R Data Import/Export

Version 1.2.0 (2000-11-14)

R Development Core Team

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.

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

foreign	Thomas Lumley, Saikat DebRoy and Douglas Bates
hdf5	Marcus Daniels
netCDF	Thomas Lumley
RmSQL	Torsten Hothorn
RMySQL	David James and Saikat DebRoy
RODBC	Michael Lapsley
RPgSQL	Timothy Keitt
Rstreams	Brian Ripley and Duncan Murdoch
stataread	Thomas Lumley

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 (and hence in the Devel section on CRAN) but they already provide useful functionality.

Unless otherwise stated, everything described in this manual is available for both Unix/Linux and Windows versions of 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 the other system do the work! (For example Therneau & Grambsch (2000) comment that they prefer to do data manipulation in SAS and then use **survival5** in S for the analysis.)

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 5.

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 commonest applications on their computers for that purpose). However, this is not always possible, and Chapter 3 [Importing from other statistical systems], page 8 discusses what facilities are available to access such files directly from R.

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 17.

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 9. Importing data via network connections is discussed in Chapter 6 [Network connections], page 19.

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 See Chapter 5 [Binary files], page 17.)

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`.

The commonest 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**

   These functions are based on `cat` not `print`, and the precision to which numbers are printed is governed by the current setting of `options(digits)`. It may be necessary to increase this to avoid losing precision. 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
      ...
   ```

   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. Unfortunately, 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.

   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 variables. Factors are never quoted (and most character variables in data frames are factors).

   Some care is needed if the strings contain embedded quotes.

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

   is much as one might expect, but is probably not acceptable to other programs. Better solutions include

```
> equote <- function(x)
    if(is.character(x) gsub('"', '\\\\"', x)) else x
> write.table(lapply(df, equote))
"a"
"1" "a \" quote"
> dquote <- function(x)
    if(is.character(x) gsub('"', '""', x)) else x
> write.table(lapply(df, dquote))
"a"
"1" "a "" quote"
```

   the latter being the form of escape commonly used by spreadsheets.

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.

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.

Some of the issues to consider are:

1. **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).

2. **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 `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.

3. **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.

4. **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.

5. **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`.

6. **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`.

7. **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 connection with `fill = TRUE`, perhaps to indicate missing data in a regular layout.

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

```
row.lens=
 [1] 4 4 5 5 5 4 6 4 4 4 5 4 4 5 5
Error in read.table("file.dat", header = TRUE) :
        all rows must have the same length.
```

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

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.

2.3 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.

One reason can be to reduce memory usage. Function `read.table` reads the whole data file into memory and processes it into a data frame, thereby creating another copy of the data. If the post-processing is not required (for example if most of the variables are to be discarded) it may be feasible to use `scan` but not to use `read.table`.

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.

2.4 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.

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(7, 1, 28), 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.

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 Stata

Import and export facilities for Stata '`.dta`' files and provided by package **stataread** on CRAN. This is a binary file format, and files from versions 5.0 and 6.0 of Stata can be read and (despite the package name) written by functions `read.dta` and `write.dta`.

3.2 Minitab

Package **foreign** in the Devel section on CRAN provides a function `read.mtp` to read a 'Minitab Portable Worksheet'. This returns the components of the worksheet as an R list.

3.3 SAS

Package **foreign** in the Devel section on CRAN provides a function `read.xport` to read a file in SAS Transport (XPORT) format and return a list of data frames.

3.4 S-PLUS

Package **Rstreams** on CRAN contains a function `readSfile` which can read binary objects produced by S-PLUS 3.x, 4.x or 2000 on 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.

3.5 SPSS

There are currently no facilities to import SPSS data files, but it may be possible to use the PSPP package (`http://pspp.stat.wisc.edu`) to convert these files to a text format.

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 (tens of) 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, `www.informix.com`; Oracle, `www.oracle.com`; Sysbase, `www.sysbase.com`; 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 Sysbase 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 all of these data sources. It originated on Windows (`http://www.microsoft.com/data/odbc/`) but is also implemented on Linux.

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: SQL-92 is ISO/IEC 9075, also known as ANSI X3.135-1992) for an interface language called SQL (Structured Query Language, sometimes pronounced 'sequel': see Bowman *et al.* 1996) 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'.

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).

name	description
`float(p)`	Real number, with optional precision
`integer`	32-bit integer
`smallint`	16-bit integer
`character(n)`	fixed-length character string
`character varying(n)`	variable-length character string
`boolean`	true or false.
`date`	calendar date
`time`	time of day
`timestamp`	date and time

There are variants on `time` and `timestamp`, `with timezone`.

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

4.3 R interface packages

There are four 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 (except **RmSQL**) to retrieve the result as a whole as a data frame or in pieces (usually as groups of rows, but **RPgSQL** can retrieve columns). All except **RODBC** are (currently) tied to one DBMS.

4.3.1 Package RPgSQL

Package **RPgSQL** at `http://rpgsql.sourceforge.net/` and on CRAN provides an interface to PostgreSQL (`http://www.postgresql.org`).

PostgreSQL is described by its developers as 'the most advanced open source database server' (Momjian, 2000). It would appear to be buildable for most Unix-alike OSes and Windows (under Cygwin or U/Win). PostgreSQL has most of the features of the commercial RDBMSs.

RPgSQL is the most mature and comprehensive of these RDBMS interfaces.

To make use of **RPgSQL**, first open a connection to a database using `db.connect`. (Currently only one connection can be open at a time.) Once a connection is open an R data

frame can be copied to a PostgreSQL table by `db.write.table`, whereas `db.read.table` copies a PostgreSQL table to an R data frame.

RPgSQL has the interesting concept of a *proxy data frame*. A data frame proxy is an R object that inherits from the `"data.frame"` class, but contains no data. All accesses to the proxy data frame generate the appropriate SQL query and retrieve the resulting data from the database. A proxy data frame is set up by a call to `bind.db.proxy`. To remove the proxy, just remove the object which `bind.db.proxy` created.

A finer level of control is available via sending SQL queries to the PostgreSQL server via `db.execute`. Once this is done, `db.fetch.result` can be used to fetch the whole result as a data frame. Functions such as `db.result.columns` and `db.result.rows` will report the number of columns and rows in the resulting table, and `db.read.column` will fetch a single column (as a vector). An individual cell in the table can be read by `db.result.get.value`.

One disadvantage is that SQL queries get mapped to lower case, so for maximal flexibility, only use lower case in R names. Functions `sql.insert` and `sql.select` provide convenience wrappers for the INSERT and SELECT queries.

We can explore these functions in a simple example. The database '`testdb`' had already been set up, and as PostgreSQL was running on a standalone machine no further authentication was required to connect.

```
> library(RPgSQL)
> db.connect(dbname="testdb")   # add authentication as needed
Connected to database "testdb" on ""
> data(USArrests)
> usarrests <- USArrests
> names(usarrests) <- tolower(names(USArrests))
> db.write.table(USArrests, write.row.names = TRUE)
> db.write.table(usarrests, write.row.names = TRUE)
> rm(USArrests, usarrests)
## db.ls lists tables in the database.
> db.ls()
[1] "USArrests"  "usarrests"
> db.read.table("USArrests")
               Murder Assault UrbanPop Rape
Alabama          13.2     236       58 21.2
Alaska           10.0     263       48 44.5
   ...
## set up a proxy data frame. Remember USArrests has been removed
> bind.db.proxy("USArrests")
## USArrests is now a proxy, so all accesses are to the database
> USArrests[, "Rape"]
   Rape
1  21.2
2  44.5
   ...
> rm(USArrests) # remove proxy
> db.execute("SELECT rpgsql_row_names, murder FROM usarrests",
             "WHERE rape > 30 ORDER BY murder", clear=FALSE)
> db.fetch.result()
```

```
            murder
Colorado      7.9
Arizona       8.1
California    9.0
Alaska       10.0
New Mexico   11.4
Michigan     12.1
Nevada       12.2
Florida      15.4
> db.rm("USArrests", "usarrests")  # use ask=F to skip confirmation
Destroy table USArrests? y
Destroy table usarrests? y
> db.ls()
character(0)
> db.disconnect()
```

Notice how the row names are mapped if `write.row.names = TRUE` to a field `rpgsql_row_names` in the database table and transparently restored provided we preserve that field in the query.

RPgSQL provides means to extend its mapping between R classes within a data frame and PostgreSQL types.

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 Linux and Windows, and many database systems provide support ODBC, including most of those on Windows (such as Microsoft Access), and MySQL, Oracle and PostgreSQL on Unix/Linux.

You will need an ODBC Driver Manager such as unixODBC (`http://www.unixODBC.org`) or iOBDC (`http://www.iODBC.org`) on Unix/Linux, and an installed driver for your database system. The FreeODBC project (`http://www.jepstone.net/FreeODBC/`) is a repository of information related to ODBC.

Two groups of interface functions are provided. The `odbc*` group provide a low-level interface to the basic ODBC functions: see the help page (`?RODBC`) for details. The `sql*` group provide an interface between R data frames and SQL tables.

Up to 16 simultaneous connections are possible. A connection is opened by a call to `odbcConnect` which returns a handle used for subsequent access to the database. A connection is closed by `odbcClose`. 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. `sqlGetResults` returns a data frame, but the raw results as a character matrix can be obtained from `odbcGetResults`.

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 Windows NT (where MySQL also maps names to lowercase). Under Windows, DSNs are set up in the ODBC applet in the Control Panel.

```
> library(RODBC)
## tell it to map names to l/case
> (channel <- odbcConnect("testdb", uid="ripley", case="tolower"))
[1] 0
## check it worked as failure is indicated by returning -1
## load a data frame into the database
> data(USArrests)
> sqlSave(channel, USArrests, rownames="state")
> rm(USArrests)
## list the tables in the database
> sqlTables(channel)
  TABLE_QUALIFIER TABLE_OWNER TABLE_NAME TABLE_TYPE REMARKS
1              NA          NA  usarrests      TABLE      NA
## list it
> sqlFetch(channel,"USArrests")
            state murder assault urbanpop rape
1         Alabama   13.2     236       58 21.2
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)
```

4.3.3 Package RMySQL

Package **RMySQL** in the Devel section on CRAN provides an interface to the MySQL database system (see `http://www.mysql.com` and Dubois, 2000.). This is part of a project to provide a common API for access from R to relational DBMSs using SQL. Currently this provides lower-level facilities than **RPgSQL** or **RODBC**.

MySQL exists on Unix/Linux and Windows. (Its current status appears to be that the source is freely available and beta versions are free but release binary versions are commercial on Windows.) 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.)

A call to the function `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 `close`.

SQL queries can be sent by either `dbExec` or `dbExecStatement`. `dbExec` sends the query and retrieves the results as a list. (It will fail unless the results are from a SELECT-like statement.) `dbExecStatement` sends the query and returns an object of class `"MySQLResultSet"` which can be used to retrieve the results, and subsequently used to `close` the result.

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

There is no simple way to load a data frame into the database, so we have to work harder using direct SQL statements.

```
> library(RMySQL)
## open a connection to a MySQL database
> con <- dbConnect(MySQL(), dbname = "test")
## list the tables in the database
> getTables(con)
## load a data frame into the database, deleting any existing copy
> data(USArrests)
## write the values to a file
> write.table(USArrests, "/tmp/arrests.dat", sep="\t",
              row.names = FALSE, col.names = FALSE)
> if(!is.na(match("arrests", getTables(con)[[1]])))
## delete any existing table
    dbExecStatement(con, "drop table arrests")
## character string here needs to be on one line
> dbExecStatement(con,
    "create table arrests (Murder float, Assault float,
     UrbanPop float, Rape float)")
> dbExecStatement(con,
    'load data infile "/tmp/arrests.dat" into table arrests')
> unlink("/tmp/arrests.dat")
## Select from the loaded database.
> as.data.frame(dbExec(con,
    "select * from arrests where Rape > 30 order by Murder"))
  Murder Assault UrbanPop Rape
1    7.9     204       78 38.7
2    8.1     294       80 31.0
3    9.0     276       91 40.6
4   10.0     263       48 44.5
5   11.4     285       70 32.1
6   12.1     255       74 35.1
7   12.2     252       81 46.0
```

```
8   15.4     335       80 31.9
> close(con)
```

4.3.4 Package RmSQL

Package **RmSQL** on CRAN provides an interface to the Mini SQL database system (also known as mSQL, `http://www.hughes.com.au`, Yarger *et al.*, 1999). The package documentation describes mSQL as

> Note that mSQL is NOT GPL licenced but free of charge for universities and noncommercial organisations.

RmSQL provides the most basic interface of those in this chapter.

A database connection is opened by first selecting a host with `msqlConnect` and then a database by `msqlSelect`. The connection is closed by a call to `msqlClose`. Then an SQL query is sent by a call to `msqlQuery`, and the results stored by a call to `msqlStoreResult`. When a query is finished with, the result can be freed by `msqlFreeResult`.

Once the result of a query has been stored, the values can be retrieved row by row using `msqlFetchRow`. This fetches the rows in order unless the position is reset by a call to `msqlDataSeek`. A call to `msqlNumRows` gives the total number of rows in the result.

5 Binary files

5.1 Package Rstreams

Package **Rstreams** on CRAN provides a low-level interface to read from and write to binary files. It has been used to read in data from MRI experiments and Windows sound files, for example.

Rstreams' view of the file is as a stream of bytes. A file can be opened by the function `openstream` for either reading or writing. The return value is a number (in fact the file descriptor) that is used to reference the open file until it is closed by `closestream`.

Once a stream is open for reading, bytes can be transferred from it to an R object by one of the functions

```
readint(stream, n, size = 4, signed = TRUE, swapbytes = FALSE)
readfloat(stream, n, size = 8, swapbytes = FALSE)
readcomplex(stream, n, size = 8, swapbytes = FALSE)
readchar(stream, n = 1, len = NA, bufsize = 256)
```

These return an R object of an appropriate mode and storage mode. (Integers too large to be represented in storage mode `"integer"` will be read into a vector of storage mode `"double"`.) Here `n` is the number of items to be read, but fewer will be read if there is insufficient data on the file.

The size of the data components on file need not be the same as that in the machine running R, and data written on a little-endian machine can be read on a big-endian machine or *vice versa* by setting `swapbytes = TRUE`.

Character data can be read either as fixed-length blocks of bytes (by specifying `len`) or as ASCII-zero-delimited strings.

A file open for writing can be written to by any of

```
writeint(stream, data, size = 4, swapbytes = FALSE)
writefloat(stream, data, size = 8, swapbytes = FALSE)
writecomplex(stream, data, size = 8, swapbytes = FALSE)
writechar(stream, data, asciiz = FALSE)
```

where `data` is an R vector of an appropriate mode. There is no separator between character strings on the file unless `asciiz = TRUE` is specified.

Function `copystream` can copy a specified number of bytes from one open stream to another.

Function `truncate` truncates a stream at its current position, so all data after that point is lost.

5.2 Binary data formats

Packages **hdf5** and **netCDF** in the Devel area on CRAN provide experimental 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/`), respectively.

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

The R interface can only read netCDF, not write it.

6 Network connections

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

6.1 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.

The low-level interface is given by functions `make.socket`, `read.socket`, `write.socket` and `close.socket`. The function `httpclient` builds on these to read (directly) from an HTTP server.

6.2 Using `download.file`

There are functions `read.table.url` and `scan.url` that appear to work like `read.table` and `scan` but access a Web resource at a specified URL. Appearances are deceptive, however, as these functions (and `url.show` and `source.url`) first download the Web resource to a local file and then apply the standard functions. Various methods can be used to download the file, including the programs `wget` and `lynx` and the direct use of BSD sockets.

6.3 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 which ships with the Windows version of R 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.

The program comes with a Visual Basic demonstration, and there is an Excel plug-in by Erich Neuwirth available in the same area on CRAN. 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.

6.4 CORBA interface

An interface to R providing communication via CORBA objects is available from the Omegahat Project (`http://www.omeghat.org`).

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.

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

B. Momjian (2000) *PostgreSQL: Introduction and Concepts.* Draft book from `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

U

W

Concept index