `textConnection`, which copies the current contents of the character vector to an internal buffer at the time of creation.

Text connections can also be used to capture R output to a character vector. `textConnection` can be asked to create a new character object or append to an existing one, in both cases in the user's workspace. The connection is opened by the call to `textConnection`, and at all times the complete lines output to the connection are available in the R object. Closing the connection writes any remaining output to a final element of the character vector.

Pipes are a special form of file that connects to another process, and pipe connections are created by the function `pipe`. Opening a pipe connection for writing (it makes no sense to append to a pipe) runs an OS command, and connects its standard input to whatever R then writes to that connection. Conversely, opening a pipe connection for input runs an OS command and makes its standard output available for R input from that connection.

URLs of types '`http://`', '`ftp://`' and '`file://`' can be read from using the function `url`. For convenience, `file` will also accept these as the file specification and call `url`.

Sockets can also be used as connections via function `socketConnection` on platforms which support Berkeley-like sockets (most Unix systems, Linux and Windows). Sockets can be written to or read from, and both client and server sockets can be used.

6.2 Output to connections

We have described functions `cat`, `write`, `write.table` and `sink` as writing to a file, possibly appending to a file if argument `append = TRUE`, and this is what they did prior to R version 1.2.0.

The current behaviour is equivalent, but what actually happens is that when the `file` argument is a character string, a file connection is opened (for writing or appending) and closed again at the end of the function call. If we want to repeatedly write to the same file, it is more efficient to explicitly declare and open the connection, and pass the connection object to each call to an output function. This also makes it possible to write to pipes, which was implemented earlier in a limited way via the syntax `file = "|cmd"` (which can still be used).

There is a function `writeLines` to write complete text lines to a connection.

Some simple examples are

```
zz <- file("ex.data", "w")  # open an output file connection
cat("TITLE extra line", "2 3 5 7", "", "11 13 17",
    file = zz, sep = "\n")
cat("One more line\n", file = zz)
close(zz)

## convert decimal point to comma in output, using a pipe (Unix)
## both R strings and (probably) the shell need \ doubled
zz <- pipe(paste("sed s/\\\\./,/ >", "outfile"), "w")
cat(format(round(rnorm(100), 4)), sep = "\n", file = zz)
close(zz)
## now look at the output file:
file.show("outfile", delete.file = TRUE)

## capture R output: use examples from help(lm)
zz <- textConnection("ex.lm.out", "w")
sink(zz)
example(lm, prompt.echo = "> ")
sink()
close(zz)
## now 'ex.lm.out' contains the output for futher processing.
## Look at it by, e.g.,
cat(ex.lm.out, sep = "\n")
```

6.3 Input from connections

The basic functions to read from connections are `scan` and `readLines`. These take a character string argument and open a file connection for the duration of the function call,

but explicitly opening a file connection allows a file to be read sequentially in different formats.

Other functions that call `scan` can also make use of connections, in particular `read.table`.

Some simple examples are

```
## read in file created in last examples
readLines("ex.data")
unlink("ex.data")

## read listing of current directory (Unix)
readLines(pipe("ls -1"))

# remove trailing commas from an input file.
# Suppose we are given a file 'data' containing
450, 390, 467, 654,  30, 542, 334, 432, 421,
357, 497, 493, 550, 549, 467, 575, 578, 342,
446, 547, 534, 495, 979, 479
# Then read this by
scan(pipe("sed -e s/,$// data"), sep=",")
```

For convenience, if the `file` argument specifies a FTP or HTTP URL, the URL is opened for reading via `url`. Specifying files via '`file://foo.bar`' is also allowed.

6.3.1 Pushback

C programmers may be familiar with the `ungetc` function to push back a character onto a text input stream. R connections have the same idea in a more powerful way, in that an (essentially) arbitrary number of lines of text can be pushed back onto a connection via a call to `pushBack`.

Pushbacks operate as a stack, so a read request first uses each line from the most recently pushbacked text, then those from earlier pushbacks and finally reads from the connection itself. Once a pushbacked line is read completely, it is cleared. The number of pending lines pushed back can be found via a call to `pushBackLength`.

A simple example will show the idea.

```
> zz <- textConnection(LETTERS)
> readLines(zz, 2)
[1] "A" "B"
> scan(zz, "", 4)
Read 4 items
[1] "C" "D" "E" "F"
> pushBack(c("aa", "bb"), zz)
> scan(zz, "", 4)
Read 4 items
[1] "aa" "bb" "G"  "H"
> close(zz)
```

Pushback is only available for connections opened for input in text mode.

6.4 Listing and manipulating connections

A summary of all the connections currently opened by the user can be found by `showConnections()`, and a summary of all connections, including closed and terminal connections, by `showConnections(all = TRUE)`

The generic function `seek` can be used to read and (on some connections) reset the current position for reading or writing. Unfortunately it depends on OS facilities which may be unreliable (e.g. with text files under Windows). Function `isSeekable` reports if `seek` can change the position on the connection given by its argument.

The function `truncate` can be used to truncate a file opened for writing at its current position. It works only for `file` connections, and is not implemented on all platforms.

6.5 Binary connections

Functions `readBin` and `writeBin` read to and write from binary connections. A connection is opened in binary mode by appending `"b"` to the mode specification, that is using mode `"rb"` for reading, and mode `"wb"` or `"ab"` (where appropriate) for writing. The functions have arguments

```
readBin(con, what, n = 1, size = NA, endian = .Platform$endian)
writeBin(object, con, size = NA, endian = .Platform$endian)
```

In each case `con` is a connection which will be opened if necessary for the duration of the call, and if a character string is given it is assumed to specify a file name.

It is slightly simpler to describe writing, so we will do that first. `object` should be an atomic vector object, that is a vector of mode `numeric`, `integer`, `logical`, `character`, `complex` or `raw`, without attributes. By default this is written to the file as a stream of bytes exactly as it is represented in memory.

`readBin` reads a stream of bytes from the file and interprets them as a vector of mode given by `what`. This can be either an object of the appropriate mode (e.g. `what=integer()`) or a character string describing the mode (one of the five given in the previous paragraph or `"double"` or `"int"`). Argument `n` specifies the maximum number of vector elements to read from the connection: if fewer are available a shorter vector will be returned. Argument `signed` allows 1-byte and 2-byte integers to be read as signed (the default) or unsigned integers.

The remaining two arguments are used to write or read data for interchange with another program or another platform. By default binary data is transferred directly from memory to the connection or *vice versa*. This will not suffice if the file is to be transferred to a machine with a different architecture, but between almost all R platforms the only change needed is that of byte-order. Common PCs (‘`ix86`’-based and ‘`x86_64`’-based machines), Compaq Alpha and Vaxen are *little-endian*, whereas Sun Sparc, mc680x0 series, IBM R6000, Apple Macintosh, SGI and most others are *big-endian*. (Network byte-order (as used by XDR, eXternal Data Representation) is big-endian.) To transfer to or from other programs we may need to do more, for example to read 16-bit integers or write single-precision real numbers. This can be done using the `size` argument, which (usually) allows sizes 1, 2, 4, 8 for integers and logicals, and sizes 4, 8 and perhaps 12 or 16 for reals. Transferring at different sizes can lose precision, and should not be attempted for vectors containing `NA`’s.

Character strings are read and written in C format, that is as a string of bytes terminated by a zero byte. Functions `readChar` and `writeChar` provide greater flexibility.

6.5.1 Special values

Functions `readBin` and `writeBin` will pass missing and special values, although this should not be attempted if a size change is involved.

The missing value for R logical and integer types is `INT_MIN`, the smallest representable `int` defined in the C header '`limits.h`', normally corresponding to the bit pattern `0x80000000`.

The representation of the special values for R numeric and complex types is machine-dependent, and possibly also compiler-dependent. The simplest way to make use of them is to link an external application against the standalone `Rmath` library which exports double constants `NA_REAL`, `R_PosInf` and `R_NegInf`, and include the header '`Rmath.h`' which defines the macros `ISNAN` and `R_FINITE`.

If that is not possible, on all common platforms IEC 60559 (aka IEEE 754) arithmetic is used, so standard C facilities can be used to test for or set `Inf`, `-Inf` and `NaN` values. On such platforms `NA` is represented by the `NaN` value with low-word `0x7a2` (1954 in decimal).

Character missing values are written as `NA`, and there are no provision to recognize character values as missing (as this can be done by re-assigning them once read).

7 Network interfaces

Some limited facilities are available to exchange data at a lower level across network connections.

7.1 Reading from sockets

Base R comes with some facilities to communicate *via* BSD sockets on systems that support them (including the common Linux, Unix and Windows ports of R). One potential problem with using sockets is that these facilities are often blocked for security reasons or to force the use of Web caches, so these functions may be more useful on an intranet than externally. For new projects it is suggested that socket connections are used instead.

The earlier low-level interface is given by functions `make.socket`, `read.socket`, `write.socket` and `close.socket`.

7.2 Using `download.file`

Function `download.file` is provided to read a file from a Web resource via FTP or HTTP and write it to a file. Often this can be avoided, as functions such as `read.table` and `scan` can read directly from a URL, either by explicitly using `url` to open a connection, or implicitly using it by giving a URL as the `file` argument.

7.3 DCOM interface

DCOM is a Windows protocol for communicating between different programs, possibly on different machines. Thomas Baier's `StatConnector` program available from CRAN under Software->Other->Non-standard provides an interface to the proxy DLL in package **rscproxy** and makes an DCOM server. This can be used to pass simple objects (vectors and matrices) to and from R and to submit commands to R. (It is not clear if this is still functional: there is another version in the **RExcelInstaller** package.)

The program comes with a Visual Basic demonstration and an Excel plug-in by Erich Neuwirth available. This interface is in the other direction to most of those considered here in that it is another application (Excel, or written in Visual Basic) that is the client and R is the server.

Another (D)COM server is available from http://www.omegahat.org/, which allows R objects to be exported as COM values. That site also has packages `RDCOMClient` and `SWinTypeLibs` which allow R to act as a (D)COM client.

7.4 CORBA interface

CORBA (Common Object Request Broker Architecture) is similar to DCOM, allowing applications to call methods, or operations, in server objects running in other applications, potentially programmed in different languages and running on different machines. There is a **CORBA** package available from the Omegahat project (at http://www.omegahat.org/RSCORBA/), currently for Unix but a Windows version looks to be possible.

This package allows R commands to be used to locate available CORBA servers, query the methods they provide, and dynamically invoke methods on these objects. R values given

as arguments in these calls are exported in the call and made available to that operation invocation. Primitive data types (vectors and lists) are exported by default, while more complex objects are exported by reference. Examples of using this include communicating with the Gnumeric (http://www.gnumeric.org) spreadsheet, and also interacting with the data visualization system ggobi.

One can also create CORBA servers within R, allowing other applications to call these methods. For example, one might offer access to a particular dataset or to some of R's modelling software. This is done dynamically by combining R data objects and functions. This allows one to explicitly export data and functionality from R.

One can also use the **CORBA** package to achieve distributed, parallel computing in R. One R session acts as a manager and dispatches tasks to different servers running in other R worker sessions. This uses the ability to invoke asynchronous or background CORBA calls in R. More information is available from the Omegahat project at http://www.omegahat.org/RSCORBA/.

8 Reading Excel spreadsheets

The most common R data import/export question seems to be 'how do I read an Excel spreadsheet'. This chapter collects together advice and options given earlier. Note that most of the advice is for pre-Excel 2007 spreadsheets: currently the only one of these methods that reads the '`.xlsx`' format is that *via* **RODBC**.

The first piece of advice is to avoid doing so if possible! If you have access to Excel, export the data you want from Excel in tab-delimited or comma-separated form, and use `read.delim` or `read.csv` to import it into R. (You may need to use `read.delim2` or `read.csv2` in a continental European locale that uses comma as the decimal point.) Exporting a DIF file and reading it using `read.DIF` is another possibility.

If you do not have Excel, many other programs are able to read such spreadsheets and export in a text format on both Windows and Unix, for example Gnumeric (`http://www.gnome.org/projects/gnumeric/`) and OpenOffice (`http://www.openoffice.org`). You can also cut-and-paste between the display of a spreadsheet in such a program and R: `read.table` will read from the R console or, under Windows, from the clipboard (via `file = "clipboard"` or `readClipboard`). The `read.DIF` function can also read from the clipboard.

Note that an Excel '`.xls`' file is not just a spreadsheet: such files can contain many sheets, and the sheets can contain formulae, macros and so on. Not all readers can read other than the first sheet, and may be confused by other contents of the file.

Windows users can use `odbcConnectExcel` in package **RODBC**. This can select rows and columns from any of the sheets in an Excel spreadsheet file (at least from Excel 97–2003, depending on your ODBC drivers: by calling `odbcConnect` directly versions back to Excel 3.0 can be read). The version `odbcConnectExcel2007` will read the Excel 2007 formats as well as earlier ones (provided the drivers are installed: see Section 4.3.2 [RODBC], page 18).

`Perl` users have contributed a module `OLE::SpreadSheet::ParseExcel` And a program `xls2csv.pl` to convert Excel 95–2003 spreadsheets to CSV files. Package **gdata** provides a basic wrapper in its `read.xls` function.

Packages **dataframes2xls** and **WriteXLS** each contain a function to *write* one or more data frames to an '`.xls`' file:, using Python and Perl respectively.

Appendix A References

R. A. Becker, J. M. Chambers and A. R. Wilks (1988) *The New S Language. A Programming Environment for Data Analysis and Graphics.* Wadsworth & Brooks/Cole.

J. Bowman, S. Emberson and M. Darnovsky (1996) *The Practical SQL Handbook. Using Structured Query Language.* Addison-Wesley.

J. M. Chambers (1998) *Programming with Data. A Guide to the S Language.* Springer-Verlag.

P. Dubois (2000) *MySQL.* New Riders.

M. Henning and S. Vinoski (1999) *Advanced CORBA Programming with C++.* Addison-Wesley.

K. Kline and D. Kline (2001) *SQL in a Nutshell.* O'Reilly.

B. Momjian (2000) *PostgreSQL: Introduction and Concepts.* Addison-Wesley. Also downloadable at http://www.postgresql.org/docs/awbook.html.

T. M. Therneau and P. M. Grambsch (2000) *Modeling Survival Data. Extending the Cox Model.* Springer-Verlag.

E. J. Yarger, G. Reese and T. King (1999) *MySQL & mSQL.* O'Reilly.

Function and variable index

T

U

W

Concept index